Applied Network Science with R

George G. Vega Yon, Ph.D.

2018-01-19

Table of contents

1 Preface **6**

1.1 About the project 6

1.2 About the Author 7

1.3 AI Disclosure 7

2 Introduction **8**

3 R Basics **9**

3.1 Getting R 9

3.2 How to install packages 9

3.3 A ~~gentle~~ Quick n' Dirty Introduction to R 10

I Applications **20**

4 School networks **21**

4.1 Data preprocessing 21

4.1.1 Reading the data into R 21

4.1.2 Creating a unique id for each participant 22

4.2 Creating a network 24

4.2.1 From survey to edgelist 24

4.2.2 igraph network 29

4.3 Network descriptive stats 31

4.4 Plotting the network in igraph 35

4.4.1 Single plot 35

4.4.2 Multiple plots 39

4.5 Statistical tests 43

4.5.1 Is nomination number correlated with indegree? 43

5 Simulation and vizualization **45**
5.1 Random Graph Models . . . 46
5.2 Social Networks in Schools . . . 47
5.3 Reading a network . . . 50
5.4 Visualizing the network . . . 55
5.4.1 Vertex size . . . 55
5.4.2 Vertex color . . . 59
5.4.3 Vertex shape . . . 61

6 Egocentric networks **66**
6.1 Network files (graphml) . . . 67
6.2 Person files . . . 72
6.3 Ego files . . . 75
6.4 Edgelist files . . . 78
6.5 Putting all together . . . 79
6.5.1 Generating statistics using igraph . . . 79
6.5.2 Generating statistics based on ergm . . . 80
6.6 Saving the data . . . 82

7 Network diffusion **84**
7.1 Network diffusion of innovation . . . 84
7.1.1 Diffusion networks . . . 84
7.1.2 Thresholds . . . 84
7.2 The netdiffuseR R package . . . 85
7.2.1 Overview . . . 85
7.2.2 Datasets . . . 85
7.2.3 Visualization methods . . . 87
7.2.4 Problems . . . 91
7.3 Simulation of diffusion processes . . . 91
7.3.1 Simulating diffusion networks . . . 91
7.3.2 Rumor spreading . . . 93
7.3.3 Difussion . . . 95
7.3.4 Mentor Matching . . . 96
7.3.5 Example by changing threshold . . . 97
7.3.6 Problems . . . 102
7.4 Statistical inference . . . 102
7.4.1 Moran's I . . . 102

7.4.2 Using geodesics . . . 103
7.4.3 Structural dependence and permutation tests . . . 104
7.4.4 Idea . . . 105
7.4.5 Regression analysis . . . 108
7.4.6 Problems . . . 112

II Statistical inference 113

8 Exponential Random Graph Models 114
8.1 A naïve example . . . 116
8.2 Estimation of ERGMs . . . 118
8.3 The `ergm` package . . . 120
8.4 Running ERGMs . . . 124
8.5 Model Goodness-of-Fit . . . 129
8.6 More on MCMC convergence . . . 143
8.7 Mathematical Interpretation . . . 144
8.8 Markov independence . . . 145

9 Using constraints in ERGMs 147
9.1 Example 1: Interlocking egos and disconnected alters . . . 147
9.2 Example 2: Bi-partite networks . . . 154

10 Temporal Exponential Family Random Graph Models 161

11 Hypothesis testing in networks 162
11.1 Comparing networks . . . 162
11.1.1 Network bootstrap . . . 162
11.1.2 When the statistic is normal . . . 163
11.1.3 When the statistic is NOT normal . . . 164
11.2 Examples . . . 165
11.2.1 Average of node-level stats . . . 165

12 Stochastic Actor Oriented Models 166

13 Power calculation in network studies 167
13.1 Example 1: Spillover effects in egocentric studies . . . 167
13.2 Example 2: Spillover effects pre-post effect . . . 172

13.3 Example 3: First difference 177

III Foundations **181**

14 Bayes' Rule **182**

15 Markov Chain **184**

15.1 Metropolis Algorithm 184
15.2 Metropolis-Hastings 185
15.3 Likelihood-free MCMC 185

16 Power and sample size **186**

16.1 Error types 186
16.2 Example 1: Sample size for a proportion 187
16.3 Example 2: Sample size for a proportion (vis) 189

IV Appendix **194**

17 Datasets **195**

17.1 SNS data 195
17.1.1 About the data 195
17.1.2 Variables 195

References **197**

1 Preface

Statistical methods for networked systems are present in most disciplines. Despite language differences between areas, many methods developed to study specific problems can be helpful outside their original context; this is the premise of this book. **Applied Network Science with R** provides examples using the R programming language to study networked systems. Although most cases deal with social network analysis, the methods presented here can be applied to contexts such as biological networks, transportation networks, and many others.

The entire book was written using quarto–a literate programming system that allows mixing text and code–meaning that all the code presented is 100% executable and, thus, reproducible. The source code is available on GitHub at https://github.com/gvegayon/appliedsnar. Readers are encouraged to download the code and execute it on their machines using either RStudio or VScode.

Besides the R programming, we will be using RStudio. For data management, we will use `dplyr` and `data.table`. The network data management and visualization packages we will use are `igraph`, netdiffuseR, the statnet suite, and `netplot`.

1.1 About the project

This project began over six years ago as a part of a series of workshops and tutorials I ran at USC's **Center for Applied Network Analysis**. Today, I use it to gather and study statistical methods to analyze networks, emphasizing social and biological systems. Moreover, the book will use statistical computing methods as a core component.

1.2 About the Author

I am a Research Assistant Professor at the **University of Utah's Division of Epidemiology**, where I work on studying Complex Systems using Statistical Computing. I was born and raised in Chile. I have over ten years of experience developing scientific software focusing on high-performance computing, data visualization, and social network analysis. My training is in Public Policy (M.A. UAI, 2011), Economics (M.Sc. Caltech, 2015), and Biostatistics (Ph.D. USC, 2020).

I obtained my Ph.D. in Biostatistics under the supervision of **Prof. Paul Marjoram** and **Prof. Kayla de la Haye**, with my dissertation titled "*Essays on Bioinformatics and Social Network Analysis: Statistical and Computational Methods for Complex Systems.*"

If you'd like to learn more about me, please visit my website at https://ggvy.cl.

1.3 AI Disclosure

Starting in mid-2023, I have been using AI to help me write this book. Mainly, I use a combination of GitHub co-pilot, which aids with code and text, and Grammarly, which aids with grammar and style. AI's role has been to help me write faster and more accurately, but it has not been involved in the book's conceptualization or the development of the methods presented here.

2 Introduction

Social Network Analysis and Network Science have a long scholarly tradition. From social diffusion models to protein-interaction networks, these complex systems disciplines cover various problems across scientific fields. Yet, although these could be seen as wildly different, the object under the microscope is the same: networks.

With a long history (and insufficient inter-discipline collaboration, if you allow me to say) of scientific advances happening somewhat isolatedly, the potential for cross-pollination between disciplines within network science is immense.

This book attempts to compile the many methods available in the realm of complexity sciences, provide an in-depth mathematical examination–when possible–and provide a few examples illustrating their usage.

3 R Basics

R (R Core Team 2024) is a programming language oriented to statistical computing. R has become the *de facto* programming language in the social network community due to the large number of packages available for network analysis. R packages are collections of functions, data, and documentation that extend R. A good reference book for both novice and advanced users is "The Art of R programming" Matloff (2011)[1].

3.1 Getting R

You can get R from the Comprehensive R Archive Network website [CRAN] (link). CRAN is a network of servers worldwide that store identical, up-to-date versions of code and documentation for R. CRAN website also has a lot of information about R, including manuals, FAQs, and mailing lists.

Although R comes with a Graphical User Interface [GUI], I recommend getting an alternative like RStudio or VSCode. RStudio and VSCode are excellent companions for programming in R. While RStudio is more common among R users, VSCode is a more general-purpose IDE that can be used for many other programming languages, including Python and C++.

3.2 How to install packages

Nowadays, there are two ways of installing R packages (that I'm aware of), either using `install.packages`, which is a function shipped with R, or using the `devtools` R package to install a package from some remote repository other than CRAN, here are a few examples:

[1] Here a free pdf version distributed by the author.

```
# This will install the igraph package from CRAN
> install.packages("netdiffuseR")

# This will install the bleeding-edge version from the project's GitHub repo!
> devtools::install_github("USCCANA/netdiffuseR")
```

The first one, using `install.packages`, installs the CRAN version of netdiffuseR, whereas the line of code installs whatever version is published on https://github.com/USCCANA/netdiffuseR, which is usually called the development version.

In some cases, users may want/need to install packages from the command line as some packages need extra configuration to be installed. But we won't need to look at it now.

3.3 A ~~gentle~~ Quick n' Dirty Introduction to R

Some common tasks in R

0. Getting help (and reading the manual) is *THE MOST IMPORTANT* thing you should know about. For example, if you want to read the manual (help file) of the `read.csv` function, you can type either of these:

   ```
   ?read.csv
   ?"read.csv"
   help(read.csv)
   help("read.csv")
   ```

   If you are not fully aware of what is the name of the function, you can always use the *fuzzy search*

   ```
   help.search("linear regression")
   ??"linear regression"
   ```

1. In R, you can create new objects by either using the assign operator (`<-`) or the equal sign `=`, for example, the following two are equivalent: `r a <- 1 a = 1` Historically, the assign operator is the most commonly used.

2. R has several types of objects. The most basic structures in R are `vectors`, `matrix`, `list`, `data.frame`. Here is an example of creating several of these (each line is enclosed with parenthesis so that R prints the resulting element):

```
(a_vector     <- 1:9)
```

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

```
(another_vect <- c(1, 2, 3, 4, 5, 6, 7, 8, 9))
```

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

```
(a_string_vec <- c("I", "like", "netdiffuseR"))
```

```
[1] "I"           "like"        "netdiffuseR"
```

```
(a_matrix     <- matrix(a_vector, ncol = 3))
```

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

```
# Matrices can be of strings too
(a_string_mat <- matrix(letters[1:9], ncol=3))
```

```
     [,1] [,2] [,3]
[1,] "a"  "d"  "g"
[2,] "b"  "e"  "h"
[3,] "c"  "f"  "i"
```

```
# The `cbind` operator does "column bind"
(another_mat  <- cbind(1:4, 11:14))
```

```
     [,1] [,2]
[1,]    1   11
[2,]    2   12
[3,]    3   13
[4,]    4   14
```

```r
# The `rbind` operator does "row bind"
(another_mat2 <- rbind(1:4, 11:14))
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]   11   12   13   14
```

```r
(a_string_mat <- matrix(letters[1:9], ncol = 3))
```

```
     [,1] [,2] [,3]
[1,] "a"  "d"  "g"
[2,] "b"  "e"  "h"
[3,] "c"  "f"  "i"
```

```r
(a_list       <- list(a_vector, a_matrix))
```

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

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

```r
# same but with names!
(another_list <- list(my_vec = a_vector, my_mat = a_matrix))
```

```
$my_vec
[1] 1 2 3 4 5 6 7 8 9

$my_mat
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```r
# Data frames can have multiple types of elements; it
# is a collection of lists
(a_data_frame <- data.frame(x = 1:10, y = letters[1:10]))
```

```
    x y
1   1 a
2   2 b
3   3 c
4   4 d
5   5 e
6   6 f
7   7 g
8   8 h
9   9 i
10 10 j
```

3. Depending on the type of object, we can access its components using indexing:

```r
# First 3 elements
a_vector[1:3]
```

```
[1] 1 2 3
```

```r
# Third element
a_string_vec[3]
```

```
[1] "netdiffuseR"
```

```r
# A sub matrix
a_matrix[1:2, 1:2]
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

```r
# Third column
a_matrix[,3]
```

```
[1] 7 8 9
```

```
# Third row
a_matrix[3,]
```

```
[1] 3 6 9
```

```
# First 6 elements of the matrix. R stores matrices
# by column.
a_string_mat[1:6]
```

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

```
# These three are equivalent
another_list[[1]]
```

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

```
another_list$my_vec
```

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

```
another_list[["my_vec"]]
```

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

```
# Data frames are just like lists
a_data_frame[[1]]
```

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

```
a_data_frame[,1]
```

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

```
a_data_frame[["x"]]
```

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

```
a_data_frame$x
```

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

4. Control-flow statements

```
# The oldfashion forloop
for (i in 1:10) {
  print(paste("I'm step", i, "/", 10))
}
```

```
[1] "I'm step 1 / 10"
[1] "I'm step 2 / 10"
[1] "I'm step 3 / 10"
[1] "I'm step 4 / 10"
[1] "I'm step 5 / 10"
[1] "I'm step 6 / 10"
[1] "I'm step 7 / 10"
[1] "I'm step 8 / 10"
[1] "I'm step 9 / 10"
[1] "I'm step 10 / 10"
```

```
# A nice ifelse

for (i in 1:10) {

  if (i %% 2) # Modulus operand
    print(paste("I'm step", i, "/", 10, "(and I'm odd)"))
  else
    print(paste("I'm step", i, "/", 10, "(and I'm even)"))

}
```

```
[1] "I'm step 1 / 10 (and I'm odd)"
[1] "I'm step 2 / 10 (and I'm even)"
[1] "I'm step 3 / 10 (and I'm odd)"
[1] "I'm step 4 / 10 (and I'm even)"
[1] "I'm step 5 / 10 (and I'm odd)"
[1] "I'm step 6 / 10 (and I'm even)"
[1] "I'm step 7 / 10 (and I'm odd)"
[1] "I'm step 8 / 10 (and I'm even)"
[1] "I'm step 9 / 10 (and I'm odd)"
```

```
[1] "I'm step 10 / 10 (and I'm even)"
```

```
# A while
i <- 10
while (i > 0) {
  print(paste("I'm step", i, "/", 10))
  i <- i - 1
}
```

```
[1] "I'm step 10 / 10"
[1] "I'm step 9 / 10"
[1] "I'm step 8 / 10"
[1] "I'm step 7 / 10"
[1] "I'm step 6 / 10"
[1] "I'm step 5 / 10"
[1] "I'm step 4 / 10"
[1] "I'm step 3 / 10"
[1] "I'm step 2 / 10"
[1] "I'm step 1 / 10"
```

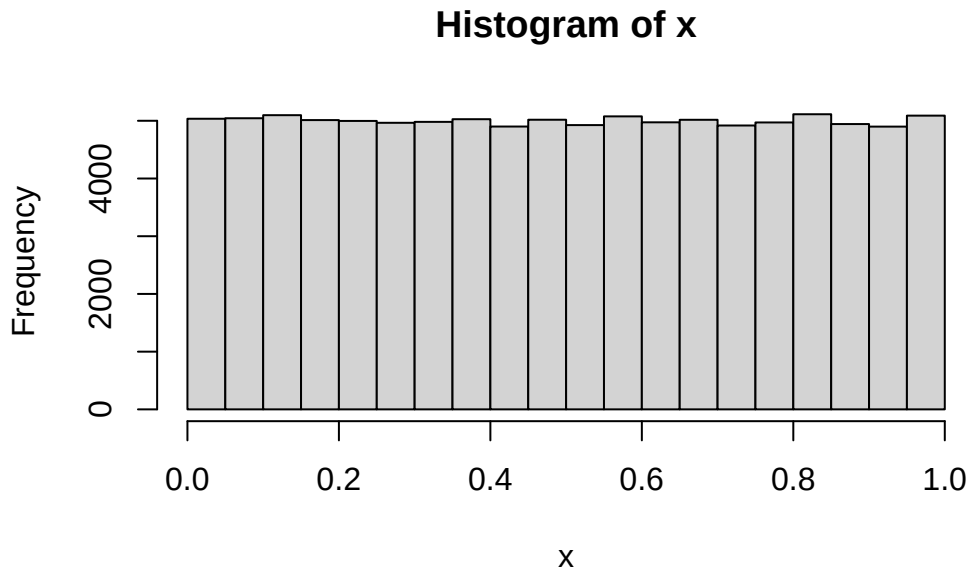
5. R has a compelling set of pseudo-random number generation functions. In general, distribution functions have the following name structure:

   a. Random Number Generation: `r[name-of-the-distribution]`, *e.g.*, `rnorm` for normal, `runif` for uniform.
   b. Density function: `d[name-of-the-distribution]`, e.g. `dnorm` for normal, `dunif` for uniform.
   c. Cumulative Distribution Function (CDF): `p[name-of-the-distribution]`, *e.g.*, `pnorm` for normal, `punif` for uniform.
   d. Inverse (quantile) function: `q[name-of-the-distribution]`, e.g. `qnorm` for the normal, `qunif` for the uniform.

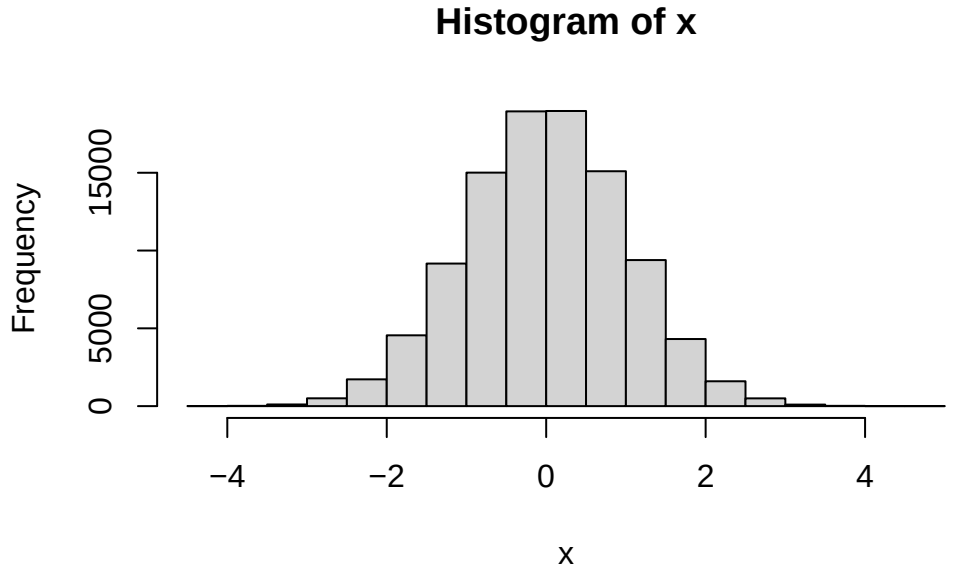
   Here are some examples:

```
# To ensure reproducibility
set.seed(1231)

# 100,000 Unif(0,1) numbers
```

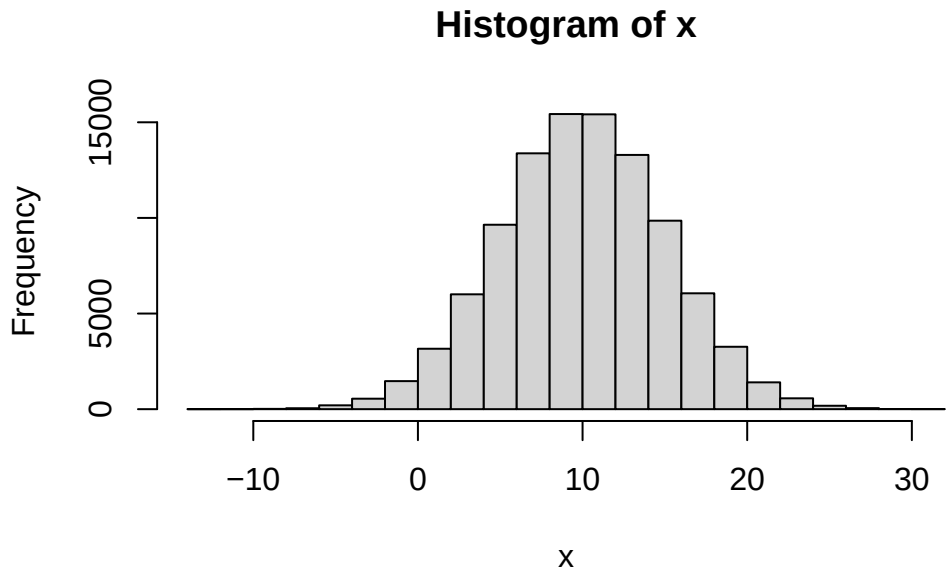
```r
x <- runif(1e5)
hist(x)
```



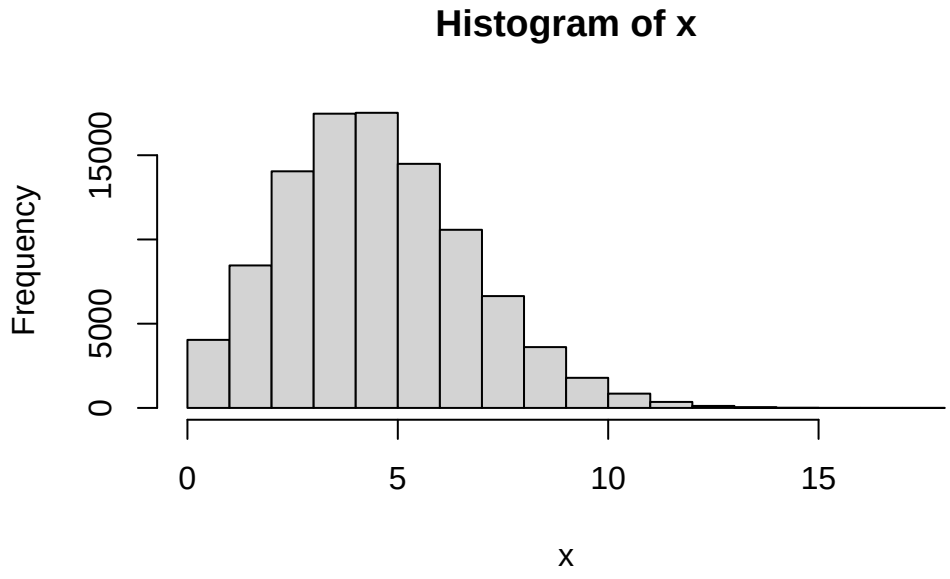

```r
# 100,000 N(0,1) numbers
x <- rnorm(1e5)
hist(x)
```



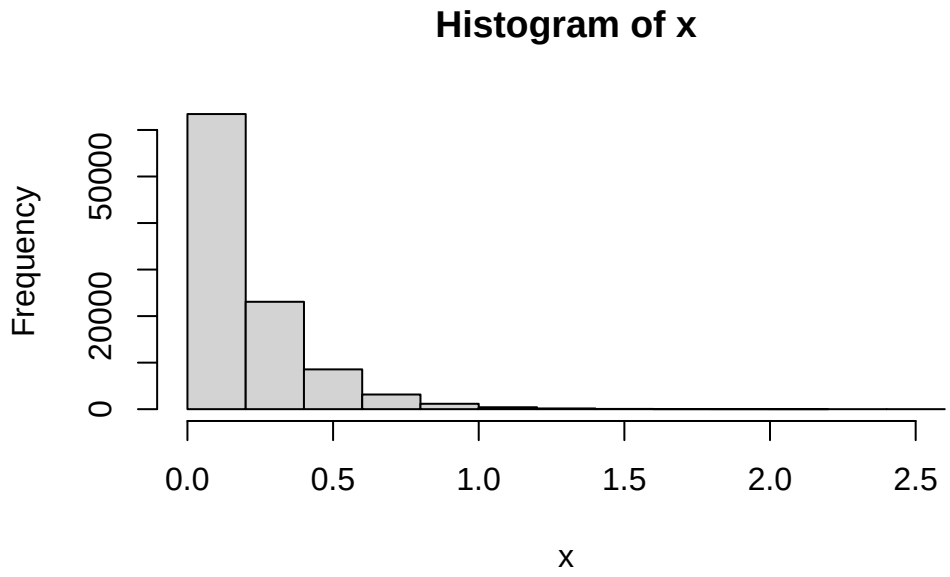
```r
# 100,000 N(10,25) numbers
x <- rnorm(1e5, mean = 10, sd = 5)
hist(x)
```




```r
# 100,000 Poisson(5) numbers
x <- rpois(1e5, lambda = 5)
hist(x)
```



```r
# 100,000 rexp(5) numbers
x <- rexp(1e5, 5)
hist(x)
```



More distributions are available at `??Distributions`.

For a nice intro to R, take a look at "The Art of R Programming" by Norman Matloff. For more advanced users, take a look at "Advanced R" by Hadley Wickham.

Part I

Applications

4 School networks

This chapter provides a start-to-finish example for processing survey-type data in R. The chapter features the Social Network Study [SNS] dataset. You can download the data for this chapter here, and the codebook for the data provided here is in the appendix.

The goals for this chapter are:

1. Read the data into R.
2. Create a network with it.
3. Compute descriptive statistics.
4. Visualize the network.

4.1 Data preprocessing

4.1.1 Reading the data into R

R has several ways of reading data. Your data can be Raw plain files like CSV, tab-delimited, or specified by column width. To read plain-text data, you can use the `readr` package (Wickham, Hester, and Bryan 2024). In the case of binary files, like Stata, Octave, or SPSS files, you can use the R package `foreign` (R Core Team 2023). If your data is formatted as Microsoft spreadsheets, the `readxl` R package (Wickham and Bryan 2023) is the alternative to use. In our case, the data for this session is in Stata format:

```r
library(foreign)

# Reading the data
dat <- foreign::read.dta("03-sns.dta")

# Taking a look at the data's first 5 columns and 5 rows
dat[1:5, 1:10]
```

```
  photoid school hispanic female1 female2 female3 female4 grades1 grades2
1       1    111        1      NA      NA       0       0      NA      NA
2       2    111        1       0      NA      NA       0     3.0      NA
3       7    111        0       1       1       1       1     5.0     4.5
4      13    111        1       1       1       1       1     2.5     2.5
5      14    111        1       1       1       1      NA     3.0     3.5
  grades3
1     3.5
2      NA
3     4.0
4     2.5
5     3.5
```

4.1.2 Creating a unique id for each participant

We must create a unique `id` using the school and photo `id`. Since both variables are numeric, encoding the id is a good way of doing this. For example, the last three numbers are the `photoid`, and the first numbers are the school `id`. To do this, we need to take into account the range of the variables:

```r
(photo_id_ran <- range(dat$photoid))
```

```
[1]    1 2074
```

As the variable spans up to 2074, we need to set the last 4 units of the variable to store the `photoid`. We will use `dplyr` (Wickham et al. 2023) to create this variable and call it `id`:

```
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
# Creating the variable
dat <- dat |>
  mutate(id = school*10000 + photoid)

# First few rows
dat |>
  head() |>
  select(school, photoid, id)
```

```
  school photoid      id
1    111       1 1110001
2    111       2 1110002
3    111       7 1110007
4    111      13 1110013
5    111      14 1110014
6    111      15 1110015
```

Wow, what happened in the last lines of code?! What is that `|>`? Well, that's the pipe operator[1], and it is an appealing way of writing nested function calls. In this case, instead of writing something like:

[1]Introduced in R version 4.1.0, base R's pipe operator `|>` works similarly to the `magrittr` pipe `%>%`. The key differences between these two are explained in https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/.

```
dat_filtered$id <- dat_filtered$school*10000 + dat_filtered$photoid
subset(head(dat_filtered), select = c(school, photoid, id))
```

4.2 Creating a network

- We want to build a social network. For that, we either use an adjacency matrix or an edgelist.
- Each individual of the SNS data nominated 19 friends from school. We will use those nominations to create the social network.
- In this case, we will create the network by coercing the dataset into an edgelist.

4.2.1 From survey to edgelist

Let's start by loading a couple of handy R packages. We will load `tidyr` (Wickham, Vaughan, and Girlich 2024) and `stringr` (Wickham 2023). We will use the first, `tidyr`, to reshape the data. The second, `stringr`, will help us processing strings using *regular expressions*[2].

```
library(tidyr)
library(stringr)
```

Optionally, we can use the `tibble` type of object, an alternative to the actual `data.frame`. This object provides *more efficient methods for matrices and data frames*.

```
dat <- as_tibble(dat)
```

What I like about `tibbles` is that when you print them on the console, these look nice:

```
dat
```

[2]Please refer to the help file `?'regular expression'` in R. The R package `rex` (Ushey, Hester, and Krzyzanowski 2021) is a friendly companion for writing regular expressions. There's also a neat (but experimental) RStudio add-in that can be very helpful for understanding how regular expressions work, the regexplain add-in.

```
# A tibble: 2,164 x 100
   photoid school hispanic female1 female2 female3 female4 grades1 grades2
     <int>  <int>    <dbl>   <int>   <int>   <int>   <int>   <dbl>   <dbl>
 1       1    111        1      NA      NA       0       0    NA      NA
 2       2    111        1       0      NA      NA       0     3      NA
 3       7    111        0       1       1       1       1     5       4.5
 4      13    111        1       1       1       1       1     2.5     2.5
 5      14    111        1       1       1       1      NA     3       3.5
 6      15    111        1       0       0       0       0     2.5     2.5
 7      20    111        1       1       1       1       1     2.5     2.5
 8      22    111        1      NA      NA       0       0    NA      NA
 9      25    111        0       1       1      NA       1     4.5     3.5
10      27    111        1       0      NA       0       0     3.5    NA
# i 2,154 more rows
# i 91 more variables: grades3 <dbl>, grades4 <dbl>, eversmk1 <int>,
#   eversmk2 <int>, eversmk3 <int>, eversmk4 <int>, everdrk1 <int>,
#   everdrk2 <int>, everdrk3 <int>, everdrk4 <int>, home1 <int>, home2 <int>,
#   home3 <int>, home4 <int>, sch_friend11 <int>, sch_friend12 <int>,
#   sch_friend13 <int>, sch_friend14 <int>, sch_friend15 <int>,
#   sch_friend16 <int>, sch_friend17 <int>, sch_friend18 <int>, ...
```

```r
# Maybe too much piping... but its cool!
net <- dat |>
  select(id, school, starts_with("sch_friend")) |>
  gather(key = "varname", value = "content", -id, -school) |>
  filter(!is.na(content)) |>
  mutate(
    friendid = school*10000 + content,
    year     = as.integer(str_extract(varname, "(?<=[a-z])[0-9]")),
    nnom     = as.integer(str_extract(varname, "(?<=[a-z][0-9])[0-9]+"))
  )
```

Let's take a look at this step by step:

1. First, we subset the data: We want to keep `id, school, sch_friend*`. For the latter, we use the function `starts_with` (from the `tidyselect` package). The latter

allows us to select all variables that start with the word "`sch_friend`", which means that `sch_friend11, sch_friend12, ...` will be selected.

```
dat |>
  select(id, school, starts_with("sch_friend"))
```

```
# A tibble: 2,164 x 78
        id school sch_friend11 sch_friend12 sch_friend13 sch_friend14
     <dbl>  <int>        <int>        <int>        <int>        <int>
 1 1110001    111           NA           NA           NA           NA
 2 1110002    111          424          423          426          289
 3 1110007    111          629          505           NA           NA
 4 1110013    111          232          569           NA           NA
 5 1110014    111          582          134           41          592
 6 1110015    111           26          488           81          138
 7 1110020    111          528           NA          492          395
 8 1110022    111           NA           NA           NA           NA
 9 1110025    111          135          185          553           84
10 1110027    111          346          168          559            5
# i 2,154 more rows
# i 72 more variables: sch_friend15 <int>, sch_friend16 <int>,
#   sch_friend17 <int>, sch_friend18 <int>, sch_friend19 <int>,
#   sch_friend110 <int>, sch_friend111 <int>, sch_friend112 <int>,
#   sch_friend113 <int>, sch_friend114 <int>, sch_friend115 <int>,
#   sch_friend116 <int>, sch_friend117 <int>, sch_friend118 <int>,
#   sch_friend119 <int>, sch_friend21 <int>, sch_friend22 <int>, ...
```

2. Then, we reshape it to *long* format: By transposing all the `sch_friend*` to long format. We do this using the function `gather` (from the `tidyr` package); an alternative to the `reshape` function, which I find easier to use. Let's see how it works:

```
dat |>
  select(id, school, starts_with("sch_friend")) |>
  gather(key = "varname", value = "content", -id, -school)
```

```
# A tibble: 164,464 x 4
        id school varname      content
     <dbl>  <int> <chr>          <int>
```

```
 1 1110001    111 sch_friend11      NA
 2 1110002    111 sch_friend11     424
 3 1110007    111 sch_friend11     629
 4 1110013    111 sch_friend11     232
 5 1110014    111 sch_friend11     582
 6 1110015    111 sch_friend11      26
 7 1110020    111 sch_friend11     528
 8 1110022    111 sch_friend11      NA
 9 1110025    111 sch_friend11     135
10 1110027    111 sch_friend11     346
# i 164,454 more rows
```

In this case, the `key` parameter sets the name of the variable that will contain the name of the variable that was reshaped, while `value` is the name of the variable that will hold the content of the data (that's why I named those like that). The `-id, -school` bit tells the function to "drop" those variables before reshaping. In other words, "reshape everything but `id` and `school`."

Also, notice that we passed from 2164 rows to 19 (nominations) * 2164 (subjects) * 4 (waves) = 164464 rows, as expected.

3. As the nomination data can be empty for some cells, we need to take care of those cases, the `NA`s, so we filter the data:

```
dat |>
  select(id, school, starts_with("sch_friend")) |>
  gather(key = "varname", value = "content", -id, -school) |>
  filter(!is.na(content))
```

```
# A tibble: 39,561 x 4
        id school varname      content
     <dbl>  <int> <chr>          <int>
 1 1110002    111 sch_friend11     424
 2 1110007    111 sch_friend11     629
 3 1110013    111 sch_friend11     232
 4 1110014    111 sch_friend11     582
 5 1110015    111 sch_friend11      26
 6 1110020    111 sch_friend11     528
```

```
 7 1110025    111 sch_friend11     135
 8 1110027    111 sch_friend11     346
 9 1110029    111 sch_friend11     369
10 1110030    111 sch_friend11     462
# i 39,551 more rows
```

4. And finally, we create three new variables from this dataset: `friendid,`, `year`, and `nom_num` (nomination number). All using regular expressions:

```
dat |>
  select(id, school, starts_with("sch_friend")) |>
  gather(key = "varname", value = "content", -id, -school) |>
  filter(!is.na(content)) |>
  mutate(
    friendid = school*10000 + content,
    year     = as.integer(str_extract(varname, "(?<=[a-z])[0-9]")),
    nnom     = as.integer(str_extract(varname, "(?<=[a-z][0-9])[0-9]+"))
    )
```

```
# A tibble: 39,561 x 7
        id school varname      content friendid  year  nnom
     <dbl>  <int> <chr>          <int>    <dbl> <int> <int>
 1 1110002    111 sch_friend11     424  1110424     1     1
 2 1110007    111 sch_friend11     629  1110629     1     1
 3 1110013    111 sch_friend11     232  1110232     1     1
 4 1110014    111 sch_friend11     582  1110582     1     1
 5 1110015    111 sch_friend11      26  1110026     1     1
 6 1110020    111 sch_friend11     528  1110528     1     1
 7 1110025    111 sch_friend11     135  1110135     1     1
 8 1110027    111 sch_friend11     346  1110346     1     1
 9 1110029    111 sch_friend11     369  1110369     1     1
10 1110030    111 sch_friend11     462  1110462     1     1
# i 39,551 more rows
```

The regular expression `(?<=[a-z])` matches a string preceded by any letter from *a* to *z*. In contrast, the expression `[0-9]` matches a single number. Hence, from the string `"sch_friend12"`, the regular expression will only match the `1`, as it is the only number followed by a letter. The expression `(?<=[a-z][0-9])` matches a string

preceded by a lowercase letter and a one-digit number. Finally, the expression `[0-9]+` matches a string of numbers–so it could be more than one. Hence, from the string `"sch_friend12"`, we will get 2:

```
str_extract("sch_friend12", "(?<=[a-z])[0-9]")
```

```
[1] "1"
```

```
str_extract("sch_friend12", "(?<=[a-z][0-9])[0-9]+")
```

```
[1] "2"
```

And finally, the `as.integer` function coerces the returning value from the `str_extract` function from `character` to `integer`. Now that we have this edgelist, we can create an igraph object

4.2.2 igraph network

For coercing the edgelist into an igraph object, we will be using the `graph_from_data_frame` function in igraph (Csárdi et al. 2024). This function receives the following arguments: a data frame where the two first columns are "source" (ego) and "target" (alter), an indicator of whether the network is directed or not, and an optional data frame with vertices, in which's first column should contain the vertex ids.

Using the optional `vertices` argument is a good practice: It tells the function what `ids` should expect. Using the original dataset, we will create a data frame with name vertices:

```
vertex_attrs <- dat |>
  select(id, school, hispanic, female1, starts_with("eversmk"))
```

Now, let's now use the function `graph_from_data_frame` to create an `igraph` object:

```
library(igraph)

ig_year1 <- net |>
  filter(year == "1") |>
  select(id, friendid, nnom) |>
```

```
  graph_from_data_frame(
    vertices = vertex_attrs
  )
```

```
Error in graph_from_data_frame(select(filter(net, year == "1"), id, friendid, : Some ver
```

Ups! It seems that individuals are nominating other students not included in the survey. How to solve that? Well, it all depends on what you need to do! In this case, we will go for the *quietly-remove-em'-and-don't-tell* strategy:

```
library(igraph)

ig_year1 <- net |>
  filter(year == "1") |>

  # Extra line, all nominations must be in ego too.
  filter(friendid %in% id) |>

  select(id, friendid, nnom) |>
  graph_from_data_frame(
    vertices = vertex_attrs
    )

ig_year1
```

```
IGRAPH 2020242 DN-- 2164 9514 --
+ attr: name (v/c), school (v/n), hispanic (v/n), female1 (v/n),
| eversmk1 (v/n), eversmk2 (v/n), eversmk3 (v/n), eversmk4 (v/n), nnom
| (e/n)
+ edges from 2020242 (vertex names):
 [1] 1110007->1110629 1110013->1110232 1110014->1110582 1110015->1110026
 [5] 1110025->1110135 1110027->1110346 1110029->1110369 1110035->1110034
 [9] 1110040->1110390 1110041->1110557 1110044->1110027 1110046->1110030
[13] 1110050->1110086 1110057->1110263 1110069->1110544 1110071->1110167
[17] 1110072->1110289 1110073->1110014 1110075->1110352 1110084->1110305
```

```
[21] 1110086->1110206 1110093->1110040 1110094->1110483 1110095->1110043
+ ... omitted several edges
```

So we have our network with 2164 nodes and 9514 edges. The following steps: get some descriptive stats and visualize our network.

4.3 Network descriptive stats

While we could do all networks at once, in this part, we will focus on computing some network statistics for one of the schools only. We start by school 111. The first question that you should be asking yourself now is, "how can I get that information from the igraph object?." Vertex and edges attributes can be accessed via the `V` and `E` functions, respectively; moreover, we can list what vertex/edge attributes are available:

```
vertex_attr_names(ig_year1)
```

```
[1] "name"     "school"   "hispanic" "female1"  "eversmk1" "eversmk2" "eversmk3"
[8] "eversmk4"
```

```
edge_attr_names(ig_year1)
```

```
[1] "nnom"
```

Just like we would do with data frames, accessing vertex attributes is done via the dollar sign operator `$`. Together with the `V` function; for example, accessing the first ten elements of the variable `hispanic` can be done as follows:

```
V(ig_year1)$hispanic[1:10]
```

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

Now that you know how to access vertex attributes, we can get the network corresponding to school 111 by identifying which vertices are part of it and pass that information to the `induced_subgraph` function:

```r
# Which ids are from school 111?
school111ids <- which(V(ig_year1)$school == 111)

# Creating a subgraph
ig_year1_111 <- induced_subgraph(
  graph = ig_year1,
  vids  = school111ids
)
```

The `which` function in R returns a vector of indices indicating which elements pass the test, returning true and false, otherwise. In our case, it will result in a vector of indices of the vertices which have the attribute `school` equal to 111. With the subgraph, we can compute different centrality measures[3] for each vertex and store them in the igraph object itself:

```r
# Computing centrality measures for each vertex
V(ig_year1_111)$indegree   <- degree(ig_year1_111, mode = "in")
V(ig_year1_111)$outdegree  <- degree(ig_year1_111, mode = "out")
V(ig_year1_111)$closeness  <- closeness(ig_year1_111, mode = "total")
V(ig_year1_111)$betweeness <- betweenness(ig_year1_111, normalized = TRUE)
```

From here, we can *go back* to our old habits and get the set of vertex attributes as a data frame so we can compute some summary statistics on the centrality measurements that we just got

```r
# Extracting each vertex features as a data.frame
stats <- as_data_frame(ig_year1_111, what = "vertices")

# Computing quantiles for each variable
stats_degree <- with(stats, {
 cbind(
   indegree   = quantile(indegree, c(.025, .5, .975), na.rm = TRUE),
   outdegree  = quantile(outdegree, c(.025, .5, .975), na.rm = TRUE),
   closeness  = quantile(closeness, c(.025, .5, .975), na.rm = TRUE),
   betweeness = quantile(betweeness, c(.025, .5, .975), na.rm = TRUE)
```

[3]For more information about the different centrality measurements, please take a look at the "Centrality" article on Wikipedia.

```
 )
})

stats_degree
```

```
      indegree outdegree    closeness  betweeness
2.5%         0         0 0.0005915148 0.000000000
50%          4         4 0.0007487833 0.001879006
97.5%       16        16 0.0008838413 0.016591048
```

The `with` function is somewhat similar to what `dplyr` allows us to do when we want to work with the dataset but without mentioning its name everytime that we ask for a variable. Without using the `with` function, the previous could have been done as follows:

```
stats_degree <-
 cbind(
   indegree   = quantile(stats$indegree, c(.025, .5, .975), na.rm = TRUE),
   outdegree  = quantile(stats$outdegree, c(.025, .5, .975), na.rm = TRUE),
   closeness  = quantile(stats$closeness, c(.025, .5, .975), na.rm = TRUE),
   betweeness = quantile(stats$betweeness, c(.025, .5, .975), na.rm = TRUE)
 )
```

Next, we will compute some statistics at the graph level:

```
cbind(
  size    = vcount(ig_year1_111),
  nedges  = ecount(ig_year1_111),
  density = edge_density(ig_year1_111),
  recip   = reciprocity(ig_year1_111),
  centr   = centr_betw(ig_year1_111)$centralization,
  pathLen = mean_distance(ig_year1_111)
  )
```

```
     size nedges     density     recip      centr pathLen
[1,]  533   2638 0.009303277 0.3731513 0.02179154 4.23678
```

Triadic census

```
triadic <- triad_census(ig_year1_111)
triadic
```

```
 [1] 24059676   724389   290849     3619     3383     4401     3219     2997
 [9]      407       33      836      235      163      137      277       85
```

To get a nicer view of this, we can use a table that I retrieved from `?triad_census`. Moreover, we can normalize the `triadic` object by its sum instead of looking at raw counts. That way, we get proportions instead[4]

```
knitr::kable(cbind(
  Pcent = triadic/sum(triadic)*100,
  read.csv("triadic_census.csv")
  ), digits = 2)
```

Pcent	code	description
95.88	003	A,B,C, the empty graph.
2.89	012	A->B, C, the graph with a single directed edge.
1.16	102	A<->B, C, the graph with a mutual connection between two vertices.
0.01	021D	A<-B->C, the out-star.
0.01	021U	A->B<-C, the in-star.
0.02	021C	A->B->C, directed line.
0.01	111D	A<->B<-C.
0.01	111U	A<->B->C.
0.00	030T	A->B<-C, A->C.
0.00	030C	A<-B<-C, A->C.
0.00	201	A<->B<->C.
0.00	120D	A<-B->C, A<->C.
0.00	120U	A->B<-C, A<->C.
0.00	120C	A->B->C, A<->C.
0.00	210	A->B<->C, A<->C.

[4] During our workshop, Prof. De la Haye suggested using $\binom{n}{3}$ as a normalizing constant. It turns out that `sum(triadic) = choose(n, 3)`! So either approach is correct.

Pcent	code	description
0.00	300	A<->B<->C, A<->C, the complete graph.

4.4 Plotting the network in igraph

4.4.1 Single plot

Let's take a look at how does our network looks like when we use the default parameters in the plot method of the igraph object:

```
plot(ig_year1)
```

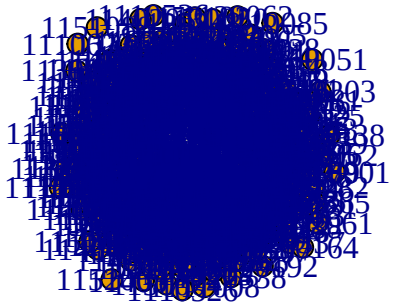


Figure 4.1: A not very nice network plot. This is what we get with the default parameters in igraph.

Not very nice, right? A couple of things with this plot:

1. We are looking at all schools simultaneously, which does not make sense. So, instead of plotting `ig_year1`, we will focus on `ig_year1_111`.

2. All the vertices have the same size and are overlapping. Instead of using the default size, we will size the vertices by indegree using the `degree` function and passing the vector of degrees to `vertex.size`.[5]

3. Given the number of vertices in these networks, the labels are not useful here. So we will remove them by setting `vertex.label = NA`. Moreover, we will reduce the size of the arrows' tip by setting `edge.arrow.size = 0.25`.

4. And finally, we will set the color of each vertex to be a function of whether the individual is Hispanic or not. For this last bit we need to go a bit more of programming:

```
col_hispanic <- V(ig_year1_111)$hispanic + 1
col_hispanic <- coalesce(col_hispanic, 3)
col_hispanic <- c("steelblue", "tomato", "white")[col_hispanic]
```

Line by line, we did the following:

1. The first line added one to all no `NA` values so that the 0s (non-Hispanic) turned to 1s and the 1s (Hispanic) turned to 2s.

2. The second line replaced all `NA`s with the number three so that our vector `col_hispanic` now ranges from one to three with no `NA`s in it.

3. In the last line, we created a vector of colors. Essentially, what we are doing here is telling R to create a vector of length `length(col_hispanic)` by selecting elements by index from the vector `c("steelblue", "tomato", "white")`. This way, if, for example, the first element of the vector `col_hispanic` was a 3, our new vector of colors would have a `"white"` in it.

To make sure we know we are right, let's print the first 10 elements of our new vector of colors together with the original `hispanic` column:

```
cbind(
  original = V(ig_year1_111)$hispanic[1:10],
  colors   = col_hispanic[1:10]
  )
```

[5] Figuring out what is the optimal vertex size is a bit tricky. Without getting too technical, there's no other way of getting *nice* vertex size other than just playing with different values of it. A nice solution to this is using `netdiffuseR::igraph_vertex_rescale` which rescales the vertices so that these keep their aspect ratio to a predefined proportion of the screen.

```
      original colors
 [1,] "1"      "tomato"
 [2,] "1"      "tomato"
 [3,] "0"      "steelblue"
 [4,] "1"      "tomato"
 [5,] "1"      "tomato"
 [6,] "1"      "tomato"
 [7,] "1"      "tomato"
 [8,] "1"      "tomato"
 [9,] "0"      "steelblue"
[10,] "1"      "tomato"
```

With our nice vector of colors, now we can pass it to `plot.igraph` (which we call implicitly by just calling `plot`), via the `vertex.color` argument:

```
# Fancy graph
set.seed(1)
plot(
  ig_year1_111,
  vertex.size     = degree(ig_year1_111)/10 +1,
  vertex.label    = NA,
  edge.arrow.size = .25,
  vertex.color    = col_hispanic
  )
```

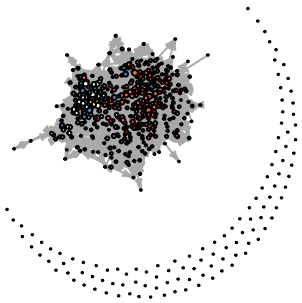


Figure 4.2: Friends network in time 1 for school 111.

Nice! So it does look better. The only problem is that we have a lot of isolates. Let's try again by drawing the same plot without isolates. To do so, we need to filter the graph, for which we will use the function `induced_subgraph`

```
# Which vertices are not isolates?
which_ids <- which(degree(ig_year1_111, mode = "total") > 0)

# Getting the subgraph
ig_year1_111_sub <- induced_subgraph(ig_year1_111, which_ids)

# We need to get the same subset in col_hispanic
col_hispanic <- col_hispanic[which_ids]
```

```
# Fancy graph
set.seed(1)
plot(
  ig_year1_111_sub,
  vertex.size     = degree(ig_year1_111_sub)/5 +1,
  vertex.label    = NA,
```

```
  edge.arrow.size = .25,
  vertex.color    = col_hispanic
  )
```

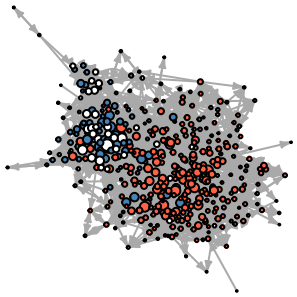


Figure 4.3: Friends network in time 1 for school 111. The graph excludes isolates.

Now that's better! An interesting pattern that shows up is that individuals seem to cluster by whether they are Hispanic or not.

We can write this as a function to avoid copying and pasting the code n times (supposing that we want to create a plot similar to this n times). We do the latter in the following subsection.

4.4.2 Multiple plots

When you are repeating yourself repeatedly, it is a good idea to write down a sequence of commands as a function. In this case, since we will be running the same type of plot for all schools/waves, we write a function in which the only things that change are: (a) the school id, and (b) the color of the nodes.

```r
myplot <- function(
  net,
  schoolid,
  mindgr = 1,
  vcol   = "tomato",
  ...) {

  # Creating a subgraph
  subnet <- induced_subgraph(
    net,
    which(degree(net, mode = "all") >= mindgr & V(net)$school == schoolid)
  )

  # Fancy graph
  set.seed(1)
  plot(
    subnet,
    vertex.size     = degree(subnet)/5,
    vertex.label    = NA,
    edge.arrow.size = .25,
    vertex.color    = vcol,
    ...
    )
}
```

The function definition:

1. The `myplot <- function([arguments]) {[body of the function]}` tells R that we are going to create a function called `myplot`.

2. We declare four specific arguments: `net`, `schoolid`, `mindgr`, and `vcol`. These are an igraph object, the school id, the minimum degree that vertices must have to be included in the figure, and the color of the vertices. Observe that, compared to other programming languages, R does not require declaring the data types.

3. The ellipsis object, `...`, is an especial object in R that allows us to pass other arguments without specifying which. If you take a look at the `plot` bit in the function body, you

will see that we also added `...`. We use the ellipsis to pass extra arguments (different from the ones that we explicitly defined) directly to `plot`. In practice, this implies that we can, for example, set the argument `edge.arrow.size` when calling `myplot`, even though we did not include it in the function definition! (See `?dotsMethods` in R for more details).

In the following lines of code, using our new function, we will plot each schools' network in the same plotting device (window) with the help of the `par` function, and add legend with the `legend`:

```
# Plotting all together
oldpar <- par(no.readonly = TRUE)
par(mfrow = c(2, 3), mai = rep(0, 4), oma= c(1, 0, 0, 0))
myplot(ig_year1, 111, vcol = "tomato")
myplot(ig_year1, 112, vcol = "steelblue")
myplot(ig_year1, 113, vcol = "black")
myplot(ig_year1, 114, vcol = "gold")
myplot(ig_year1, 115, vcol = "white")
par(oldpar)

# A fancy legend
legend(
  "bottomright",
  legend = c(111, 112, 113, 114, 115),
  pt.bg  = c("tomato", "steelblue", "black", "gold", "white"),
  pch    = 21,
  cex    = 1,
  bty    = "n",
  title  = "School"
  )
```

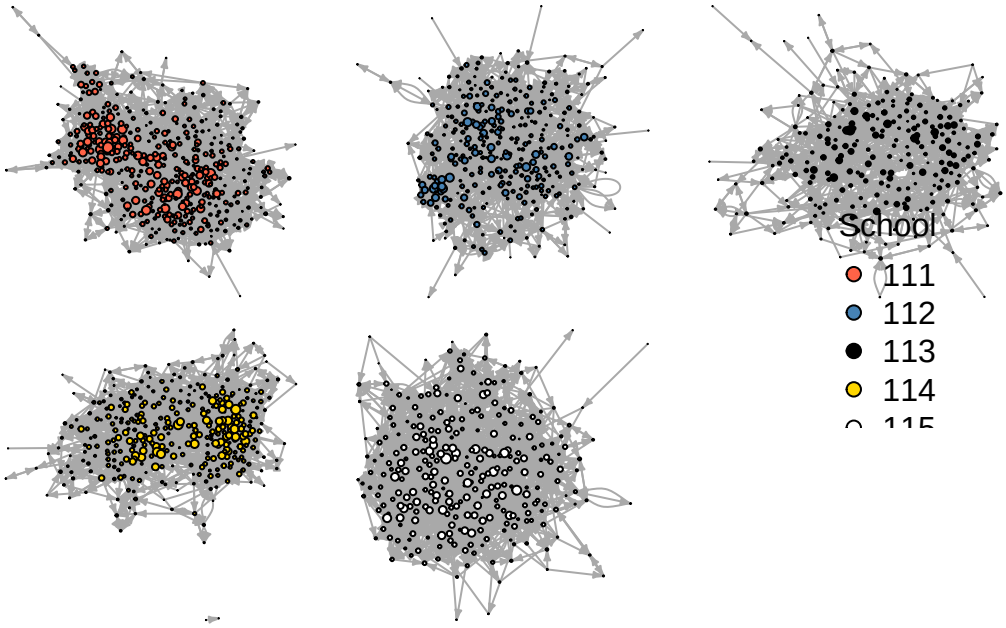


Figure 4.4: All 5 schools in time 1. Again, the graphs exclude isolates.

So what happened here?

- `oldpar <- par(no.readonly = TRUE)` This line stores the current parameters for plotting. Since we are going to be changing them, we better make sure we are able to go back!.

- `par(mfrow = c(2, 3), mai = rep(0, 4), oma=rep(0, 4))` Here we are setting various things at the same time. `mfrow` specifies how many *figures* will be drawn, and in what order. In particular, we are asking the plotting device to make room for 2*3 = 6 figures organized in two rows and three columns drawn by row.

  `mai` specifies the size of the margins in inches, setting all margins equal to zero (which is what we are doing now) gives more space to the graph. The same is true for `oma`. See `?par` for more info.

- `myplot(ig_year1, ...)` This is simply calling our plotting function. The neat part of this is that, since we set `mfrow = c(2, 3)`, R takes care of *distributing* the plots in the device.

- `par(oldpar)` This line allows us to restore the plotting parameters.

4.5 Statistical tests

4.5.1 Is nomination number correlated with indegree?

Hypothesis: Individuals that, on average, are among the first nominations of their peers are more popular

```
# Getting all the data in long format
edgelist <- as_long_data_frame(ig_year1) |>
  as_tibble()

# Computing indegree (again) and average nomination number
# Include "On a scale from one to five how close do you feel"
# Also for egocentric friends (A. Friends)
indeg_nom_cor <- group_by(edgelist, to, to_name, to_school) |>
  summarise(
    indeg   = length(nnom),
    nom_avg = 1/mean(nnom)
  ) |>
  rename(
    school = to_school
  )
```

```
`summarise()` has grouped output by 'to', 'to_name'. You can override using the
`.groups` argument.
```

```
indeg_nom_cor
```

```
# A tibble: 1,561 x 5
# Groups:   to, to_name [1,561]
      to to_name school indeg nom_avg
   <dbl> <chr>    <int> <int>   <dbl>
 1     2 1110002    111    22   0.222
 2     3 1110007    111     7   0.175
 3     4 1110013    111     6   0.171
```

```
 4     5 1110014    111    19   0.134
 5     6 1110015    111     3   0.15
 6     7 1110020    111     6   0.154
 7     9 1110025    111     6   0.214
 8    10 1110027    111    13   0.220
 9    11 1110029    111    14   0.131
10    12 1110030    111     6   0.222
# i 1,551 more rows
```

```
# Using pearson's correlation
with(indeg_nom_cor, cor.test(indeg, nom_avg))
```

```

    Pearson's product-moment correlation

data:  indeg and nom_avg
t = -12.254, df = 1559, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.3409964 -0.2504653
sample estimates:
       cor
-0.2963965
```

```
save.image("03.rda")
```

5 Simulation and vizualization

In this chapter, we will build and visualize artificial networks using Exponential Random Graph Models [ERGMs.] Together with chapter 3, this will be an extended example of how to read network data and visualize it using some of the available R packages out there.

For this chapter, we will be using the following R packages:

- `ergm`: To simulate and estimate ERGMs.
- `sna`: To visualize networks.
- `igraph`: Also to visualize networks.
- `intergraph`: To convert between `igraph` and `network` objects.
- `netplot`: Again, for visualization.
- `netdiffuseR`: For a single function we use for adjusting vertex size in igraph.
- `rgexf`: For building interactive (html) figures.

You can use the following codeblock to install any missing package:

```
# Creating the list to install
pkgs <- c(
  "ergm", "sna", "igraph", "intergraph", "netplot", "netdiffuseR", "rgexf"
  )

# Checking if we can load them and install them if not available
for (pkg in pkgs) {
  if (!require(pkg, character.only = TRUE)) {

    # If not present, will install it
    install.packages(pkg, character.only = TRUE)

    # And load it!
```

```
    library(pkg, character.only = TRUE)

  }
}
```

A recorded version is available here.

5.1 Random Graph Models

While there are tons of social network data, we will use an artificial one for this chapter. We do this as it is always helpful to have more examples simulating Random networks. For this chapter, we will classify random graph models for sampling and generating networks into three categories:

1. **Exogenous**: Graphs where the structure is determined by a macro rule, e.g., expected density, degree distribution, or degree-sequence. In these cases, ties are assigned to comply with a macro-property.

2. **Endogenous**: This category includes all Random Graphs generated based on endogenous information, e.g., small-world, scale-free, etc. Here, a tie creation rule gives origin to a macro property, for example, preferential attachment in scale-free networks.

3. **Exponential Random Graph Models**: Overall, since ERGMs compose a family of statistical models, we can always (or almost always) find a model specification that matches the previous categories. Whereas we are thinking about degree sequence, preferential attachment, or a mix of both, ERGMs can be the baseline for any of those models.

The latter, ERGMs, are a generalization that covers all classes. Because of that, we will use ERGMs to generate our artificial network.

5.2 Social Networks in Schools

A common type of network we analyze is friendship networks. In this case, we will use ERGMs to simulate friendship networks within a school. In our simulated world, these networks will be dominated by the following phenomena

- Low density,
- Race homophily,
- Structural balance,
- And age homophily.

If you have been paying attention to the previous chapters, you will notice that, out of these five properties, only one constitutes Markov graphs. Within a tie, homophily and density only depend on ego and alter. In race homophily, only ego and alter's race matter for the tie formation, but, in the case of Structural balance, ego is more likely to befriend alter if a fried of ego is friends with alter, i.e., "the friend of my friend is my friend."

The simulation steps are as follows:

1. Draw a population of n students and randomly distribute race and age across them.
2. Create a `network` object.
3. Simulate the ties in the empty network.

Here is the code

```
set.seed(712)
n <- 200

# Step 1: Students
race   <- sample(c("white", "non-white"), n, replace = TRUE)
age    <- sample(c(10, 14, 17), n, replace = TRUE)

# Step 2: Create an empty network
library(ergm)
library(network)
net <- network.initialize(n)
```

```
net %v% "race"   <- race
net %v% "age"    <- age

# Step 3: Simulate a graph
net_sim <- simulate(
    net ~ edges +
    nodematch("race") +
    ttriad +
    absdiff("age"),
    coef = c(-4, .5, .25, -.5)
    )
```

What just happened? Here is a line-by-line breakout:

1. `set.seed(712)` Since this is a random simulation, we need to fix a seed so it is reproducible. Otherwise, results would change with every iteration.
2. `n <- 200` We are assigning the value 200 to the object `n`. This will make things easier as, if needed, changing the size of the networks can be done at the top of the code.
3. `race <- sample(c("white", "non-white"), n, replace = TRUE)` We are sampling 200, or actually, `n` values from the vector `c("white", "non-white")` with replacement.
4. `age <- sample(c(10, 14, 17), n, replace = TRUE)` Same as before, but with ages!
5. `library(ergm)` Loading the ergm R package, which we need to simulate the networks!
6. `library(network)` Loading the `network` R package, which we need to create the empty graph.
7. `net <- network.initialize(n)` Creating an empty graph of size `n`.
8. `net %v% "race" <- race` Using the `%v%` operator, we can access vertices features in the network object. Since race does not exist in the network yet, the operator just creates it. Notice that the number of vertices matches the length of the race vector.
9. `net %v% "age" <- age` Same as with race!

10. `net_sim <- simulate(` Simulating an ERGM! A couple of observations here:

    a. The LHS (left-hand-side) of the equation has the network, `net`

    b. The RHS (you guessed it) has the terms that govern the process.

    c. For low density, we used the `edges` term with a corresponding -4.0 for the parameter.

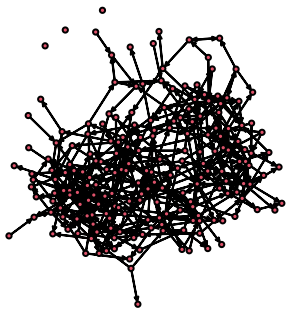
    d. For race homophily, we used the `nodematch("race")` with a corresponding 0.5 parameter value.

    e. For structural balance, we use the `ttriad` term with parameter 0.25.

    f. For age homophily, we use the `absdiff("age")` term with parameter -0.5. This is, in rigor, a term capturing heterophily. Nonetheless, heterophily is the opposite of homophily.

Let's take a quick look at the resulting graph

```
library(sna)
gplot(net_sim)
```



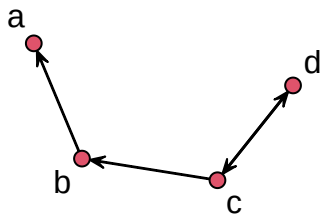
We can now start to see whether we got what we wanted! Before that, let's save the network as a plain-text file so we can practice reading networks back in R!

```
write.csv(
  x         = as.edgelist(net_sim),
  file      = "06-edgelist.csv",
  row.names = FALSE
  )

write.csv(
  x         = as.data.frame(net_sim, unit = "vertices"),
  file      = "06-nodes.csv",
  row.names = FALSE
  )
```

5.3 Reading a network

The first step to analyzing network data is to read it in. Many times you'll find data in the form of an adjacency matrix. Other times, data will come in the form of an edgelist. Another common format is the adjacency list, which is a compressed version of an edgelist. Let's see how the formats look like for the following network:

```
example_graph <- matrix(0L, 4, 4, dimnames = list(letters[1:4], letters[1:4]))
example_graph[c(2, 7)] <- 1L
example_graph["c", "d"] <- 1L
example_graph["d", "c"] <- 1L
example_graph <- as.network(example_graph)
set.seed(1231)
gplot(example_graph, label = letters[1:4])
```



- **Adjacency matrix** a matrix of size n by n where the ij-th entry represents the tie between i and j. In a directed network, we say i connects to j, so the i-th row shows the ties i sends to the rest of the network. Likewise, in a directed graph, the j-th column shows the ties sent to j. For undirected graphs, the adjacency matrix is usually upper or lower diagonal. The adjacency matrix of an undirected graph is symmetric, so we don't need to report the same information twice. For example:

```
as.matrix(example_graph)
```

```
  a b c d
a 0 0 0 0
b 1 0 0 0
c 0 1 0 1
d 0 0 1 0
```

- **Edge list** a matrix of size $|E|$ by 2, where $|E|$ is the number of edges. Each entry represents a tie in the graph.

```
as.edgelist(example_graph)
```

```
     [,1] [,2]
[1,]    2    1
[2,]    3    2
[3,]    3    4
[4,]    4    3
attr(,"n")
[1] 4
attr(,"vnames")
[1] "a" "b" "c" "d"
attr(,"directed")
[1] TRUE
attr(,"bipartite")
[1] FALSE
attr(,"loops")
[1] FALSE
attr(,"class")
[1] "matrix_edgelist" "edgelist"        "matrix"          "array"
```

The command turns the `network` object into a matrix with a set of attributes (which are also printed.)

- **Adjacency list** This data format uses less space than edgelists as ties are grouped by ego (source.)

```
igraph::as_adj_list(intergraph::asIgraph(example_graph))
```

```
[[1]]
+ 1/4 vertex, from c7f2a77:
[1] 2

[[2]]
+ 2/4 vertices, from c7f2a77:
[1] 1 3

[[3]]
+ 3/4 vertices, from c7f2a77:
```

```
[1] 2 4 4

[[4]]
+ 2/4 vertices, from c7f2a77:
[1] 3 3
```

The function `igraph::as_adj_list` turns the igraph object into a list of type adjacency list. In plain text it would look something like this:

```
2
1 3
2 4 4
3 3
```

Here we will deal with an edgelist that includes node information. In my opinion, this is one of the best ways to share network data. Let's read the data into R using the function `read.csv`:

```
edges <- read.csv("06-edgelist.csv")
nodes <- read.csv("06-nodes.csv")
```

We now have two objects of class `data.frame`, edges and nodes. Let's inspect them using the `head` function:

```
head(edges)
```

```
  V1  V2
1  2   7
2  2  41
3  3   5
4  3  16
5  4 138
6  5   9
```

```
head(nodes)
```

```
  vertex.names      race age
1            1 non-white  10
2            2     white  10
3            3     white  17
4            4 non-white  14
5            5 non-white  17
6            6 non-white  14
```

It is always important to look at the data before creating the network. Most common errors happen before reading the data in and could go undetected in many cases. A few examples:

- Headers in the file could be treated as data, or the files may not have headers.
- Ego/alter columns may show in the wrong order. Both the `igraph` and `network` packages take the first and second columns of edgelists as ego and alter.
- Isolates, which wouldn't show in the edgelist, may be missing from the node information set. This is one of the most common errors.
- Nodes showing in the edgelist may be missing from the nodelist.

Both `igraph` and network have functions to read edgelist with a corresponding nodelist; the functions `graph_from_data_frame` and `as.nework`, respectively. Although , for both cases, you can avoid using a nodelist, it is highly recommended as then you will (a) make sure that isolates are included and (b) become aware of possible problems in the data. A frequent error in `graph_from_data_frame` is nodes present in the edgelist but not in the set of nodes.

```
net_ig <- igraph::graph_from_data_frame(
  d        = edges,
  directed = TRUE,
  vertices = nodes
)
```

Using `as.network` from the `network` package:

```
net_net <- network::as.network(
  x        = edges,
  directed = TRUE,
  vertices = nodes
)
```

As you can see, both syntaxes are very similar. The main point here is that the more explicit we are, the better. Nevertheless, R can be brilliant; being *shy*, i.e., not throwing warnings or errors, is not uncommon. In the next section, we will finally start visualizing the data.

5.4 Visualizing the network

We will focus on three different attributes that we can use for this visualization: Node size, node shape, and node color. While there are no particular rules, some ideas you can follow are:

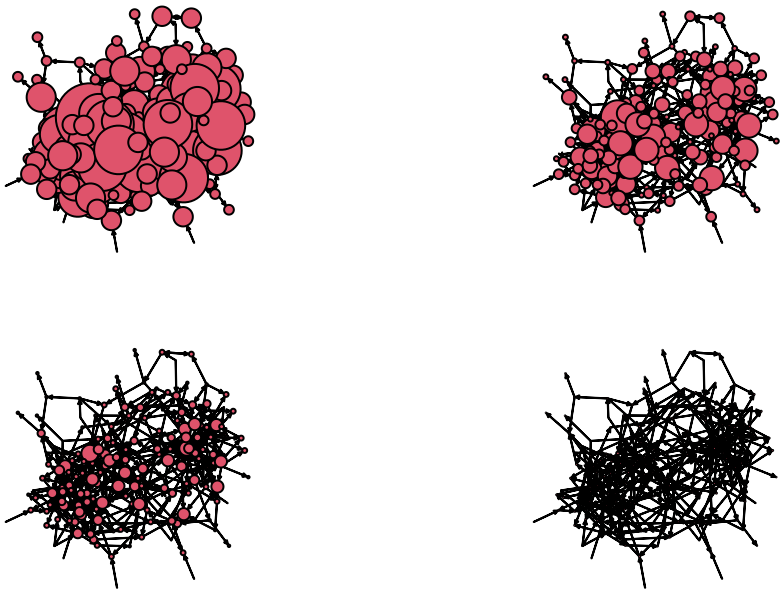
- **Node size** Use it to describe a continuous measurement. This feature is often used to highlight important nodes, e.g., using one of the many available degree measurements.
- **Node shape** Shapes can be used to represent categorical values. A good figure will not feature too many of them; less than four would make sense.
- **Node color** Like shapes, colors can be used to represent categorical values, so the same idea applies. Furthermore, it is not crazy to use both shape and color to represent the same feature.

Notice that we have not talked about layout algorithms. The R packages to build graphs usually have internal rules to decide what algorithm to use. We will discuss that later on. Let's start by size.

5.4.1 Vertex size

Finding the right scale can be somewhat difficult. We will draw the graph four times to see what size would be the best:

```r
# Sized by indegree
net_sim %v% "indeg" <- sna::degree(net_sim, cmode = "indegree")

# Changing device config
op <- par(mfrow = c(2, 2), mai = c(.1, .1, .1, .1))

# Plotting
glayout <- gplot(net_sim, vertex.cex = (net_sim %v% "indeg") * 2)
gplot(net_sim, vertex.cex = net_sim %v% "indeg", coord = glayout)
gplot(net_sim, vertex.cex = (net_sim %v% "indeg")/2, coord = glayout)
gplot(net_sim, vertex.cex = (net_sim %v% "indeg")/10, coord = glayout)
```



```r
# Restoring device config
par(op)
```

Line-by-line we did the following:

1. `net_sim %v% "indeg" <- degree(net_sim, cmode = "indegree")` Created a new vertex attribute called indegree and assigned it to the network object. The indegree is calculated using the `degree` function from the `sna` package. Since `igraph` also has

a `degree` function, we are making sure that R uses `sna`'s and not `igraph`'s. The `package::function` notation is useful for these cases.

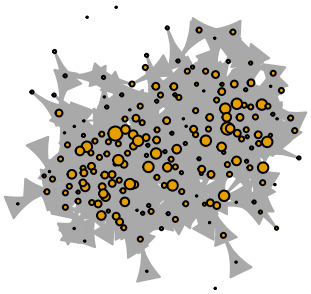
2. `op <- par(mfrow = c(2, 2), mai = c(.1, .1, .1, .1))` This changes the graphical device information to (a) `mfrow = c(2,2)` have a 2x2 grid by row, meaning that new figures will be added left to right and then top to bottom, and (b) set the margins in the figure to be 0.1 inches in all four sizes.

3. `glayout <- gplot(net_sim, vertex.cex = (net_sim %v% "indeg") * 2)` generating the plot **and** recording the layout. The `gplot` function returns a matrix of size `# vertices` by 2 with the positions of the vertices. We are also passing the `vertex.cex` argument, which we use to specify the size of each vertex. In our case, we decided to size the vertices proportional to their indegree *times two*.

4. `gplot(net_sim, vertex.cex = net_sim %v% "indeg", coord = glayout)`, again, we are drawing the graph using the coordinates of the previous draw, but now the vertices are half the size of the original figure.

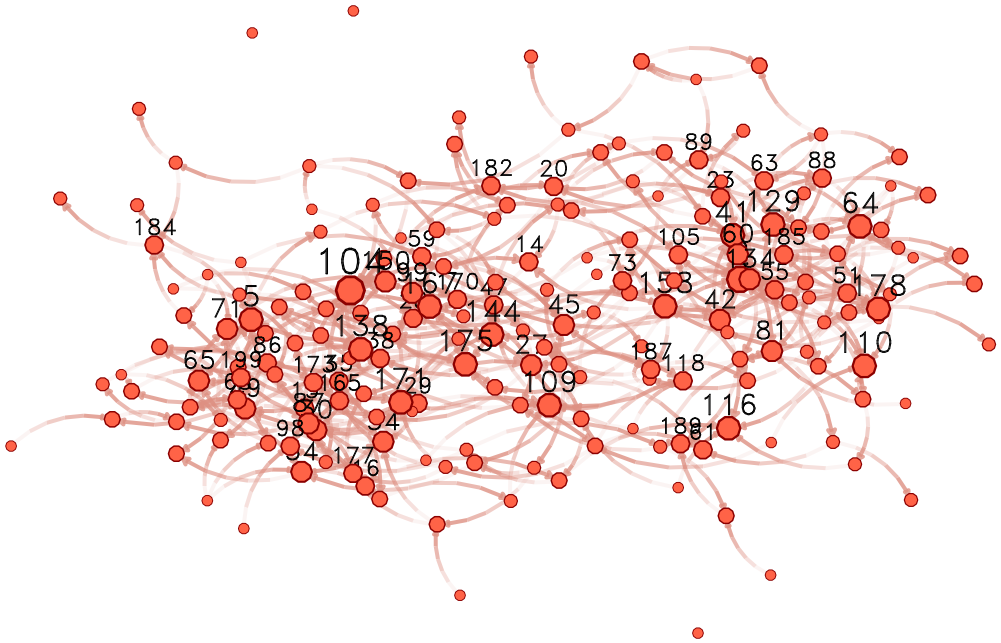
The other two calls are similar to four. If we used igraph, setting the size can be more accessible thanks to the netdiffuseR R package. Let's start by converting our network to an igraph object with the R package `intergraph`.

```
library(intergraph)
library(igraph)

# Converting the network object to an igraph object
net_sim_i <- asIgraph(net_sim)

# Plotting with igraph
plot(
  net_sim_i,
  vertex.size = netdiffuseR::rescale_vertex_igraph(
    vertex.size = V(net_sim_i)$indeg,
    minmax.relative.size = c(.01, .1)
  ),
  layout       = glayout,
  vertex.label = NA
)
```



We could also have tried netplot, which should make things easier and make a better use of the space:

```
library(netplot)
nplot(
  net_sim, layout = glayout,
  vertex.color = "tomato",
  vertex.frame.color = "darkred"
  )
```



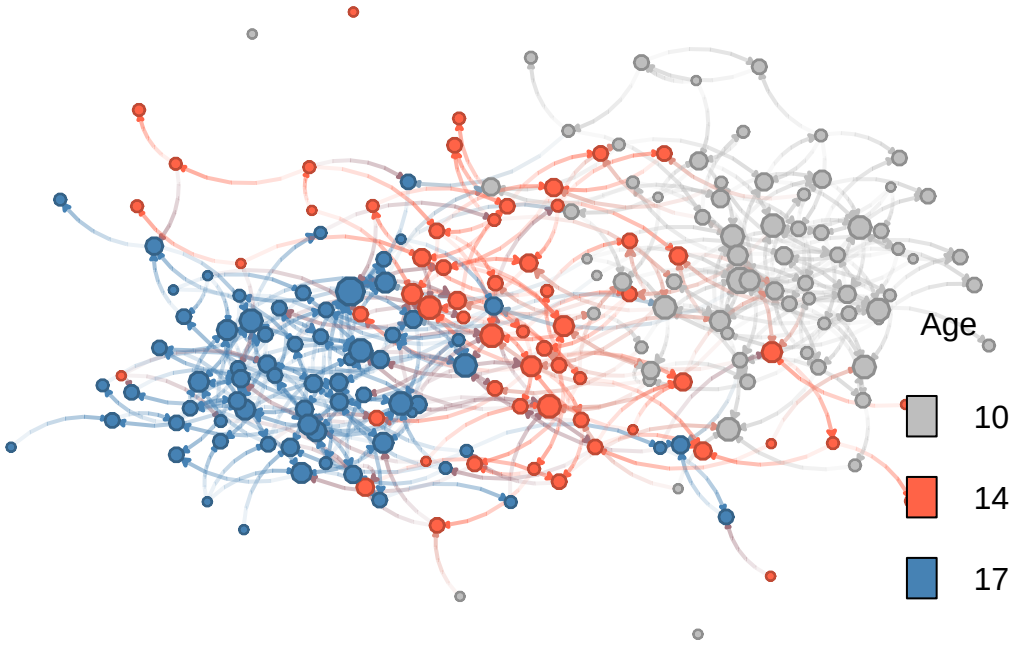
With a good idea for size, we can now start looking into vertex color.

5.4.2 Vertex color

For the color, we will use vertex age. Although age is, by definition, continuous, we only have three values for age. Because of this, we can treat age as categorical. Instead of using `nplot` we will go ahead with `nplot_base`. As of this version of the book, the `netplot` package does not have an easy way to add legends with the core function, `nplot`; therefore, we use `nplot_base` which is compatible with the R function `legend`, as we will now see:

```
# Specifying colors for each vertex
vcolors_palette <- c("10" = "gray", "14" = "tomato", "17" = "steelblue")
vcolors <- vcolors_palette[as.character(net_sim %v% "age")]
net_sim %v% "color" <- vcolors

# Plotting
nplot_base(
  net_ig,
  layout = glayout,
  vertex.color = net_sim %v% "color",
  )
```

```r
# Color legend
legend(
  "bottomright",
  legend = names(vcolors_palette),
  fill   = vcolors_palette,
  bty    = "n",
  title  = "Age"
  )
```



Line by line, this is what we just did:

1. `vcolors <- c("10" = "gray", "14" = "tomato", "17" = "steelblue")` we created a character vector with three elements, `"gray"`, `"tomato"`, and `"blue"`. Furthermore, the vector has names assigned to it, `"10"`, `"14"`, and `"17"`– the ages we have in the network–so that we can access its elements by indexing by name, e.g., if we type `vcolors["10"]` R returns the value `"gray"`.

2. `vcolors <- vcolors[as.character(net_sim %v% "age")]` there are several things going on in this line. First, we extract the attribute "age" from the network using the `%v%` operator. We then transform the resulting vector from integer type to a character type with the function `as.character`. Finally, using the resulting **character vector**

with values `"10", "14", "17", ...`, we retrieve values from `vcolors` name-indexing. The resulting vector is of length equal to the vertex count in the network.

3. `net_sim %v% "color" <- vcolors` creates a new vertex attribute, `color`. The assigned value is the result from subsetting `vcolors` by the ages of each vertex.

4. `nplot_base(...` finally draws the network. We pass the previously computed vertex coordinates and vertex colors with the new attribute `color`.

5. `legend(...)` Let's see one parameter at a time:

   a. `"bottomright"` tells the overall position of the legend

   b. `legend = names(vcolors)` passes the actual legend (text); in our case the ages of individuals.

   c. `fill = vcolors` passes the colors associated with the text.

   d. `bty = "n"` suppresses wrapping the legend within a box.

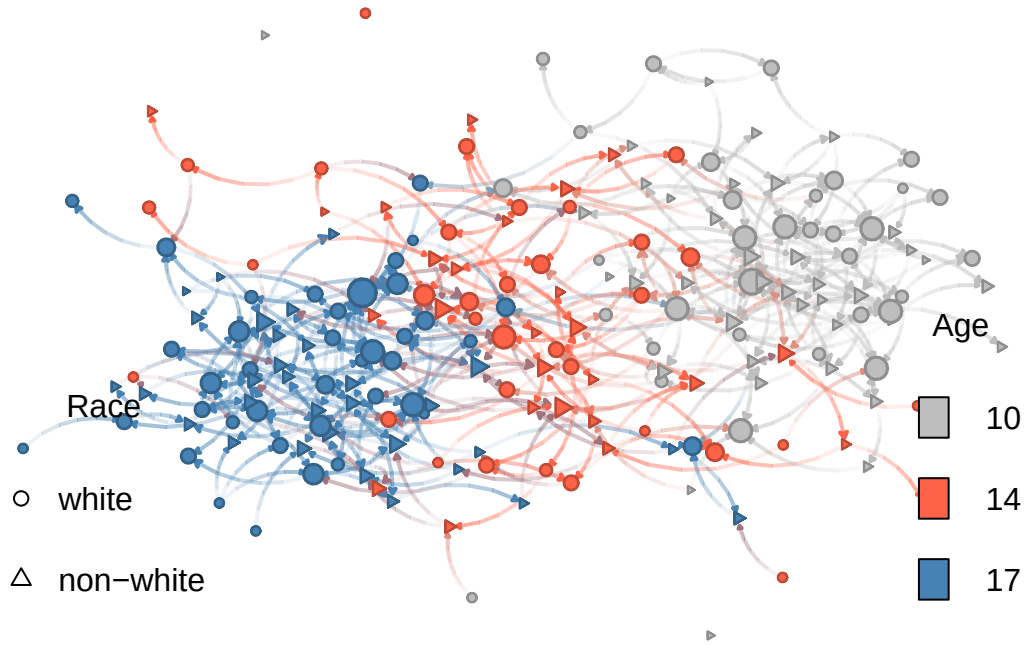
   e. `title = "Age"` sets the title to be "Age".

5.4.3 Vertex shape

For the color, we will use vertex age. Although age is, by definition, continuous, we only have three values for age. Because of this, we can treat age as categorical.

```
# Specifying the shapes for each vertex
vshape_list <- c("white" = 15, "non-white" = 3)
vshape      <- vshape_list[as.character(net_sim %v% "race")]
net_sim %v% "shape" <- vshape

# Plotting
nplot_base(
  net_ig,
  layout = glayout,
  vertex.color = net_sim %v% "color",
  vertex.nsides = net_sim %v% "shape"
  )
```

```r
# Color legend
legend(
  "bottomright",
  legend = names(vcolors_palette),
  fill   = vcolors_palette,
  bty    = "n",
  title  = "Age"
  )

# Shape legend
legend(
  "bottomleft",
  legend = names(vshape_list),
  pch    = c(1, 2),
  bty    = "n",
  title  = "Race"
  )
```



Let's now compare the figure to our original ERGM:

1. **Low density (`edges`)** Without low density, the figure would be a hairball.

2. **Race homophily (`nodematch("race")`)** Although not surprisingly evident, nodes tend to form small clusters by shape, which, in our model, represents race.

3. **Structural balance (`ttriad`)** A force, in this case, opposite to low density, higher prevalence of transitive triads makes individuals cluster.

4. **Age homophily (`absdiff("age")`)** This is the most prominent feature of the graph. In it, nodes are clustered by age.

Of the four features, **age homophily** is the one that stands out. Why is this tha case? If we look again at the parameters used in the ERGM and how these interact with vertices' attributes, we will find the answer:

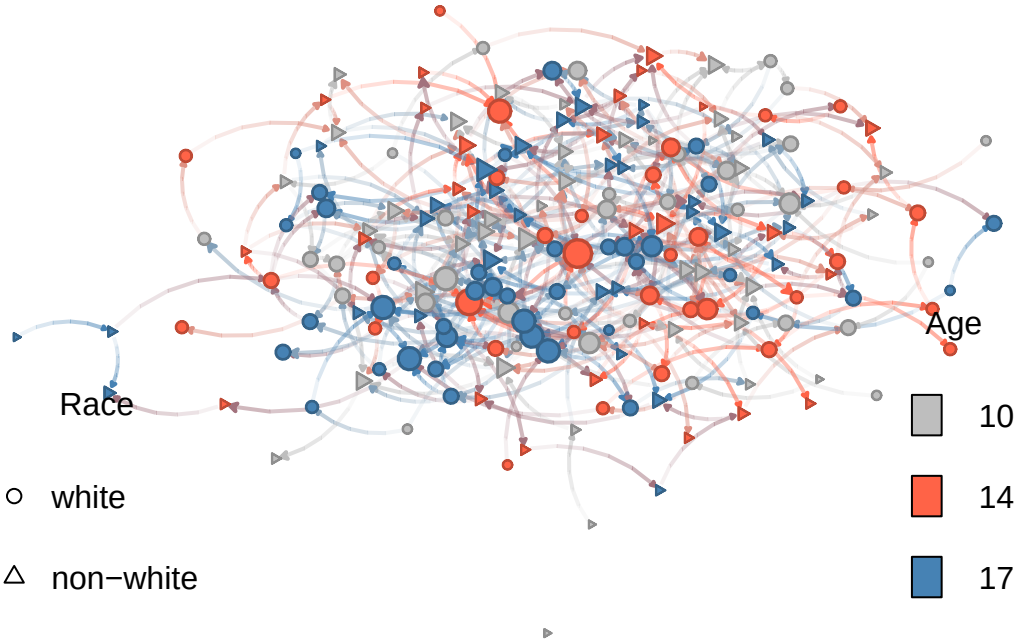
- The log-odds of a new race-homophilic tie are $1 \times \theta_{\text{race-homophily}} = 0.5$.
- But, the log-odd of an age heterophilic tie between, say, 14 and 17 year olds is $|17 - 14|\theta_{\text{age-homophily}} = 3 \times -0.5 = -1.5$.
- Therefore, the effect of heterophily (which is just the opposite of homophily) is significantly larger, actually three times in this case, than the race-homophily effect.

This observation becomes clear if we run another simulation with the same seed, but adjusting for the maximum size the effect of age-homophily can take. A quick-n-dirty way to achieve this is to re-run the simulation with the `nodematch` term instead of the `absdiff` term. This way, we (a) explicitly operationalize the term as homophily (before it was heterophily,) and (b) have both homophily effects have the same influence in the model:

```
net_sim2 <- simulate(
    net ~ edges +
    nodematch("race") +
    ttriad +
    nodematch("age"),
    coef = c(-5, .5, .25, .5) # This line changed
    )
```

Re-doing the plot. From the previous graph-drawing, only the graph structure changed. The vertex attributes are the same so we can go ahead and re-use them. Like I mentioned earlier, the `nplot_base` function currently supports `igraph` objects, so we will use `intergraph::asIgraph` to make it work:

```r
# Plotting
nplot_base(
  asIgraph(net_sim2),
  # We comment this out to allow for a new layout
  # layout = glayout,
  vertex.color = net_sim %v% "color",
  vertex.nsides = net_sim %v% "shape"
  )

# Color legend
legend(
  "bottomright",
  legend = names(vcolors_palette),
  fill   = vcolors_palette,
  bty    = "n",
  title  = "Age"
  )

# Shape legend
legend(
  "bottomleft",
  legend = names(vshape_list),
  pch    = c(1, 2),
  bty    = "n",
  title  = "Race"
  )
```



As expected, there is no longer a dominant effect in homophily. One important thing we can learn from this final example is that phenomena will not always show themselves in graph visualization. Careful analysis in complex networks is a must.

6 Egocentric networks

In egocentric social network analysis (ESNA, for our book,) instead of dealing with a single network, we have as many networks as participants in the study. Egos–the main study subjects–are analyzed from the perspective of their local social network. For a more extended view of ESNA, look at Raffaele Vacca's "*Egocentric network analysis with R*".

In this chapter, I show how to work with one particular type of ESNA data: information generated by the tool Network Canvas. You can download an "artificial" ZIP file containing the outputs from a Network Canvas project here[1]. We assume the ZIP file was extracted to the folder `data-raw/egonets`. You can go ahead and extract the ZIP by point-and-click or use the following R code to automate the process:

```
[1] FALSE
```

```
unzip(
  zipfile = "data-raw/networkCanvasExport-fake.zip",
  exdir   = "data-raw/egonets"
  )
```

This will extract all the files in `networkCanvasExport-fake.zip` to the subfolder `egonets`. Let's take a look at the first few files:

```
head(list.files(path = "data-raw/egonets"))
## [1] "I_-59190_BRB9111_attributeList_Person.csv"
## [2] "I_-59190_BRB9111_edgeList_Knows.csv"
## [3] "I_-59190_BRB9111_ego.csv"
## [4] "I_-59190_BRB9111.graphml"
```

[1] I thank Jacqueline M. Kent-Marvick, who provided me with what I used as a baseline to generate the artificial Network Canvas export.

```
## [5] "I-100BB_00B95-90_attributeList_Person.csv"
## [6] "I-100BB_00B95-90_edgeList_Knows.csv"
```

As you can see, for each ego in the dataset, there are four files:

- `...attributeList_Person.csv`: Attributes of the alters.
- `...edgeList_Knows.csv`: Edgelist indicating the ties between the alters.
- `...ego.csv`: Information about the egos.
- `...graphml`: And a `graphml` file that contains the egonets.

The next sections will illustrate, file by file, how to read the information into R, apply any required processing, and store the information for later use. We start with the `graphml` files.

6.1 Network files (graphml)

The `graphml` files can be read directly with `igraph`'s `read_graph` function. The key is to take advantage of R's lists to avoid writing over and over the same block of code, and, instead, manage the data through lists.

Just like any data-reading function, `read_graph` function requires a file path to the network file. **The function we will use to list the required files is `list.files()`**:

```
# We start by loading igraph
library(igraph)

# Listing all the graphml files
graph_files <- list.files(
  path       = "data-raw/egonets", # Where are these files
  pattern    = "*.graphml",        # Specify a pattern for only listing graphml
  full.names = TRUE                # And we make sure we use the full name
                                   # (path.) Otherwise, we would only get names.
  )
```

```
# Taking a look at the first three files we got
graph_files[1:3]
## [1] "data-raw/egonets/I_-59190_BRB9111.graphml"
## [2] "data-raw/egonets/I-100BB_00B95-90.graphml"
## [3] "data-raw/egonets/I-1BB79950-0-7.graphml"

# Applying igraph's read_graph
graphs <- lapply(
  X      = graph_files,       # List of files to read
  FUN    = read_graph,        # The function to apply
  format = "graphml"          # Argument passed to read_graph
  )
```

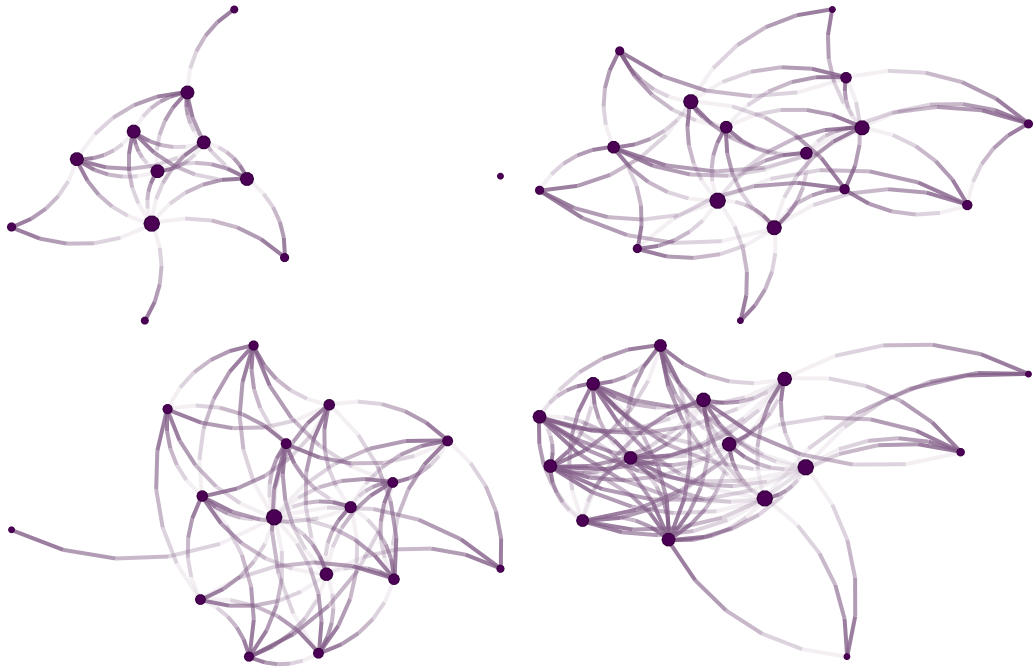
If the operation succeeded, the previous code block should generate a list of `igraph` objects named `graphs`. Let's take a peek at the first two:

```
graphs[[1]]
## IGRAPH 5743656 U--- 12 25 --
## + attr: age (v/n), healthy_diet (v/n), gender_1 (v/l), eat_with_2
## | (v/l), id (v/c)
## + edges from 5743656:
##  [1] 1-- 3 1-- 2 1-- 6 1-- 5 1-- 4 1-- 8 1--11 1--10 2-- 3 3-- 7 3-- 4 3-- 5
## [13] 3-- 6 2-- 7 2-- 4 2-- 5 2-- 6 5-- 6 6--10 7-- 9 4-- 5 5-- 7 4--11 6-- 7
## [25] 4-- 7
graphs[[2]]
## IGRAPH 971cd35 U--- 16 47 --
## + attr: age (v/n), healthy_diet (v/n), gender_1 (v/l), eat_with_2
## | (v/l), id (v/c)
## + edges from 971cd35:
##  [1]  7--13  1-- 5  1-- 6  1-- 4  1-- 2  7--15  1-- 3 11--13  1--10  1--16
## [11]  4-- 6  2-- 6  6-- 7  1--11 11--15  6-- 9  6-- 8  3-- 9  5--15  4-- 5
## [21]  2-- 5  5-- 8  5-- 7  5--10  3-- 5  6--14 12--13  6--13  3--13  2-- 3
## [31]  3-- 4  3--16  3--11 10--14  7--14  2-- 4  2--10  2--15 10--12  4-- 7
## [41]  6--10  5--11  9--10  1-- 9  1--12  3--12  4--14
```

As always, one of the first things we do with networks is visualize them. We will use the

`netplot` R package (by yours truly) to draw the figures:

```
library(netplot)
library(gridExtra)

# Graph layout is random
set.seed(1231)

# The grid.arrange allows putting multiple netplot graphs into the same page
grid.arrange(
  nplot(graphs[[1]]),
  nplot(graphs[[2]]),
  nplot(graphs[[3]]),
  nplot(graphs[[4]]),
  ncol = 2, nrow = 2
)
```



Great! Since nodes in our network have features, we can add a little color. We will use the `eat_with_2` variable, coded as `TRUE` or `FALSE`. Vertex colors can be specified using the `vertex.color` argument of the `nplot` function. In our case, we will specify colors passing a vector with length equal to the number of nodes in the graph. Furthermore, since we will be doing this multiple times, it is worthwhile writing a function:

```r
# A function to color by the eat with variable
color_it <- function(net) {

  # Coding eat_with_2 to be 1 (FALSE) or 2 (TRUE)
  eatswith <- V(net)$eat_with_2

  # Subsetting the color
  ifelse(eatswith, "purple", "darkgreen")

}
```

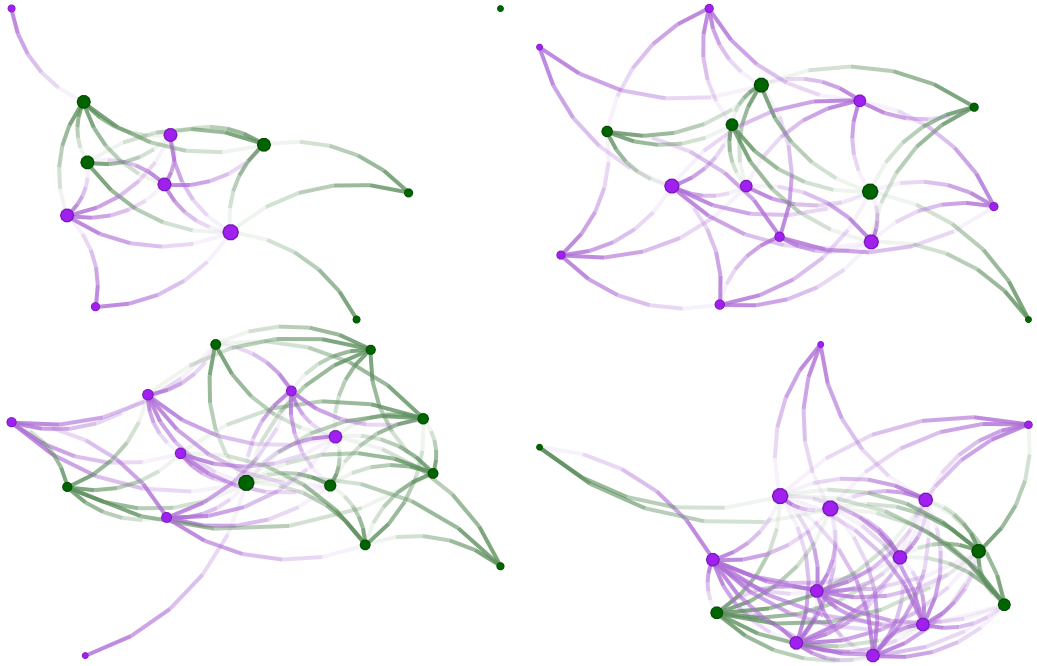
This function takes two arguments: a network and a vector of two colors. Vertex attributes in `igraph` can be accessed through the `V(...)$...` function. For this example, to access the attribute `eat_with_2` in the network `net`, we type `V(net)$eat_with_2`. Finally, individuals with `eat_with_2` equal to true will be colored `purple`; otherwise, if equal to `FALSE`, they will be colored `darkgreen`. Before plotting the networks, let's see what we get when we access the `eat_with_2` attribute in the first graph:

```r
V(graphs[[1]])$eat_with_2
```

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

A logical vector. Now let's redraw the figures:

```r
grid.arrange(
  nplot(graphs[[1]], vertex.color = color_it(graphs[[1]])),
  nplot(graphs[[2]], vertex.color = color_it(graphs[[2]])),
  nplot(graphs[[3]], vertex.color = color_it(graphs[[3]])),
  nplot(graphs[[4]], vertex.color = color_it(graphs[[4]])),
  ncol = 2, nrow = 2
)
```



Since most of the time, we will be dealing with many egonets; you may want to draw each network independently; the following code block does that. First, if needed, will create a folder to store the networks. Then, using the `lapply` function, it will use `netplot::nplot()` to draw the networks, add a legend, and save the graph as `.../graphml_[number].png`, where `[number]` will go from `01` to the total number of networks in `graphs`.

```
if (!dir.exists("egonets/figs/egonets"))
  dir.create("egonets/figs/egonets", recursive = TRUE)

lapply(seq_along(graphs), function(i) {

  # Creating the device
  png(sprintf("egonets/figs/egonets/graphml_%02i.png", i))

  # Drawing the plot
  p <- nplot(
    graphs[[i]],
    vertex.color = color_it(graphs[[i]])
    )

  # Adding a legend
```

```
  p <- nplot_legend(
    p,
    labels = c("eats with: FALSE", "eats with: TRUE"),
    pch    = 21,
    packgrob.args = list(side = "bottom"),
    gp            = gpar(
      fill = c("darkgreen", "purple")
    ),
    ncol = 2
  )

  print(p)

  # Closing the device
  dev.off()
})
```

6.2 Person files

Like before, we list the files ending in `Person.csv` (with the full path,) and read them into R. While R has the function `read.csv`, here I use the function `fread` from the `data.table` R package. Alongside `dplyr`, `data.table` is one of the most popular data-wrangling tools in R. Besides syntax, the biggest difference between the two is performance; `data.table` is significantly faster than any other data management package in R, and is a great alternative for handling large datasets. The following code block loads the package, lists the files, and reads them into R.

```
# Loading data.table
library(data.table)

# Listing the files
person_files <- list.files(
  path       = "data-raw/egonets",
  pattern    = "*Person.csv",
```

```
  full.names = TRUE
  )

# Loading all into a single list
persons <- lapply(person_files, fread)

# Looking into the first element
persons[[1]]
##     nodeID   age
##      <int> <int>
##  1:      1    45
##  2:      2    32
##  3:      3    31
##  4:      4    45
##  5:      5    43
##  6:      6    47
##  7:      7    45
##  8:      8    62
##  9:      9    28
## 10:     10    41
## 11:     11    41
## 12:     12    46
## 13:     13    46
## 14:     14    46
## 15:     15    62
## 16:     16    41
```

A common task is adding an identifier to each dataset in `persons` so we know from to which ego they belong. Again, the `lapply` function is our friend:

```
persons <- lapply(seq_along(persons), function(i) {
  persons[[i]][, dataset_num := i]
})
```

In `data.table`, variables are created using the `:=` symbol. The previous code chunk is equivalent to this:

```
for (i in 1:length(persons)) {
  persons[[i]]$dataset_num <- i
}
```

If needed, we can transform the list `persons` into a `data.table` object (i.e., a single `data.frame`) using the `rbindlist` function[2]. The next code block uses that function to combine the `data.table`s into a single dataset.

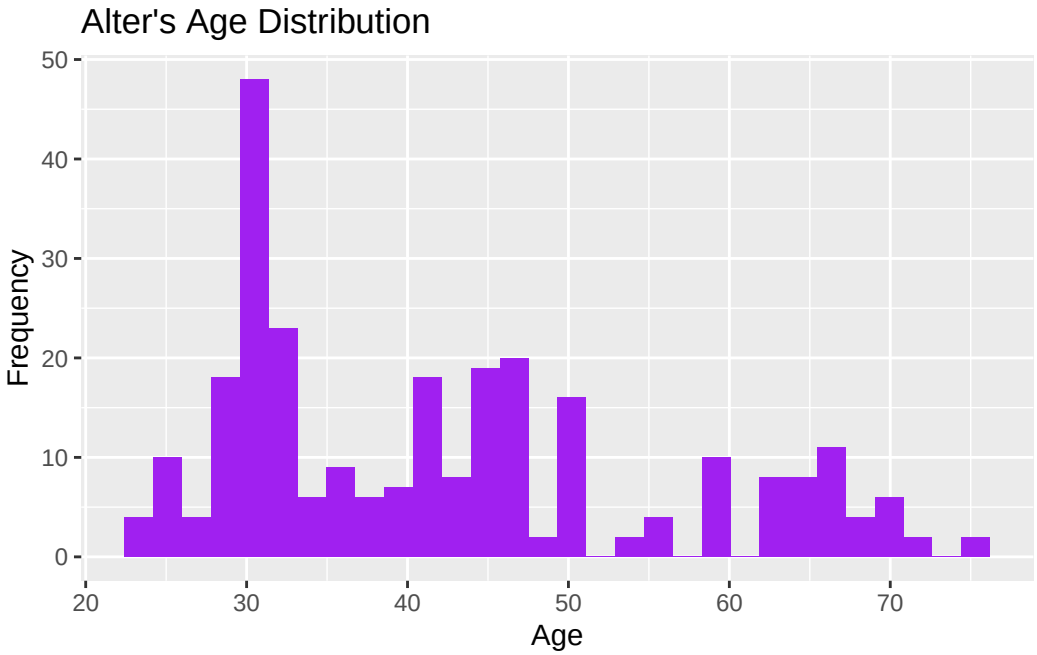
```
# Combining the datasets
persons <- rbindlist(persons)
persons
##       nodeID   age dataset_num
##        <int> <int>       <int>
##    1:      1    45           1
##    2:      2    32           1
##    3:      3    31           1
##    4:      4    45           1
##    5:      5    43           1
##   ---
## 271:      7    43          19
## 272:      8    48          19
## 273:      9    70          19
## 274:     10    46          19
## 275:     11    50          19
```

Now that we have a single dataset, we can do some data exploration. For example, we can use the package `ggplot2` to draw a histogram of alters' ages.

```
# Loading the ggplot2 package
library(ggplot2)

# Histogram of age
ggplot(persons, aes(x = age)) +            # Starting off the plot
```

[2] Although not the same, `rbindlist` (almost always) yields the same result as calling the function `do.call`. In particular, instead of executing the call `rbindlist(persons)`, we could have used `do.call(rbind, persons)`.

```
  geom_histogram(fill = "purple") +      # Adding a histogram
  labs(x = "Age", y = "Frequency") +       # Changing the x/y axis labels
  labs(title = "Alter's Age Distribution") # Adding a title
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



6.3 Ego files

The ego files contain information about egos (duh!.) Again, we will read them all at once using `list.files` + `lapply`:

```
# Listing files ending with *ego.csv
ego_files <- list.files(
  path       = "data-raw/egonets",
  pattern    = "*ego.csv",
  full.names = TRUE
  )

# Reading the files with fread
egos <- lapply(ego_files, fread)
```

```
# Combining them
egos <- rbindlist(egos)
head(egos)
##                   networkCanvasEgoUUID networkCanvasCaseID
##                                 <char>              <char>
## 1: I-11ca3a78c-62f131f37169-c139217a1f6    I_-59190_BRB9111
## 2: I-fef-ab-4-5a--7-35c4f23-96eb32-34ea    I-100BB_00B95-90
## 3: I2f1bd0b6d-f71f4664cf-d-26-97408f22d      I-1BB79950-0-7
## 4: Id36bb-3b2bcbd2a6239b1103134c6b3d1d6    I000091I_RB010B5
## 5: I436d32fc67fb5c6-23-244f353849b120cd    I019051R0_RRR0-0
## 6: Ibf1f-2-34162bb5f2c36b8241--316a-fff    I01B11-I1101_44R
##                  networkCanvasSessionID
##                                 <char>
## 1: I612b7a1af---0880b-70698204-b-8dbf09
## 2: If5e0-f-26cbec070760f-e6b6d26ebfb06f
## 3: I825c293a1304-e5-cbea8a80aae05b305fa
## 4: I1b8a7d0f6b4-8298c9-848-9186d68a7f3c
## 5: Ie620be37b75983c49ac63-38-425227c959
## 6: Ie3-134323ed40-0e-d954b3d-febbcb9363
##                       networkCanvasProtocolName        sessionStart
##                                           <char>              <POSc>
## 1: Postpartum social networks with sociogram_V5 2023-02-22 23:41:59
## 2: Postpartum social networks with sociogram_V5 2023-02-10 21:46:02
## 3: Postpartum social networks with sociogram_V5 2023-03-01 16:52:09
## 4: Postpartum social networks with sociogram_V5 2023-01-26 20:38:07
## 5: Postpartum social networks with sociogram_V5 2023-02-06 14:55:57
## 6: Postpartum social networks with sociogram_V5 2023-03-16 18:20:02
##          sessionFinish     sessionExported
##                 <POSc>              <POSc>
## 1: 2023-02-23 01:47:00 2023-02-23 01:47:08
## 2: 2023-02-11 01:29:32 2023-02-11 01:34:12
## 3: 2023-03-02 16:51:20 2023-03-02 17:04:42
## 4: 2023-01-26 22:03:20 2023-01-26 22:03:34
## 5: 2023-02-06 15:49:38 2023-02-06 15:56:42
## 6: 2023-03-17 21:11:09 2023-03-17 21:16:15
```

A cool thing about `data.table` is that, within square brackets, we can manipulate the data referring to the variables directly. For example, if we wanted to calculate the difference between `sessionFinish` and `sessionStart`, using base R we would do the following:

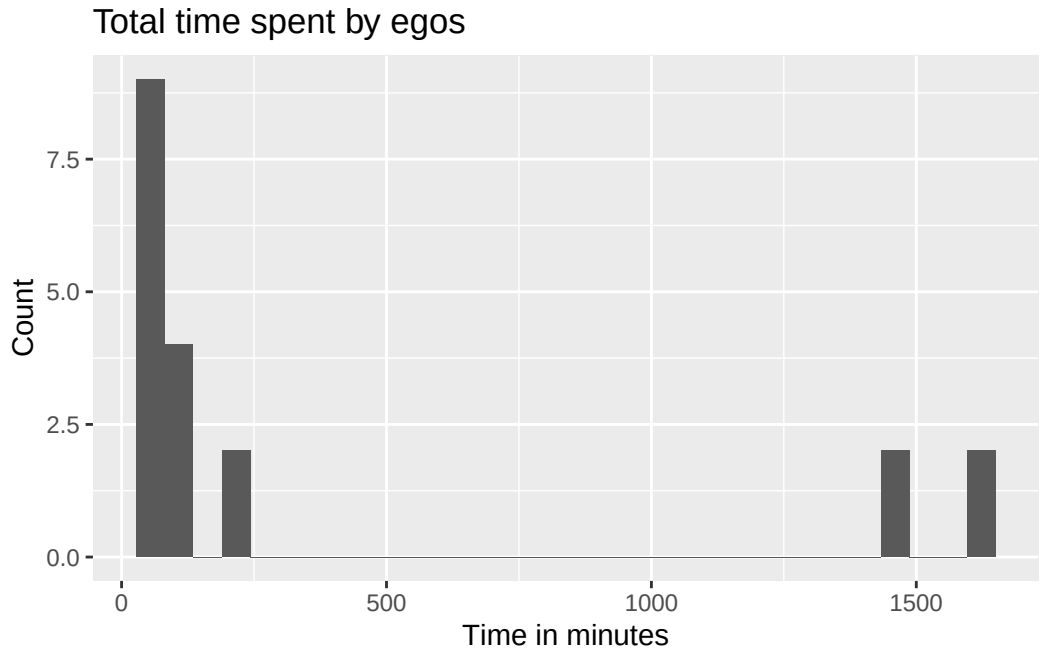
```
egos$total_time <- egos$sessionFinish - egos$sessionStart
```

Whereas with `data.table`, variable creation is much more straightforward (notice that instead of using `<-` or `=` to assign a variable, we use the `:=` operator):

```
# How much time?
egos[, total_time := sessionFinish - sessionStart]
```

We can also visualize this using `ggplot2`:

```
ggplot(egos, aes(x = total_time)) +
  geom_histogram() +
  labs(x = "Time in minutes", y = "Count") +
  labs(title = "Total time spent by egos")
## Don't know how to automatically pick scale for object of type <difftime>.
## Defaulting to continuous.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



6.4 Edgelist files

As I mentioned earlier, since we are reading the `graphml` files, using the edgelist may not be needed. Nevertheless, the process to import the edgelist file to R is the same we have been applying: list the files and read them all at once using `lapply`:

```
# Listing all files ending in Knows.csv
edgelist_files <- list.files(
  path = "data-raw/egonets",
  pattern = "*Knows.csv",
  full.names = TRUE
  )

# Reading all files at once
edgelists <- lapply(edgelist_files, fread)
```

To avoid confusion, we can also add ids corresponding to the file number. Once we do that, we can combine all files into a single `data.table` object using `rbindlist`:

```
edgelists <- lapply(seq_along(edgelists), function(i) {
  edgelists[[i]][, dataset_num := i]
})

edgelists <- rbindlist(edgelists)

head(edgelists)
##    edgeID  from    to                networkCanvasEgoUUID
##     <int> <int> <int>                              <char>
## 1:      1     1     5 I839f-8fa8f8aeb8-eaf---ba8-cf3908f3a
## 2:      2     1    10 If81a9c0f-9f4f28ccf-c4c923a-8-0f5fce
## 3:      3     1     9 I899ffe-27-3-a3-ca2fb7f7-ca8e7715ce9
## 4:      4     1    10 I814efaba88cbb02caa8c89790-83beeaf9-
## 5:      5     7     6 Ifd-0eec2e08974eaf2b79f-9efb7e3-8998
## 6:      6     2     6 I-28fe89cc-fc5db3825b92-ae87c-c18e3d
##                      networkCanvasUUID             networkCanvasSourceUUID
##                                 <char>                              <char>
```

```
## 1: I720400eb19bccce-77cee773289b02-fe7e I4d5--16a08f8ba463c6458f8979e-65fa9d
## 2: I-b469c0-60f8bbb543-32-628-216f9-038 I-6cf8-f3da-4-96-87efaf5daaa48ba5e5c
## 3: Ifa4933-9baaf5fc-f-e4f5c5e5-ff34-f-f I5-f69a6eaa-5956e8897ca999-ffb6ed-e1
## 4: I4cb-904496b1-6194bcb51b58444b40-ef8 I3e5-6c8d5e0f086--e-5ab45-4-5aaa5-0e
## 5: I0ab7--b7a0ee71e54c1e93cdb-4ca5ab1-b I5-b-9-7eca5ab5-91915ba9b6565a6e42cc
## 6: Ic80142fc4c431009e84b3-ab3f-9b0eab03 Ie0a24eea4e01a4340343a0-66723-a-9970
##                  networkCanvasTargetUUID dataset_num
##                                   <char>       <int>
## 1: Id1c8befd46bdd195c-ce91a8-bc0---4f0e           1
## 2: I757b4a-3ea4d95--b9ebb9db3d55dcbaf-c           1
## 3: I92a62925ff9-e2f27-6ef97d-29fb729624           1
## 4: I7f--da48-46a64-b972c-ef6bbec--64cb4           1
## 5: I-eaa7e95659-9cf01a4f5fd69af54e6-d60           1
## 6: I69060e8a-454609-faa04cd3eeb-5-9550-           1
```

6.5 Putting all together

In this last part of the chapter, we will use the `igraph` and `ergm` packages to generate features (covariates, controls, independent variables, or whatever you call them) at the ego-network level. Once again, the `lapply` function is our friend

6.5.1 Generating statistics using igraph

The `igraph` R package has multiple high-performing routines to compute graph-level statistics. For now, we will focus on the following statistics: vertex count, edge count, number of isolates, transitivity, and modularity based on betweenness centrality:

```
net_stats <- lapply(graphs, function(g) {

  # Calculating modularity
  groups <- cluster_edge_betweenness(g)

  # Computing the stats
  data.table(
```

```
    size      = vcount(g),
    edges     = ecount(g),
    nisolates = sum(degree(g) == 0),
    transit   = transitivity(g, type = "global"),
    modular   = modularity(groups)
  )
})
```

Observe we count isolates using the `degree()` function. We can combine the statistics into a single `data.table` using the `rbindlist` function:

```
net_stats <- rbindlist(net_stats)

head(net_stats)
##     size edges nisolates   transit     modular
##    <num> <num>     <int>     <num>       <num>
## 1:    12    25         1 0.6750000 0.012000000
## 2:    16    47         0 0.4332130 0.003395201
## 3:    16    58         0 0.5612009 0.002675386
## 4:    15    75         0 0.8515112 0.000000000
## 5:    15    52         0 0.5780488 0.000000000
## 6:    17    68         0 0.6291161 0.025735294
```

6.5.2 Generating statistics based on ergm

The `ergm` R package has a much larger set of graph-level statistics we can add to our models.[3] The key to generating statistics based on the `ergm` package is the `summary_formula` function. Before we start using that function, we first need to convert the `igraph` networks to `network` objects, which are the native object class for the `ergm` package. We use the `intergraph` R package for that, and in particular, the `asNetwork` function:

```
# Loading the required packages
library(intergraph)
library(ergm)
```

[3] There's an obvious reason, ERGMs are all about graph-level statistics!

```
## Loading required package: network
##
## 'network' 1.18.2 (2023-12-04), part of the Statnet Project
## * 'news(package="network")' for changes since last version
## * 'citation("network")' for citation information
## * 'https://statnet.org' for help, support, and other information
##
## Attaching package: 'network'
## The following objects are masked from 'package:igraph':
##
##     %c%, %s%, add.edges, add.vertices, delete.edges, delete.vertices,
##     get.edge.attribute, get.edges, get.vertex.attribute, is.bipartite,
##     is.directed, list.edge.attributes, list.vertex.attributes,
##     set.edge.attribute, set.vertex.attribute
##
## 'ergm' 4.6.0 (2023-12-17), part of the Statnet Project
## * 'news(package="ergm")' for changes since last version
## * 'citation("ergm")' for citation information
## * 'https://statnet.org' for help, support, and other information
## 'ergm' 4 is a major update that introduces some backwards-incompatible
## changes. Please type 'news(package="ergm")' for a list of major
## changes.

# Converting all "igraph" objects in graphs to network "objects"
graphs_network <- lapply(graphs, asNetwork)
```

With the network objects ready, we can proceed to compute graph-level statistics using the `summary_formula` function. Here we will only look into: the number of triangles, gender homophily, and healthy-diet homophily:

```
net_stats_ergm <- lapply(graphs_network, function(n) {

  # Computing the statistics
  s <- summary_formula(
    n ~ triangles +
      nodematch("gender_1") +
```

```
      nodematch("healthy_diet")
    )

  # Saving them as a data.table object
  data.table(
    triangles       = s[1],
    gender_homoph   = s[2],
    healthyd_homoph = s[3]
  )
})
```

Once again, we use `rbindlist` to combine all the network statistics into a single `data.table` object:

```
net_stats_ergm <- rbindlist(net_stats_ergm)
head(net_stats_ergm)
##    triangles gender_homoph healthyd_homoph
##        <num>         <num>            <num>
## 1:        27            11                3
## 2:        40            30               20
## 3:        81            40               29
## 4:       216            33               38
## 5:        79            44               19
## 6:       121            38               16
```

6.6 Saving the data

We end the chapter saving all our work into four datasets:

- Network statistics (as a csv file)
- Igraph objects (as a rda file, which we can read back using `read.rds`)
- Network objects (idem)
- Person files (alter's information, as a csv file.)

CSV files can be saved either using `write.csv` or, as we do here, `fwrite` from the `data.table` package:

```
# Checking directory exists
if (!dir.exists("data"))
  dir.create("data")

# Network attributes
master <- cbind(egos, net_stats, net_stats_ergm)
fwrite(master, file = "data/network_stats.csv")

# Networks
saveRDS(graphs, file = "data/networks_igraph.rds")
saveRDS(graphs_network, file = "data/networks_network.rds")

# Attributes
fwrite(persons, file = "data/persons.csv")
```

7 Network diffusion

This chapter is based on the 2018 and 2019 tutorials of netdiffuseR at the Sunbelt conference. The source code of the tutorials, taught by Thomas W. Valente and George G. Vega Yon (author of this book), is available here.

7.1 Network diffusion of innovation

7.1.1 Diffusion networks

- Explains how new ideas and practices (innovations) spread within and between communities.
- While a lot of factors have been shown to influence diffusion (Spatial, Economic, Cultural, Biological, etc.), Social Networks are prominent.
- There are many components in the diffusion network model, including network exposures, thresholds, infectiousness, susceptibility, hazard rates, diffusion rates (bass model), clustering (Moran's I), and so on.

7.1.2 Thresholds

- One of the canonical concepts is the network threshold. Network thresholds (Valente, 1995; 1996), τ, are defined as the required proportion or number of neighbors that lead you to adopt a particular behavior (innovation), $a = 1$. In (very) general terms

$$a_i = \begin{cases} 1 & \text{if } \tau_i \leq E_i \\ 0 & \text{Otherwise} \end{cases} \qquad E_i \equiv \frac{\sum_{j \neq i} \mathbf{X}_{ij} a_j}{\sum_{j \neq i} \mathbf{X}_{ij}}$$

Where E_i is i's exposure to the innovation and $\mathbf{X}$ is the adjacency matrix (the network).

- This can be generalized and extended to include covariates and other network weighting schemes (that's what **netdiffuseR** is all about).

7.2 The netdiffuseR R package

7.2.1 Overview

netdiffuseR is an R package that:

- It is designed to Visualize, Analyze, and simulate network diffusion data (in general).
- Depends on some pretty popular packages:
  - *RcppArmadillo*: So it's fast,
  - *Matrix*: So it's big,
  - *statnet* and *igraph*: So it's not from scratch
- Can handle big graphs, e.g., an adjacency matrix with more than 4 billion elements (PR for RcppArmadillo)
- Already on CRAN with ~6,000 downloads since its first version, Feb 2016,
- A lot of features to make it easy to read network (dynamic) data, making it a companion of other net packages.

7.2.2 Datasets

- **netdiffuseR** has the three classic Diffusion Network Datasets:
  - `medInnovationsDiffNet` Doctors and the innovation of Tetracycline (1955).
  - `brfarmersDiffNet` Brazilian farmers and the innovation of Hybrid Corn Seed (1966).
  - `kfamilyDiffNet` Korean women and Family Planning methods (1973).

```
brfarmersDiffNet
```

```
Dynamic network of class -diffnet-
 Name               : Brazilian Farmers
 Behavior           : Adoption of Hybrid Corn Seeds
 # of nodes         : 692 (1001, 1002, 1004, 1005, 1007, 1009, 1010, 1020, ...)
 # of time periods  : 21 (1946 - 1966)
 Type               : directed
 Final prevalence   : 1.00
 Static attributes  : village, idold, age, liveout, visits, contact, coo... (146)
 Dynamic attributes : -
```

```
medInnovationsDiffNet
```

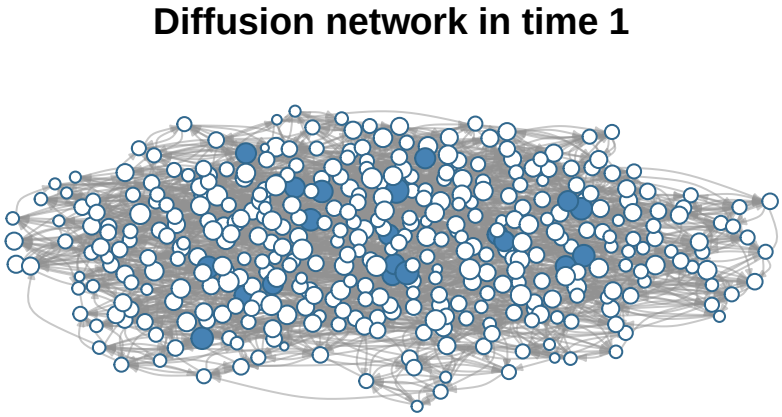
```
Dynamic network of class -diffnet-
 Name               : Medical Innovation
 Behavior           : Adoption of Tetracycline
 # of nodes         : 125 (1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, ...)
 # of time periods  : 18 (1 - 18)
 Type               : directed
 Final prevalence   : 1.00
 Static attributes  : city, detail, meet, coll, attend, proage, length, ... (58)
 Dynamic attributes : -
```

```
kfamilyDiffNet
```

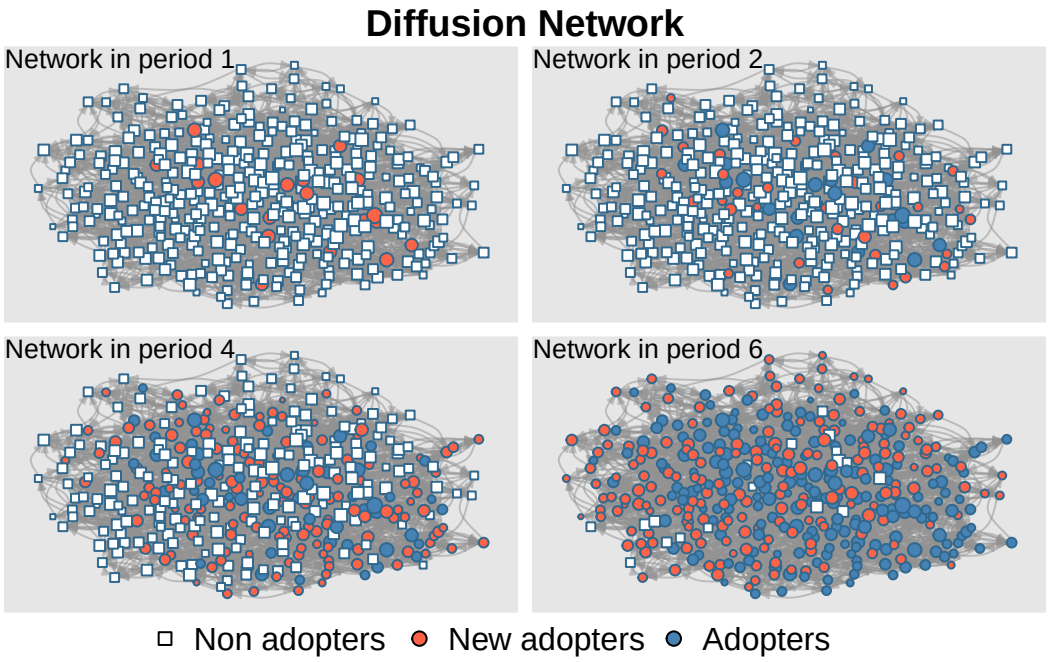
```
Dynamic network of class -diffnet-
 Name               : Korean Family Planning
 Behavior           : Family Planning Methods
 # of nodes         : 1047 (10002, 10003, 10005, 10007, 10010, 10011, 10012, 10014,
 # of time periods  : 11 (1 - 11)
 Type               : directed
 Final prevalence   : 1.00
 Static attributes  : village, recno1, studno1, area1, id1, nmage1, nmag... (430)
 Dynamic attributes : -
```

7.2.3 Visualization methods

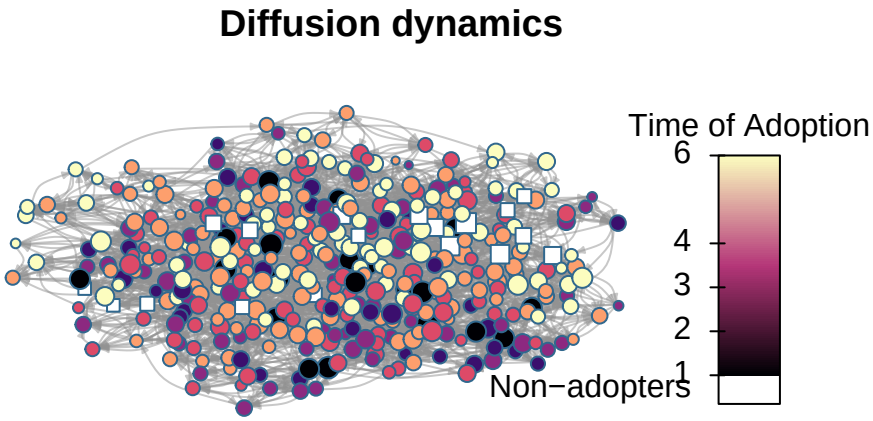
```
set.seed(12315)
x <- rdiffnet(
  400, t = 6, rgraph.args = list(k=6, p=.3),
  seed.graph = "small-world",
  seed.nodes = "central", rewire = FALSE, threshold.dist = 1/4
  )
plot(x)
```



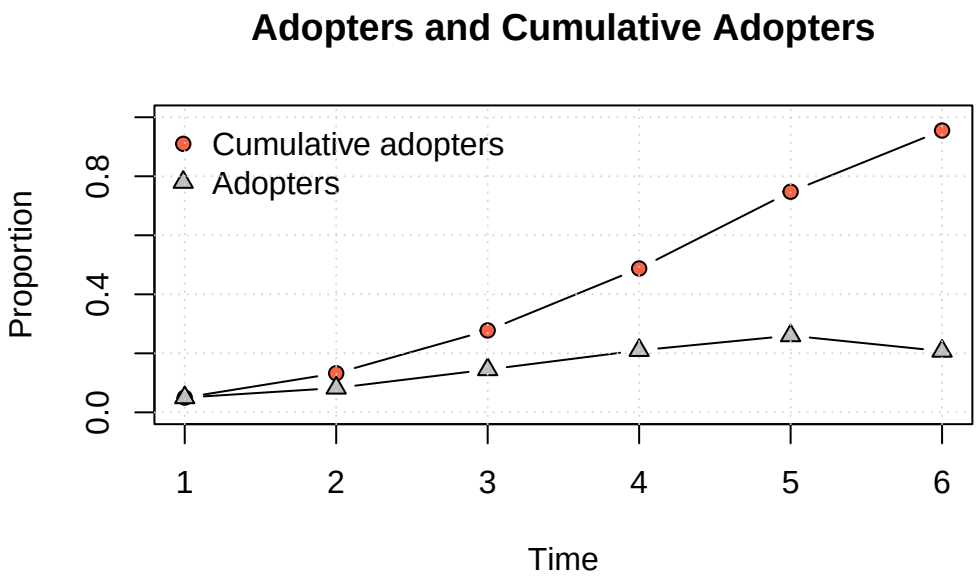

```
plot_diffnet(x)
```



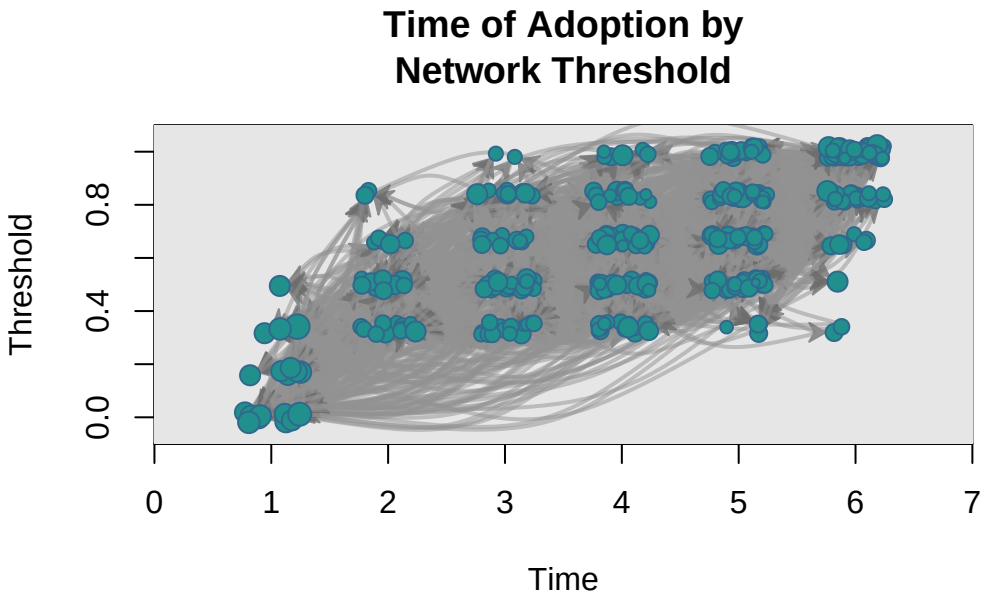

```
plot_diffnet2(x)
```




```
plot_adopters(x)
```

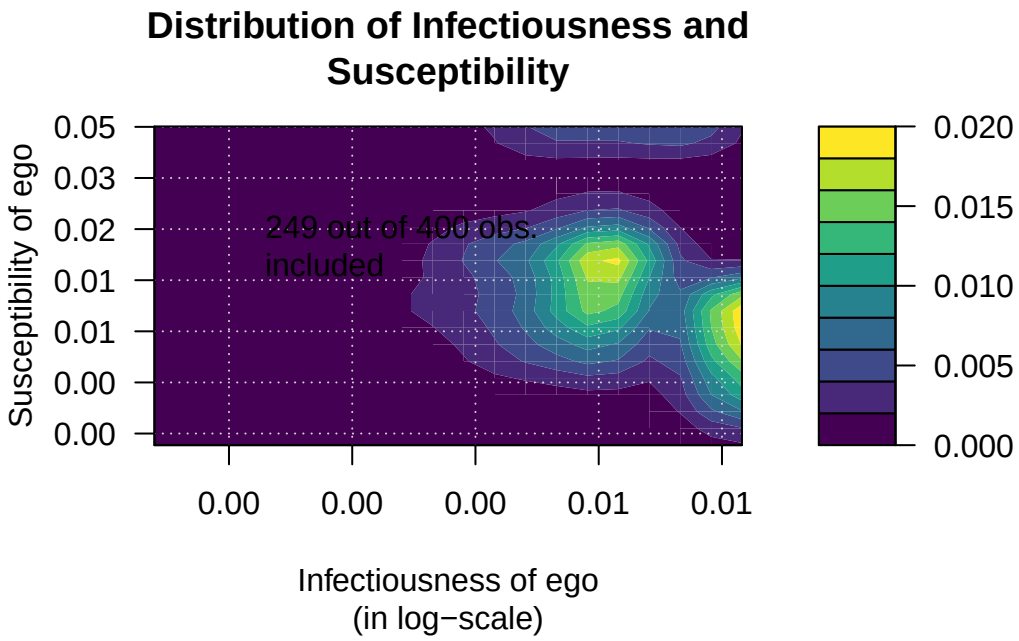



```
plot_threshold(x)
```

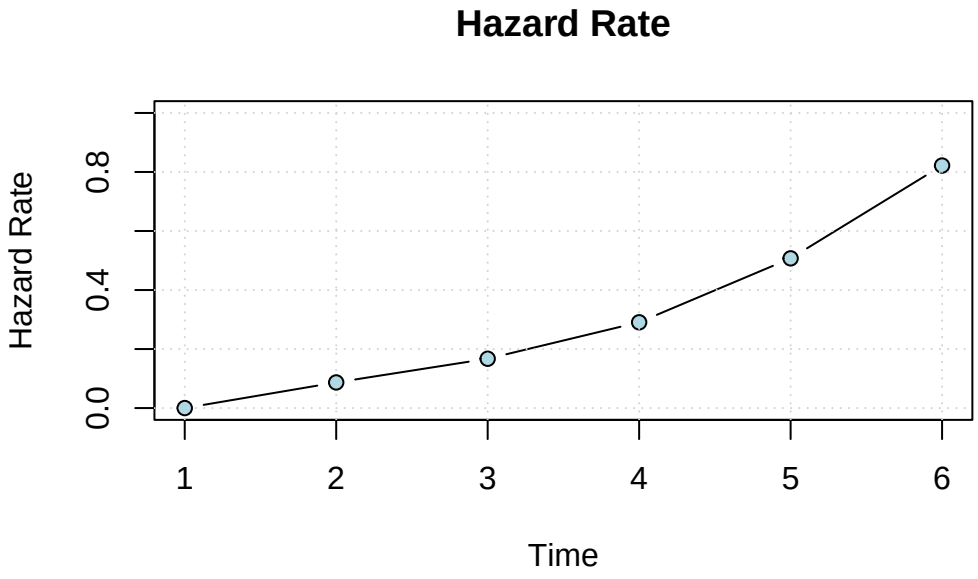



```
plot_infectsuscep(x, K=2)
```

```
Warning in plot_infectsuscep.list(graph$graph, graph$toa, t0, normalize, : When
applying logscale some observations are missing.
```




```
plot_hazard(x)
```



7.2.4 Problems

1. Using the diffnet object in `intro.rda`, use the function `plot_threshold` specifying shapes and colors according to the variables ItrustMyFriends and Age. Do you see any pattern?

7.3 Simulation of diffusion processes

Before we start, a review of the concepts we will be using here

1. Exposure: Proportion/number of neighbors that have adopted an innovation at each point in time.
2. Threshold: The proportion/number of your neighbors who had adopted at or one time period before ego (the focal individual) adopted.
3. Infectiousness: How much i's adoption affects her alters.
4. Susceptibility: How much i's alters' adoption affects her.
5. Structural equivalence: How similar is i to j in terms of position in the network.

7.3.1 Simulating diffusion networks

We will simulate a diffusion network with the following parameters:

1. Will have 1,000 vertices,
2. Will span 20 time periods,
3. The initial adopters (seeds) will be selected at random,
4. Seeds will be a 10% of the network,
5. The graph (network) will be small-world,
6. Will use the WS algorithm with $p = .2$ (probability of rewiring).
7. Threshold levels will be uniformly distributed between [0.3, 0.7]

To generate this diffusion network, we can use the `rdiffnet` function included in the package:

```r
# Setting the seed for the RNG
set.seed(1213)

# Generating a random diffusion network
net <- rdiffnet(
  n              = 1e3,                          # 1.
  t              = 20,                           # 2.
  seed.nodes     = "random",                     # 3.
  seed.p.adopt   = .1,                           # 4.
  seed.graph     = "small-world",                # 5.
  rgraph.args    = list(p=.2),                   # 6.
  threshold.dist = function(x) runif(1, .3, .7) # 7.
  )
```

```
Warning in (function (graph, p, algorithm = "endpoints", both.ends = FALSE, :
The option -copy.first- is set to TRUE. In this case, the first graph will be
treated as a baseline, and thus, networks after T=1 will be replaced with T-1.
```

- The function `rdiffnet` generates random diffusion networks. Main features:

    1. Simulating random graph or using your own,
    2. Setting threshold levels per node,
    3. Network rewiring throughout the simulation, and
    4. Setting the seed nodes.

- The simulation algorithm is as follows:

    1. If required, a baseline graph is created,
    2. Set of initial adopters and threshold distribution are established,
    3. The set of t networks is created (if required), and
    4. Simulation starts at t=2, assigning adopters based on exposures and thresholds:

    a. For each $i \in N$, if its exposure at $t-1$ is greater than its threshold, then adopts, otherwise, continue without change.

    b. next i

7.3.2 Rumor spreading

```
library(netdiffuseR)

set.seed(09)
diffnet_rumor <- rdiffnet(
  n = 5e2,
  t = 5,
  seed.graph = "small-world",
  rgraph.args = list(k = 4, p = .3),
  seed.nodes = "random",
  seed.p.adopt = .05,
  rewire = TRUE,
  threshold.dist = function(i) 1L,
  exposure.args = list(normalized = FALSE)
  )
```

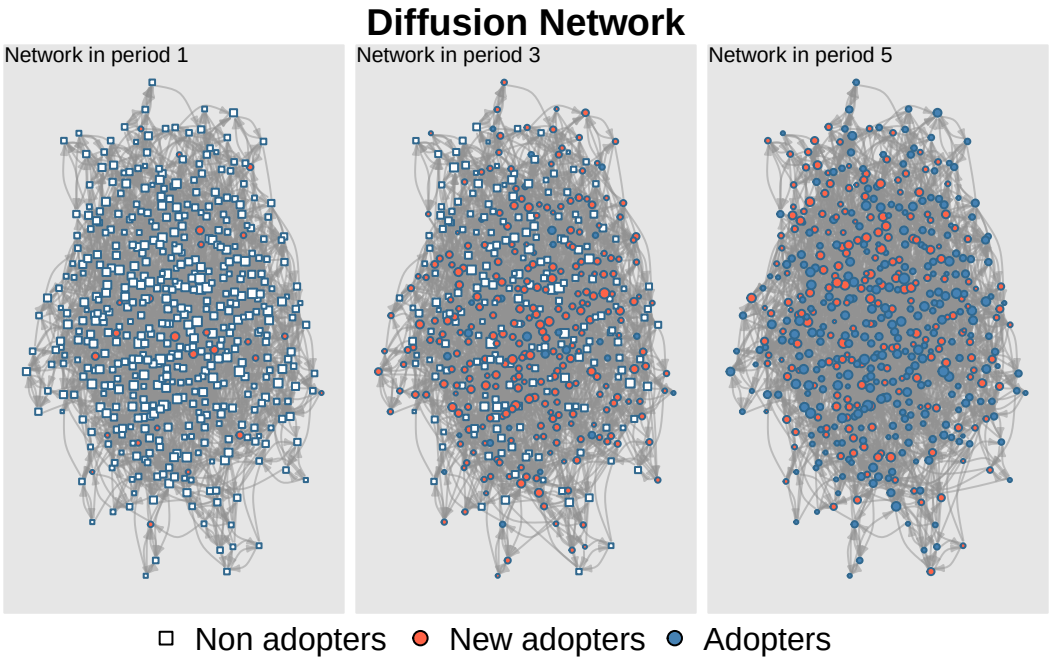
```
Warning in (function (graph, p, algorithm = "endpoints", both.ends = FALSE, :
The option -copy.first- is set to TRUE. In this case, the first graph will be
treated as a baseline, and thus, networks after T=1 will be replaced with T-1.
```

```
summary(diffnet_rumor)
```

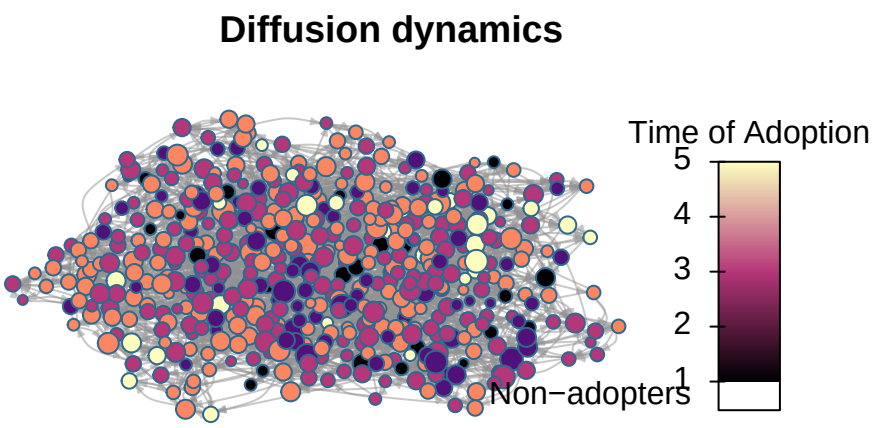
```
Diffusion network summary statistics
Name     : A diffusion network
Behavior : Random contagion
---------------------------------------------------------------------------
 Period   Adopters   Cum Adopt. (%)   Hazard Rate   Density   Moran's I (sd)
-------- ---------- ---------------- ------------ --------- ----------------
      1         25        25 (0.05)            -      0.01 -0.00 (0.00)
      2         78       103 (0.21)         0.16      0.01  0.01 (0.00) ***
      3        187       290 (0.58)         0.47      0.01  0.01 (0.00) ***
      4        183       473 (0.95)         0.87      0.01  0.01 (0.00) ***
      5         27       500 (1.00)         1.00      0.01                -
---------------------------------------------------------------------------
```

```
Left censoring   : 0.05 (25)
Right centoring : 0.00 (0)
# of nodes       : 500

Moran's I was computed on contemporaneous autocorrelation using 1/geodesic
values. Significane levels  *** <= .01, ** <= .05, * <= .1.
```

```
plot_diffnet(diffnet_rumor, slices = c(1, 3, 5))
```



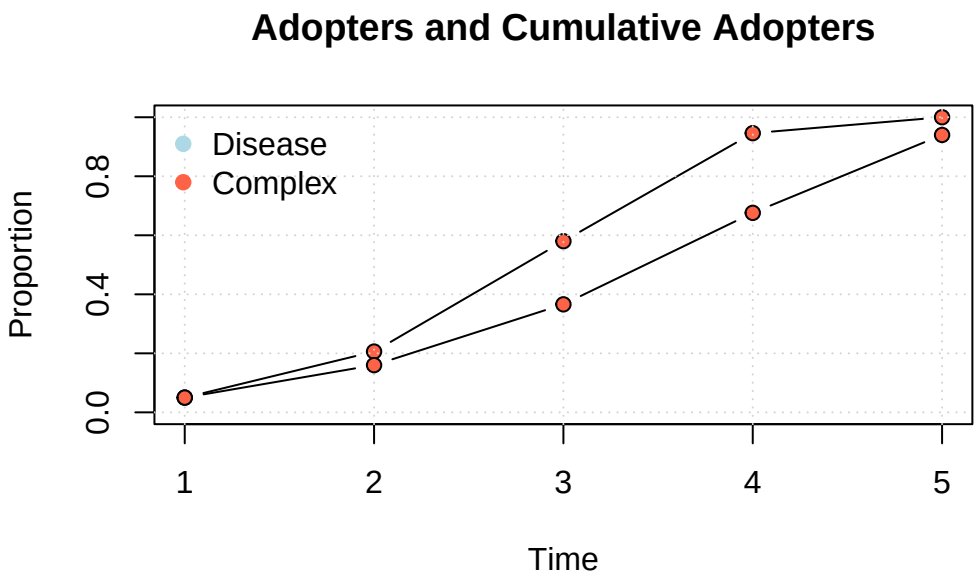

```
# We want to use igraph to compute layout
igdf <- diffnet_to_igraph(diffnet_rumor, slices=c(1,2))[[1]]
pos <- igraph::layout_with_drl(igdf)

plot_diffnet2(diffnet_rumor, vertex.size = dgr(diffnet_rumor)[,1], layout=pos)
```



7.3.3 Difussion

```
set.seed(09)
diffnet_complex <- rdiffnet(
  seed.graph = diffnet_rumor$graph,
  seed.nodes = which(diffnet_rumor$toa == 1),
  rewire = FALSE,
  threshold.dist = function(i) rbeta(1, 3, 10),
  name = "Diffusion",
  behavior = "Some social behavior"
)
```

```
plot_adopters(diffnet_rumor, what = "cumadopt", include.legend = FALSE)
plot_adopters(diffnet_complex, bg="tomato", add=TRUE, what = "cumadopt")
legend("topleft", legend = c("Disease", "Complex"), col = c("lightblue", "tomato"),
       bty = "n", pch=19)
```



7.3.4 Mentor Matching

```
# Finding mentors
mentors <- mentor_matching(diffnet_rumor, 25, lead.ties.method = "random")

# Simulating diffusion with these mentors
set.seed(09)
diffnet_mentored <- rdiffnet(
  seed.graph = diffnet_complex,
  seed.nodes = which(mentors$`1`$isleader),
  rewire = FALSE,
  threshold.dist = diffnet_complex[["real_threshold"]],
  name = "Diffusion using Mentors"
)

summary(diffnet_mentored)
```

```
Diffusion network summary statistics
Name     : Diffusion using Mentors
```

```
Behavior : Random contagion
-------------------------------------------------------------------------------
 Period   Adopters   Cum Adopt. (%)   Hazard Rate   Density   Moran's I (sd)
-------- ---------- ---------------- ------------ --------- ----------------
      1         25        25 (0.05)            -      0.01 -0.00 (0.00)
      2         92       117 (0.23)         0.19      0.01  0.01 (0.00) ***
      3        152       269 (0.54)         0.40      0.01  0.01 (0.00) ***
      4        150       419 (0.84)         0.65      0.01  0.01 (0.00) ***
      5         73       492 (0.98)         0.90      0.01 -0.00 (0.00) **
-------------------------------------------------------------------------------
 Left censoring  : 0.05 (25)
 Right centoring : 0.02 (8)
 # of nodes      : 500

 Moran's I was computed on contemporaneous autocorrelation using 1/geodesic
 values. Significane levels  *** <= .01, ** <= .05, * <= .1.
```

```
cumulative_adopt_count(diffnet_complex)
```

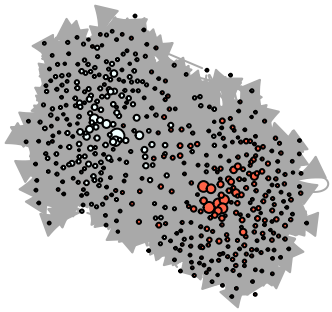
```
         1     2        3           4           5
num  25.00 80.00 183.0000 338.0000000 470.0000000
prop  0.05  0.16   0.3660   0.6760000   0.9400000
rate  0.00  2.20   1.2875   0.8469945   0.3905325
```

```
cumulative_adopt_count(diffnet_mentored)
```

```
         1       2          3           4           5
num  25.00 117.000 269.000000 419.0000000 492.0000000
prop  0.05   0.234   0.538000   0.8380000   0.9840000
rate  0.00   3.680   1.299145   0.5576208   0.1742243
```

7.3.5 Example by changing threshold

```r
# Simulating a scale-free homophilic network
set.seed(1231)
X <- rep(c(1,1,1,1,1,0,0,0,0,0), 50)
net <- rgraph_ba(t = 499, m=4, eta = X)

# Taking a look in igraph
ig  <- igraph::graph_from_adjacency_matrix(net)
plot(ig, vertex.color = c("azure", "tomato")[X+1], vertex.label = NA,
     vertex.size = sqrt(dgr(net)))
```



```r
# Now, simulating a bunch of diffusion processes
nsim <- 500L
ans_1and2 <- vector("list", nsim)
set.seed(223)
for (i in 1:nsim) {
  # We just want the cum adopt count
  ans_1and2[[i]] <-
    cumulative_adopt_count(
      rdiffnet(
```

```
        seed.graph = net,
        t = 10,
        threshold.dist = sample(1:2, 500L, TRUE),
        seed.nodes = "random",
        seed.p.adopt = .10,
        exposure.args = list(outgoing = FALSE, normalized = FALSE),
        rewire = FALSE
        )
      )

  # Are we there yet?
  if (!(i %% 50))
    message("Simulation ", i," of ", nsim, " done.")
}
## Simulation 50 of 500 done.
## Simulation 100 of 500 done.
## Simulation 150 of 500 done.
## Simulation 200 of 500 done.
## Simulation 250 of 500 done.
## Simulation 300 of 500 done.
## Simulation 350 of 500 done.
## Simulation 400 of 500 done.
## Simulation 450 of 500 done.
## Simulation 500 of 500 done.

# Extracting prop
ans_1and2 <- do.call(rbind, lapply(ans_1and2, "[", i="prop", j=))

ans_2and3 <- vector("list", nsim)
set.seed(223)
for (i in 1:nsim) {
  # We just want the cum adopt count
  ans_2and3[[i]] <-
    cumulative_adopt_count(
      rdiffnet(
        seed.graph = net,
```

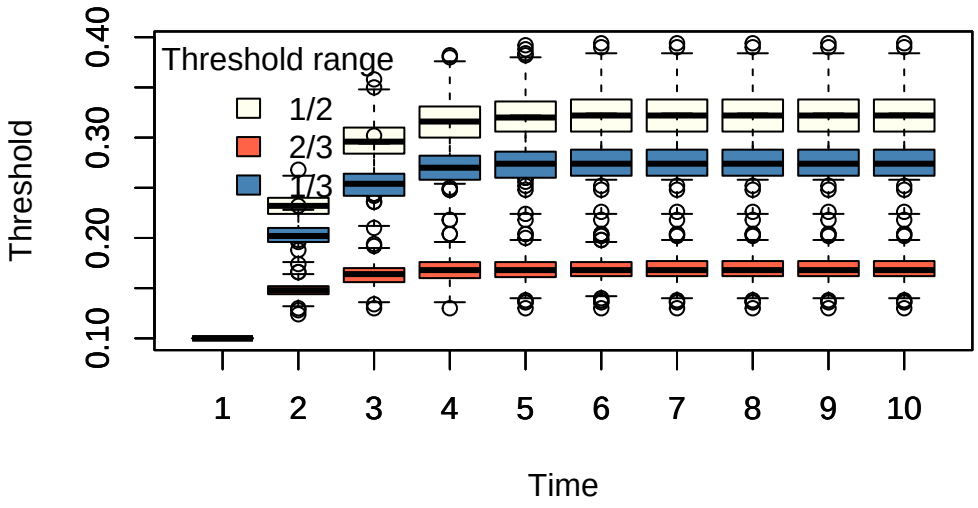
```
        t = 10,
        threshold.dist = sample(2:3, 500L, TRUE),
        seed.nodes = "random",
        seed.p.adopt = .10,
        exposure.args = list(outgoing = FALSE, normalized = FALSE),
        rewire = FALSE
        )
      )

  # Are we there yet?
  if (!(i %% 50))
    message("Simulation ", i," of ", nsim, " done.")
}
## Simulation 50 of 500 done.
## Simulation 100 of 500 done.
## Simulation 150 of 500 done.
## Simulation 200 of 500 done.
## Simulation 250 of 500 done.
## Simulation 300 of 500 done.
## Simulation 350 of 500 done.
## Simulation 400 of 500 done.
## Simulation 450 of 500 done.
## Simulation 500 of 500 done.

ans_2and3 <- do.call(rbind, lapply(ans_2and3, "[", i="prop", j=))
```

We can simplify by using the function `rdiffnet_multiple`. The following lines of code accomplish the same as the previous code avoiding the for-loop (from the user's perspective). Besides of the usual parameters passed to `rdiffnet`, the `rdiffnet_multiple` function requires `R` (number of repetitions/simulations), and `statistic` (a function that returns the statistic of interest). Optionally, the user may choose to specify the number of clusters to run it in parallel (multiple CPUs):

```
ans_1and3 <- rdiffnet_multiple(
  # Num of sim
  R              = nsim,
```

```
  # Statistic
  statistic      = function(d) cumulative_adopt_count(d)["prop",],
  seed.graph     = net,
  t              = 10,
  threshold.dist = sample(1:3, 500, TRUE),
  seed.nodes     = "random",
  seed.p.adopt   = .1,
  rewire         = FALSE,
  exposure.args  = list(outgoing=FALSE, normalized=FALSE),
  # Running on 4 cores
  ncpus          = 4L
  )
```

```
boxplot(ans_1and2, col="ivory", xlab = "Time", ylab = "Threshold")
boxplot(ans_2and3, col="tomato", add=TRUE)
boxplot(t(ans_1and3), col = "steelblue", add=TRUE)
legend(
  "topleft",
  fill = c("ivory", "tomato", "steelblue"),
  legend = c("1/2", "2/3", "1/3"),
  title = "Threshold range",
  bty ="n"
)
```



7.3.6 Problems

1. Given the following types of networks: Small-world, Scale-free, Bernoulli, what set of n initiators maximizes diffusion?

7.4 Statistical inference

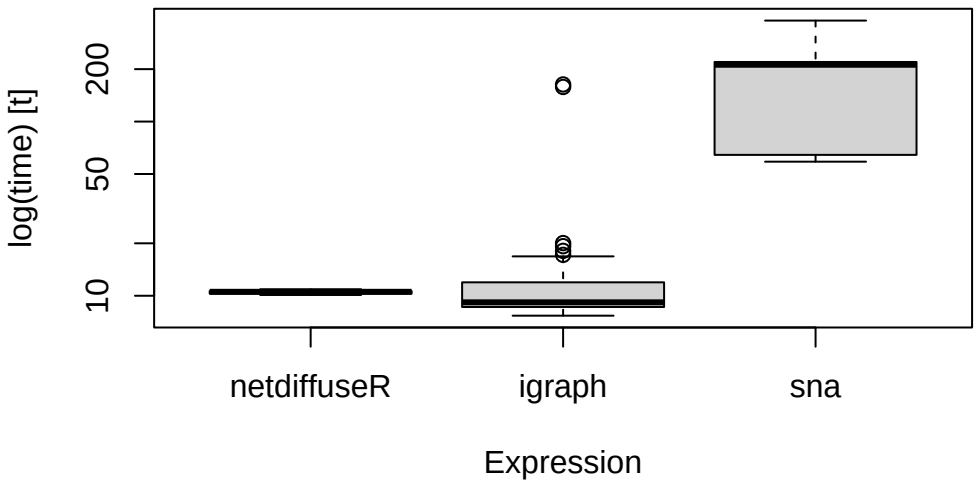
7.4.1 Moran's I

- Moran's I tests for spatial autocorrelation.
- **netdiffuseR** implements the test in `moran`, which is suited for sparse matrices.
- We can use Moran's I as a first look to whether there is something happening: let that be influence or homophily.

7.4.2 Using geodesics

- One approach is to use the geodesic (shortest path length) matrix to account for indirect influence.
- **netdiffuseR** has a function to do so, the `approx_geodesic` function, which, using graph powers, computes the shortest path up to `n` steps. This could be faster (if you only care up to `n` steps) than `igraph` or `sns`:

```
# Extracting the large adjacency matrix (stacked)
dgc <- diag_expand(medInnovationsDiffNet$graph)
ig  <- igraph::graph_from_adjacency_matrix(dgc)
mat <- network::as.network(as.matrix(dgc))

# Measuring times
times <- microbenchmark::microbenchmark(
  netdiffuseR = netdiffuseR::approx_geodesic(dgc),
  igraph = igraph::distances(ig),
  sna = sna::geodist(mat),
  times = 50, unit="ms"
)
```



- The `summary.diffnet` method already runs Moran's for you. What happens under the hood is:

```
# For each time point we compute the geodesic distances matrix
W <- approx_geodesic(medInnovationsDiffNet$graph[[1]])

# We get the element-wise inverse
W@x <- 1/W@x

# And then compute moran
moran(medInnovationsDiffNet$cumadopt[,1], W)
```

```
$observed
[1] 0.06624028

$expected
[1] -0.008064516

$sd
[1] 0.03265066

$p.value
[1] 0.02286087

attr(,"class")
[1] "diffnet_moran"
```

7.4.3 Structural dependence and permutation tests

- A novel statistical method (work-in-progress) for testing network influence effects.
- Included in the package, tests whether a particular network statistic depends on network structure
- Suitable to be applied to network thresholds (you can't use thresholds in regression-like models!)

7.4.4 Idea

- Let $\mathcal{G} = (V, E)$ be a graph, γ a vertex attribute, and $\beta = f(\gamma, \mathcal{G})$, then

$$\gamma \perp \mathcal{G} \implies \mathbb{E}\left[\beta(\gamma, \mathcal{G})|\mathcal{G}\right] = \mathbb{E}\left[\beta(\gamma, \mathcal{G})\right]$$

- For example, if time of adoption is independent of the structure of the network, then the average threshold level will be independent from the network structure as well.
- Another way of looking at this is that the test will allow us to see how probable it is to have this combination of network structure and network threshold (if it is uncommon, then we say that the diffusion model is highly likely)

7.4.4.1 Example Not random TOA

- To use this test, **netdiffuseR** has the `struct_test` function.
- It simulates networks with the same density, and computes a particular statistic every time, generating an EDF (Empirical Distribution Function) under the Null hypothesis (p-values).

```
# Simulating network
set.seed(1123)
net <- rdiffnet(n=500, t=10, seed.graph = "small-world")
```

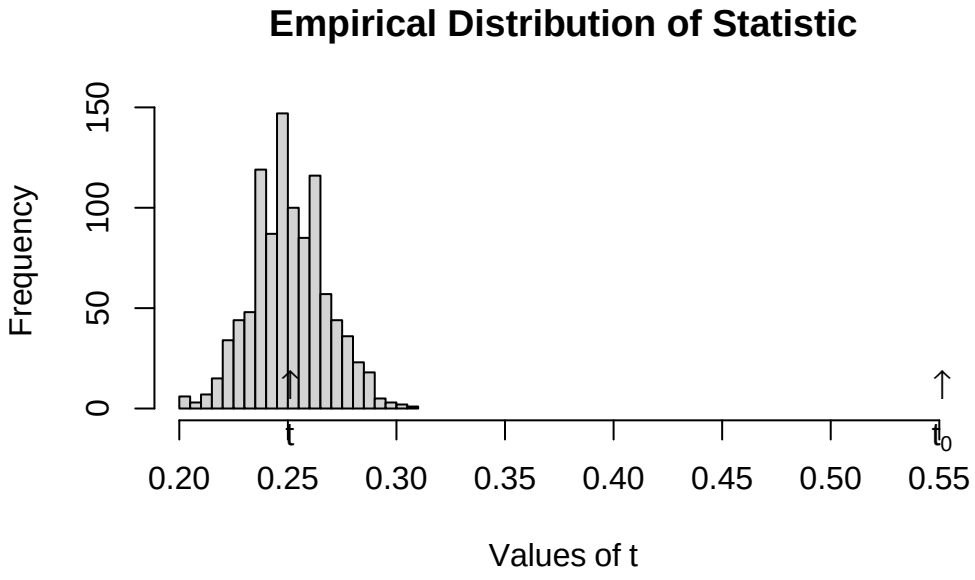
```
Warning in (function (graph, p, algorithm = "endpoints", both.ends = FALSE, :
The option -copy.first- is set to TRUE. In this case, the first graph will be
treated as a baseline, and thus, networks after T=1 will be replaced with T-1.
```

```
# Running the test
test <- struct_test(
  graph     = net,
  statistic = function(x) mean(threshold(x), na.rm = TRUE),
  R         = 1e3,
  ncpus=4, parallel="multicore"
  )
```

```
Warning in (function (graph, p, algorithm = "endpoints", both.ends = FALSE, :
The option -copy.first- is set to TRUE. In this case, the first graph will be
treated as a baseline, and thus, networks after T=1 will be replaced with T-1.
```

```r
# See the output
test
```

```
Structure dependence test
# Simulations     : 1,000
# nodes           : 500
# of time periods : 10
--------------------------------------------------------------------------------
 H0: E[beta(Y,G)|G] - E[beta(Y,G)] = 0 (no structure dependency)
    observed    expected       p.val
      0.5513      0.2510      0.0000
```



- Now we shuffle times of adoption, so that is random

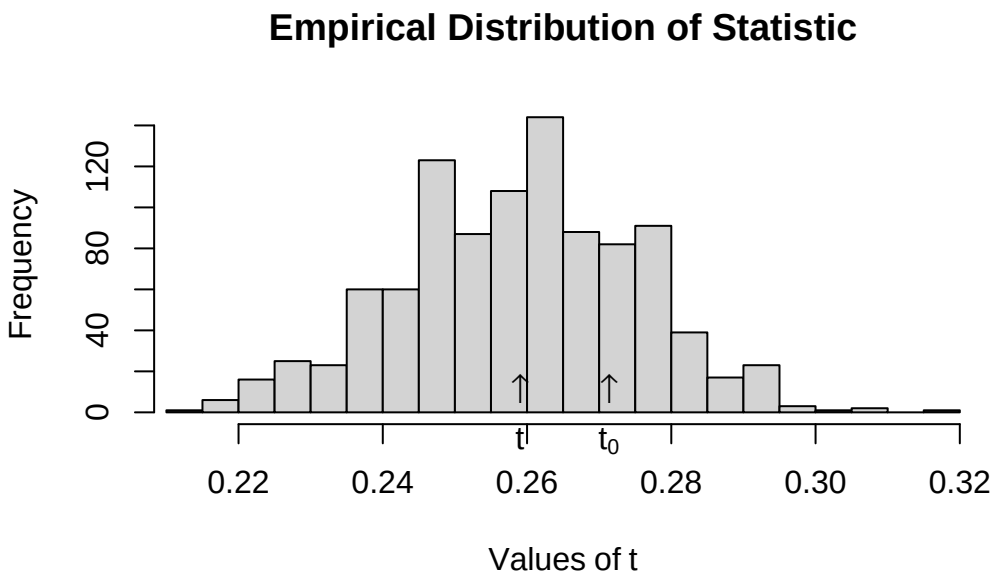
```r
# Resetting TOAs (now will be completely random)
diffnet.toa(net) <- sample(diffnet.toa(net), nnodes(net), TRUE)

# Running the test
```

```r
test <- struct_test(
  graph     = net,
  statistic = function(x) mean(threshold(x), na.rm = TRUE),
  R         = 1e3,
  ncpus=4, parallel="multicore"
  )
```

```
Warning in (function (graph, p, algorithm = "endpoints", both.ends = FALSE, :
The option -copy.first- is set to TRUE. In this case, the first graph will be
treated as a baseline, and thus, networks after T=1 will be replaced with T-1.
```

```r
# See the output
test
```

```
Structure dependence test
# Simulations     : 1,000
# nodes           : 500
# of time periods : 10
--------------------------------------------------------------------------------
 H0: E[beta(Y,G)|G] - E[beta(Y,G)] = 0 (no structure dependency)
    observed    expected       p.val
      0.2714      0.2590      0.4380
```



7.4.5 Regression analysis

- In regression analysis, we want to see if exposure, once we control for other covariates, had any effect on adopting a behavior.
- The big problem is when we have a latent variable that co-determines both network and behavior.
- Regression analysis will be generically biased Unless we can control for that variable.
- On the other hand, if you can claim that either such variable doesn't exist or you actually can control for it, then we have two options: lagged exposure models or contemporaneous exposure models. We will focus on the former.

7.4.5.1 Lagged exposure models

- In this type of model, we usually have the following

$$y_t = f(W_{t-1}, y_{t-1}, X_i) + \varepsilon$$

Furthermore, in the case of adoption, we have

$$y_{it} = \begin{cases} 1 & \text{if} \quad \rho \sum_{j \neq i} \frac{W_{ijt-1} y_{it-1}}{\sum_{j \neq i} W_{ijt-1}} + X_{it}\beta > 0 \\ 0 & \text{otherwise} \end{cases}$$

- In netdiffuseR, it is as easy as doing the following:

```
# fakedata
set.seed(121)

W   <- rgraph_ws(1e3, 8, .2)
X   <- cbind(var1 = rnorm(1e3))
toa <- sample(c(NA,1:5), 1e3, TRUE)

dn  <- new_diffnet(W, toa=toa, vertex.static.attrs = X)
```

```
Warning in new_diffnet(W, toa = toa, vertex.static.attrs = X): -graph- is
static and will be recycled (see ?new_diffnet).
```

```
# Computing exposure and adoption for regression
dn[["cohesive_expo"]] <- cbind(NA, exposure(dn)[,-nslices(dn)])
dn[["adopt"]]         <- dn$cumadopt

# Generating the data and running the model
dat <- as.data.frame(dn)
ans <- glm(adopt ~ cohesive_expo + var1 + factor(per),
           data = dat,
           family = binomial(link="probit"),
           subset = is.na(toa) | (per <= toa))
summary(ans)
```

```
Call:
glm(formula = adopt ~ cohesive_expo + var1 + factor(per), family = binomial(link =
    data = dat, subset = is.na(toa) | (per <= toa))

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)   -0.92777    0.05840 -15.888  < 2e-16 ***
```

```
cohesive_expo  0.23839    0.17514   1.361 0.173452
var1          -0.04623    0.02704  -1.710 0.087334 .
factor(per)3   0.29313    0.07715   3.799 0.000145 ***
factor(per)4   0.33902    0.09897   3.425 0.000614 ***
factor(per)5   0.59851    0.12193   4.909 9.18e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2745.1  on 2317  degrees of freedom
Residual deviance: 2663.5  on 2312  degrees of freedom
  (1000 observations deleted due to missingness)
AIC: 2675.5

Number of Fisher Scoring iterations: 4
```

Alternatively, we could have used the new function `diffreg`

```
ans <- diffreg(dn ~ exposure + var1 + factor(per), type = "probit")
summary(ans)
```

```
Call:
glm(formula = Adopt ~ exposure + var1 + factor(per), family = binomial(link = "prob
    data = dat, subset = ifelse(is.na(toa), TRUE, toa >= per))

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  -0.92777    0.05840 -15.888  < 2e-16 ***
exposure      0.23839    0.17514   1.361 0.173452
var1         -0.04623    0.02704  -1.710 0.087334 .
factor(per)3  0.29313    0.07715   3.799 0.000145 ***
factor(per)4  0.33902    0.09897   3.425 0.000614 ***
factor(per)5  0.59851    0.12193   4.909 9.18e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2745.1  on 2317  degrees of freedom
Residual deviance: 2663.5  on 2312  degrees of freedom
  (1000 observations deleted due to missingness)
AIC: 2675.5

Number of Fisher Scoring iterations: 4
```

7.4.5.2 Contemporaneous exposure models

- Similar to the lagged exposure models, we usually have the following

$$y_t = f(W_t, y_t, X_t) + \varepsilon$$

Furthermore, in the case of adoption, we have

$$y_{it} = \begin{cases} 1 & \text{if} \quad \rho \sum_{j \neq i} \frac{W_{ijt} y_{it}}{\sum_{j \neq i} W_{ijt}} + X_{it}\beta > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Unfortunately, since y_t is in both sides of the equation, these models cannot be fitted using a standard probit or logit regression.
- Two alternatives to solve this:
  a. Using Instrumental Variables Probit (ivprobit in both R and Stata)
  b. Use a Spatial Autoregressive (SAR) Probit (SpatialProbit and ProbitSpatial in R).
- We won't cover these here.

7.4.6 Problems

Using the dataset stats.rda:

1. Compute Moran's I as the function `summary.diffnet` does. To do so, you'll need to use the function `toa_mat` (which calculates the cumulative adoption matrix), and `approx_geodesic` (which computes the geodesic matrix). (see `?summary.diffnet` for more details).

2. Read the data as diffnet object, and fit the following logit model $adopt = Exposure * \gamma + Measure * \beta + \varepsilon$. What happens if you exclude the time-fixed effects?

Part II

Statistical inference

8 Exponential Random Graph Models

I strongly suggest reading the vignette in the `ergm` R package.

```
vignette("ergm", package="ergm")
```

> The purpose of ERGMs, in a nutshell, is to describe parsimoniously the local selection forces that shape the global structure of a network. To this end, a network dataset, like those depicted in Figure 1, may be considered as the response in a regression model, where the predictors are things like "propensity for individuals of the same sex to form partnerships" or "propensity for individuals to form triangles of partnerships". In Figure 1(b), for example, it is evident that the individual nodes appear to cluster in groups of the same numerical labels (which turn out to be students' grades, 7 through 12); thus, an ERGM can help us quantify the strength of this intra-group effect.
>
> — (David R. Hunter et al. 2008)

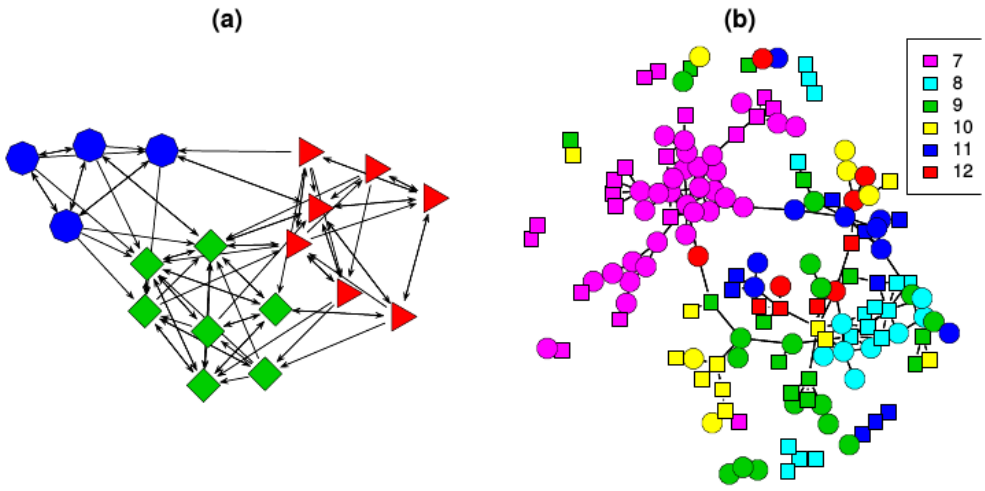


Figure 1: The **(a)** `samplike` and **(b)** `faux.mesa.high` networks described in Section 2. The values of nodal covariates may be indicated using various colors, shapes, and labels of nodes.

Figure 8.1: Source: Hunter et al. (2008)

In a nutshell, we use ERGMs as a parametric interpretation of the distribution of $\mathbf{Y}$, which takes the canonical form:

$$\mathbb{P}\left(\mathbf{Y}=\mathbf{y}|\theta,\mathcal{Y}\right)=\frac{\exp\left\{\theta^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\}}{\kappa\left(\theta,\mathcal{Y}\right)},\quad \mathbf{y}\in\mathcal{Y} \tag{8.1}$$

Where $\theta\in\Omega\subset\mathbb{R}^q$ is the vector of model coefficients and $\mathbf{g}(\mathbf{y})$ is a q-vector of statistics based on the adjacency matrix $\mathbf{y}$.

Model Equation 8.1 may be expanded by replacing $\mathbf{g}(\mathbf{y})$ with $\mathbf{g}(\mathbf{y},\mathbf{X})$ to allow for additional covariate information $\mathbf{X}$ about the network. The denominator $\kappa\left(\theta,\mathcal{Y}\right)=\sum_{\mathbf{y}\in\mathcal{Y}}\exp\left\{\theta^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\}$ is the normalizing factor that ensures that equation Equation 8.1 is a legitimate probability distribution. Even after fixing $\mathcal{Y}$ to be all the networks that have size n, the size of $\mathcal{Y}$ makes this type of statistical model hard to estimate as there are $N=2^{n(n-1)}$ possible networks! (David R. Hunter et al. 2008)

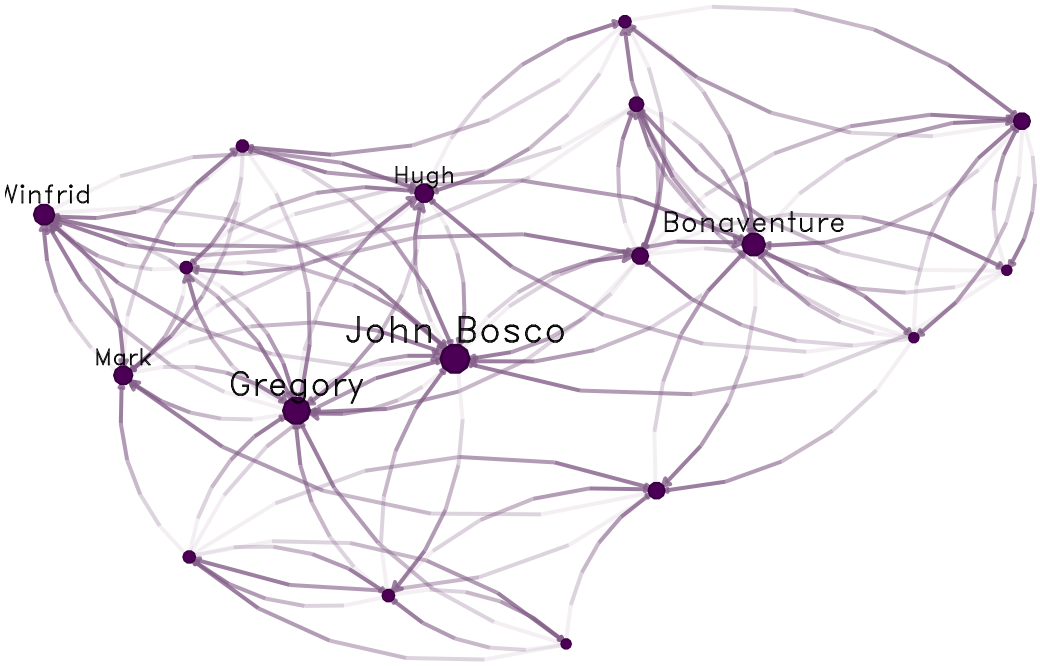
Later developments include new dependency structures to consider more general neighborhood effects. These models relax the one-step Markovian dependence assumptions, allowing investigation of longer-range configurations, such as longer paths in the network or larger cycles (Pattison and Robins 2002). Models for bipartite (Faust and Skvoretz 1999) and tripartite (Mische and Robins 2000) network structures have been developed. (David R. Hunter et al. 2008, 9)

8.1 A naïve example

In the simplest case, ERGMs equate a logistic regression. By simple, I mean cases with no Markovian terms–motifs involving more than one edge–for example, the Bernoulli graph. In the Bernoulli graph, ties are independent, so the presence/absence of a tie between nodes i and j won't affect the presence/absence of a tie between nodes k and l.

Let's fit an ERGM using the `sampson` dataset in the `ergm` package.

```
library(ergm)
library(netplot)
data("sampson")
nplot(samplike)
```



Using `ergm` to fit a Bernoulli graph requires using the `edges` term, which counts how many ties are in the graph:

```
ergm_fit <- ergm(samplike ~ edges)
## Starting maximum pseudolikelihood estimation (MPLE):
## Obtaining the responsible dyads.
## Evaluating the predictor and response matrix.
## Maximizing the pseudolikelihood.
```

```
## Finished MPLE.
## Evaluating log-likelihood at the estimate.
```

Since this is equivalent to a logistic regression, we can use the `glm` function to fit the same model. First, we need to prepare the data so we can pass it to `glm`:

```
y <- sort(as.vector(as.matrix(samplike)))
y <- y[-c(1:18)] # We remove the diagonal from the model, which is all 0.
y
##   [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 0 0 0 0 0 0 0 0 0 0
##  [38] 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 0 0 0 0 0 0 0 0 0
##  [75] 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 0 0 0 0 0 0 0 0 0
## [112] 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 0 0 0 0 0 0 0 0 0
## [149] 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 0 0 0 0 0 0 0 0 0
## [186] 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 0 0 0 0 0 1 1 1 1
## [223] 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 1 1 1 1 1 1 1 1 1 1 1
## [260] 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 1 1 1 1 1 1 1 1 1 1 1
## [297] 1 1 1 1 1 1 1 1 1 1
```

We can now fit the GLM model:

```
glm_fit <- glm(y~1, family=binomial("logit"))
```

The coefficients of both ERGM and GLM should match:

```
glm_fit
##
## Call:  glm(formula = y ~ 1, family = binomial("logit"))
##
## Coefficients:
## (Intercept)
##     -0.9072
##
## Degrees of Freedom: 305 Total (i.e. Null);  305 Residual
## Null Deviance:       367.2
```

```
## Residual Deviance: 367.2     AIC: 369.2
ergm_fit
##
## Call:
## ergm(formula = samplike ~ edges)
##
## Maximum Likelihood Coefficients:
##   edges
## -0.9072
```

Furthermore, in the case of the Bernoulli graph, we can get the estimate using the Logit function:

```
pr <- mean(y)
# Logit function:
# Alternatively we could have used log(pr) - log(1-pr)
qlogis(pr)
## [1] -0.9071582
```

Again, the same result. The Bernoulli graph is not the only ERGM model that can be fitted using a Logistic regression. Moreover, if all the terms of the model are non-Markov terms, `ergm` automatically defaults to a Logistic regression.

8.2 Estimation of ERGMs

The ultimate goal is to perform statistical inference on the proposed model. In a *standard* setting, we could use Maximum Likelihood Estimation (MLE), which consists of finding the model parameters θ that, given the observed data, maximize the likelihood of the model. For the latter, we generally use Newton's method. Newton's method requires computing the model's log-likelihood, which can be challenging in ERGMs.

For ERGMs, since part of the likelihood involves a normalizing constant that is a function of all possible networks, this is not as straightforward as in the regular setting. Because of this, most estimation methods rely on simulations.

In `statnet`, the default estimation method is based on a method proposed by (Geyer and Thompson 1992), Markov-Chain MLE, which uses Markov-Chain Monte Carlo for simulating networks and a modified version of the Newton-Raphson algorithm to estimate the parameters.

The idea of MC-MLE for this family of statistical models is to approximate the expectation of normalizing constant ratios using the law of large numbers. In particular, the following:

$$\begin{aligned}
\frac{\kappa(\theta, \mathcal{Y})}{\kappa(\theta_0, \mathcal{Y})} &= \frac{\sum_{\mathbf{y}\in\mathcal{Y}} \exp\left\{\theta^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\}}{\sum_{\mathbf{y}\in\mathcal{Y}} \exp\left\{\theta_0^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\}} \\
&= \sum_{\mathbf{y}\in\mathcal{Y}} \left(\frac{1}{\sum_{\mathbf{y}\in\mathcal{Y}} \exp\{\theta_0^{\mathrm{T}}\mathbf{g}(\mathbf{y})\}} \times \exp\left\{\theta^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\} \right) \\
&= \sum_{\mathbf{y}\in\mathcal{Y}} \left(\frac{\exp\left\{\theta_0^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\}}{\sum_{\mathbf{y}\in\mathcal{Y}} \exp\{\theta_0^{\mathrm{T}}\mathbf{g}(\mathbf{y})\}} \times \exp\left\{(\theta-\theta_0)^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\} \right) \\
&= \sum_{\mathbf{y}\in\mathcal{Y}} \left(\mathbb{P}\left(Y = y|\mathcal{Y}, \theta_0\right) \times \exp\left\{(\theta-\theta_0)^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\} \right) \\
&= \mathrm{E}_{\theta_0}\left(\exp\left\{(\theta-\theta_0)^{\mathrm{T}}\mathbf{g}(\mathbf{y})\right\} \right)
\end{aligned}$$

In particular, the MC-MLE algorithm uses this fact to maximize the log-likelihood ratio. The objective function itself can be approximated by simulating m networks from the distribution with parameter θ_0:

$$l(\theta) - l(\theta_0) \approx (\theta-\theta_0)^{\mathrm{T}}\mathbf{g}(\mathbf{y}_{obs}) - \log\left[\frac{1}{m} \sum_{i=1}^{m} \exp\left\{(\theta-\theta_0)^{\mathrm{T}}\right\} \mathbf{g}(\mathbf{Y}_i) \right]$$

For more details, see (David R. Hunter et al. 2008). A sketch of the algorithm follows:

1. Initialize the algorithm with an initial guess of θ, call it $\theta^{(t)}$ (must be a rather OK guess)

2. While (no convergence) do:

a. Using $\theta^{(t)}$, simulate M networks by means of small changes in the $\mathbf{Y}_{obs}$ (the observed network). This part is done by using an importance-sampling method which weights each proposed network by its likelihood conditional on $\theta^{(t)}$

b. With the networks simulated, we can do the Newton step to update the parameter $\theta^{(t)}$ (this is the iteration part in the `ergm` package): $\theta^{(t)} \rightarrow \theta^{(t+1)}$.

c. If convergence has been reached (which usually means that $\theta^{(t)}$ and $\theta^{(t+1)}$ are not very different), then stop; otherwise, go to step a.

(Lusher, Koskinen, and Robins 2012; Admiraal and Handcock 2006; T. A. Snijders 2002; Wang et al. 2009) provides details on the algorithm used by PNet (the same as the one used in `RSiena`), and (Lusher, Koskinen, and Robins 2012) provides a short discussion on the differences between `ergm` and `PNet`.

8.3 The `ergm` package

The `ergm` R package (Handcock et al. 2018)

From the previous section:[1]

```
library(igraph)

library(dplyr)

load("03.rda")
```

In this section, we will use the `ergm` package (from the `statnet` suit of packages (Handcock et al. 2018),) and the `intergraph` (Bojanowski 2023) package. The latter provides functions to go back and forth between `igraph` and `network` objects from the `igraph` and `network` packages respectively[2]

```
library(ergm)
library(intergraph)
```

As a rather important side note, the order in which R packages are loaded matters. Why is this important to mention now? Well, it turns out that at least a couple of functions in the `network` package have the same name as some functions in the `igraph` package. When the

[1] You can download the 03.rda file from this link.
[2] Yes, the classes have the same name as the packages.

`ergm` package is loaded, since it depends on `network`, it will load the `network` package first, which will *mask* some functions in `igraph`. This becomes evident once you load `ergm` after loading `igraph`:

```
The following objects are masked from 'package:igraph':

  add.edges, add.vertices, %c%, delete.edges, delete.vertices, get.edge.attribute, get.e
  get.vertex.attribute, is.bipartite, is.directed, list.edge.attributes, list.vertex.att
  set.edge.attribute, set.vertex.attribute
```

What are the implications of this? If you call the function `list.edge.attributes` for an object of class `igraph` R will return an error as the first function that matches that name comes from the `network` package! To avoid this you can use the double colon notation:

```
igraph::list.edge.attributes(my_igraph_object)
network::list.edge.attributes(my_network_object)
```

Anyway... Using the `asNetwork` function, we can coerce the `igraph` object into a network object so we can use it with the `ergm` function:

```
# Creating the new network
network_111 <- intergraph::asNetwork(ig_year1_111)

# Running a simple ergm (only fitting edge count)
ergm(network_111 ~ edges)
## [1] "Warning:  This network contains loops"
## Starting maximum pseudolikelihood estimation (MPLE):
## Obtaining the responsible dyads.
## Evaluating the predictor and response matrix.
## Maximizing the pseudolikelihood.
## Finished MPLE.
## Evaluating log-likelihood at the estimate.
##
## Call:
## ergm(formula = network_111 ~ edges)
##
```

```
## Maximum Likelihood Coefficients:
##  edges
## -4.734
```

So what happened here? We got a warning. It turns out that our network has loops (didn't think about it before!). Let's take a look at that with the `which_loop` function

```
E(ig_year1_111)[which_loop(ig_year1_111)]
## + 1/2638 edge from 58d21c9 (vertex names):
## [1] 1110111->1110111
```

We can get rid of these using the `igraph::-.igraph`. Let's remove the isolates using the same operator

```
# Creating the new network
network_111 <- ig_year1_111

# Removing loops
network_111 <- network_111 - E(network_111)[which(which_loop(network_111))]

# Removing isolates
network_111 <- network_111 - which(degree(network_111, mode = "all") == 0)

# Converting the network
network_111 <- intergraph::asNetwork(network_111)
```

`asNetwork(simplify(ig_year1_111))` `ig_year1_111 |> simplify() |> asNetwork()`

A problem that we have with this data is the fact that some vertices have missing values in the variables `hispanic`, `female1`, and `eversmk1`. For now, we will proceed by imputing values based on the averages:

```
for (v in c("hispanic", "female1", "eversmk1")) {
  tmpv <- network_111 %v% v
  tmpv[is.na(tmpv)] <- mean(tmpv, na.rm = TRUE) > .5
  network_111 %v% v <- tmpv
}
```

Let's take a look at the network

```
nplot(
  network_111,
  vertex.color = ~ hispanic
  )
```

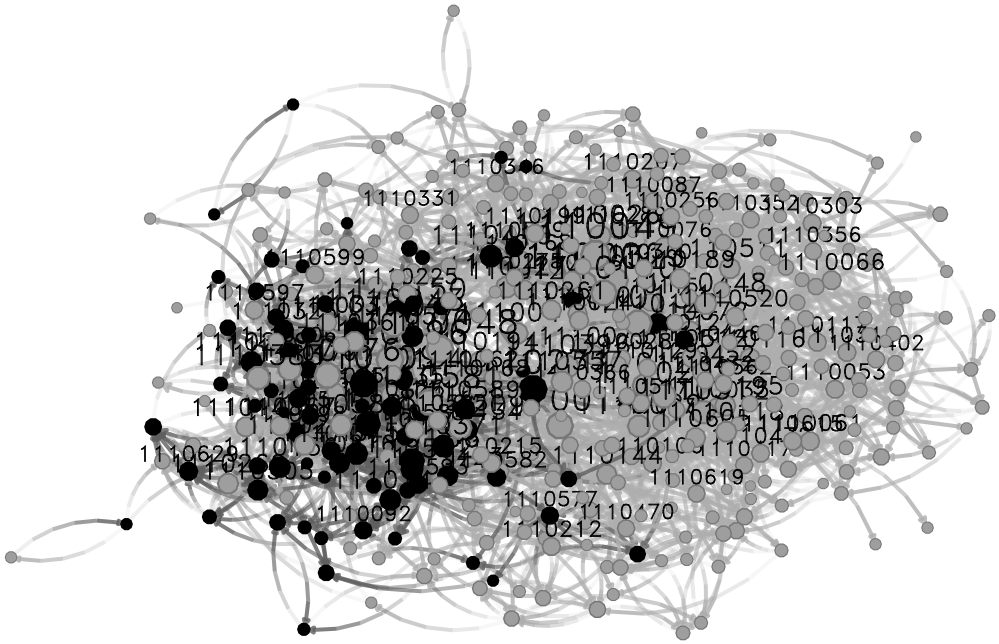


Figure 8.2

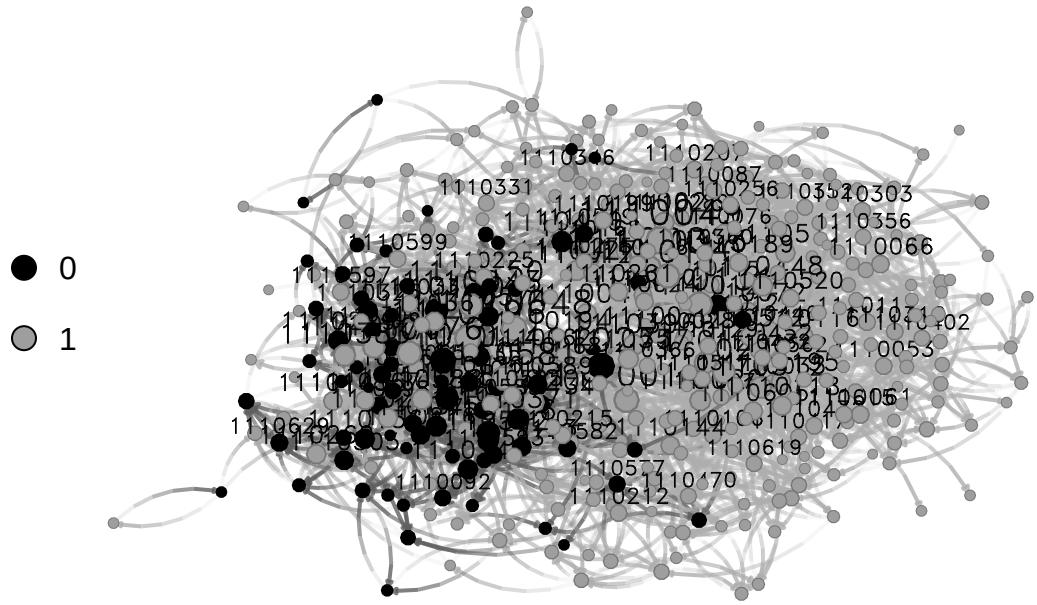


Figure 8.3

8.4 Running ERGMs

Proposed workflow:

1. Estimate the simplest model, adding one variable at a time.
2. After each estimation, run the `mcmc.diagnostics` function to see how good (or bad) behaved the chains are.
3. Run the `gof` function and verify how good the model matches the network's structural statistics.

What to use:

1. `control.ergms`: Maximum number of iterations, seed for Pseudo-RNG, how many cores
2. `ergm.constraints`: Where to sample the network from. Gives stability and (in some cases) faster convergence as by constraining the model you are reducing the sample size.

Here is an example of a couple of models that we could compare[3]

```
ans0 <- ergm(
  network_111 ~
    edges +
    nodematch("hispanic") +
    nodematch("female1") +
    nodematch("eversmk1") +
    mutual,
  constraints = ~bd(maxout = 19),
  control = control.ergm(
    seed        = 1,
    MCMLE.maxit = 10,
    parallel    = 4,
    CD.maxit    = 10
    )
  )
## Warning: 'glpk' selected as the solver, but package 'Rglpk' is not available;
## falling back to 'lpSolveAPI'. This should be fine unless the sample size and/or
## the number of parameters is very big.
## Warning in nobs.ergm(object, ...): The number of observed dyads in this network
## is ill-defined due to complex constraints on the sample space. Disable this
## warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
## Warning in nobs.ergm(object, ..., verbose = verbose): The number of observed
## dyads in this network is ill-defined due to complex constraints on the sample
## space. Disable this warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
```

So what are we doing here:

1. The model is controlling for:

    a. `edges` Number of edges in the network (as opposed to its density)

    b. `nodematch("some-variable-name-here")` Includes a term that controls for homophily/heterophily

[3]Notice that this document may not include the usual messages that the `ergm` command generates during the estimation procedure. This is just to make it more printable-friendly.

c. `mutual` Number of mutual connections between $(i, j), (j, i)$. This can be related to, for example, triadic closure.

For more on control parameters, see (Morris, Handcock, and Hunter 2008).

```
ans1 <- ergm(
  network_111 ~
    edges +
    nodematch("hispanic") +
    nodematch("female1") +
    nodematch("eversmk1")
    ,
  constraints = ~bd(maxout = 19),
  control = control.ergm(
    seed        = 1,
    MCMLE.maxit = 10,
    parallel    = 4,
    CD.maxit    = 10
    )
  )
```

This example takes longer to compute

```
ans2 <- ergm(
  network_111 ~
    edges +
    nodematch("hispanic") +
    nodematch("female1") +
    nodematch("eversmk1") +
    mutual +
    balance
    ,
  constraints = ~bd(maxout = 19),
  control = control.ergm(
    seed        = 1,
    MCMLE.maxit = 10,
    parallel    = 4,
```

```
    CD.maxit    = 10
    )
  )
```

Now, a nice trick to see all regressions in the same table, we can use the `texreg` package (Leifeld 2013) which supports `ergm` ouputs!

```
library(texreg)
## Version:  1.39.3
## Date:     2023-11-09
## Author:   Philip Leifeld (University of Essex)
##
## Consider submitting praise using the praise or praise_interactive functions.
## Please cite the JSS article in your publications -- see citation("texreg").
screenreg(list(ans0, ans1, ans2))
## Warning in nobs.ergm(object): The number of observed dyads in this network is
## ill-defined due to complex constraints on the sample space. Disable this
## warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
## Warning: This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing
## it with version 4.2 or later may return incorrect results or fail.
## Warning in nobs.ergm(object): The number of observed dyads in this network is
## ill-defined due to complex constraints on the sample space. Disable this
## warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
## Warning: This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing
## it with version 4.2 or later may return incorrect results or fail.
## Warning in nobs.ergm(object): The number of observed dyads in this network is
## ill-defined due to complex constraints on the sample space. Disable this
## warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
##
## ============================================================
##                      Model 1        Model 2        Model 3
## ------------------------------------------------------------
## edges                -5.63 ***      -5.49 ***      -5.60 ***
##                      (0.06)         (0.06)         (0.06)
## nodematch.hispanic    0.22 ***       0.30 ***       0.22 ***
##                      (0.04)         (0.05)         (0.04)
```

```
## nodematch.female1         0.87 ***       1.17 ***       0.87 ***
##                          (0.04)         (0.05)         (0.04)
## nodematch.eversmk1        0.33 ***       0.45 ***       0.34 ***
##                          (0.04)         (0.04)         (0.04)
## mutual                    4.12 ***                      1.75 ***
##                          (0.07)                        (0.14)
## balance                                                 0.01 ***
##                                                        (0.00)
## ------------------------------------------------------------------
## AIC                  -40020.94      -37511.87      -39989.59
## BIC                  -39970.60      -37471.60      -39929.18
## Log Likelihood        20015.47       18759.94       20000.79
## ==================================================================
## *** p < 0.001; ** p < 0.01; * p < 0.05
```

Or, if you are using rmarkdown, you can export the results using LaTeX or html, let's try the latter to see how it looks like here:

```
library(texreg)
texreg(list(ans0, ans1, ans2))
## Warning in nobs.ergm(object): The number of observed dyads in this network is
## ill-defined due to complex constraints on the sample space. Disable this
## warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
## Warning: This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing
## it with version 4.2 or later may return incorrect results or fail.
## Warning in nobs.ergm(object): The number of observed dyads in this network is
## ill-defined due to complex constraints on the sample space. Disable this
## warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
## Warning: This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing
## it with version 4.2 or later may return incorrect results or fail.
## Warning in nobs.ergm(object): The number of observed dyads in this network is
## ill-defined due to complex constraints on the sample space. Disable this
## warning with 'options(ergm.loglik.warn_dyads=FALSE)'.
```

	Model 1	Model 2	Model 3
edges	−5.63***	−5.49***	−5.60***
	(0.06)	(0.06)	(0.06)
nodematch.hispanic	0.22***	0.30***	0.22***
	(0.04)	(0.05)	(0.04)
nodematch.female1	0.87***	1.17***	0.87***
	(0.04)	(0.05)	(0.04)
nodematch.eversmk1	0.33***	0.45***	0.34***
	(0.04)	(0.04)	(0.04)
mutual	4.12***		1.75***
	(0.07)		(0.14)
balance			0.01***
			(0.00)
AIC	−40020.94	−37511.87	−39989.59
BIC	−39970.60	−37471.60	−39929.18
Log Likelihood	20015.47	18759.94	20000.79

***$p < 0.001$; **$p < 0.01$; *$p < 0.05$

Table 8.1: Statistical models

8.5 Model Goodness-of-Fit

In raw terms, once each chain has reached stationary distribution, we can say that there are no problems with autocorrelation and that each sample point is iid. The latter implies that, since we are running the model with more than one chain, we can use all the samples (chains) as a single dataset.

> Recent changes in the ergm estimation algorithm mean that these plots can no longer be used to ensure that the mean statistics from the model match the observed network statistics. For that functionality, please use the GOF command: `gof(object, GOF=~model)`.
>
> —?ergm::mcmc.diagnostics

Since `ans0` is the best model, let's look at the GOF statistics. First, let's see how the MCMC did. We can use the `mcmc.diagnostics` function included in the package. The function is a wrapper of a couple of functions from the `coda` package (Plummer et al. 2006), which are called upon the `$sample` object holding the *centered* statistics from the sampled networks. At first, it can be confusing to look at the `$sample` object; it neither matches the observed statistics nor the coefficients.

When calling `mcmc.diagnostics(ans0, centered = FALSE)`, you will see many outputs, including a couple of plots showing the trace and posterior distribution of the *uncentered* statistics (`centered = FALSE`). The following code chunks will reproduce the output from the `mcmc.diagnostics` function step by step using the coda package. First, we need to *uncenter* the sample object:

```
# Getting the centered sample
sample_centered <- ans0$sample

# Getting the observed statistics and turning it into a matrix so we can add it
# to the samples
observed <- summary(ans0$formula)
observed <- matrix(
  observed,
  nrow  = nrow(sample_centered[[1]]),
  ncol  = length(observed),
  byrow = TRUE
  )

# Now we uncenter the sample
sample_uncentered <- lapply(sample_centered, function(x) {
  x + observed
})

# We have to make it an mcmc.list object
sample_uncentered <- coda::mcmc.list(sample_uncentered)
```

Under the hood:

1. *Empirical means and sd, and quantiles*:

::: {.cell}

```
summary(sample_uncentered)
##
## Iterations = 655360:12255232
## Thinning interval = 65536
```

```
## Number of chains = 4
## Sample size per chain = 178
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                       Mean    SD Naive SE Time-series SE
## edges                2473.9 58.60   2.1961          3.846
## nodematch.hispanic 1826.6 50.63   1.8976          3.602
## nodematch.female1  1875.4 50.54   1.8941          3.847
## nodematch.eversmk1 1755.6 50.66   1.8985          3.315
## mutual              483.5 21.97   0.8232          1.634
##
## 2. Quantiles for each variable:
##
##                     2.5%  25%  50%    75%  97.5%
## edges               2366 2431 2473 2516.0 2583.4
## nodematch.hispanic 1725 1790 1828 1862.0 1923.4
## nodematch.female1  1774 1839 1876 1910.0 1974.5
## nodematch.eversmk1 1661 1718 1755 1791.0 1857.0
## mutual              441  469  483  499.2  527.2
```

:::

2. *Cross correlation*:

::: {.cell}

```
coda::crosscorr(sample_uncentered)
##                         edges nodematch.hispanic nodematch.female1
## edges               1.0000000          0.8565106         0.8788820
## nodematch.hispanic 0.8565106          1.0000000         0.7565759
## nodematch.female1  0.8788820          0.7565759         1.0000000
## nodematch.eversmk1 0.8645339          0.7342901         0.7441760
## mutual              0.7359569          0.6684517         0.6943574
##                     nodematch.eversmk1    mutual
```

```
## edges                          0.8645339 0.7359569
## nodematch.hispanic             0.7342901 0.6684517
## nodematch.female1              0.7441760 0.6943574
## nodematch.eversmk1             1.0000000 0.6648087
## mutual                         0.6648087 1.0000000
```

:::

3. Autocorrelation_: For now, we will only look at autocorrelation for chain one. Autocorrelation should be small (in a general MCMC setting). If autocorrelation is high, then it means that your sample is not idd (no Markov property). A way to solve this is *thinning* the sample.

::: {.cell}

```
coda::autocorr(sample_uncentered)[[1]]
## , , edges
##
##                   edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0        1.00000000        0.836037936      0.8821700893         0.86741018
## Lag 65536    0.29546092        0.336082499      0.3126843913         0.28693021
## Lag 327680  -0.06232415       -0.006038374     -0.0825766181         0.02595267
## Lag 655360   0.07015730        0.053058193      0.0960399588         0.03077291
## Lag 3276800 -0.00989517       -0.009090348      0.0001679658        -0.05633911
##                  mutual
## Lag 0        0.73293230
## Lag 65536    0.46321953
## Lag 327680  -0.02060089
## Lag 655360   0.12705728
## Lag 3276800  0.01462118
##
## , , nodematch.hispanic
##
##                   edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0        0.83603794         1.00000000       0.772940115         0.72892437
## Lag 65536    0.24693356         0.35329161       0.258747609         0.26629645
```

```
## Lag 327680  -0.01122090         0.01199279      -0.003794365         0.06759060
## Lag 655360   0.08604041         0.06242513       0.088828307         0.05919026
## Lag 3276800 -0.01659566        -0.03235612       0.000974174        -0.04643465
##                    mutual
## Lag 0         0.6716895580
## Lag 65536     0.3791243900
## Lag 327680    0.0261119585
## Lag 655360    0.0607146690
## Lag 3276800 -0.0001893006
##
## , , nodematch.female1
##
##                    edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0         0.88217009         0.77294011        1.00000000         0.74934143
## Lag 65536     0.27898163         0.34275682        0.33764682         0.27489365
## Lag 327680   -0.08876005        -0.06253290       -0.11524990        -0.01943560
## Lag 655360    0.06074076         0.02953052        0.09298027         0.04512914
## Lag 3276800   0.01357995        -0.01878670        0.03039838        -0.04875359
##                   mutual
## Lag 0         0.68557322
## Lag 65536     0.40180603
## Lag 327680   -0.11239238
## Lag 655360    0.08831210
## Lag 3276800   0.01514836
##
## , , nodematch.eversmk1
##
##                    edges nodematch.hispanic nodematch.female1
## Lag 0         0.867410177        0.728924371       0.749341430
## Lag 65536     0.308858124        0.329895859       0.352890856
## Lag 327680   -0.059305945       -0.006457098      -0.067161252
## Lag 655360    0.084904907        0.086115019       0.110833574
## Lag 3276800 -0.008811374        0.016144057      -0.005430382
##              nodematch.eversmk1      mutual
## Lag 0              1.000000000  0.65218802
## Lag 65536          0.365355720  0.44773193
```

```
## Lag 327680          0.049205563 -0.02967711
## Lag 655360          0.006413673  0.07422969
## Lag 3276800        -0.066505311  0.02476719
##
## , , mutual
##
##                   edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0        0.73293230        0.671689558        0.68557322        0.652188019
## Lag 65536    0.38912404        0.457808562        0.44041048        0.357851343
## Lag 327680  -0.05892551        0.015349925       -0.08588321        0.053255709
## Lag 655360   0.05482938        0.080365573        0.08786954        0.031917431
## Lag 3276800  0.02014438       -0.006550518        0.01638320        0.009361014
##                   mutual
## Lag 0        1.000000000
## Lag 65536    0.539527252
## Lag 327680  -0.002392256
## Lag 655360   0.064381438
## Lag 3276800 -0.007195045
```

:::

4. *Geweke Diagnostic*: From the function's help file:

   "If the samples are drawn from the stationary distribution of the chain, the two means are equal and Geweke's statistic has an asymptotically standard normal distribution. […] The Z-score is calculated under the assumption that the two parts of the chain are asymptotically independent, which requires that the sum of frac1 and frac2 be strictly less than 1.""

   —?coda::geweke.diag

Let's take a look at a single chain:

::: {.cell}

```
coda::geweke.diag(sample_uncentered)[[1]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##               edges nodematch.hispanic  nodematch.female1 nodematch.eversmk1
##              0.4535             0.5490             0.5069             0.8436
##              mutual
##              0.5864
```

:::

5. *(not included) Gelman Diagnostic*: From the function's help file:

   > Gelman and Rubin (1992) propose a general approach to monitoring convergence of MCMC output in which m > 1 parallel chains are run with starting values that are overdispersed relative to the posterior distribution. Convergence is diagnosed when the chains have 'forgotten' their initial values, and the output from all chains is indistinguishable. The gelman.diag diagnostic is applied to a single variable from the chain. It is based a comparison of within-chain and between-chain variances, and is similar to a classical analysis of variance.
   > —?coda::gelman.diag

As a difference from the previous diagnostic statistic, this uses all chains simultaneously:

::: {.cell}

```
coda::gelman.diag(sample_uncentered)
## Potential scale reduction factors:
##
##                     Point est. Upper C.I.
## edges                     1.00       1.02
## nodematch.hispanic        1.01       1.03
## nodematch.female1         1.00       1.01
## nodematch.eversmk1        1.00       1.01
## mutual                    1.01       1.03
##
```

```
## Multivariate psrf
##
## 1.03
```

:::

As a rule of thumb, values in the $[.9, 1.1]$ are good.

One nice feature of the `mcmc.diagnostics` function is the nice trace and posterior distribution plots that it generates. If you have the R package `latticeExtra` (Sarkar and Andrews 2022), the function will override the default plots used by `coda::plot.mcmc` and use lattice instead, creating nicer-looking plots. The next code chunk calls the `mcmc.diagnostic` function, but we suppress the rest of the output (see figure **?@fig-coda-plots**).

```
# [2022-03-13] This line is failing for what it could be an ergm bug
# mcmc.diagnostics(ans0, center = FALSE) # Suppressing all the output
```

If we call the function `mcmc.diagnostics`, this message appears at the end:

> MCMC diagnostics shown here are from the last round of simulation, prior to computation of final parameter estimates. Because the final estimates are refinements of those used for this simulation run, these diagnostics may understate model performance. To directly assess the performance of the final model on in-model statistics, please use the GOF command: gof(ergmFitObject, GOF=~model).
>
> —`mcmc.diagnostics(ans0)`

Not that bad (although the `mutual` term could do better)![4] First, observe that in the figure, we see four different lines; why is that? Since we were running in parallel using four cores, the algorithm ran four chains of the MCMC algorithm. An eyeball test is to see if all the chains moved at about the same place; in such a case, we can start thinking about model convergence from the MCMC perspective.

Once we are sure to have reach convergence on the MCMC algorithm, we can start thinking about how well does our model predicts the observed network's proterties. Besides the statistics that define our ERGM, the `gof` function's default behavior show GOF for:

[4] The statnet wiki website as a very nice example of (very) bad and good MCMC diagnostics plots here.

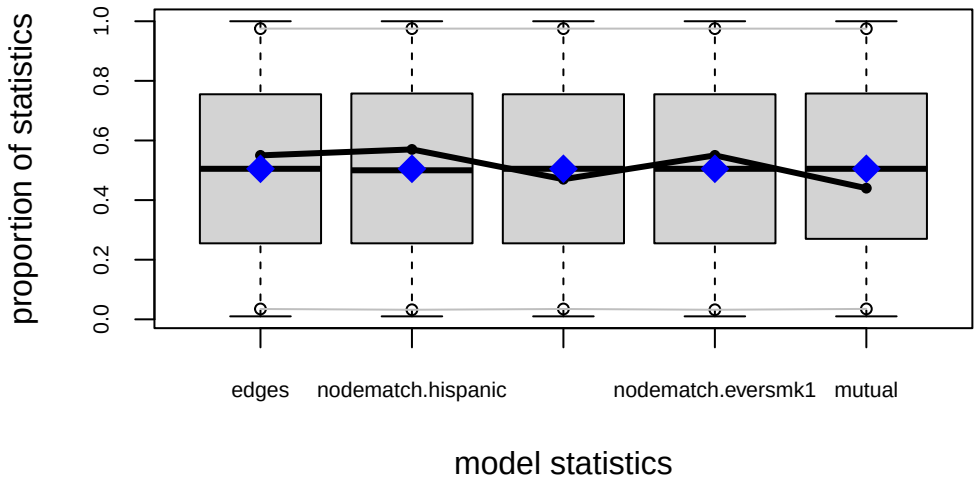
a. In degree distribution,
b. Out degree distribution,
c. Edge-wise shared partners, and
d. Geodesics

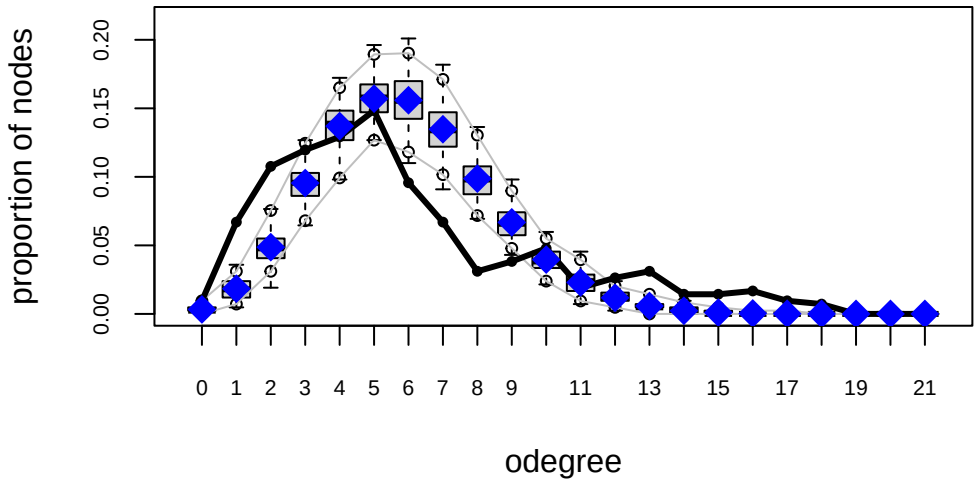
Let's take a look at it

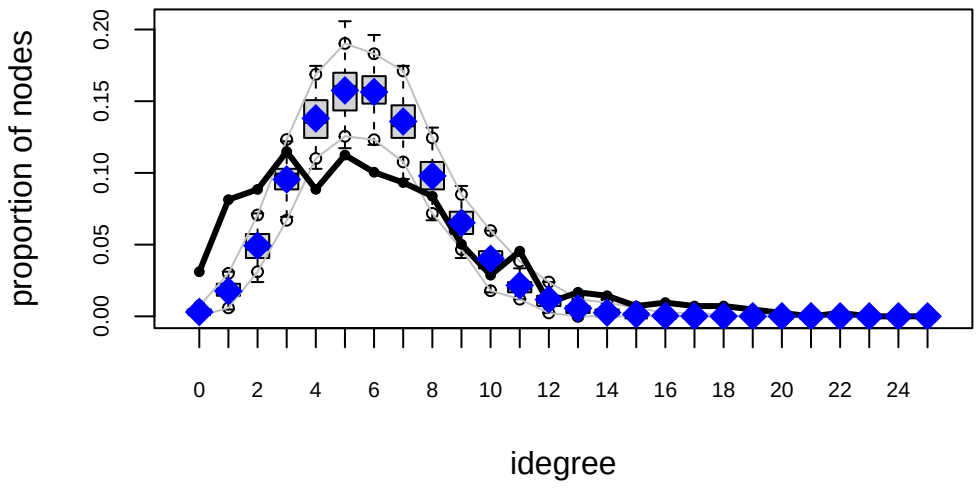
```
# Computing and printing GOF estatistics
ans_gof <- gof(ans0)
ans_gof
##
## Goodness-of-fit for in-degree
##
##            obs min  mean max MC p-value
## idegree0    13   0  1.27   4       0.00
## idegree1    34   2  7.37  15       0.00
## idegree2    37  10 20.57  30       0.00
## idegree3    48  24 39.88  56       0.22
## idegree4    37  43 57.68  80       0.00
## idegree5    47  49 65.84  86       0.00
## idegree6    42  47 65.37  86       0.00
## idegree7    39  40 56.79  73       0.00
## idegree8    35  24 40.92  55       0.34
## idegree9    21  17 27.27  38       0.18
## idegree10   12   6 16.76  29       0.34
## idegree11   19   1  9.03  18       0.00
## idegree12    4   1  4.91  12       0.94
## idegree13    7   0  2.33   6       0.00
## idegree14    6   0  1.09   5       0.00
## idegree15    3   0  0.59   3       0.06
## idegree16    4   0  0.19   1       0.00
## idegree17    3   0  0.08   1       0.00
## idegree18    3   0  0.03   1       0.00
## idegree19    2   0  0.01   1       0.00
## idegree20    1   0  0.00   0       0.00
## idegree21    0   0  0.02   1       1.00
## idegree22    1   0  0.00   0       0.00
```

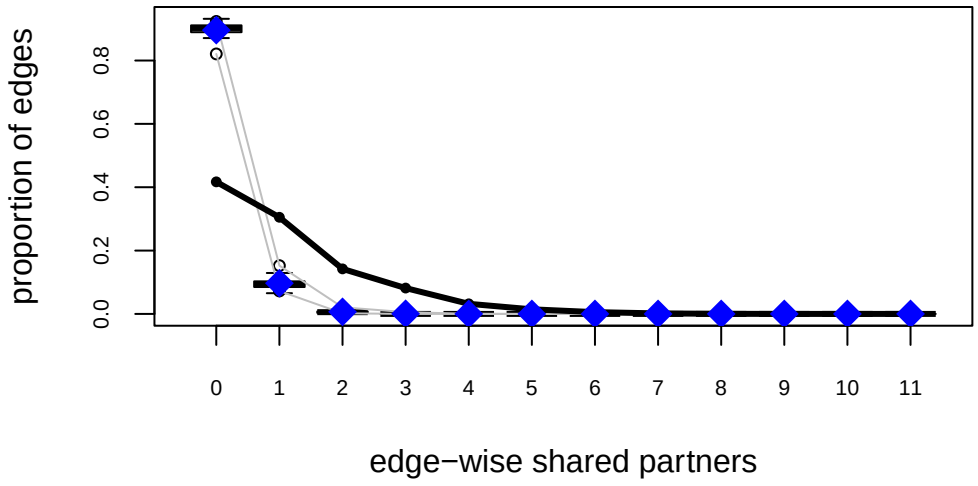
```
##
## Goodness-of-fit for out-degree
##
##           obs min  mean max MC p-value
## odegree0    4   0  1.37   5       0.08
## odegree1   28   2  7.70  15       0.00
## odegree2   45   8 20.36  35       0.00
## odegree3   50  22 39.91  56       0.14
## odegree4   54  39 57.28  72       0.64
## odegree5   62  53 65.65  85       0.60
## odegree6   40  46 65.17  90       0.00
## odegree7   28  38 56.32  76       0.00
## odegree8   13  29 41.26  57       0.00
## odegree9   16  18 27.93  41       0.00
## odegree10  20   9 16.57  25       0.46
## odegree11   8   3  9.60  19       0.74
## odegree12  11   0  4.92  10       0.00
## odegree13  13   0  2.42   7       0.00
## odegree14   6   0  0.96   4       0.00
## odegree15   6   0  0.39   3       0.00
## odegree16   7   0  0.14   2       0.00
## odegree17   4   0  0.05   1       0.00
## odegree18   3   0  0.00   0       0.00
##
## Goodness-of-fit for edgewise shared partner
##
##           obs  min    mean  max MC p-value
## esp.OTP0 1032 1975 2218.24 2333          0
## esp.OTP1  755  159  240.50  440          0
## esp.OTP2  352    3   16.40   79          0
## esp.OTP3  202    0    1.20   20          0
## esp.OTP4   79    0    0.07    1          0
## esp.OTP5   36    0    0.00    0          0
## esp.OTP6   14    0    0.00    0          0
## esp.OTP7    4    0    0.00    0          0
## esp.OTP8    1    0    0.00    0          0
```

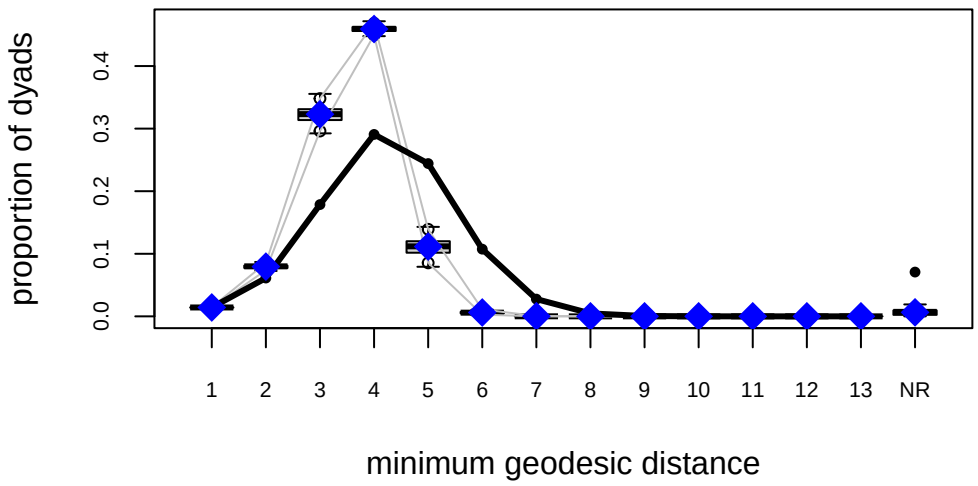
```
##
## Goodness-of-fit for minimum geodesic distance
##
##       obs   min     mean   max MC p-value
## 1    2475  2373  2476.41  2601       0.94
## 2   10672 12655 13847.68 15148       0.00
## 3   31134 50975 56296.27 61965       0.00
## 4   50673 77045 80030.39 82203       0.00
## 5   42563 13813 19437.28 24922       0.00
## 6   18719   327  1072.27  2060       0.00
## 7    4808     1    29.61   209       0.00
## 8     822     0     0.55    27       0.00
## 9     100     0     0.00     0       0.00
## 10      7     0     0.00     0       0.00
## Inf 12333     0  1115.54  3320       0.00
##
## Goodness-of-fit for model statistics
##
##                     obs  min    mean  max MC p-value
## edges              2475 2373 2476.41 2601       0.94
## nodematch.hispanic 1832 1769 1841.97 1941       0.88
## nodematch.female1  1879 1773 1874.83 1990       0.94
## nodematch.eversmk1 1755 1668 1758.81 1842       0.92
## mutual              486  447  483.09  514       0.88

# Plotting GOF statistics
plot(ans_gof)
```


proportion of statistics
0.0
0.2
0.4
0.6
0.8
1.0
edges
nodematch.hispanic
nodematch.eversmk1
mutual
model statistics


proportion of nodes
0.00
0.05
0.10
0.15
0.20
0
1
2
3
4
5
6
7
8
9
11
13
15
17
19
21
odegree


proportion of nodes
0.00
0.05
0.10
0.15
0.20
0
2
4
6
8
10
12
14
16
18
20
22
24
idegree


proportion of edges
0.0
0.2
0.4
0.6
0.8
0
1
2
3
4
5
6
7
8
9
10
11
edge−wise shared partners



Try the following configuration instead

```
ans0_bis <- ergm(
  network_111 ~
    edges +
    nodematch("hispanic") +
    nodematch("female1") +
    mutual +
    esp(0:3) +
    idegree(0:10)
    ,
  constraints = ~bd(maxout = 19),
  control = control.ergm(
    seed        = 1,
    MCMLE.maxit = 15,
    parallel    = 4,
    CD.maxit    = 15,
    MCMC.samplesize = 2048*4,
    MCMC.burnin = 30000,
    MCMC.interval = 2048*4
    )
```

```
)
```

Increase the sample size, so the curves are smoother, longer intervals (thinning), which reduces autocorrelation, and a larger burin. All this together to improve the Gelman test statistic. We also added idegree from 0 to 10, and esp from 0 to 3 to explicitly match those statistics in our model.

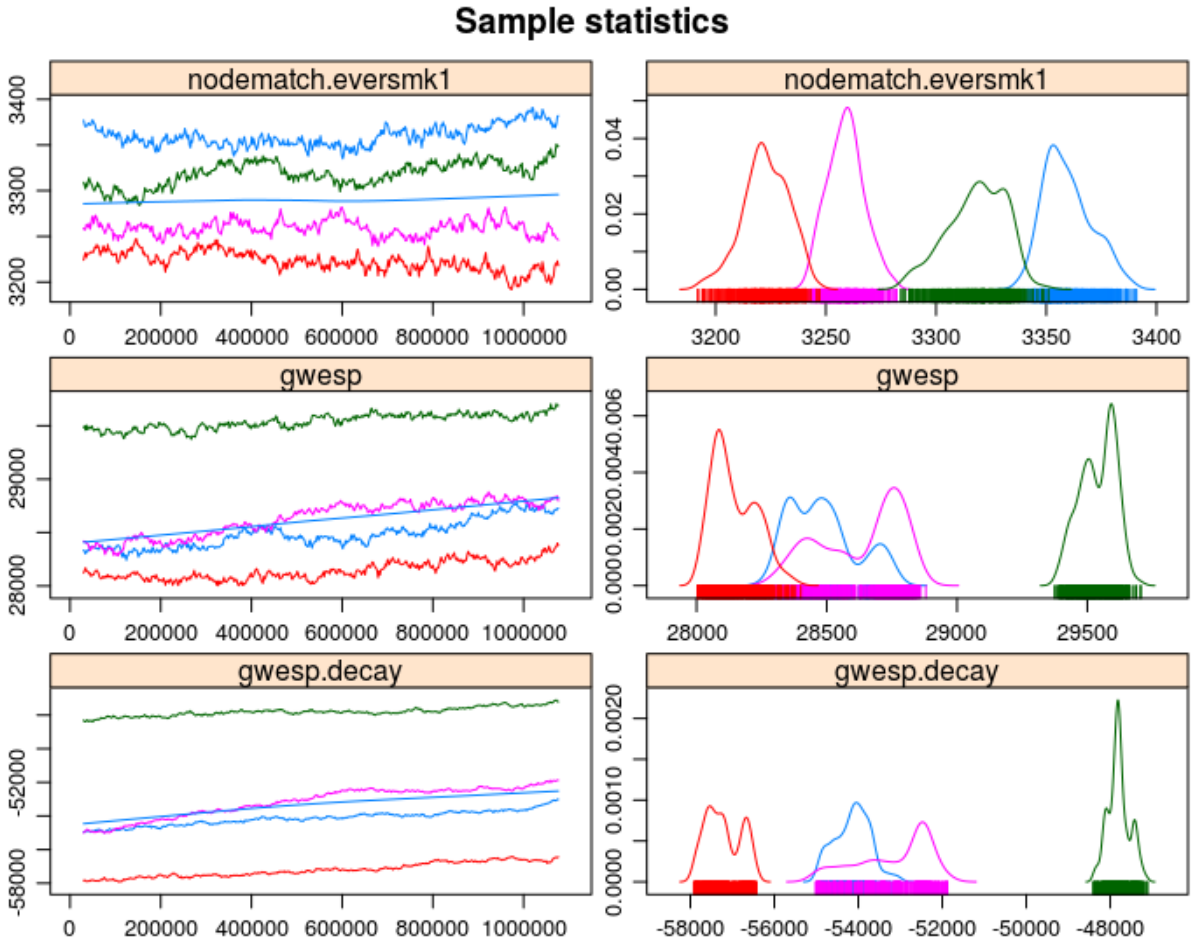


Figure 8.4: An example of a terrible ERGM (no convergence at all). Also, a good example of why running multiple chains can be useful

8.6 More on MCMC convergence

For more on this issue, I recommend reviewing chapter 1 and chapter 6 from the Handbook of MCMC (Brooks et al. 2011). Both chapters are free to download from the book's website.

For GOF take a look at section 6 of ERGM 2016 Sunbelt tutorial, and for a more technical review, you can take a look at (David R. Hunter, Goodreau, and Handcock 2008).

8.7 Mathematical Interpretation

One of the most critical parts of statistical modeling is interpreting the results, if not the most important. In the case of ERGMs, a key aspect is based on change statistics. Suppose that we would like to know how likely the tie y_{ij} is to happen, given the rest of the network. We can compute such probabilities using what literature sometimes describes as the Gibbs-sampler.

In particular, the log-odds of the ij tie ocurring conditional on the rest of the network can be written as:

$$\text{logit}\left(\mathbb{P}\left(y_{ij}=1|y_{-ij}\right)\right)=\theta^{\mathbf{t}}\Delta\delta\left(y_{ij}:0\to 1\right), \tag{8.2}$$

with $\delta\left(y_{ij}:0\to 1\right)\equiv s\left(\mathbf{y}\right)_{\text{ij}}^{+}-s\left(\mathbf{y}\right)_{\text{ij}}^{-}$ as the vector of change statistics, in other words, the difference between the sufficient statistics when $y_{ij}=1$ and its value when $y_{ij}=0$. To show this, we write the following:

$$\begin{aligned}\mathbb{P}\left(y_{ij}=1|y_{-ij}\right)&=\frac{\mathbb{P}\left(y_{ij}=1,x_{-ij}\right)}{\mathbb{P}\left(y_{ij}=1,y_{-ij}\right)+\mathbb{P}\left(y_{ij}=0,y_{-ij}\right)}\\&=\frac{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{+}\right\}}{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{+}\right\}+\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{-}\right\}}\end{aligned}$$

Applying the logit function to the previous equation, we obtain:

$$\begin{aligned}&=\log\left\{\frac{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{+}\right\}}{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{+}\right\}+\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{-}\right\}}\right\}-\log\left\{\frac{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{-}\right\}}{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{+}\right\}+\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{-}\right\}}\right\}\\&=\log\left\{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{+}\right\}\right\}-\log\left\{\exp\left\{\theta^{\mathbf{t}}s\left(\mathbf{y}\right)_{ij}^{-}\right\}\right\}\\&=\theta^{\mathbf{t}}\left(s\left(\mathbf{y}\right)_{ij}^{+}-s\left(\mathbf{y}\right)_{ij}^{-}\right)\end{aligned}$$

$$= \theta^{\mathbf{t}} \Delta\delta \left(y_{ij} : 0 \to 1 \right)$$

Henceforth, the conditional probability of node n gaining function k can be written as:

$$\mathbb{P}\left(y_{ij} = 1 | y_{-ij} \right) = \frac{1}{1 + \exp\left\{ -\theta^{\mathbf{t}} \Delta\delta \left(y_{ij} : 0 \to 1 \right) \right\}} \tag{8.3}$$

i.e., a logistic probability.

8.8 Markov independence

The challenge of analyzing networks is their interdependent nature. Nonetheless, in the absence of such interdependence, ERGMs are equivalent to logistic regression. Conceptually, if all the statistics included in the model do not involve two or more dyads, then the model is non-Markovian in the sense of Markov graphs.

Mathematically, to see this, it suffices to show that the ERGM probability can be written as the product of each dyads' probabilities.

$$\mathbb{P}\left(\mathbf{y} | \theta \right) = \frac{\exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right) \right\}}{\sum_y \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right) \right\}} = \frac{\prod_{ij} \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{ij} \right\}}{\sum_y \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right) \right\}}$$

Where $s\left(\right)_{ij}$ is a function such that $s\left(\mathbf{y} \right) = \sum_{ij} s\left(\mathbf{y} \right)_{ij}$. We now need to deal with the normalizing constant. To see how that can be saparated, let's start from the result:

$$\begin{aligned}
&= \prod_{ij} \left(1 + \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{ij} \right\} \right) \\
&= \left(1 + \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{11} \right\} \right) \left(1 + \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{12} \right\} \right) ... \left(1 + \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{nn} \right\} \right) \\
&= 1 + \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{11} \right\} + \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{11} \right\} \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{12} \right\} + \cdots + \prod_{ij} \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{ij} \right\} \\
&= 1 + \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{11} \right\} + \exp\left\{ \theta^{\mathbf{t}} \left(s\left(\mathbf{y} \right)_{11} + s\left(\mathbf{y} \right)_{12} \right) \right\} + \cdots + \prod_{ij} \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right)_{ij} \right\} \\
&= \sum_{\mathbf{y} \in \mathcal{Y}} \exp\left\{ \theta^{\mathbf{t}} s\left(\mathbf{y} \right) \right\}
\end{aligned}$$

Where the last equality follows from the fact that the sum *is* the sum over all possible combinations of networks, starting from $exp(0) = 1$, up to $exp(all)$. This way, we can now write:

$$\frac{\prod_{ij} \exp\left\{\theta^{\mathbf{t}} s\left(\mathbf{y}\right)_{ij}\right\}}{\sum_{y} \exp\left\{\theta^{\mathbf{t}} s\left(\mathbf{y}\right)\right\}} = \prod_{ij} \frac{\exp\left\{\theta^{\mathbf{t}} s\left(\mathbf{y}\right)_{ij}\right\}}{1 + \exp\left\{\theta^{\mathbf{t}} s\left(\mathbf{y}\right)_{ij}\right\}} \tag{8.4}$$

Related to this, block-diagonal ERGMs can be estimated as independent models, one per block. To see more about this, read (SNIJDERS 2010). Likewise, since independence depends–pun intended–on partitioning the objective function, as pointed by Snijders, non-linear functions make the model dependent, e.g., $s\left(\mathbf{y}\right) = \sqrt{\sum_{ij} y_{ij}}$, the square root of the edgecount is no longer a bernoulli graph.

9 Using constraints in ERGMs

Exponential Random Graph Models [ERGMs] can represent a variety of network classes. We often look at "regular" social networks like students in schools, colleagues in the workplace, or families. Nonetheless, some social networks we study have features that restrict how connections can occur. Typical examples are bi-partite graphs and multilevel networks. There are two classes of vertices in bi-partite networks, and ties can only occur between classes. On the other hand, Multilevel networks may feature multiple classes with inter-class ties somewhat restricted. In both cases, structural constraints exist, meaning that some configurations may not be plausible.

Mathematically, what we are trying to do is, instead of assuming that all network configurations are possible:

$$\{\mathbf{y} \in \mathcal{Y} : y_{ij} = 0, \forall i = j\}$$

we want to go a bit further avoiding loops, namely:

$$\{\mathbf{y} \in \mathcal{Y} : y_{ij} = 0, \forall i = j; \mathbf{y} \in C\}$$

,

where C is a constraint, for example, only networks with no triangles. The `ergm` R package has built-in capabilities to deal with some of these cases. Nonetheless, we can specify models with arbitrary structural constraints built into the model. The key is in using offset terms.

9.1 Example 1: Interlocking egos and disconnected alters

Imagine that we have two sets of vertices. The first, group `E`, are egos part of an egocentric study. The second group, called `A`, is composed of people mentioned by egos in `E` but were

not surveyed. Assume that individuals in `A` can only connect to individuals in `E`; moreover, individuals in `E` have no restrictions on connecting. In other words, only two types of ties exist: `E-E` and `A-E`. The question is now, how can we enforce such a constraint in an ERGM?

Using offsets, and in particular, setting coefficients to `-Inf` provides an easy way to restrict the support set of ERGMs. For example, if we wanted to constrain the support to include networks with no triangles, we would add the term `offset(triangle)` and use the option `offset.coef = -Inf` to indicate that realizations including triangles are not possible. Using R:

```
ergm(net ~ edges + offset(triangle), offset.coef = -Inf)
```

In this model, a Bernoulli graph, we reduce the sample space to networks with no triangles. In our example, such a statistic should only take non-zero values whenever ties within the `A` class happen. We can use the `nodematch()` term to do that. Formally

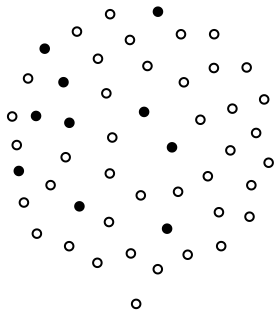
$$\text{NodeMatch}(x) = \sum_{i,j} y_{ij} \mathbf{1}(x_i = x_j)$$

This statistic will sum over all ties in which source (i) and target (j)'s X attribute is equal. One way to make this happen is by creating an auxiliary variable that equals, e.g., 0 for all vertices in `A`, and a unique value different from zero otherwise. For example, if we had 2 `A`s and three `E`s, the data would look something like this: $\{0, 0, 1, 2, 3\}$. The following code block creates an empty graph with 50 nodes, 10 of which are in group `E` (ego).

```
library(ergm, quietly =  TRUE)
library(sna, quietly =  TRUE)

n <- 50
n_egos <- 10
net <- as.network(matrix(0, ncol = n, nrow = n), directed = TRUE)

# Let's assing the groups
net %v% "is.ego" <- c(rep(TRUE, n_egos), rep(FALSE, n - n_egos))
net %v% "is.ego"
```

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE 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 FALSE FALSE FALSE
[49] FALSE FALSE
```

```r
gplot(net, vertex.col = net %v% "is.ego")
```



To create the auxiliary variable, we will use the following function:

```r
# Function that creates an aux variable for the ergm model
make_aux_var <- function(my_net, is_ego_dummy) {

  n_vertex <- length(my_net %v% is_ego_dummy)
  n_ego_   <- sum(my_net %v% is_ego_dummy)

  # Creating an auxiliary variable to identify the non-informant non-informant ties
  my_net %v% "aux_var" <- ifelse(
    !my_net %v% is_ego_dummy, 0, 1:(n_vertex - n_ego_)
    )
```

```
  my_net
}
```

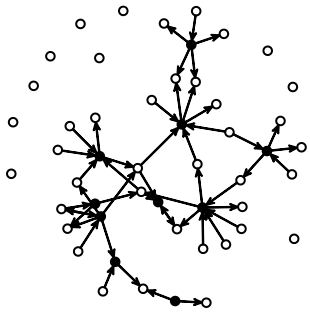
Calling the function in our data results in the following:

```
net <- make_aux_var(net, "is.ego")

# Taking a look over the first 15 rows of data
cbind(
  Is_Ego = net %v% "is.ego",
  Aux    = net %v% "aux_var"
) |> head(n = 15)
```

```
      Is_Ego Aux
 [1,]      1   1
 [2,]      1   2
 [3,]      1   3
 [4,]      1   4
 [5,]      1   5
 [6,]      1   6
 [7,]      1   7
 [8,]      1   8
 [9,]      1   9
[10,]      1  10
[11,]      0   0
[12,]      0   0
[13,]      0   0
[14,]      0   0
[15,]      0   0
```

We can now use this data to simulate a network in which ties between `A`-class vertices are not possible:

```
set.seed(2828)
net_sim <- simulate(net ~ edges + nodematch("aux_var"), coef = c(-3.0, -Inf))
gplot(net_sim, vertex.col = net_sim %v% "is.ego")
```



As you can see, this network has only ties of the type `E-E` and `A-E`. We can double-check by (i) looking at the counts and (ii) visualizing each induced-subgraph separately:

```
summary(net_sim ~ edges + nodematch("aux_var"))
```

```
             edges nodematch.aux_var
                49                  0
```

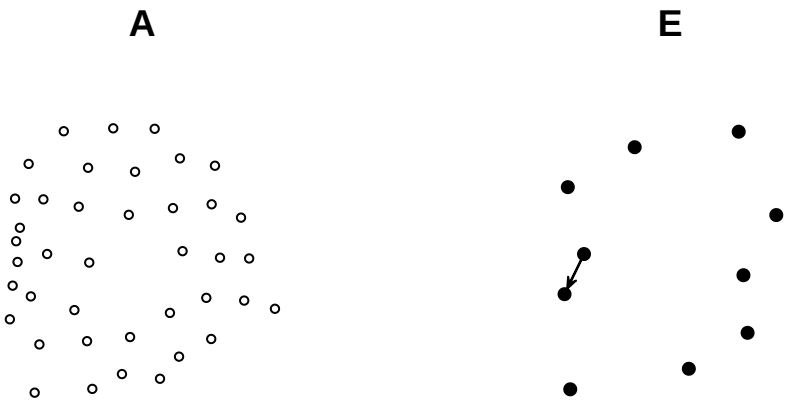
```
net_of_alters <- get.inducedSubgraph(
  net_sim, which((net_sim %v% "aux_var") == 0)
  )

net_of_egos <- get.inducedSubgraph(
  net_sim, which((net_sim %v% "aux_var") != 0)
  )

# Counts
summary(net_of_alters ~ edges + nodematch("aux_var"))
```

```
             edges nodematch.aux_var
                 0                  0
```

```
summary(net_of_egos ~ edges + nodematch("aux_var"))
```

```
             edges nodematch.aux_var
                 1                 0
```

```
# Figures
op <- par(mfcol = c(1, 2))
gplot(net_of_alters, vertex.col = net_of_alters %v% "is.ego", main = "A")
gplot(net_of_egos, vertex.col = net_of_egos %v% "is.ego", main = "E")
```




```
par(op)
```

Now, to fit an ERGM with this constraint, we simply need to make use of the offset terms. Here is an example:

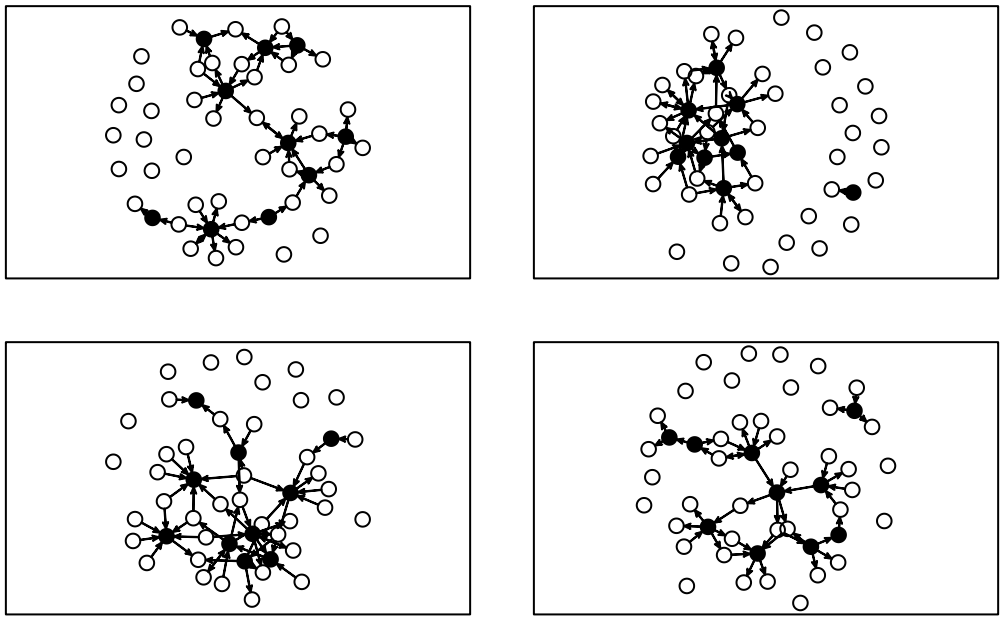
```
ans <- ergm(
  net_sim ~ edges + offset(nodematch("aux_var")), # The model (notice the offset)
  offset.coef = -Inf                              # The offset coefficient
  )
## Starting maximum pseudolikelihood estimation (MPLE):
```

```
## Obtaining the responsible dyads.
## Evaluating the predictor and response matrix.
## Maximizing the pseudolikelihood.
## Finished MPLE.
## Evaluating log-likelihood at the estimate.
summary(ans)
## Call:
## ergm(formula = net_sim ~ edges + offset(nodematch("aux_var")),
##     offset.coef = -Inf)
##
## Maximum Likelihood Results:
##
##                          Estimate Std. Error MCMC % z value Pr(>|z|)
## edges                      -2.843      0.147      0  -19.34   <1e-04 ***
## offset(nodematch.aux_var)     -Inf      0.000      0    -Inf   <1e-04 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##      Null Deviance: 1233.8  on 890  degrees of freedom
##  Residual Deviance:  379.4  on 888  degrees of freedom
##
## AIC: 381.4  BIC: 386.2  (Smaller is better. MC Std. Err. = 0)
##
##  The following terms are fixed by offset and are not estimated:
##   offset(nodematch.aux_var)
```

This ERGM model–which by the way only featured dyadic-independent terms, and thus can be reduced to a logistic regression–restricts the support by excluding all networks in which ties within the class `A` exists. To finalize, let's look at a few simulations based on this model:

```
set.seed(1323)
op <- par(mfcol = c(2,2), mar = rep(1, 4))
for (i in 1:4) {
  gplot(simulate(ans), vertex.col = net %v% "is.ego", vertex.cex = 2)
```

```
  box()
}
```



```
par(op)
```

All networks with no ties between `A` nodes.

9.2 Example 2: Bi-partite networks

In the case of bipartite networks (sometimes called affiliation networks,) we can use `ergm`'s terms for bipartite graphs to corroborate what we discussed here. For example, the two-star term. Let's start simulating a bipartite network using the `edges` and `two-star` parameters. Since the `k-star` term is usually complex to fit (tends to generate degenerate models,) we will take advantage of the `Log()` transformation function in the `ergm` package to smooth the term.[1]

[1] After writing this example, it became apparent the use of the `Log()` transformation function may not be ideal. Since many terms used in ERGMs can be zero, e.g., triangles, the term `Log(~ ostar(2))` is undefined when `ostar(2) = 0`. In practice, the ERGM package sets a lower limit for the log of 0, so, instead of having `Log(0) ~ -Inf`, they set it to be a really large negative number. This causes all sorts of issues to the estimates; in our example, an overestimation of the population parameter and a positive log-likelihood. Therefore, I wouldn't recommend using this transformation too often.

The bipartite network that we will be simulating will have 100 actors and 50 entities. Actors, which we will map to the first level of the `ergm` terms, this is, `b1star` `b1nodematch`, etc. will send ties to the entities, the second level of the bipartite ERGM. To create a bipartite network, we will create an empty matrix of size `nactors x nentitites`; thus, actors are represented by rows and entities by columns.

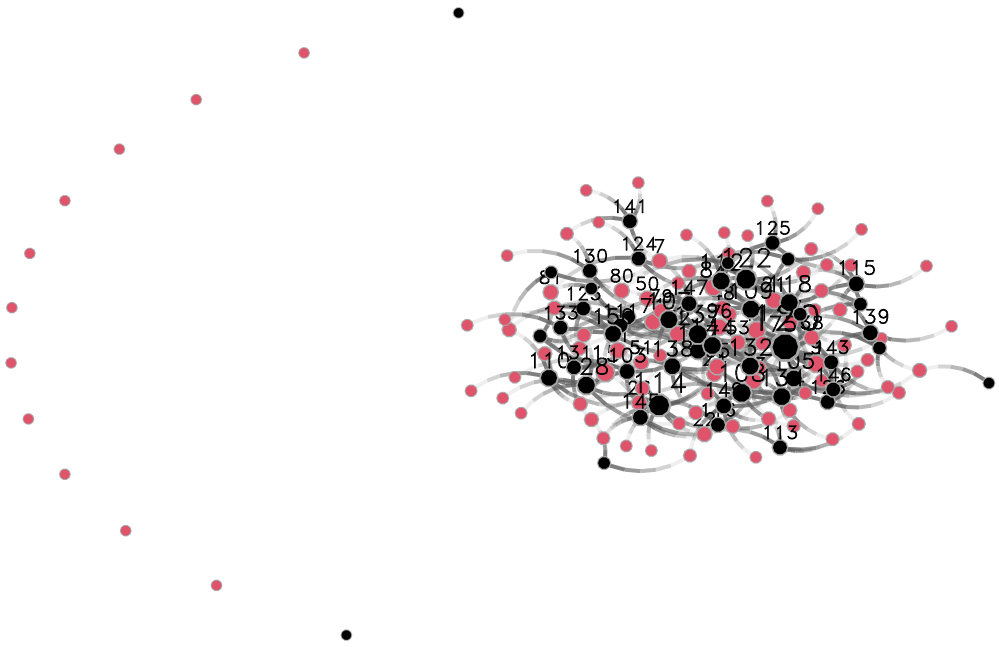
```
# Parameters for the simulation
nactors   <- 100
nentities <- floor(nactors/2)
n         <- nactors + nentities

# Creating an empty bipartite network (baseline)
net_b <- network(
  matrix(0, nrow = nactors, ncol = nentities), bipartite = TRUE
)

# Simulating the bipartite ERGM,
net_b <- simulate(net_b ~ edges + Log(~b1star(2)), coef = c(-3, 1.5), seed = 55)
```

Let's see what we got here:

```
summary(net_b ~ edges + Log(~b1star(2)))
```

```
      edges Log~b1star2
 245.000000    5.746203
```

```
netplot::nplot(net_b, vertex.col = (1:n <= nactors) + 1)
```



Notice that the first `nactors` vertices in the network are the actors, and the remaining are the entities. Now, although the `ergm` package features bipartite network terms, we can still fit a bipartite ERGM without explicitly declaring the graph as such. In such case, the `b1star(2)` term of a bipartite network is equivalent to an `ostar(2)` in a directed graph. Likewise, `b2star(2)` in a bipartite graph matches the `istar(2)` term in a directed graph. This information will be relevant when fitting the ERGM. Let's transform the bipartite network into a directed graph. The following code block does so:

```
# Identifying the edges
net_not_b <- which(as.matrix(net_b) != 0, arr.ind = TRUE)

# We need to offset the endpoint of the ties by nactors
# so that the ids go from 1 through (nactors + nentitites)
net_not_b[,2] <- net_not_b[,2] + nactors

# The resulting graph is a directed network
net_not_b <- network(net_not_b, directed = TRUE)
```

Now we are almost done. As before, we need to use node-level covariates to put the constraints in our model. For this ERGM to reflect an ERGM on a bipartite network, we need two constraints:

1. Only ties from actors to entities are allowed, and
2. entities can only receive ties.

The corresponding offset terms for this model are: `nodematch("is.actor") ~ -Inf`, and `nodeocov("isnot.actor") ~ -Inf`. Mathematically:

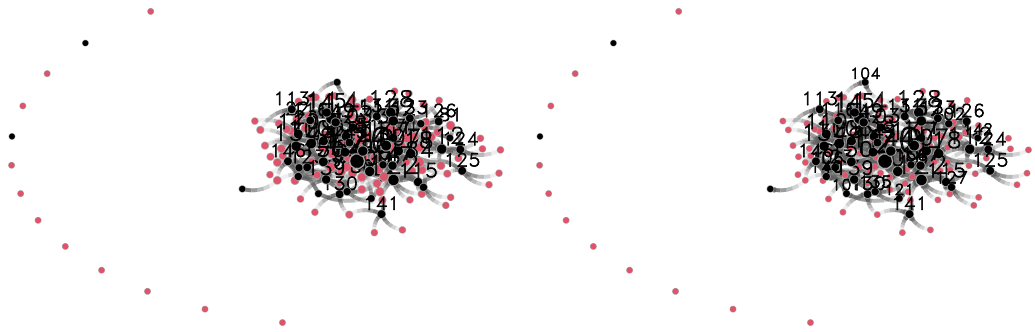
$$\text{NodeMatch}(\text{x} = \text{"is.actor"}) = \sum_{i<j} y_{ij} \mathbb{1}\left(x_i = x_j\right)$$
$$\text{NodeOCov}(\text{x} = \text{"isnot.actor"}) = \sum_{i} x_i \times \sum_{j<i} y_{ij}$$

In other words, we are setting that ties between nodes of the same class are forbidden, and outgoing ties are forbidden for entities. Let's create the vertex attributes needed to use the aforementioned terms:

```
net_not_b %v% "is.actor" <- as.integer(1:n <= nactors)
net_not_b %v% "isnot.actor" <- as.integer(1:n > nactors)
```

Finally, to make sure we have done all well, let's look how both networks–bipartite and unimodal–look side by side:

```
# First, let's get the layout
fig <- netplot::nplot(net_b, vertex.col = (1:n <= nactors) + 1)
gridExtra::grid.arrange(
  fig,
  netplot::nplot(
    net_not_b, vertex.col = (1:n <= nactors) + 1,
    layout = fig$.layout
     ),
  ncol = 2, nrow = 1
)
```



```
# Looking at the counts
summary(net_b ~ edges + b1star(2) + b2star(2))
```

```
  edges b1star2 b2star2
    245     313     645
```

```
summary(net_not_b ~ edges + ostar(2) + istar(2))
```

```
 edges ostar2 istar2
   245    313    645
```

With the two networks matching, we can now fit the ERGMs with and without offset terms and compare the results between the two models:

```
# ERGM with a bipartite graph
res_b     <- ergm(
  # Main formula
  net_b ~ edges + Log(~b1star(2)),

  # Control parameters
```

```
  control = control.ergm(seed = 1)
  )
```

```
Warning: 'glpk' selected as the solver, but package 'Rglpk' is not available;
falling back to 'lpSolveAPI'. This should be fine unless the sample size and/or
the number of parameters is very big.
```

```
# ERGM with a digraph with constraints
res_not_b <- ergm(
  # Main formula
  net_not_b ~ edges + Log(~ostar(2)) +

  # Offset terms
  offset(nodematch("is.actor")) + offset(nodeocov("isnot.actor")),
  offset.coef = c(-Inf, -Inf),

  # Control parameters
  control = control.ergm(seed = 1)
  )
```

Here are the estimates (using the `texreg` R package for a prettier output):

```
texreg::screenreg(list(Bipartite = res_b, Directed = res_not_b))
```

```
==========================================================================
                              Bipartite    Directed
--------------------------------------------------------------------------
edges                          -3.14 ***                  -3.11 ***
                               (0.15)                     (0.14)
Log~b1star2                    21.89
                              (17.13)
Log~ostar2                                                19.66
                                                         (16.75)
offset(nodematch.is.actor)                                -Inf
```

```
offset(nodeocov.isnot.actor)                                    -Inf

---------------------------------------------------------------------
AIC                            1958.00      -2134192392498170112.00
BIC                            1971.03      -2134192392498170112.00
Log Likelihood                 -977.00       1067096196249085056.00
=====================================================================
*** p < 0.001; ** p < 0.01; * p < 0.05
```

As expected, both models yield the "same" estimate. The minor differences observed between the models are how the `ergm` package performs the sampling. In particular, in the bipartite case, `ergm` has special routines for making the sampling more efficient, having a higher acceptance rate than that of the model in which the bipartite graph was not explicitly declared. We can tell this by inspecting rejection rates:

```
data.frame(
  Bipartite = coda::rejectionRate(res_b$sample[[1]]) * 100,
  Directed  = coda::rejectionRate(res_not_b$sample[[1]][, -c(3,4)]) * 100
) |> knitr::kable(digits = 2, caption = "Rejection rate (percent)")
```

Table 9.1: Rejection rate (percent)

	Bipartite	Directed
edges	2.48	3.67
Log~b1star2	1.24	2.04

The ERGM fitted with the offset terms has a much higher rejection rate than that of the ERGM fitted with the bipartite ERGM.

Finally, the fact that we can fit ERGMs using offset does not mean that we need to use it ALL the time. Unless there is a very good reason to go around `ergm`'s capabilities, I wouldn't recommend fitting bipartite ERGMs as we just did, as the authors of the package have included (MANY) features to make our job easier.

10 Temporal Exponential Family Random Graph Models

This tutorial is great! https://statnet.org/trac/raw-attachment/wiki/Sunbelt2016/tergm_tutorial.pdf

11 Hypothesis testing in networks

Overall, there are many ways in which we can see hypothesis testing within the networks context:

1. **Comparing two or more networks**, e.g., we want to see if the density of two networks are *equal*.
2. **Prevalence of a motif/pattern**, e.g., check whether the observed number of transitive triads is different from that expected as of by chance.
3. **Multivariate using ERGMs**, e.g., jointly test whether homophily and two stars are the motifs that drive network structure.

The latter we already review in the ERGM chapter. In this part, we will look at types one and two; both using non-parametric methods.

11.1 Comparing networks

Imagine that we have two graphs, $(G_1, G_2) \in \mathcal{G}$, and we would like to assess whether a given statistic $s(\cdot)$, e.g., density, is equal in both of them. Formally, we would like to asses whether $H_0 : s(G_1) - s(G_2) = k$ vs $H_a : s(G_1) - s(G_2) \neq k$.

As usual, the true distribution of $s(\cdot)$ is unknown, thus, one approach that we could use is a non-parametric bootstrap test.

11.1.1 Network bootstrap

The non parametric bootstrap and jackknife methods for social networks were introduced by (T. A. B. Snijders and Borgatti 1999). The method itself is used to generate standard errors for network level statistics. Both methods are implemented in the R package `netdiffuseR`.

11.1.2 When the statistic is normal

When the we deal with things that are normally distributed, e.g., sample means like density[1], we can make use of the Student's distribution for making inference. In particular, we can use Bootstrap/Jackknife to approximate the standard errors of the statistic for each network:

1. Since $s(G_i) \sim \text{N}(\mu_i, \sigma_i^2/m_i)$ for $i \in \{1, 2\}$, in the case of the density, $m_i = n_i * (n_i - 1)$. The statistic is then:

   $$s(G_1) - s(G_0) \sim \text{N}(\mu_1 - \mu_0, \sigma_1^2/m_1 + \sigma_1^2/m_2)$$

   Thus

   $$\frac{s(G_1) - s(G_0) - \mu_1 + \mu_2}{\sqrt{\sigma_1^2/m_1 + \sigma_1^2/m_2}} \sim t_{m_1+m_2-2}$$

   But, if we are testing $H_0 : \mu_1 - \mu_2 = k$, then, under the null

   $$\frac{s(G_1) - s(G_0) - k}{\sqrt{\sigma_1^2/m_1 + \sigma_1^2/m_2}} \sim t_{m_1+m_2-2}$$

   Where We now proceede to approximate the variances.

2. Using the *plugin principle* (Efron and Tibshirani 1994), we can approximate the variances using Bootstrap/Jackknife, i.e., compute $\hat{\sigma}_1^2 \approx \sigma_1^2/m_1$ and $\hat{\sigma}_2^2 \approx \sigma_2^2/m_2$. Using netdiffuseR

```
library(netdiffuseR)

# Obtain a 100 replicates
sg1 <- bootnet(g1, function(i, ...) sum(i)/(nnodes(i) * (nnodes(i) - 1)), R = 100)
sg2 <- bootnet(g2, function(i, ...) sum(i)/(nnodes(i) * (nnodes(i) - 1)), R = 100)

# Retrieving the variances
hat_sigma1 <- sg1$var_t
hat_sigma2 <- sg2$var_t
```

[1] Density is indeed a sample mean as we are, in principle computing the average of a sequence of Bernoulli variables. Formally: $\text{density}(G) = \frac{1}{n(n-1)} \sum_{ij} A_{ij}$.

```r
# And the actual values
sg1 <- sg1$t0
sg2 <- sg2$t0
```

3. With the approximates in hand, we can then use the the "t-test table" to retrieve the corresponding value, in R:

```r
# Building the statistic
k <- 0 # For equal variances
tstat <- (sg1 - sg2 - k)/(sqrt(hat_sigma1 + hat_sigma2))

# Computing the pvalue
m1 <- nnodes(g1)*(nnodes(g1) - 1)
m2 <- nnodes(g2)*(nnodes(g2) - 1)
pt(tstat, df = m1 + m2 - 2)
```

11.1.3 When the statistic is NOT normal

In the case that the statistic is not normally distributed, we cannot use the t-statistic any longer. Nevertheless, the Bootstrap can come to help. While in general it is better to use distributions of pivot statistics (see (Efron and Tibshirani 1994)), we can still leverage the power of this method to make inferences. For this example, $s(\cdot)$ will be the range of the threshold in a diffusion graph.

As before, imagine that we are dealing with an statistic $s(\cdot)$ for two different networks, and we would like to asses whether we can reject H_0 or fail to reject it. The procedure is very similar:

1. One approach that we can test is whether $k \in \text{ConfInt}(s(G_1) - s(G_2))$. Building confidence intervals with bootstrap could be more intuitive.

2. Like before, we use bootstrap to generate a distribution of $s(G_1)$ and $s(G_2)$, in R:

```r
# Obtain a 1000 replicates
sg1 <- bootnet(g1, function(i, ...) range(threshold(i)), R = 1000)
sg2 <- bootnet(g2, function(i, ...) range(threshold(i)), R = 1000)

# Retrieving the distributions
```

```r
sg1 <- sg1$boot$t
sg2 <- sg2$boot$t

# Define the statistic
sdiff <- sg1 - sg2
```

3. Once we have `sdiff`, we can proceed and compute the, for example, 95% confidence interval, and evaluate whether k falls within. In R:

```r
diff_ci <- quantile(sdiff, probs = c(0.025, .975))
```

This corresponds to what Efron and Tibshirani call "percentile interval." This is easy to compute, but a better approach is using the "BCa" method, "Bias Corrected and Accelerated." (TBD)

11.2 Examples

11.2.1 Average of node-level stats

Supposed that we would like to compare something like average indegree. In particular, for both networks, G_1 and G_2, we compute the average indegree per node:

$$s(G_1) = \text{AvgIndeg}(G_1) = \frac{1}{n}\sum_i \sum_{j \neq i} A^1_{ji}$$

where A^1_{ji} equals one if vertex j sends a tie to i. In this case, since we are looking at an average, we have that $\text{AvgIndeg}(G_1) \sim N(\mu_1, \sigma_1^2/n)$. Thus, taking advantage of the normality of the statistic, we can build a test statistic as follows:

$$\frac{s(G_1) - s(G_2) - k}{\sqrt{\hat{\sigma}_1^2 + \hat{\sigma}_2^2}} \sim t_{n_1+n_2-2}$$

Where $\hat{\sigma}_i$ is the bootstrap standard error, and $k = 0$ when we are testing equality. This distributes t with $n_1 + n_2 - 2$ degrees of freedom. As a difference from the previous example using density, the degrees of freedom for this test are less as, instead of having an average across all entries of the adjacency matrix, we have an average across all vertices.

12 Stochastic Actor Oriented Models

Stochastic Actor Oriented Models (SOAM), also known as Siena models were introduced by CITATION NEEDED.

As a difference from ERGMs, Siena models look at the data generating process from the individuals' point of view. Based on McFadden's ideas of probabilistic choice, the model is founded in the following equation

$$U_i(x) - U_i(x') \sim \text{Extreame Value Distribution}$$

In other words, individuals choose between states x and x' in a probabilistic way (with some noise),

$$\frac{\exp\left\{f_i^Z(\beta^z, x, z)\right\}}{\sum_{Z' \in \mathcal{C}} \exp\left\{f_i^Z(\beta, x, z')\right\}}$$

snijders_(sociological methodology 2001)

Ripley et al. (2011)

13 Power calculation in network studies

In survey and study design, calculating the required sample size is critical. Nowadays, tools and methods for calculating the required sample size abound; nonetheless, sample size calculation for studies involving social networks is still underdeveloped. This chapter will illustrate how we can use computer simulations to estimate the required sample size. Chapter (**part2-power?**) provides a general overview of power analysis.

13.1 Example 1: Spillover effects in egocentric studies[1]

Suppose we want to run an intervention over a particular population, and we are interested in the effects of such intervention on the egos' alters. In economics, this problem, which they call the "spillover effect," is actively studied.

We assume that alters only get exposed if egos acquire the behavior for the power calculation. Furthermore, for this first run, we will assume that there is no social reinforcement or influence between alters. We will later relax this assumption. To calculate power, we will do the following:

1. Simulate egos' behavior following a logit distribution.
2. Randomly drop some egos as a result of attrition.
3. Simulate alters' behavior using their egos as the treatment.
4. Fit a logistic regression based on the previous model.
5. Accept/reject the null and store the result.

[1]The original problem was posed by Dr. Shinduk Lee from the School of Nursing at the University of Utah.

The previous steps will be repeated 500 for each value of n we analyze. We will finalize by plotting power against sample sizes. Let's first start by writing down the simulation parameters:

```
# Design
n_sims    <- 500 # Number of simulations
n_a       <- 4   # Number of alters
sizes     <-     # Sizes to try
  seq(from = 100, to = 200, by = 25)

# Assumptions
odds_h_1  <- 2.0 # Odds of Increase/
attrition <- .3
baseline  <- .2  # Low prevalence in 1s

# Parameters
alpha    <- .05
beta_pow <- 0.2
```

As we discuss in (**part2-power?**), it is always a good idea to encapsulate the simulation into a function:

```
# The odds turned to a prob
theta_h_1 <- plogis(log(odds_h_1))

# Simulation function
sim_data <- function(n) {

  # Treatment assignment
  tr  <- c(rep(1, n/2), rep(0, n/2))

  # Step 1: Sampling population of egos
  y_ego <- runif(n) < c(
    rep(theta_h_1, n/2),
    rep(0.5, n/2)
  )
```

```
  # Step 2: Simulating attrition
  todrop <- order(runif(n))[1:(n * attrition)]
  y_ego  <- y_ego[-todrop]
  tr     <- tr[-todrop]
  n      <- n - length(todrop)

  # Step 3: Simulating alter's effect. We assume the same as in
  # ego
  tr_alter <- rep(y_ego * tr, n_a)
  y_alter  <- runif(n * n_a) < ifelse(tr_alter, theta_h_1, 0.5)

  # Step 4: Computing test statistic
  res_ego   <- tryCatch(glm(y_ego ~ tr, family = binomial("logit")), error = function(e)
  res_alter <- tryCatch(glm(y_alter ~ tr_alter, family = binomial("logit")), error = fun

  if (inherits(res_ego, "error") | inherits(res_alter, "error"))
    return(c(ego =  NA, alter = NA))

  # Step 5: Reject?
  c(
    ego   = summary(res_ego)$coefficients["tr", "Pr(>|z|)"] < alpha,
    alter = summary(res_alter)$coefficients["tr_alter", "Pr(>|z|)"] < alpha
  )


}
```

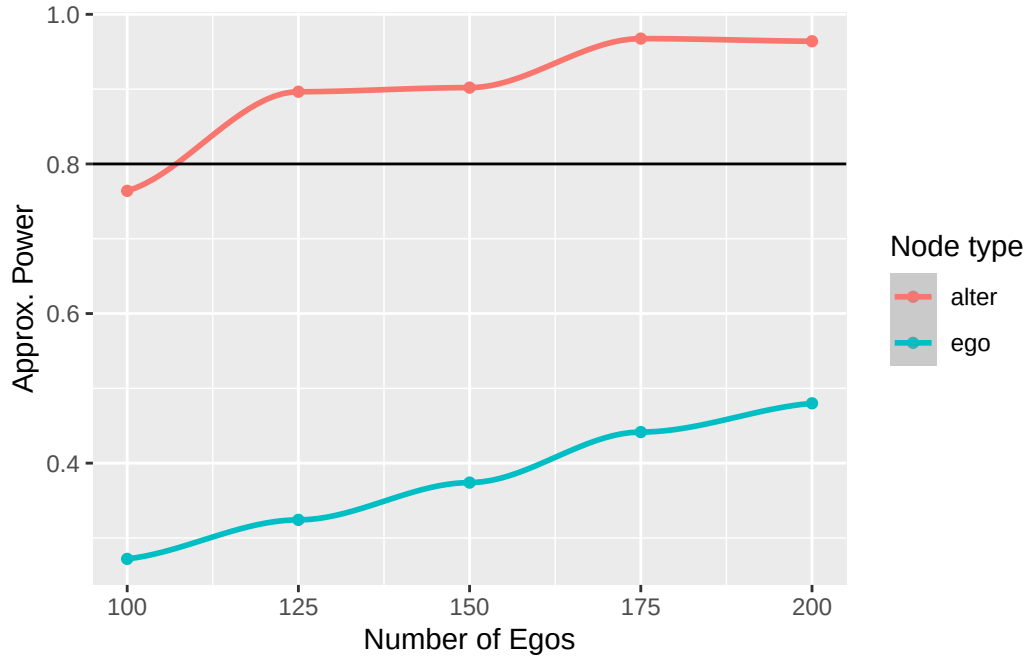
Now that we have the data generating function, we can run the simulations to approximate statistical power given the sample size. The results will be stored in the matrix `spower`. Since we are simulating data, it is crucial to set the seed so we can reproduce the results.

```
# We always set the seed
set.seed(88)

# Making space, and running!
```

```
spower <- NULL
for (s in sizes) {

  # Run the simulation for size s
  simres <- rowMeans(replicate(n_sims, sim_data(s)), na.rm = TRUE)

  # And store the results
  spower <- rbind(spower, simres)

}
```

The following figure shows the approximate power for finding effects at both levels, ego and alter:

```
library(ggplot2)

spower <- rbind(
  data.frame(size = sizes, power = spower[,"ego"], type =  "ego"),
  data.frame(size = sizes, power = spower[,"alter"], type =  "alter")
)

spower |>
  ggplot(aes(x = size, y = power, colour = type)) +
  geom_point() +
  geom_smooth(method = "loess", formula = y ~ x) +
  labs(x = "Number of Egos", y = "Approx. Power", colour = "Node type") +
  geom_hline(yintercept = 1 - beta_pow)
```



As shown in Chapter (**part2-power?**), we can use a linear regression model to predict sample size as a function of statistical power:

```
# Fitting the model
power_model <- glm(
  size ~ power + I(power^2),
  data = spower, family = gaussian(), subset = type == "alter"
)

summary(power_model)
```

```
Call:
glm(formula = size ~ power + I(power^2), family = gaussian(),
    data = spower, subset = type == "alter")

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)     1460       1342   1.088    0.390
power          -3532       3124  -1.131    0.376
I(power^2)      2293       1805   1.270    0.332
```

```
(Dispersion parameter for gaussian family taken to be 317.0856)

    Null deviance: 6250.00  on 4  degrees of freedom
Residual deviance:  634.17  on 2  degrees of freedom
AIC: 46.404

Number of Fisher Scoring iterations: 2
```

```r
# Predict
predict(power_model, newdata = data.frame(power = .8), type = "response") |>
  ceiling()
```

```
  1
102
```

From the figure, it becomes apparent that, although there is not enough power to identify effects at the ego level, because each ego brings in five alters, the alter sample size is high enough that we can reach above 0.8 statistical power with relatively small sample size.

13.2 Example 2: Spillover effects pre-post effect

Now the dynamics are different. Instead of having a treated and control group, we have a single group over which we will measure behavioral change. We will simulate individuals in their initial state, still 0/1, and then simulate that the intervention will make them more likely to have $y = 1$. We will also assume that subjects generally don't change their behavior and that the baseline prevalence of zeros is higher. The simulation steps are as follows:

1. For each individual in the population, draw the underlying probability that $y = 1$. With that probability, assign the value of y. This applies to both ego and alter.
2. Randomly drop some egos, and their corresponding alters due to attrition.
3. Simulate alters' behavior using their egos as the treatment. Both ego and alter's underlying probability are increased by the chosen odds.

4. To control for the underlying probability that an individual has $y = 1$, we use conditional logistic regression (also known as matched case-control logit,) to estimate the treatment effects.

5. Accept/reject the null and store the result.

```
beta_pars <- c(4, 6)
odds_h_1  <- 2.0
```

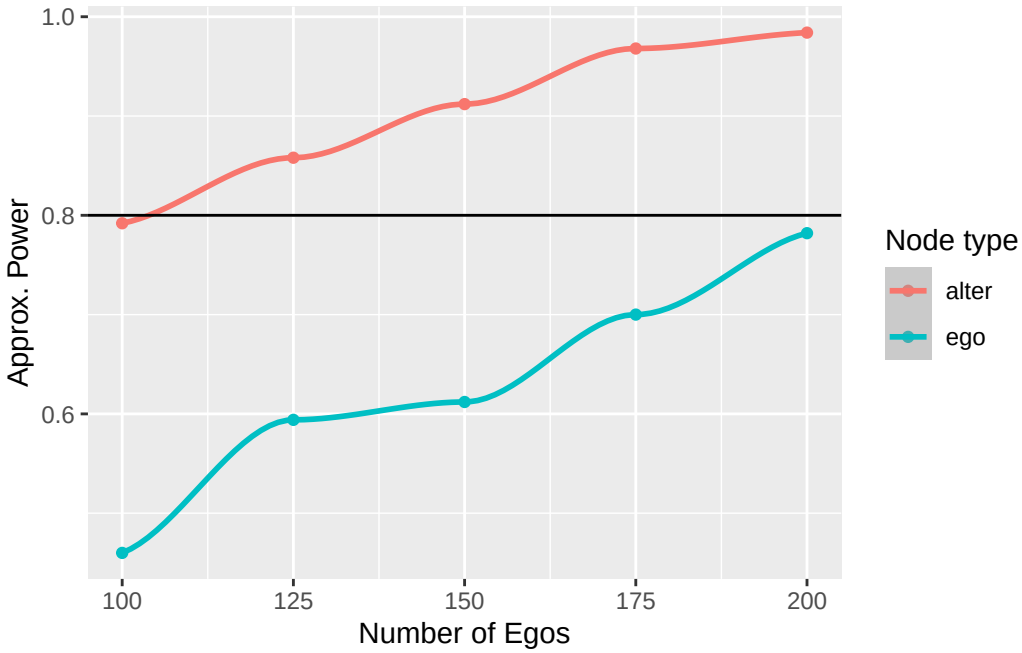
```
# Simulation function
library(survival)
sim_data_prepost <- function(n) {



  # Step 1: Sampling population of egos
  y_ego_star <- rbeta(n, beta_pars[1], beta_pars[2])
  y_ego_0    <- runif(n) < y_ego_star

  # Step 2: Simulating attrition
  todrop     <- order(runif(n))[1:(n * attrition)]
  y_ego_0    <- y_ego_0[-todrop]
  n          <- n - length(todrop)
  y_ego_star <- y_ego_star[-todrop]

  # Step 3: Simulating alter's effect. We assume the same as in
  # ego
  y_alter_star <- rbeta(n * n_a, beta_pars[1], beta_pars[2])
  y_alter_0    <- runif(n * n_a) < y_alter_star

  # Simulating post
  y_ego_1   <- runif(n) < plogis(qlogis(y_ego_star) + log(odds_h_1))
  tr_alter  <- as.integer(rep(y_ego_1, n_a))
  y_alter_1 <- runif(n * n_a) < plogis(qlogis(y_alter_star) + log(odds_h_1) * tr_alter)

  # Step 4: Computing test statistic
  y_ego_0 <- as.integer(y_ego_0)
  y_ego_1 <- as.integer(y_ego_1)
```

```r
y_alter_0 <- as.integer(y_alter_0)
y_alter_1 <- as.integer(y_alter_1)

d <- data.frame(
  y  = c(y_ego_0, y_ego_1),
  tr = c(rep(0, n), rep(1, n)),
  g  = c(1:n, 1:n)
)

res_ego   <- tryCatch(
  clogit(y ~ tr + strata(g), data = d, method = "exact"),
  error = function(e) e
  )

d <- data.frame(
  y  = c(y_alter_0, y_alter_1),
  tr = c(rep(0, n * n_a), tr_alter),
  g  = c(1:(n * n_a), 1:(n * n_a))
)

res_alter <- tryCatch(
  clogit(y ~ tr + strata(g), data = d, method = "exact"),
  error = function(e) e
  )

if (inherits(res_ego, "error") | inherits(res_alter, "error"))
  return(c(ego =  NA, alter = NA))

# Step 5: Reject?
c(
  # ego        = res_ego$p.value < alpha,
  ego        = summary(res_ego)$coefficients["tr", "Pr(>|z|)"] < alpha,
  alter      = summary(res_alter)$coefficients["tr", "Pr(>|z|)"] < alpha,
  ego_test   = coef(res_ego),
  alter_glm  = coef(res_alter)
)
```

```
}
```

```
# We always set the seed
set.seed(88)

# Making space and running!
spower <- NULL
for (s in sizes) {

  # Run the simulation for size s
  simres <- rowMeans(
    replicate(n_sims, sim_data_prepost(s)),
    na.rm = TRUE
    )

  # And store the results
  spower <- rbind(spower, simres)

}
```

```
library(ggplot2)

spowerd <- rbind(
  data.frame(size = sizes, power = spower[,"ego"], type =  "ego"),
  data.frame(size = sizes, power = spower[,"alter"], type =  "alter")
)

spowerd |>
  ggplot(aes(x = size, y = power, colour = type)) +
  geom_point() +
  geom_smooth(method = "loess", formula = y ~ x) +
  labs(x = "Number of Egos", y = "Approx. Power", colour = "Node type") +
  geom_hline(yintercept = 1 - beta_pow)
```



As shown in Chapter (**part2-power?**), we can use a linear regression model to predict sample size as a function of statistical power:

```
# Fitting the model
power_model <- glm(
  size ~ power + I(power^2),
  data = spowerd, family = gaussian(), subset = type == "alter"
)

summary(power_model)
```

```
Call:
glm(formula = size ~ power + I(power^2), family = gaussian(),
    data = spowerd, subset = type == "alter")

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    611.4      666.3   0.918    0.456
power        -1553.8     1504.7  -1.033    0.410
I(power^2)    1147.9      844.8   1.359    0.307
```

```
(Dispersion parameter for gaussian family taken to be 52.1536)

    Null deviance: 6250.00  on 4  degrees of freedom
Residual deviance:  104.31  on 2  degrees of freedom
AIC: 37.379

Number of Fisher Scoring iterations: 2
```

```r
# Predict
predict(power_model, newdata = data.frame(power = .8), type = "response") |>
  ceiling()
```

```
  1
104
```

13.3 Example 3: First difference

Now, instead of looking at a dichotomous outcome, let's evaluate what happens if the variable is continuous. The effects we are interested to identify are the ego and alter effect, γ_{ego} and γ_{alter}, respectively. Furthermore, the data generating process is

$$
\begin{aligned}
y_{itg} &= \alpha_i + \kappa_g + X_i\beta + \varepsilon_{itg} \\
y_{itg} &= \alpha_i + \kappa_g + X_i\beta + D_i^{ego}\gamma_{ego} + D_i^{alter}\gamma_{alter} + \varepsilon_{itg}
\end{aligned}
$$

Where $D_i^{ego/alter}$ is an indicator variable. Here, ego and alter behavior are correlated through a fixed effect. In other words, within each group, we are assuming that there's a shared baseline prevalence of the outcome. The main difference is that ego and alter may have different results regarding the effect size of the treatment. Another way of approaching the group-level correlation could be through an autocorrelation process, like in a spatial Autocorrelated model; nonetheless, estimating such models is computationally expensive, so we opted to use the former.

For simplicity, we assume that there is no time effect. Two essential components here, α_i and κ_g are individual and group-level unobserved fixed effects. The most straightforward approach here is to use a first difference estimator:

$$(y_{it+1g} - y_{itg}) = D_i^{ego}\gamma_{ego} + D_i^{alter}\gamma_{alter} + \varepsilon_i', \quad \varepsilon_i' = \varepsilon_{it+1g} - \varepsilon_{itg}$$

By taking the first difference, the fixed effects are removed from the equation, and we can proceed with a regular linear model.

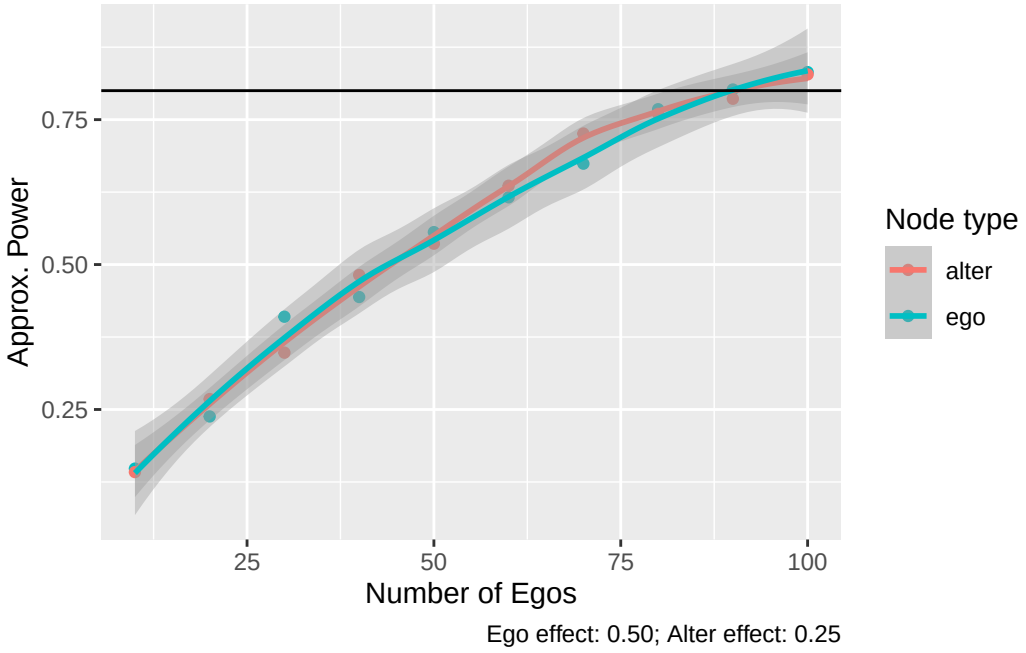
```
effect_size_ego   <- 0.5
effect_size_alter <- 0.25
sizes <- seq(10, 100, by = 10)
```

```
# Simulation function
sim_data_prepost <- function(n) {

  # Applying attrition
  n <- floor(n * (1 - attrition))

  # Step 1: Sampling fixed effects
  alpha_i <- rnorm(n * (n_a + 1))
  kappa_g <- rep(rnorm(n_a + 1), n)

  # Step 2: Generating the outcome at t = 1
  is_ego   <- rep(c(1, rep(0, n_a)), n)
  is_alter <- 1 - is_ego
  y_0      <- alpha_i + kappa_g + rnorm(n * (n_a + 1))
  y_1      <- alpha_i + kappa_g +
    is_ego * effect_size_ego +
    is_alter * effect_size_alter +
    rnorm(n * (n_a + 1))

  # Step 4: Computing test statistic
  res <- tryCatch(
    glm(I(y_1 - y_0) ~ -1 + is_ego + is_alter, family = gaussian("identity")),
    error = function(e) e
```

```
  )

  if (inherits(res, "error"))
    return(c(ego =  NA, alter = NA))

  # Step 5: Reject?
  c(
    # ego      = res_ego$p.value < alpha,
    ego        = summary(res)$coefficients["is_ego", "Pr(>|t|)"] < alpha,
    alter      = summary(res)$coefficients["is_alter", "Pr(>|t|)"] < alpha,
    coef(res)[1],
    coef(res)[2]
  )


}
```

```
# We always set the seed
set.seed(88)

# Making space and running!
spower <- NULL
for (s in sizes) {

  # Run the simulation for size s
  simres <- rowMeans(
    replicate(n_sims, sim_data_prepost(s)),
    na.rm = TRUE
    )

  # And store the results
  spower <- rbind(spower, simres)

}
```

```r
library(ggplot2)

spowerd <- rbind(
  data.frame(size = sizes, power = spower[,"ego"], type =  "ego"),
  data.frame(size = sizes, power = spower[,"alter"], type =  "alter")
)

spowerd |>
  ggplot(aes(x = size, y = power, colour = type)) +
  geom_point() +
  geom_smooth(method = "loess", formula = y ~ x) +
  labs(x = "Number of Egos", y = "Approx. Power", colour = "Node type") +
  geom_hline(yintercept = 1 - beta_pow) +
  labs(
    caption = sprintf(
      "Ego effect: %.2f; Alter effect: %.2f", effect_size_ego, effect_size_alter)
      )
```



From the inferential point of view, we could still use a demean operator to estimate individual-level effects. In particular, we would require to use the demean operator at the group level and then fit a fixed effect model to estimate individual-level parameters.

Part III

Foundations

14 Bayes' Rule

Bayes' Rule is a fundamental equation in Bayesian statistics. With it, we can reformulate inferential problems by writing probabilities in terms of known quantities. Bayes' rule can be stated as follows:

$$\mathbb{P}(X = x|Y = y) = \frac{\mathbb{P}(Y = y|X = y)\mathbb{P}(X = x)}{\mathbb{P}(Y = y)} \tag{14.1}$$

Here, we say that the conditional probability of X given Y can be expressed in terms of the conditional probability of Y given X. For example, let X be an unknown vector of parameters $\theta \in \Theta$ and Y a dataset $D \sim f(\theta)$ whose data generating process depends on the unobserved θ. As the posterior distribution of model parameters is in general, elusive, instead, we use Bayes' rule to rephrase the problem:

$$\mathbb{P}(\theta|D) = \frac{\mathbb{P}(\theta|D)\mathbb{P}(\theta)}{\mathbb{P}(D)}$$

Since the denominator of the equation does not depend on θ, we can, instead, write

$$\mathbb{P}(\theta|D) \propto \mathbb{P}(\theta|D)\mathbb{P}(\theta)$$

In the Bayesian world, the unconditional distribution of the model parameters is assumed to come from a particular distribution, whereas in the frequentist world, no distributional assumptions are made about the model parameters. The latter is then equivalent to saying that $\theta \sim \text{Uniform}(-\infty, +\infty)$; therefore, even frequentists assume something about the model parameters![1]

[1] The discussion about differences and similarities between frequentists and Bayesians has a long tradition. Bottom line, no one can say 100% that they are either-or. In rigor, frequentists say model parameters are not random but deterministic.

Bayes' rule can be derived using conditional probabilities. In particular, $\mathbb{P}(x = x|Y = y)$ is defined as $\mathbb{P}(x = x, Y = y)/Pr(Y = y)$. Likewise, $\mathbb{P}(y = y|X = x)$ is defined as $\mathbb{P}(y = y, X = x)/Pr(X = x)$, which can be re-written as $\mathbb{P}(x = x, Y = y) = \mathbb{P}(y = y|X = x)Pr(X = x)$. Replacing the last equality in the first equation, we get

$$\mathbb{P}(x = x|Y = y) = \frac{\mathbb{P}(x = x, Y = y)}{Pr(Y = y)}$$
$$\frac{\mathbb{P}(y = y|X = x)Pr(X = x)}{Pr(Y = y)}$$

15 Markov Chain

A Markov Chain is a sequence of random variables in which the conditional distribution of the n-th element only depends on $n-1$.

15.1 Metropolis Algorithm

The Metropolis Algorithm, or Metropolis MCMC, builds a Markov Chain that, under certain conditions, converges to the target distribution. The key is in accepting a proposed move from θ to θ' with probability equal to:

$$r = \min\left(1, \frac{\mathbb{P}(\theta'|D)}{\mathbb{P}(\theta|D)}\right) \tag{15.1}$$

The resulting sequence converges to the target distribution. We can prove convergence by showing that (a) the sequence is ergodic and (b) the posterior distribution matches the target distribution. Ergodicity describes three properties of a chain:

- Irreducibility: There is no zero probability of transitioning between any pair of states.
- Aperiodicity: As the term suggests, the chain has no repetitive periods/sequences.
- Non-transient: Transient refers to a chain having non-zero probability of never returning to a starting state.

The three properties are reached by any random walk based on a well-defined probability distribution, so we will focus on showing that the posterior matches the target distribution.

15.2 Metropolis-Hastings

$$\min\left(1, \frac{\mathbb{P}(d|\theta')\mathbb{P}(\theta')\mathbb{P}(\theta'|\theta)}{\mathbb{P}(d|\theta)\mathbb{P}(\theta)\mathbb{P}(\theta|\theta')}\right)$$

If the transitttion probability is symmetric, then the previous equation reduces to the Metropolis probability.

15.3 Likelihood-free MCMC

1. Initialize the algorithm with θ_0, $\theta^* = \theta_0$–the current accepted state,–and observed summary statistic $s_0 = S(D_{observed})$:
2. For $t = 1$ to T do:
    a. Draw θ_t from the proposal distribution $J(\theta_t|\theta^*)$
    b. Draw a simulated data D_t from model $M(\theta_t)$
    c. Calculate the summary statistics $s_t = S(D_t)$
    d. Accept the proposed state with probability

    If accepted, set $\theta^* = \theta_t$.

    e. Next t

16 Power and sample size

Computing power and sample size are common tasks in study design. This chapter will walk you through power analysis for network studies. First, we will start with some preliminaries regarding error types and statistical power.

16.1 Error types

One of the most important tables we'll see around is the contingency table of accept/reject the null hypothesis conditional on the true state:

	Accept H0	Reject H0
H0 is true	True positive	False negative
H1 is true	False positive	True negative

A better way, more statistically accurate version of this table would be

	Accept H0	Reject H0
H0 is true	Correct inference	Type I error
H1 is true	Type II error	Correct Inference

With $\mathbb{P}(\text{Type I error}) = \alpha$ and $\mathbb{P}(\text{Type II error}) = \beta$. This way, power can be defined as the probability of rejecting the null given the alternative is true, $\mathbb{P}(\text{Reject H0}|\text{H1 is true}) = 1 - \beta$.

16.2 Example 1: Sample size for a proportion

Let's imagine we are preparing a study in which we would like to estimate the proportion of individuals with a given status. Formally, we then say that the variable $Y \sim \text{Bernoulli}(p)$. To do so, we will need to survey n individuals and estimate such a number by taking the sample average. Furthermore, we hypothesize that under the null the proportion is $H_0 : p = p_0$.

The key here is to think about a simple rejection rule. Again, power is the probability of **rejecting the null** given that **the alternative is true**. So, to write down the equation, we need to think about acceptance and rejection regions. Let $\hat{p}$ be our estimate for the population parameter, furthermore, $\hat{p} = n^{-1} \sum_i y_i$. Our test statistic can be–and will be, most of the cases–standardized to leverage the law of large numbers; under the null, we write the following:

$$\begin{aligned} \mathbb{E}(\hat{p}) &= p_0 \\ \mathbf{Var}(\hat{p}) &= \sqrt{p_0(1-p_0)/n} \end{aligned}$$

Therefore, the statistic:

$$\frac{\hat{p} - p_0}{\sqrt{p_0(1-p_0)/n}} = \frac{\sqrt{n}(\hat{p} - p_0)}{\sqrt{p_0(1-p_0)}} \sim \text{N}(0,1)$$

Since the statistic is normally distributed, we can then say when we will reject the null. For this case, that depends on the critical value, which most of the time is defined in terms of the type I error rate. Formally, we reject the null if

$$\frac{\sqrt{n}(\hat{p} - p_0)}{\sqrt{p_0(1-p_0)}} > Z_{1-\alpha/2}$$

This is equivalent to saying that the **test statistic fell into the rejection region**. With this in hand, we can now write out the equation that we will be using for calculating the sample size. Going back to the definition of power:

$$\mathbb{P}(\text{Reject H0}|\text{H1 is true}) = 1 - \beta$$

$$\mathbb{P}\left(\frac{\sqrt{n}(\hat{p}-p_0)}{\sqrt{p_0(1-p_0)}} > Z_{1-\alpha/2}\,\middle|\, p=p_1\right) = 1-\beta$$

Observe that we cannot compute the power for all $p \neq p_0$; instead, we look at a given parameter value. A good idea is to start from one previously known or identified in other studies. The key idea here is to be able to manipulate the argument of the probability to turn it into a known distribution, for example, the normal distribution:

For a given Type I of 0.05 and power of 0.8, the required sample size can be computed as follows:

$$\begin{aligned}
1-\beta &= \mathbb{P}\left(\frac{\sqrt{n}(\hat{p}-p_0)}{\sqrt{p_0(1-p_0)}} > Z_{1-\alpha/2}\,\middle|\, p=p_1\right)\\
&= \mathbb{P}\left(\frac{\sqrt{n}(\hat{p}-p_0)}{\sqrt{p_0(1-p_0)}} < Z_{\alpha/2}\,\middle|\, p=p_1\right)\\
&= \mathbb{P}\left(\frac{\sqrt{n}(\hat{p}-p_0)}{\sqrt{p_1(1-p_1)}} < \frac{Z_{\alpha/2}\sqrt{p_0(1-p_0)}}{\sqrt{p_1(1-p_1)}}\,\middle|\, p=p_1\right)\\
&= \mathbb{P}\left(\frac{\sqrt{n}(\hat{p}-p_0+p_0-p_1)}{\sqrt{p_1(1-p_1)}} < \frac{Z_{\alpha/2}\sqrt{p_0(1-p_0)}+\sqrt{n}(p_0-p_1)}{\sqrt{p_1(1-p_1)}}\,\middle|\, p=p_1\right)\\
&= \mathbb{P}\left(\frac{\sqrt{n}(\hat{p}-p_1)}{\sqrt{p_1(1-p_1)}} < \frac{Z_{\alpha/2}\sqrt{p_0(1-p_0)}+\sqrt{n}(p_0-p_1)}{\sqrt{p_1(1-p_1)}}\,\middle|\, p=p_1\right)\\
&= \Phi\left(\frac{Z_{\alpha/2}\sqrt{p_0(1-p_0)}+\sqrt{n}(p_0-p_1)}{\sqrt{p_1(1-p_1)}}\,\middle|\, p=p_1\right)
\end{aligned}$$

The last equality follows from the quantity $\frac{\sqrt{n}(\hat{p}-p_1)}{\sqrt{p_1(1-p_1)}}$ distributing standard normal. We can now take the inverse of the cumulative distribution function (cdf) to isolate the sample size n:

$$\begin{aligned}
\Phi^{-1}(1-\beta) &= \frac{Z_{\alpha/2}\sqrt{p_0(1-p_0)}+\sqrt{n}(p_0-p_1)}{\sqrt{p_1(1-p_1)}}\\
Z_{1-\beta}\sqrt{p_1(1-p_1)} &= Z_{\alpha/2}\sqrt{p_0(1-p_0)}+\sqrt{n}(p_0-p_1)
\end{aligned}$$

$$\frac{\left(Z_{1-\beta}\sqrt{p_1(1-p_1)} - Z_{\alpha/2}\sqrt{p_0(1-p_0)}\right)^2}{(p_0-p_1)^2} = n$$

Therefore, for the parameters $(1-\beta, \alpha, p_0, p_1) = (0.8, 0.05, 0.5, 0.6)$, the required sample size is $193.8473 \sim 194$.

16.3 Example 2: Sample size for a proportion (vis)

Now, what happens if the model we are planning to estimate does not have a close form? If analytical solutions are not available, simulations can be an excellent alternative to save the day. Let's re-do the sample size calculation using simulations.

The procedure to compute sample size based on simulations is computationally intensive. The concept is straightforward, pick a set of best guesses for sample size, and for each one of them, simulate the system to estimate power. Now, for a given value of n, we:

1. Simulate a sample of size n under the alternative.
2. Compute the test statistic corresponding to the null.
3. Accept or reject accordingly to the selected α, and store the result.
4. Repeat steps 1-3 many times. The obtained average is the corresponding power.

When running simulations, it is convenient to write a function for the data generating process. In our case, the function will be called `sim_fun`. The following lines of code achieve our goal: approximate power by simulating 10,000 experiments for each sample size candidate:

```
# Model parameters
p0        <- .5
p1        <- .6
betapower <- 1 - 0.8
alpha     <- 0.05
nsims     <- 10000

# Step 1: Simulate the data under H1
```

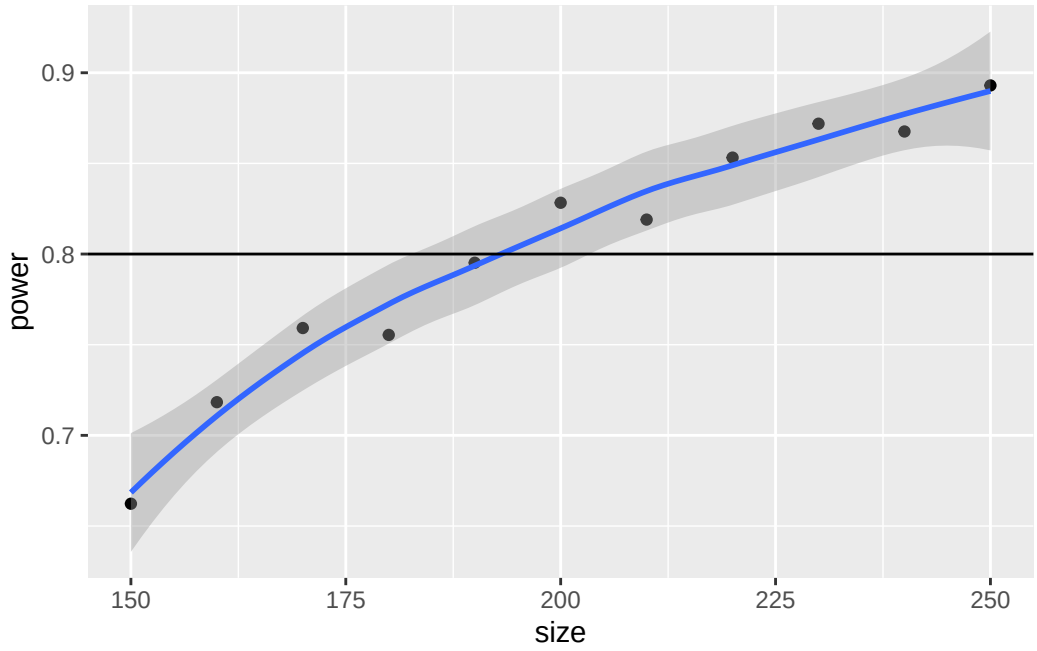
```r
z_one_minus_alpha_half <- qnorm(1 - alpha / 2)
sim_fun <- function(n) {

    # Generating the data
    y <- as.integer(runif(n) < p1)
    phat <- mean(y)

    # Accept or reject?
    sqrt(n) * (phat - p0) / sqrt(p0 * (1 - p0)) >
        z_one_minus_alpha_half

}

# Step 2: For an array of n, simulate multiple experiments
n_seq <- seq(from = 150, to = 250, by = 10)

simulations <- NULL
set.seed(12312)
for (n in n_seq) {

    # Run the nsims experiments
    res <- replicate(nsims, sim_fun(n))

    # Compute power and store the value
    simulations <- rbind(
        simulations,
        data.frame(size = n, power = mean(res))
    )
}

# Finding out what is the closets value
best <- which.min(
    abs((1 - betapower) - simulations$power)
    )

simulations[best,,drop=FALSE]
```

```
  size  power
5  190 0.7952
```

Let's visualize the power curve we generate from this simulation:

```
library(ggplot2)
ggplot(simulations, aes(x = size, y = power)) +
    geom_point() +
    geom_smooth() +
    geom_hline(yintercept = 1 - betapower)
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



Alternatively, we can fit a linear regression model where we predict power as a function of sample size using linear and quadratic effects:

$$n = \theta_0 + \theta_1(1-\beta) + \theta_2(1-\beta)^2$$

```r
# Fitting the model
power_model <- glm(
  size ~ power + I(power^2),
  data = simulations, family = gaussian()
)

# Printing the results
summary(power_model)
```

```
Call:
glm(formula = size ~ power + I(power^2), family = gaussian(),
    data = simulations)

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    632.5      232.9   2.715  0.02644 *
power        -1590.3      598.1  -2.659  0.02885 *
I(power^2)    1301.0      381.6   3.410  0.00923 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 34.83159)

    Null deviance: 11000.00  on 10  degrees of freedom
Residual deviance:   278.65  on  8  degrees of freedom
AIC: 74.769

Number of Fisher Scoring iterations: 2
```

```r
# Predict
predict(power_model, newdata = data.frame(power = .8), type = "response") |>
  ceiling()
```

```
  1
193
```

According to our simulation study, the closest to our 80% power is using a sample size equal to 193, which is very close to the analytical solution of 194.

As a final comment for this example, remember that the more simulations the better.

Part IV

Appendix

17 Datasets

17.1 SNS data

17.1.1 About the data

- This data is part of the NIH Challenge grant # RC 1RC1AA019239 "Social Networks and Networking That Puts Adolescents at High Risk".
- In general terms, the SNS's goal was(is) "Understand the network effects on risk behaviors such as smoking initiation and substance use".

17.1.2 Variables

The data has a *wide* structure, which means that there is one row per individual, and that dynamic attributes are represented as one column per time.

- `photoid` Photo id at the school level (can be repeated across schools).
- `school` School id.
- `hispanic` Indicator variable that equals 1 if the indivual ever reported himself as hispanic.
- `female1`, ..., `female4` Indicator variable that equals 1 if the individual reported to be female at the particular wave.
- `grades1`,..., `grades4` Academic grades by wave. Values from 1 to 5, with 5 been the best.
- `eversmk1`, ..., `eversmk4` Indicator variable of ever smoking by wave. A one indicated that the individual had smoked at the time of the survey.

- `everdrk1`, …, `everdrk4` Indicator variable of ever drinking by wave. A one indicated that the individual had drink at the time of the survey.
- `home1`, …, `home4` Factor variable for home status by wave. A one indicates home ownership, a 2 rent, and a 3 a "I don't know".

During the survey, participants were asked to name up to 19 of their school friends:

- `sch_friend11`, …, `sch_friend119` School friends nominations (19 in total) for wave 1. The codes are mapped to the variable `photoid`.
- `sch_friend21`, …, `sch_friend219` School friends nominations (19 in total) for wave 2. The codes are mapped to the variable `photoid`.
- `sch_friend31`, …, `sch_friend319` School friends nominations (19 in total) for wave 3. The codes are mapped to the variable `photoid`.
- `sch_friend41`, …, `sch_friend419` School friends nominations (19 in total) for wave 4. The codes are mapped to the variable `photoid`.

References

Admiraal, Ryan, and Mark S Handcock. 2006. "Sequential Importance Sampling for Bipartite Graphs with Applications to Likelihood-Based Inference." Department of Statistics, University of Washington.

Bojanowski, Michał. 2023. *intergraph: Coercion Routines for Network Data Objects*. https://mbojan.github.io/intergraph/.

Brooks, Steve, Andrew Gelman, Galin Jones, and Xiao-Li Meng. 2011. *Handbook of Markov Chain Monte Carlo*. CRC press.

Csárdi, Gábor, Tamás Nepusz, Vincent Traag, Szabolcs Horvát, Fabio Zanini, Daniel Noom, and Kirill Müller. 2024. *igraph: Network Analysis and Visualization in r*. https://doi.org/10.5281/zenodo.7682609.

Efron, Bradley, and Robert J Tibshirani. 1994. *An Introduction to the Bootstrap*. CRC press.

Geyer, Charles J., and Elizabeth A. Thompson. 1992. "Constrained Monte Carlo Maximum Likelihood for Dependent Data." *Journal of the Royal Statistical Society. Series B (Methodological)* 54 (3): 657–99. http://www.jstor.org/stable/2345852.

Handcock, Mark S., David R. Hunter, Carter T. Butts, Steven M. Goodreau, Pavel N. Krivitsky, and Martina Morris. 2018. *Ergm: Fit, Simulate and Diagnose Exponential-Family Models for Networks*. The Statnet Project (http://www.statnet.org). https://CRAN.R-project.org/package=ergm.

Hunter, David R, Steven M Goodreau, and Mark S Handcock. 2008. "Goodness of Fit of Social Network Models." *Journal of the American Statistical Association* 103 (481): 248–58. https://doi.org/10.1198/016214507000000446.

Hunter, David R., Mark S. Handcock, Carter T. Butts, Steven M. Goodreau, and Martina Morris. 2008. "ergm : A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks." *Journal of Statistical Software* 24 (3). https://doi.org/10.18637/jss.v024.i03.

Lazega, Emmanuel, and Tom AB Snijders. 2015. *Multilevel Network Analysis for the Social Sciences: Theory, Methods and Applications*. Vol. 12. Springer.

Leifeld, Philip. 2013. "texreg: Conversion of Statistical Model Output in R to LaTeX and HTML Tables." *Journal of Statistical Software* 55 (8): 1–24. https://doi.org/10.18637/jss.v055.i08.

Lusher, Dean, Johan Koskinen, and Garry Robins. 2012. *Exponential Random Graph Models for Social Networks: Theory, Methods, and Applications*. Cambridge University Press.

Matloff, Norman. 2011. *The Art of r Programming: A Tour of Statistical Software Design*. No Starch Press.

Morris, Martina, Mark Handcock, and David Hunter. 2008. "Specification of Exponential-Family Random Graph Models: Terms and Computational Aspects." *Journal of Statistical Software, Articles* 24 (4): 1–24. https://doi.org/10.18637/jss.v024.i04.

Plummer, Martyn, Nicky Best, Kate Cowles, and Karen Vines. 2006. "CODA: Convergence Diagnosis and Output Analysis for MCMC." *R News* 6 (1): 7–11. https://journal.r-project.org/archive/.

R Core Team. 2023. *Foreign: Read Data Stored by 'Minitab', 's', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', ...* https://svn.r-project.org/R-packages/trunk/foreign/.

———. 2024. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Ripley, Ruth M., Tom AB Snijders, Paulina Preciado, and Others. 2011. "Manual for RSIENA." *University of Oxford: Department of Statistics, Nuffield College*, no. 2007. https://www.uni-due.de/hummell/sna/R/RSiena%7B/_%7DManual.pdf.

Sarkar, Deepayan, and Felix Andrews. 2022. *latticeExtra: Extra Graphical Utilities Based on Lattice*. http://latticeextra.r-forge.r-project.org/.

Snijders, Tom A B, and Stephen P Borgatti. 1999. "Non-Parametric Standard Errors and Tests for Network Statistics." *Connections* 22 (2): 1–10. https://insna.org/PDF/Connections/v22/1999_I-2_61-70.pdf.

Snijders, Tom A B, Gerhard G. van de Bunt, and Christian E G Steglich. 2010. "Introduction to stochastic actor-based models for network dynamics." *Social Networks* 32 (1): 44–60. https://doi.org/10.1016/j.socnet.2009.02.004.

SNIJDERS, TOM A. B. 2010. "Conditional Marginalization for Exponential Random Graph Models." *The Journal of Mathematical Sociology* 34 (4): 239–52. https://doi.org/10.1080/0022250X.2010.485707.

Snijders, Tom AB. 2002. "Markov Chain Monte Carlo Estimation of Exponential Random Graph Models." *Journal of Social Structure* 3.

Ushey, Kevin, Jim Hester, and Robert Krzyzanowski. 2021. *Rex: Friendly Regular Expressions*. https://github.com/kevinushey/rex.

Wang, Peng, Ken Sharpe, Garry L. Robins, and Philippa E. Pattison. 2009. "Exponential

Random Graph (p*) Models for Affiliation Networks." *Social Networks* 31 (1): 12–25. https://doi.org/https://doi.org/10.1016/j.socnet.2008.08.002.

Wickham, Hadley. 2023. *Stringr: Simple, Consistent Wrappers for Common String Operations*. https://stringr.tidyverse.org.

Wickham, Hadley, and Jennifer Bryan. 2023. *Readxl: Read Excel Files*. https://readxl.tidyverse.org.

Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *Dplyr: A Grammar of Data Manipulation*. https://dplyr.tidyverse.org.

Wickham, Hadley, Jim Hester, and Jennifer Bryan. 2024. *Readr: Read Rectangular Text Data*. https://readr.tidyverse.org.

Wickham, Hadley, Davis Vaughan, and Maximilian Girlich. 2024. *Tidyr: Tidy Messy Data*. https://tidyr.tidyverse.org.