

Applied Network Science with R

George G. Vega Yon

2022-06-24

Contents

Chapter 1

About

1.0.1 The Book

Statistical methods for networked systems are present in most disciplines. Nonetheless, the language differences between disciplines, many methods developed to study specific types of problems can be helpful outside of their original context.

This project began as a part of a workshop that took place at USC's [Center for Applied Network Analysis](#). Now, it is a personal project that I use 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 when developing these topics.

In general, we will, besides R itself, we will be using R studio and the following R packages: dplyr for data management, stringr for data cleaning, and of course igraph, netdiffuseR (a bit of a bias here), and statnet for our neat network analysis.¹

You can access the book's source code at <https://github.com/gvegayon/appliedsnar>.

1.0.2 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 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."

¹Some of you may be wondering "what about ggplot2 and friends? What about [tidyverse](#)", well, my short answer is I jumped into R before all of that was that popular. When I started, plots were all about [lattice](#), and after a couple of years on that, about base R graphics. What I'm saying is that so far, I have not found a compelling reason to leave my "old practices" and embrace all the tidyverse movement (religion?).

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

Part I

Applications

Chapter 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 a wide range of 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 levels of inter-discipline collaboration, if you allow me to say) of scientific advances happening in a somewhat isolated fashion, the potential of cross-pollination between disciplines within network science is immense.

This book is an attempt 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.

Chapter 3

R Basics

3.1 What is R

A good reference book for both new and advanced user is “[The Art of R programming](#)” (Matloff 2011)¹

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 use the `devtools` R package to install a package from some remote repository other than CRAN, here is a couple of examples:

```
# 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 second 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 command line as some packages need extra configuration to be installed. But we won’t need to look at it now.

3.3 Prerequisites

To install R just follow the instructions available at <http://cran.r-project.org>

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

RStudio is the most popular Integrated Development Environment (IDE) for R that is developed by the company of the same name. While having RStudio is not a requirement for using netdiffuseR, it is highly recommended.

To get RStudio just visit <https://www.rstudio.com/products/rstudio/download/>.

3.4 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 2 are equivalent:

```
a <- 1
a = 1
```

Historically the assign operator is the most common used.

2. R has several type of objects, the most basic structures in R are vectors, matrix, list, data.frame. Here is an example 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

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

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

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

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

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

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

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

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

(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

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

## $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

# 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 to its components using indexing:

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

## [1] 1 2 3

a_string_vec[3] # Third element

## [1] "netdiffuseR"

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

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5

a_matrix[,3] # Third column

## [1] 7 8 9

a_matrix[3,] # Third row

## [1] 3 6 9

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

## [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"

```

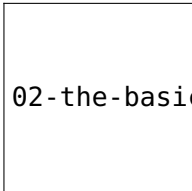
5. R has a very nice set of pseudo random number generation functions. In general, distribution functions have the following name structure:

- 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:

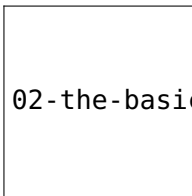
```
# To ensure reproducibility
set.seed(1231)

# 100,000 Unif(0,1) numbers
x <- runif(1e5)
hist(x)
```



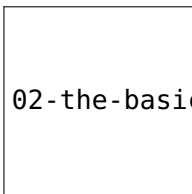
02-the-basics_files/figure-latex/random-numbers-1.pdf

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



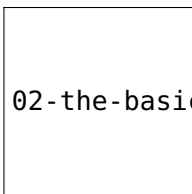
02-the-basics_files/figure-latex/random-numbers-2.pdf

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



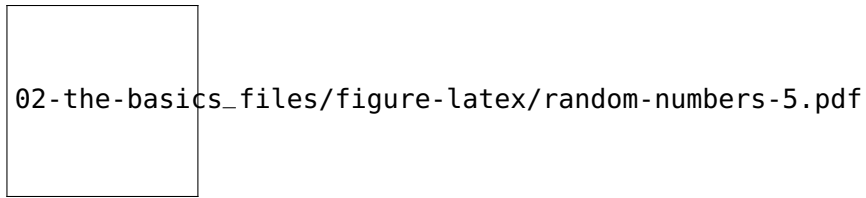
02-the-basics_files/figure-latex/random-numbers-3.pdf

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



02-the-basics_files/figure-latex/random-numbers-4.pdf

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



More distributions 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](#).

For this book, we need the following

R Core Team ([2017b](#))

1. Install R from CRAN: <https://www.r-project.org/>
2. (optional) Install Rstudio: <https://rstudio.org>

While I find RStudio extremely useful, it is not necessary to use it with R.

Chapter 4

Network Nomination Data

You can download the data for this chapter [here](#).

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 Francois 2017). In the case of binary files, like Stata, Octave, or SPSS files, you can use the R package [foreign](#) (R Core Team 2017a). If your data is formatted as Microsoft spreadsheets, the [readxl](#) R package (Wickham and Bryan 2017) is the alternative to use. In our case, the data for this session is in Stata format:

```
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

Now suppose that we want to create a unique id using the school and photo id. In this case, since both variables are numeric, a good way of doing it is to encode the id. For example, the last three numbers are the photoid and the first ones are the school id. To do this, we need to take into account the range of the variables:

```
(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. 2017) and `magrittr` (Bache and Wickham 2014)] (the pipe operator, `%>%`) to create this variable, and we will call it... `id` (mind blowing, right?):

```
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
```

```
library(magrittr)
```

```
(dat %<>% mutate(id = school*10000 + photoid)) %>%
  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 three lines of code! What is that `%>%`? Well, that's the [pipe operator](#), and it is an appealing way of writing nested function calls. In this case, instead of writing something like:

```
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 and Henry 2017) and `stringr` (Wickham 2017). We will use the first, `tidyr`, to reshape the data. The second, `stringr`, will help us processing strings using *regular expressions*¹.

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

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

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

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

```
dat
```

```
## # A tibble: 2,164 x 100
##   photoid school hispanic female1 female2 female3 female4 grades1 grades2
```

¹Please refer to the help file `?'regular expression'` in R. The R package `rex` (Ushey, Hester, and Krzyzanowski 2017) is a very nice 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.

```
##      <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
## # ... with 2,154 more rows, and 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>, sch_friend19 <int>, ...
```

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 later, 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
## # ... with 2,154 more rows, and 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>, sch_friend23 <int>, sch_friend24 <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
## # ... with 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 NAs, 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
## # ... with 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
## # ... with 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 lower case 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* (Csardi and Nepusz 2006). 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 since, by doing so, you are telling the function what ids that you are expecting to find. Using the original dataset, we will create a data frame 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(., vertices = vertex_attrs): Some vertex names in edge l
```

Ups! It seems that individuals are making nominations to 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:

```
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 832e763 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 832e763 (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 there 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:

```
list.vertex.attributes(ig_year1)
```

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

```
list.edge.attributes(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:

```
# 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 measures² for each vertex and store them in the igraph object itself:

```
# 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)$closeness <- closeness(ig_year1_111, mode = "total")
V(ig_year1_111)$betweenness <- 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

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

²For more information about the different centrality measurements, please take a look at the “Centrality” article on [Wikipedia](#).

```
# 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)
  )
})
```

```
stats_degree
```

```
##      indegree outdegree  closeness  betweeness
## 2.5%         0         0 0.0005915148 0.0000000000
## 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)
  )
```

Now we will compute some statistics at the graph level:

```
cbind(
  size      = vcount(ig_year1_111),
  nedges    = ecoun(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 `?triadic_census`. Moreover, we can normalize the triadic object by its sum instead of looking at raw counts. That way, we get proportions instead³

```
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.
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:

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

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`.⁴
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 NAs with the number three so that our vector `col_hispanic` now ranges from one to three with no NAs 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]
)
```

```
##      original colors
## [1,] "1"        "tomato"
## [2,] "1"        "tomato"
## [3,] "0"        "steelblue"
## [4,] "1"        "tomato"
## [5,] "1"        "tomato"
```

⁴Figuring 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.

```
## [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
)
```

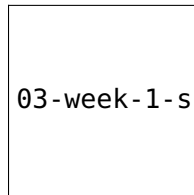


Figure 4.2: Friends network in time 1 for school 111.

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
)
```

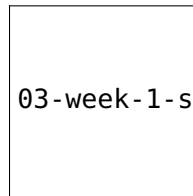


Figure 4.3: Friends network in time 1 for school 111. The graph excludes isolates.

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.

```
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"
)

```

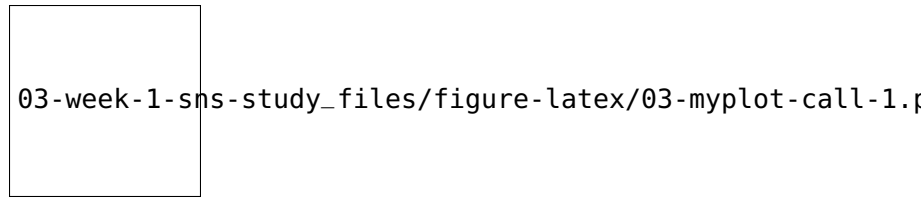


Figure 4.4: All 5 schools in time 1. Again, the graphs exclude isolates.

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 $2 \times 3 = 6$ 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`

```
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
## # ... with 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")
```


Chapter 5

Exponential Random Graph Models

I strongly suggest reading the vignette included 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](#))

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

$$\Pr(\mathbf{Y} = \mathbf{y} | \theta, \mathcal{Y}) = \frac{\exp\{\theta^T \mathbf{g}(\mathbf{y})\}}{\kappa(\theta, \mathcal{Y})}, \quad \mathbf{y} \in \mathcal{Y}$$

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 (??) 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,

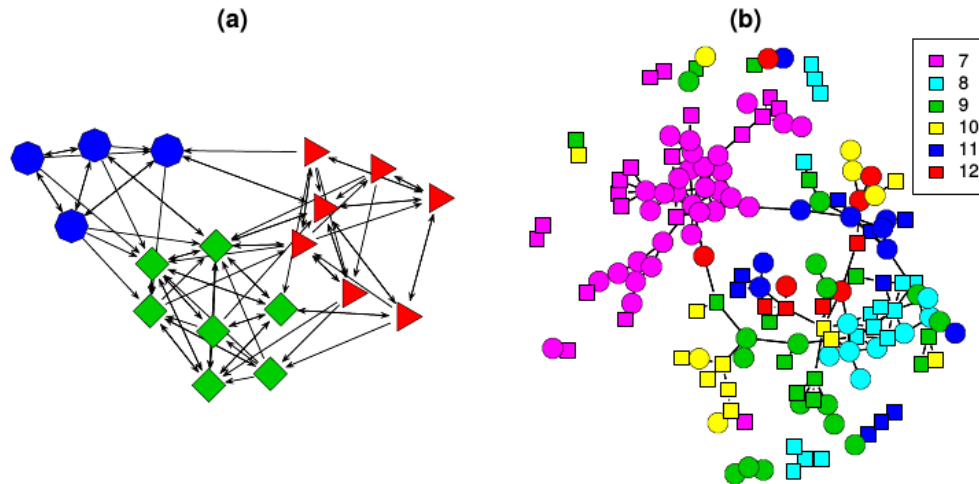


Figure 1: The (a) *samplike* and (b) *faux.mesa.high* networks described in Section 2. The values of nodal covariates may be indicated using various colors, shapes, and labels of nodes.

Figure 5.1: Source: Hunter et al. (2008)

$$\kappa(\theta, \mathcal{Y}) = \sum_{\mathbf{y} \in \mathcal{Y}} \exp \{ \theta^T \mathbf{g}(\mathbf{y}) \}$$

, is the normalizing factor that ensures that equation (??) 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)

Recent developments include new forms of dependency structures to take into account 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)

5.1 A naïve example

In the simplest case, *ergm* is equivalent to a logistic regression

```
library(ergm)
```

```
## Loading required package: network
```

```
##
```

```
## 'network' 1.17.2 (2022-05-20), 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

##
## 'ergm' 4.2.1 (2022-05-10), 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.
```

```
data("sampson")
```

```
samplike
```

```
## Network attributes:
##   vertices = 18
##   directed = TRUE
##   hyper = FALSE
##   loops = FALSE
##   multiple = FALSE
##   total edges= 88
##     missing edges= 0
##     non-missing edges= 88
##
## Vertex attribute names:
##   cloisterville group vertex.names
##
## Edge attribute names:
##   nominations

y <- sort(as.vector(as.matrix(samplike)))[-c(1:18)]
glm(y~1, family=binomial("logit"))
```

```
##
## 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(samplike ~ edges)

## Starting maximum pseudolikelihood estimation (MPLE):
## Evaluating the predictor and response matrix.
## Maximizing the pseudolikelihood.
## Finished MPLE.
## Stopping at the initial estimate.
## Evaluating log-likelihood at the estimate.
##
## Call:
## ergm(formula = samplike ~ edges)
##
## Maximum Likelihood Coefficients:
##   edges
## -0.9072

pr <- mean(y)
log(pr) - log(1-pr) # Logit function

## [1] -0.9071582

qlogis(pr)

## [1] -0.9071582

```

5.2 Estimation of ERGMs

The ultimate goal is to perform statistical inference on the proposed model. In a *standard* setting, we would be able to 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 been able to compute the log-likelihood of the model, which in ERGMs can be challenging.

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 it, 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 that we can approximate the expectation of normalizing constant ratios using the law of large numbers. In particular, the following:

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

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

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

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 it's 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.

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

5.3 The `ergm` package

The `ergm` R package (Handcock et al. 2017)

From the previous section:¹

```
library(igraph)
library(magrittr)
library(dplyr)

load("03.rda")
```

In this section we will use the `ergm` package (from the `statnet` suit of packages (Handcock et al. 2016)) suit, and the `intergraph` (Bojanowski 2015) package. The latter provides functions to go back and forth between `igraph` and `network` objects from the `igraph` and `network` packages respectively²

```
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 of some functions in the `igraph` package. When the `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.edg
get.vertex.attribute, is.bipartite, is.directed, list.edge.attributes, list.vertex.attri
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"
```

¹You can download the 03.rda file from [this link](#).

²Yes, the classes have the same name as the packages.

```
## Starting maximum pseudolikelihood estimation (MPLE):
## Evaluating the predictor and response matrix.
## Maximizing the pseudolikelihood.
## Finished MPLE.
## Stopping at the initial estimate.
## 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 thought 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 d7c9950 (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 on 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 `avareges`:

```
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
}
```

5.4 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 iteration, 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 compare³

```
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
  )
)
```

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)

³Notice 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.

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

```
library(texreg)
```

```
## Version: 1.38.6
## Date: 2022-04-06
## 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").
##
## Attaching package: 'texreg'
##
## The following object is masked from 'package:magrittr':
##
## extract
```

```
screenreg(list(ans0, ans1, ans2))
```

```
## Warning: This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing it with
## This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing it with version 4
## This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing it with version 4
```

```
##
## =====
##               Model 1           Model 2           Model 3
## -----
## edges          -5.63 ***          -5.49 ***          -5.60 ***
##                (0.05)             (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           4.10 ***                   1.75 ***
##                (0.07)                   (0.14)
## balance                                0.01 ***
##                                (0.00)
## -----
## AIC             -40017.80          -37511.87          -39989.59
## BIC             -39967.46          -37471.60          -39929.18
## Log Likelihood   20013.90          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

	Model 1	Model 2	Model 3
edges	−5.63*** (0.05)	−5.49*** (0.06)	−5.60*** (0.06)
nodematch.hispanic	0.22*** (0.04)	0.30*** (0.05)	0.22*** (0.04)
nodematch.female1	0.87*** (0.04)	1.17*** (0.05)	0.87*** (0.04)
nodematch.eversmk1	0.33*** (0.04)	0.45*** (0.04)	0.34*** (0.04)
mutual	4.10*** (0.07)		1.75*** (0.14)
balance			0.01*** (0.00)
AIC	−40017.80	−37511.87	−39989.59
BIC	−39967.46	−37471.60	−39929.18
Log Likelihood	20013.90	18759.94	20000.79

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

Table 5.1: Statistical models

latter to see how it looks like here:

```
library(texreg)
texreg(list(ans0, ans1, ans2))
```

```
## Warning: This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing it with
## This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing it with version 4
## This object was fit with 'ergm' version 4.1.2 or earlier. Summarizing it with version 4
```

5.5 Model Goodness-of-Fit

In raw terms, once each chain has reach stationary distribution, we can say that there are no problems with autocorrelation and that each sample point is iid. This implies that, since we are running the model with more than 1 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 one model which did best, let's take a look at it's GOF statistics. First, lets see how the MCMC did. For this we can use the `mcmc.diagnostics` function including in the package. This function is actually a wrapper of a couple of functions from the coda package (Plummer et al. 2006) which is called upon the `$sample` object which holds the *centered* statistics from the sampled networks. This last point is important to consider since at first look it can be confusing to look at the `$sample` object since it neither matches the observed

statistics, nor the coefficients.

When calling the function `mcmc.diagnostics(ans0, centered = FALSE)`, you will see a lot of output including a couple of plots showing the trace and posterior distribution of the *uncentered* statistics (`centered = FALSE`). In the next code chunks we 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:

```
summary(sample_uncentered)

##
## Iterations = 1769472:10944512
## Thinning interval = 65536
## Number of chains = 4
## Sample size per chain = 141
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##              Mean      SD Naive SE Time-series SE
## edges          2485 60.26   2.5372           3.753
```



```
## nodematch.hispanic 1838 51.25 2.1578 3.662
## nodematch.female1 1888 52.78 2.2224 3.779
## nodematch.eversmk1 1759 50.82 2.1400 3.072
## mutual 493 23.40 0.9855 1.967
##
## 2. Quantiles for each variable:
##
##          2.5% 25% 50% 75% 97.5%
## edges      2373 2444 2482 2530 2612
## nodematch.hispanic 1736 1803 1839 1872 1947
## nodematch.female1 1791 1851 1885 1923 1993
## nodematch.eversmk1 1662 1725 1758 1794 1858
## mutual      449 476 493 509 537
```

2. Cross correlation:

```
coda::crosscorr(sample_uncentered)
```

```
##          edges nodematch.hispanic nodematch.female1
## edges      1.0000000 0.8657369 0.8851587
## nodematch.hispanic 0.8657369 1.0000000 0.7713632
## nodematch.female1 0.8851587 0.7713632 1.0000000
## nodematch.eversmk1 0.8445651 0.7122693 0.7572735
## mutual      0.7726517 0.6801783 0.7482026
##          nodematch.eversmk1 mutual
## edges      0.8445651 0.7726517
## nodematch.hispanic 0.7122693 0.6801783
## nodematch.female1 0.7572735 0.7482026
## nodematch.eversmk1 1.0000000 0.6873242
## mutual      0.6873242 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 iid (no Markov property). A way out to solve this is *thinning* the sample.

```
coda::autocorr(sample_uncentered)[[1]]
```

```
## , , edges
##
##          edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0      1.000000000 0.861920590 0.90235072 0.86215333
## Lag 65536 0.415060923 0.326775063 0.43751588 0.38274418
## Lag 327680 0.063993999 0.002238453 0.09094189 0.05143792
## Lag 655360 0.002497326 -0.105210070 -0.02414091 0.00143358
## Lag 3276800 0.026845190 0.068616366 0.03686125 0.03652383
```

```

##                                mutual
## Lag 0                        0.785264416
## Lag 65536                    0.428519050
## Lag 327680                   0.074020671
## Lag 655360                   0.009422505
## Lag 3276800                  0.018126669
##
## , , nodematch.hispanic
##
##                                edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0                        0.86192059          1.000000000          0.76137201          0.74623272
## Lag 65536                    0.32680263          0.336764054          0.30353156          0.32690588
## Lag 327680                   0.05778076          0.004465856          0.07267341          0.03757479
## Lag 655360                   0.07704457          0.024226503          0.03252125          0.08420548
## Lag 3276800                 -0.02970399          0.021278122         -0.02753467         -0.03018601
##
##                                mutual
## Lag 0                        0.70578514
## Lag 65536                    0.35558587
## Lag 327680                   0.05282736
## Lag 655360                   0.08176601
## Lag 3276800                 -0.07743174
##
## , , nodematch.female1
##
##                                edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0                        0.902350724          0.76137201          1.000000000          0.77769826
## Lag 65536                    0.453418914          0.37756721          0.51290498          0.41954866
## Lag 327680                   0.055464012         -0.01058737          0.09841770          0.04272154
## Lag 655360                   0.009910833         -0.06123858         -0.03186870          0.04679847
## Lag 3276800                 0.004163166          0.04057544          0.01548719         -0.01288236
##
##                                mutual
## Lag 0                        0.76981085
## Lag 65536                    0.46327442
## Lag 327680                   0.03629824
## Lag 655360                   0.01987496
## Lag 3276800                 -0.00949882
##
## , , nodematch.eversmk1
##
##                                edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0                        0.86215333          0.746232721          0.77769826          1.000000000
## Lag 65536                    0.37539678          0.297591397          0.41717478          0.448697559
## Lag 327680                   0.02105523         -0.040132752          0.03760486          0.019124328

```

```
## Lag 655360 0.04566425      0.003387581      0.04761067      -0.006388743
## Lag 3276800 0.05048735      0.084790008      0.07108989      0.045582057
##
## mutual
## Lag 0      0.7053009595
## Lag 65536  0.4020746950
## Lag 327680 0.0183308894
## Lag 655360 0.0840948296
## Lag 3276800 0.0009713556
##
## , , mutual
##
## edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0      0.78526442      0.70578514      0.769810849      0.70530096
## Lag 65536  0.50645801      0.44741607      0.532817503      0.47751208
## Lag 327680 0.12979152      0.06061696      0.147380566      0.10930214
## Lag 655360 -0.06393205      -0.13217821     -0.008121728     -0.03814393
## Lag 3276800 -0.01707605      0.03244214     -0.023750630     0.02781638
##
## mutual
## Lag 0      1.000000000
## Lag 65536  0.580271013
## Lag 327680 0.091309576
## Lag 655360 -0.003521212
## Lag 3276800 -0.025558756
```

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:

```
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
##      -0.7115      -1.7204      -0.1841      0.6952
## mutual
##      -1.2891
```

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 simulatenously:

```
coda::gelman.diag(sample_uncentered)
```

```
## Potential scale reduction factors:
##
##               Point est. Upper C.I.
## edges                1.03      1.10
## nodematch.hispanic    1.03      1.10
## nodematch.female1     1.05      1.14
## nodematch.eversmk1    1.04      1.12
## mutual                1.05      1.14
##
## Multivariate psrf
##
## 1.05
```

As a rule of thumb, values that are 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 2016), the function will override the default plots used by `coda::plot.mcmc` and use `lattice` instead, creating a nicer looking plots. The next code chunk calls the `mcmc.diagnostic` function, but we suppress the rest of the output (see figure ??).

```
# [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 called 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 properties. Besides the statistics that define our ERGM, the `gof` function's default behavior show GOF for:

- 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	1.43	5		0.00
##	idegree1	34	2	8.47	16		0.00
##	idegree2	37	13	22.53	33		0.00
##	idegree3	48	29	42.25	60		0.38
##	idegree4	37	45	57.78	76		0.00
##	idegree5	47	44	66.51	88		0.02
##	idegree6	42	44	64.18	79		0.00
##	idegree7	39	40	53.58	68		0.00
##	idegree8	35	23	39.58	53		0.50
##	idegree9	21	18	27.43	40		0.28
##	idegree10	12	9	16.38	23		0.34
##	idegree11	19	1	8.82	16		0.00
##	idegree12	4	0	4.63	12		1.00
##	idegree13	7	0	2.12	7		0.02
##	idegree14	6	0	1.26	5		0.00
##	idegree15	3	0	0.44	2		0.00
##	idegree16	4	0	0.37	2		0.00
##	idegree17	3	0	0.14	1		0.00
##	idegree18	3	0	0.06	1		0.00
##	idegree19	2	0	0.02	1		0.00

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

```
## 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  1.56  5      0.10
## odegree1   28   2  8.23 17      0.00
## odegree2   45  11 22.52 33      0.00
## odegree3   50  23 40.11 56      0.10
## odegree4   54  43 59.27 76      0.52
## odegree5   62  50 67.39 93      0.48
## odegree6   40  45 63.79 79      0.00
## odegree7   28  32 54.74 73      0.00
## odegree8   13  27 39.88 50      0.00
## odegree9   16  14 26.17 43      0.04
## odegree10  20   5 16.30 25      0.36
## odegree11   8   4  8.99 17      0.76
## odegree12  11   1  4.73 10      0.00
## odegree13  13   0  2.54  8      0.00
## odegree14   6   0  1.00  3      0.00
## odegree15   6   0  0.45  3      0.00
## odegree16   7   0  0.21  2      0.00
## odegree17   4   0  0.10  2      0.00
## odegree18   3   0  0.02  1      0.00
```

```
##
```

```
## Goodness-of-fit for edgewise shared partner
```

```
##
```

```
##          obs min   mean max MC p-value
## esp0 1032 1979 2193.78 2313      0
## esp1  755  166  235.04  423      0
## esp2  352   2   16.24   83      0
## esp3  202   0    0.88    5      0
## esp4   79   0    0.02    1      0
## esp5   36   0    0.00    0      0
## esp6   14   0    0.00    0      0
## esp7    4   0    0.00    0      0
## esp8    1   0    0.00    0      0
```

```
##
```

```
## Goodness-of-fit for minimum geodesic distance
```

```
##
```

```
##          obs   min     mean   max MC p-value
```

```
## 1      2475  2329  2445.96  2573      0.52
## 2    10672 12419 13611.54 15000      0.00
## 3    31134 49756 55207.67 60888      0.00
## 4    50673 77398 79886.41 81893      0.00
## 5    42563 15561 20575.07 26629      0.00
## 6    18719   491  1265.91  2285      0.00
## 7     4808     4    38.99   178      0.00
## 8      822     0     0.77    14      0.00
## 9      100     0     0.03     1      0.00
## 10       7     0     0.00     0      0.00
## Inf 12333     0  1273.65  3321      0.00
##
## Goodness-of-fit for model statistics
##
##              obs   min    mean   max MC p-value
## edges              2475 2329 2445.96 2573      0.52
## nodematch.hispanic 1832 1740 1826.07 1978      0.86
## nodematch.female1  1879 1760 1860.64 1958      0.76
## nodematch.eversmk1 1755 1646 1733.50 1814      0.46
## mutual              486  434  472.02  498      0.42
```

```
# Plotting GOF statistics
```

```
plot(ans_gof)
```



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 burnin. 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.

```
knitr::include_graphics("awful-chains.png")
```

5.6 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](#)).

5.7 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 occurring conditional on the rest of the network can be written as:

$$\text{logit}(\mathbb{P}(y_{ij} = 1 | y_{-ij})) = \theta^t \Delta \delta(y_{ij} : 0 \rightarrow 1), \quad (5.1)$$

with $\delta(y_{ij} : 0 \rightarrow 1) \equiv s(\mathbf{y})_{ij}^+ - s(\mathbf{y})_{ij}^-$ 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:

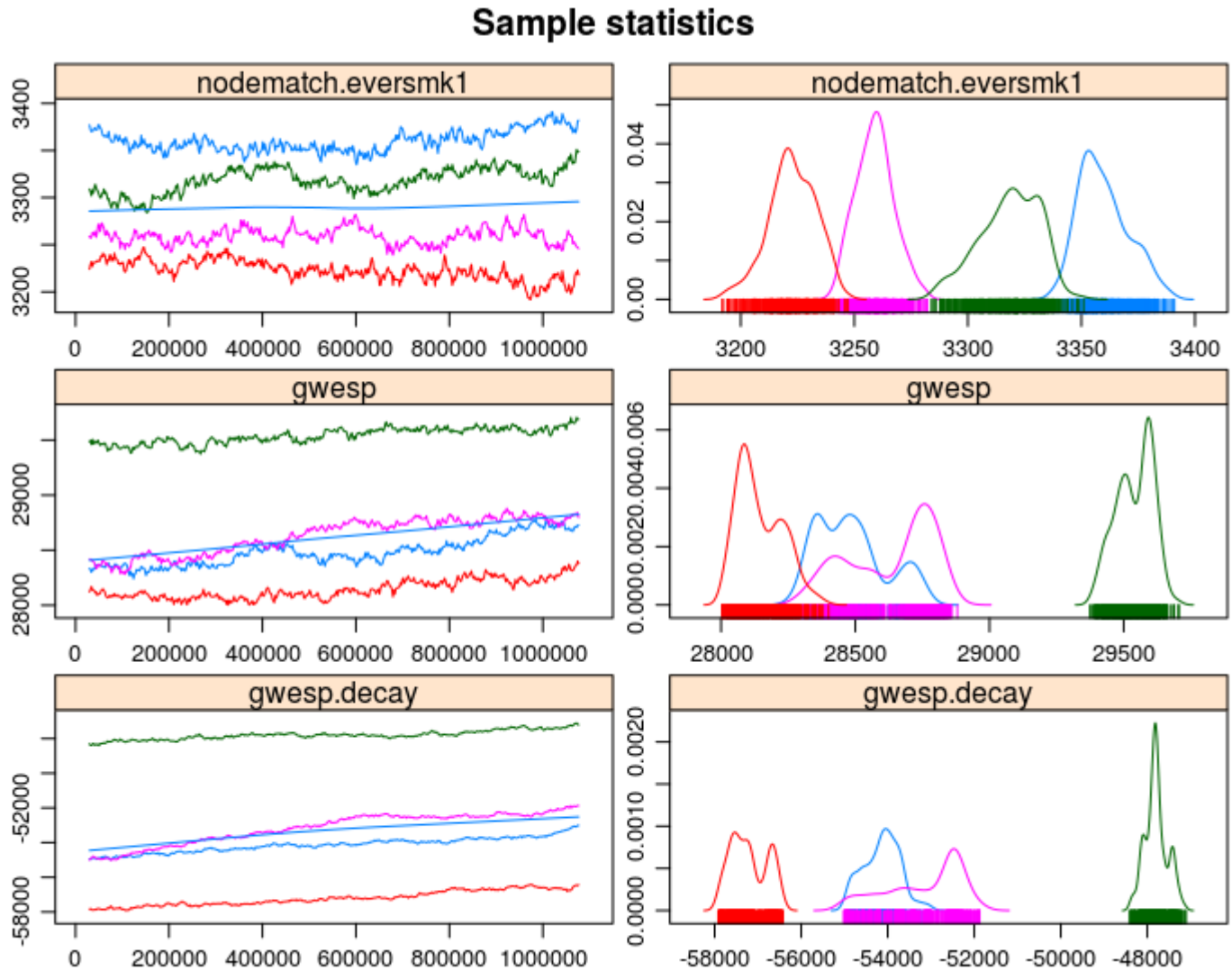


Figure 5.2: An example of a terrible ERGM (no convergence at all). Also, a good example of why running multiple chains can be useful

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

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

$$\begin{aligned}
&= \log \left\{ \frac{\exp \{ \theta^t s(\mathbf{y})_{ij}^+ \}}{\exp \{ \theta^t s(\mathbf{y})_{ij}^+ \} + \exp \{ \theta^t s(\mathbf{y})_{ij}^- \}} \right\} - \log \left\{ \frac{\exp \{ \theta^t s(\mathbf{y})_{ij}^- \}}{\exp \{ \theta^t s(\mathbf{y})_{ij}^+ \} + \exp \{ \theta^t s(\mathbf{y})_{ij}^- \}} \right\} \\
&= \log \{ \exp \{ \theta^t s(\mathbf{y})_{ij}^+ \} \} - \log \{ \exp \{ \theta^t s(\mathbf{y})_{ij}^- \} \} \\
&= \theta^t (s(\mathbf{y})_{ij}^+ - s(\mathbf{y})_{ij}^-) \\
&= \theta^t \Delta \delta(y_{ij} : 0 \rightarrow 1)
\end{aligned}$$

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

$$\mathbb{P}(y_{ij} = 1 | y_{-ij}) = \frac{1}{1 + \exp \{ -\theta^t \Delta \delta(y_{ij} : 0 \rightarrow 1) \}} \quad (5.2)$$

i.e., a logistic probability.

5.8 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.

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

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

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

Where the last equality follows from $s(\mathbf{y}) = \sum_{ij} s(\mathbf{y})_{ij}$. This way, we can now write:

$$\frac{\prod_{ij} \exp\{\theta^t s(\mathbf{y})_{ij}\}}{\sum_{\mathbf{y}} \exp\{\theta^t s(\mathbf{y})\}} = \prod_{ij} \frac{\exp\{\theta^t s(\mathbf{y})_{ij}\}}{1 + \exp\{\theta^t s(\mathbf{y})_{ij}\}} \quad (5.3)$$

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., $s(\mathbf{y}) = \sqrt{\sum_{ij} y_{ij}}$, the square root of the edgcount is no longer a bernoulli graph.

Chapter 6

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 a 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.

6.1 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 connecting to each other. 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 $-\text{Inf}$ 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 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 As and three Es, 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).

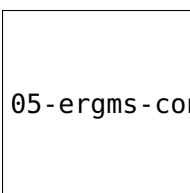
```
library(ergm, quietly = TRUE)
library(sna, quietly = TRUE)

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

# Let's assign 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
```

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



05-ergms-constrains_files/figure-latex/unnamed-chunk-1-1.pdf

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

```
# 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:

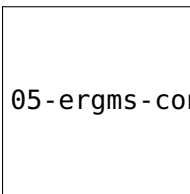
```
net <- make_aux_var(net, "is.ego")

# 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"))
```

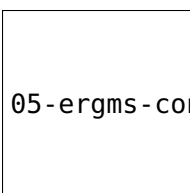
```
##          edges nodematch.aux_var
##          0              0
```

```
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)
```

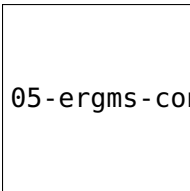
Now, to fit an ERGM with this constraint, we simply need to make use of the offset terms. Here is an example:

```
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):
## Evaluating the predictor and response matrix.
## Maximizing the pseudolikelihood.
## Finished MPLE.
## Stopping at the initial estimate.
## 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 2450  degrees of freedom
## Residual Deviance:  379.4  on 2448  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()
```

}



05-ergms-constrains_files/figure-latex/unnamed-chunk-7-1.pdf

`par(op)`

All networks with no ties between A nodes.

6.2 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

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 nentities`; 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)
```

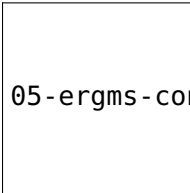
Let's see what we got here:

¹After 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 over estimation of the population parameter and a positive log-likelihood. With that said, I wouldn't recommend using this transformation too often.

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

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

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



05-ergms-constrains_files/figure-latex/05-example2-simulated-graph-1.pdf

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 `nodecov("isnot.actor") ~ -Inf`. Mathematically:

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

$$\text{NodeOCov}(x = \text{"isnot.actor"}) = \sum_i x_i \times \sum_{j < i} y_{ij}$$

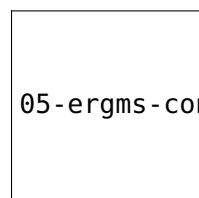
In other words, we are setting that ties between nodes of the same class are forbidden, and

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
)
```



05-ergms-constrains_files/figure-latex/05-example2-side-by-side-1.pdf

```
# 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(nodecov("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(nodecov.isnot.actor)         -Inf
##
## -----
## AIC                          1958.00          -2134192392498171136.00
## BIC                          1971.03          -2134192392498171136.00
## Log Likelihood               -977.00           1067096196249085568.00
## =====
## *** 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 (%)")
```

```
\begin{table}
```

```
\caption{Rejection rate (%)}
```

	Bipartite	Directed
edges	2.48	3.67
Log~b1star2	1.24	2.04

```
\end{table}
```

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.

Chapter 7

(Separable) Temporal Exponential Family Random Graph Models

This tutorial is great!

https://statnet.org/trac/raw-attachment/wiki/Sunbelt2016/tergm_tutorial.pdf

Chapter 8

Simulating and visualizing networks

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`, `sna`, `igraph`, `intergraph`, `netplot`, `netdiffuseR`, and `rgexf`.

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

8.2 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 friend 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") +
  triangle +
  absdiff("age"),
```

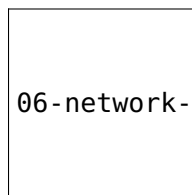
```
coef = c(-4, .5, .5, -.5)
)
```

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!

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
)
```

8.3 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])
```

06-network-simulation-and-viz_files/figure-latex/06-fake

- **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 edgelist as ties are grouped by ego (source.)

```
igraph::as_adj_list(igraph::as_igraph(example_graph))
```

```
## [[1]]
## + 1/4 vertex, from 86aelbf:
## [1] 2
##
## [[2]]
## + 2/4 vertices, from 86aelbf:
## [1] 1 3
##
## [[3]]
## + 3/4 vertices, from 86aelbf:
## [1] 2 4 4
##
## [[4]]
## + 2/4 vertices, from 86aelbf:
## [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  1  64
## 2  2  41
## 3  2 106
## 4  3  61
## 5  4  85
## 6  4 138
```

```
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.network`, 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.

8.4 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.

8.4.1 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:

```
# Sized by indegree
net_sim %>% "inddeg" <- degree(net_sim, gmode = "digraph")

# Preparing the graphical device to hold four nets.
# This line sets a 2 x 2 viz device and stores the
```

```
# original value. We will use the `op` object to reset
# the configuration
op <- par(mfrow = c(2, 2), mai = c(.1, .1, .1, .1))
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)
```

06-network-simulation-and-viz_files/figure-latex/06-size-

```
par(op)
```

If we were using igraph, setting the size can be easier 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)

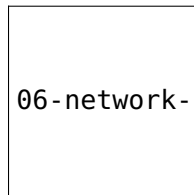
# Converting the network object to an igraph object
net_sim_i <- asIgraph(net_sim)

# 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
)
```

06-network-simulation-and-viz_files/figure-latex/06-size-

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"
)
```



06-network-simulation-and-viz_files/figure-latex/06-netplot

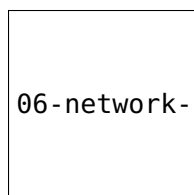
With a good idea for size, we can now start looking into vertex color.

8.4.2 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.

```
vcolors <- c("10" = "gray", "14" = "tomato", "17" = "steelblue")
net_sim %v% "color" <- vcolors[as.character(net_sim %v% "age")]
nplot_base(
  net_ig,
  layout = glayout,
  vertex.color = net_sim %v% "color",
)

legend(
  "bottomright",
  legend = names(vcolors),
  fill = vcolors,
  bty = "n",
  title = "Age"
)
```



06-network-simulation-and-viz_files/figure-latex/06-netplot

8.4.3 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.

```
vshape <- c("white" = 15, "non-white" = 3)
net_sim %v% "shape" <- vshape[as.character(net_sim %v% "race")]
nplot_base(
  net_ig,
  layout = glayout,
  vertex.color = net_sim %v% "color",
  vertex.nsides = net_sim %v% "shape"
)

legend(
  "bottomright",
  legend = names(vcolors),
  fill = vcolors,
  bty = "n",
  title = "Age"
)

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

Chapter 9

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.

9.1 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 assess whether

$$H_0 : s(G_1) - s(G_2) = k \text{ vs } H_a : s(G_1) - s(G_2) \neq k.$$

As usual, the true distribution of $s(\cdot)$ is unknown, thus, one approach that we could use is a non-parametric bootstrap test.

9.1.1 Network bootstrap

The non parametric bootstrap and jackknife methods for social networks were introduced by (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](#).

9.1.2 When the statistic is normal

When the we deal with things that are normally distributed, e.g., sample means like density¹, 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 $s(G_i) \sim N(\mu_i, \sigma_i^2/m_i)$ for $i \in \{1, 2\}$, in the case of the density, $m_i = n_i * (n_i - 1)$. The statistic is then:

$$s(G_1) - s(G_0) \sim N(\mu_1 - \mu_0, \sigma_1^2/m_1 + \sigma_1^2/m_2)$$

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
```

```
# 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:

¹Density is indeed a sample mean as we are, in principle computing the average of a sequence of Bernoulli variables. Formally: $\text{density}(G) = \frac{1}{n(n-1)} \sum_{ij} A_{ij}$.

```

# 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)

```

9.1.3 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 assess 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 $k \in \text{Conflnt}(s(G_1) - s(G_2))$. 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:

```

# 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
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:

```
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)

9.2 Examples

9.2.1 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:

$$s(G_1) = \text{AvgIndeg}(G_1) = \frac{1}{n} \sum_i \sum_{j \neq i} A_{ji}^1$$

where A_{ji}^1 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.

Chapter 10

Network diffusion with netdiffuseR

This chapter mainly was 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](#).

10.1 Network diffusion of innovation

10.1.1 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 is a prominent one.
- 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.

10.1.2 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), $\alpha = 1$. In (very) general terms

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

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).

10.2 The netdiffuseR R package

10.2.1 Overview

netdiffuseR is an R package that:

- Is designed for Visualizing, Analyzing, and Simulating 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.

10.2.2 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, reyno1, studno1, area1, id1, nmage1, nmag... (430)
## Dynamic attributes : -
```

10.2.3 Visualization methods

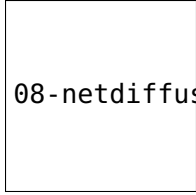
```
set.seed(12315)
x <- rdiffrnet(
  400, t = 6, rgraph.args = list(k=6, p=.3),
  seed.graph = "small-world",
  seed.nodes = "central", rewire = FALSE, threshold.dist = 1/4
)
plot(x)
```

08-netdiffuser_files/figure-latex/viz-1.pdf

```
plot_diffnet(x)
```

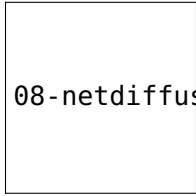
08-netdiffuser_files/figure-latex/viz-2.pdf

```
plot_diffnet2(x)
```



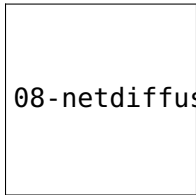
08-netdiffuser_files/figure-latex/viz-3.pdf

`plot_adopters(x)`



08-netdiffuser_files/figure-latex/viz-4.pdf

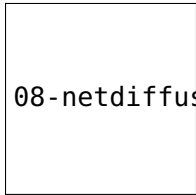
`plot_threshold(x)`



08-netdiffuser_files/figure-latex/viz-5.pdf

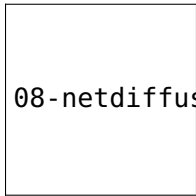
`plot_infectsuscep(x, K=2)`

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



08-netdiffuser_files/figure-latex/viz-6.pdf

`plot_hazard(x)`



08-netdiffuser_files/figure-latex/viz-7.pdf

10.2.4 Problems

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

10.3 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.

10.3.1 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:

```
# 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.
)
```

- 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

10.3.2 Rumor spreading

```
library(netdiffuser)

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

```
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. Significance 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)
```

10.3.3 Diffusion

```
set.seed(09)
```

```
diffnet_complex <- rdifffnet(
  seed.graph = diffnet_rumor$graph,
  seed.nodes = which(diffnet_rumor$toa == 1),
  rewired = 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)
```

08-netdiffuser_files/figure-latex/plot-complex-and-disease

10.3.4 Mentor Matching

```
# Finding mentors
```

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

```
# Simulating diffusion with these mentors
```

```
set.seed(09)
diffnet_mentored <- rdifffnet(
  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 censoring : 0.02 (8)
```

```
## # of nodes      : 500
```

```
##
```

```
## Moran's I was computed on contemporaneous autocorrelation using 1/geodesic
```

```
## values. Significance 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
```

10.3.5 Example by changing threshold

```
# 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)))
```

08-netdiffuser_files/figure-latex/sim-sim-1.pdf

```
# Now, simulating a bunch of diffusion processes
nsim <- 500L
ans_land2 <- vector("list", nsim)
set.seed(223)
for (i in 1:nsim) {
  # We just want the cum adopt count
  ans_land2[[i]] <-
    cumulative_adopt_count(
      rdifffnet(
        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"
)
```

10.3.6 Problems

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

10.4 Statistical inference

10.4.1 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.

10.4.2 Using geodesics

- One approach is to use the geodesic (shortest path length) matrix to account for indirect influence.
- In the case of sparse matrices, and furthermore, in the presence of structural holes it is more convenient to calculate the distance matrix taking this into account.
- **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"
)
```

08-netdiffuser_files/figure-latex/geodesic_speed-box-1.pdf

- 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"
```

10.4.3 Structural dependence and permutation tests

- A novel statistical method (work-in-progress) that allows conducting inference.
- 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!)

10.4.4 Idea

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

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

- This is, if for example time of adoption is independent on 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 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)

10.4.4.1 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

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

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

# 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        0.2511        0.0000
```

08-netdiffuser_files/figure-latex/unnamed-chunk-2-1.pdf

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

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

# Running the test
```

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

# 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        0.2579        0.3720
```

08-netdiffuser_files/figure-latex/unnamed-chunk-3-1.pdf

10.4.5 Regression analysis

- In regression analysis, we want to see if exposure, once we control for other covariates had any effect on the adoption of a behavior.
- In general, the big problem here is when we have a latent variable that co-determines both network and behavior.
- Unless we can control for such variable, regression analysis will be generically biased.
- 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.

10.4.5.1 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

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

- In `netdiffuser` 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))
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.1754  -0.8462  -0.6645   1.2878   1.9523
##
## 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 = "probit"),
##      data = dat, subset = ifelse(is.na(toa), TRUE, toa >= per))
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.1754  -0.8462  -0.6645   1.2878   1.9523
##
## 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
```

10.4.5.2 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

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

- Unfortunately, since y_t is in both sides of the equation, this 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.

10.4.6 Problems

Using the dataset [stats.rda](#):

1. Compute Moran's I as the function `summary.diffnet` does. For this 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?

Chapter 11

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),

$$\frac{\exp \{f_i^Z(\beta^Z, x, z)\}}{\sum_{Z' \in \mathcal{C}} \exp \{f_i^Z(\beta, x, z')\}}$$

snijders_(sociological methodology 2001)

(Snijders, Bunt, and Steglich [2010](#), @lazega2015, @Ripley2011)

Part II

Statistical Foundations

11.1 Bayes' Rule

$$\mathbb{P}((X = x|Y = y)) = \frac{\mathbb{P}((Y = y|X = x))\mathbb{P}((X = x))}{\mathbb{P}((Y = y))} \quad (11.1)$$

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

$$\mathbb{P}((x = x|Y = y)) = \frac{\mathbb{P}((x = x, Y = y))}{Pr(Y = y)} = \frac{\mathbb{P}((y = y|X = x))Pr(X = x)}{Pr(Y = y)}$$

11.2 Markov Chain

A Markov Chain is a sequence of random variables in which the conditional distribution of the n -th element only depends on $n - 1$.

11.2.1 Metropolis Algorithm

In the Metropolis Algorithm, or Metropolis MCMC, builds a Markov Chain that under certain conditions converges to the target distribution. The key of the Algorithm is in accepting a proposed move from θ to θ' with probability equal to:

$$r = \min\left(1, \frac{\mathbb{P}((\theta'|D))}{\mathbb{P}((\theta|D))}\right) \quad (11.2)$$

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. The following proof was adapted from "Bayesian Data Analysis:"

11.2.2 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 transition probability is symmetric, then the previous equation reduces to the Metropolis probability.

11.2.3 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

Appendix A

Datasets

A.1 SNS data

A.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”.

A.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 individual 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.
- Bache, Stefan Milton, and Hadley Wickham. 2014. *Magrittr: A Forward-Pipe Operator for R*. <https://CRAN.R-project.org/package=magrittr>.
- Bojanowski, Michal. 2015. *Intergraph: Coercion Routines for Network Data Objects*. <http://mbojan.github.io/intergraph>.
- Brooks, Steve, Andrew Gelman, Galin Jones, and Xiao-Li Meng. 2011. *Handbook of Markov Chain Monte Carlo*. CRC press.
- Csardi, Gabor, and Tamas Nepusz. 2006. "The Igraph Software Package for Complex Network Research." *InterJournal Complex Systems*: 1695. <http://igraph.org>.
- 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, Skye Bender-deMoll, and Martina Morris. 2016. *Statnet: Software Tools for the Statistical Analysis of Network Data*. The Statnet Project (<http://www.statnet.org>). CRAN.R-project.org/package=statnet.
- Handcock, Mark S., David R. Hunter, Carter T. Butts, Steven M. Goodreau, Pavel N. Krivitsky, and Martina Morris. 2017. *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. <http://www.jstatsoft.org/v55/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. 2017a. *Foreign: Read Data Stored by 'Minitab', 'S', 'Sas', 'Spss', 'Stata', 'Systat', 'Weka', 'dBase', ...* <https://CRAN.R-project.org/package=foreign>.
- . 2017b. *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{_}Manual.pdf.
- Sarkar, Deepayan, and Felix Andrews. 2016. *LatticeExtra: Extra Graphical Utilities Based on Lattice*. <https://CRAN.R-project.org/package=latticeExtra>.
- Snijders, Tom AB. 2002. "Markov Chain Monte Carlo Estimation of Exponential Random Graph Models." *Journal of Social Structure* 3.
- SNIJDETS, 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 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>.
- Ushey, Kevin, Jim Hester, and Robert Krzyzanowski. 2017. *Rex: Friendly Regular Expressions*. <https://CRAN.R-project.org/package=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. 2017. *Stringr: Simple, Consistent Wrappers for Common String Operations*.
<https://CRAN.R-project.org/package=stringr>.
- Wickham, Hadley, and Jennifer Bryan. 2017. *Readxl: Read Excel Files*.
<https://CRAN.R-project.org/package=readxl>.
- Wickham, Hadley, Romain Francois, Lionel Henry, and Kirill Müller. 2017. *Dplyr: A Grammar of Data Manipulation*. <https://CRAN.R-project.org/package=dplyr>.
- Wickham, Hadley, and Lionel Henry. 2017. *Tidyr: Easily Tidy Data with 'Spread()' and 'Gather()' Functions*. <https://CRAN.R-project.org/package=tidyr>.
- Wickham, Hadley, Jim Hester, and Romain Francois. 2017. *Readr: Read Rectangular Text Data*.
<https://CRAN.R-project.org/package=readr>.