

Applied SNA with R

George G. Vega Yon

2018-03-07

Contents

| | |
|----------------------------------------------|-----------|
| 1 About this book | 5 |
| 2 Introduction | 7 |
| 3 R Basics | 9 |
| 3.1 What is R | 9 |
| 3.2 How to install packages | 9 |
| 4 Week 1: SNS Study | 11 |
| 4.1 Data preprocessing | 11 |
| 4.2 Creating a network | 13 |
| 4.3 Network descriptive stats | 22 |
| 4.4 Plotting the network in igraph | 25 |
| 4.5 Statistical tests | 33 |
| 5 SNS Exponential Random Graph Models | 35 |
| 5.1 The ergm package | 36 |
| 6 Final Words | 55 |
| A Datasets | 57 |
| A.1 SNS data | 57 |

Chapter 1

About this book

This book will be build as part of a workshop on Applied Social Network Analysis with R. Its contents will be populated as the sessions take place, and for now there is particular program that we will follow, instead, we have the following workflow:

1. Participants will share their data and what they need to do with it.
2. Based on their data, I'll be preparing the sessions trying to show attendees how would I approach the problem, and at the same time, teach by example about the R language.
3. Materials will be published on this website and, hopefully, video recordings of the sessions.

At least in the first version, the book will be organized by session, this is, one chapter per session.

All the book materials can be downloaded from <https://github.com/gvegayon/appliedsnar>

In general, we will besides of 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

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 find a compelling reason to leave my “old-practices” and embrace all the tidyverse movement (religion?).

Chapter 2

Introduction

For this book we need the following

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 3

R Basics

3.1 What is R

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.

Chapter 4

Week 1: SNS Study

The data can be downloaded from [here](#).

The codebook for the data provided here is in [the appendix](#).

This chapter's goals 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 in. Your data can be Raw plain files like CSV, tab delimited or specified by column width, for which you can use the [readr](#) package; or it can be binary files like dta (Stata), Octave, SPSS, for which [foreign](#) can be used; or it could be excel files in which case you should be using [readxl](#). In our case, the data for this session is in Stata format:

```
library(dplyr)
library(magrittr)
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 such that, for example, the last three x 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. Here, photoid has the following range:

```
(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. Again, we use dplyr to create this variable, and we will call it... id (mind blowing, right?):

```
(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 [piping operator](#), and it is a very nice way of writing nested function calls. In this case, instead of having write 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 for this task.

```
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 claimed 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
##   <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     3.00   NA
## 3      7     111      0.      1      1      1      1     5.00   4.50
## 4     13     111      1.      1      1      1      1     2.50   2.50
## 5     14     111      1.      1      1      1     NA     3.00   3.50
## 6     15     111      1.      0      0      0      0     2.50   2.50
## 7     20     111      1.      1      1      1      1     2.50   2.50
## 8     22     111      1.     NA     NA      0      0     NA     NA
## 9     25     111      0.      1      1     NA      1     4.50   3.50
## 10    27     111      1.      0     NA      0      0     3.50   NA
## # ... with 2,154 more rows, and 91 more variables: grades3 <dbl>,
## #   grades4 <dbl>, everismk1 <int>, everismk2 <int>, everismk3 <int>,
## #   everismk4 <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>,
## #   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>, sch_friend25 <int>,
## # sch_friend26 <int>, sch_friend27 <int>, sch_friend28 <int>,
## # sch_friend29 <int>, sch_friend210 <int>, sch_friend211 <int>,
## # sch_friend212 <int>, sch_friend213 <int>, sch_friend214 <int>,
## # sch_friend215 <int>, sch_friend216 <int>, sch_friend217 <int>,
## # sch_friend218 <int>, sch_friend219 <int>, sch_friend31 <int>,
## # sch_friend32 <int>, sch_friend33 <int>, sch_friend34 <int>,
## # sch_friend35 <int>, sch_friend36 <int>, sch_friend37 <int>,
## # sch_friend38 <int>, sch_friend39 <int>, sch_friend310 <int>,
## # sch_friend311 <int>, sch_friend312 <int>, sch_friend313 <int>,
## # sch_friend314 <int>, sch_friend315 <int>, sch_friend316 <int>,
## # sch_friend317 <int>, sch_friend318 <int>, sch_friend319 <int>,
## # sch_friend41 <int>, sch_friend42 <int>, sch_friend43 <int>,
## # sch_friend44 <int>, sch_friend45 <int>, sch_friend46 <int>,
## # sch_friend47 <int>, sch_friend48 <int>, sch_friend49 <int>,
## # sch_friend410 <int>, sch_friend411 <int>, sch_friend412 <int>,
## # sch_friend413 <int>, sch_friend414 <int>, sch_friend415 <int>,
## # sch_friend416 <int>, sch_friend417 <int>, sch_friend418 <int>,
## # sch_friend419 <int>, id <dbl>
```

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). This allows us to select all variables that starts with the word “`sch_friend`”, which means that `sch_friend11`, `sch_friend12`, ... will all 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>,
## #   sch_friend25 <int>, sch_friend26 <int>, sch_friend27 <int>,
## #   sch_friend28 <int>, sch_friend29 <int>, sch_friend210 <int>,
## #   sch_friend211 <int>, sch_friend212 <int>, sch_friend213 <int>,
## #   sch_friend214 <int>, sch_friend215 <int>, sch_friend216 <int>,
## #   sch_friend217 <int>, sch_friend218 <int>, sch_friend219 <int>,
## #   sch_friend31 <int>, sch_friend32 <int>, sch_friend33 <int>,
```



```
## # sch_friend34 <int>, sch_friend35 <int>, sch_friend36 <int>,
## # sch_friend37 <int>, sch_friend38 <int>, sch_friend39 <int>,
## # sch_friend310 <int>, sch_friend311 <int>, sch_friend312 <int>,
## # sch_friend313 <int>, sch_friend314 <int>, sch_friend315 <int>,
## # sch_friend316 <int>, sch_friend317 <int>, sch_friend318 <int>,
## # sch_friend319 <int>, sch_friend41 <int>, sch_friend42 <int>,
## # sch_friend43 <int>, sch_friend44 <int>, sch_friend45 <int>,
## # sch_friend46 <int>, sch_friend47 <int>, sch_friend48 <int>,
## # sch_friend49 <int>, sch_friend410 <int>, sch_friend411 <int>,
## # sch_friend412 <int>, sch_friend413 <int>, sch_friend414 <int>,
## # sch_friend415 <int>, sch_friend416 <int>, sch_friend417 <int>,
## # sch_friend418 <int>, sch_friend419 <int>
```

2. Then, we reshape it to *long* format: By transposing all the `sch_friend*` to long. We do this by means of the function `gather` (from the `tidyr` package). This is an alternative to the `reshape` function, and I personally find it 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 this 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 that is preceded by any letter from `a` to `z`, whereas 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. On the other hand, the expression `(?<=[a-z][0-9])` matches a string that is preceded by a letter from `a` to `z` and a number from `0` to `9`; and 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`. We can actually see this

```
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`. This function receives a data frame where the two first columns are `sorce(ego)` and `target(alter)`, whether is it 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 is the set of vertex 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 that were not included on 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 fa9c268 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 fa9c268 (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 next 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 your self now is, “how can I get that information from the igraph object?”. Well, vertex attributes 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"
## [7] "eversmk3"  "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 10 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 are true. In our

case it will return a vector of indices of the vertices which have the attribute `school` equal to 111. Now that we have our subgraph, we can compute different centrality measures¹ 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")

# Computing quantiles for each variable
stats_degree <- with(stats, {
  cbind(
    indegree = quantile(indegree, c(.025, .5, .975)),
    outdegree = quantile(outdegree, c(.025, .5, .975)),
    closeness = quantile(closeness, c(.025, .5, .975)),
    betweenness = quantile(betweenness, c(.025, .5, .975))
  )
})

stats_degree
```

| ## | indegree | outdegree | closeness | betweenness |
|----------|----------|-----------|--------------|--------------|
| ## 2.5% | 0 | 0 | 3.526640e-06 | 0.0000000000 |
| ## 50% | 4 | 4 | 1.595431e-05 | 0.001879006 |
| ## 97.5% | 16 | 16 | 1.601822e-05 | 0.016591048 |

The `with` function is somewhat similar to what `dplyr` allows us to do when we want to work

¹For more information about the different centrality measurements, please take a look at the “Centrality” article on [Wikipedia](#).

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$indegree, c(.025, .5, .975)),
    outdegree  = quantile(stats$outdegree, c(.025, .5, .975)),
    closeness  = quantile(stats$closeness, c(.025, .5, .975)),
    betweeness = quantile(stats$betweeness, c(.025, .5, .975))
  )
```

Now we will compute some statistics at the graph level:

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

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

Triadic census

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

```
## [1] 24059676  724389  290849   3619   3383   4401   3219
## [8]    2997    407    33    836    235    163    137
## [15]    277    85
```

To get a nicer view of this, we can use a table that I retrieved from `?triad_census`. Moreover, instead of looking at the raw counts, we can normalize the triadic object by its sum so we get proportions instead²

²During our workshop, Prof. De la Haye suggested using $\binom{n}{3}$ as a normalizing constant. It turns out that `sum(triadic)`


```
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)
```

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

= choose(n, 3)! So either approach is correct.

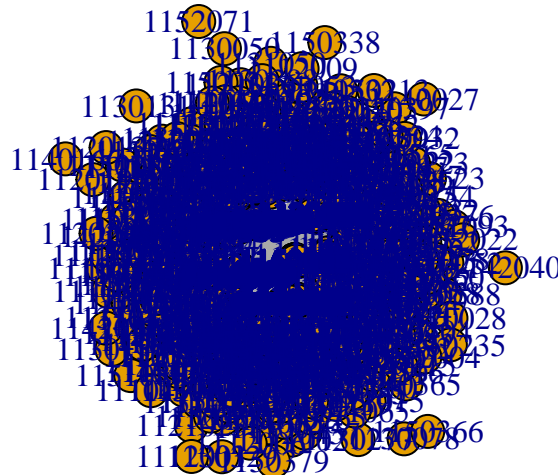


Figure 4.1: A not very nice network plot. This is what we get with the default parameters in igraph.

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 more over, are overlapping. So, 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`.³
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.

³Figuring 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.

2. The second line replaced all NAs with the number 3, so that our vector `col_hispanic` now ranges from 1 to 3 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"
## [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,
```

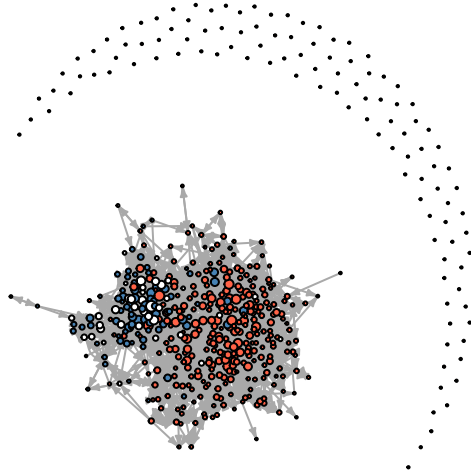


Figure 4.2: Friends network in time 1 for school 111.

```
vertex.size      = degree(ig_year1_111)/10 +1,
vertex.label     = NA,
edge.arrow.size  = .25,
vertex.color     = col_hispanic
)
```

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,
```

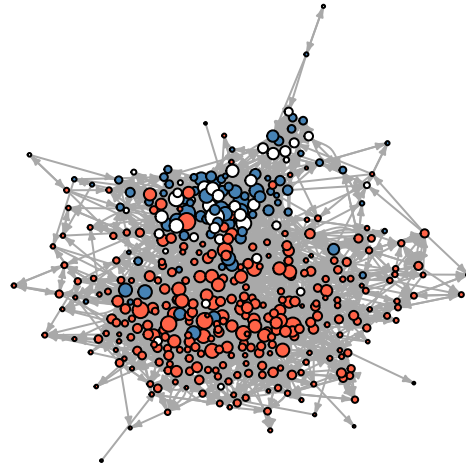


Figure 4.3: Friends network in time 1 for school 111. The graph excludes isolates.

```
vertex.size      = degree(ig_year1_111_sub)/5 +1,
vertex.label     = NA,
edge.arrow.size = .25,
vertex.color     = col_hispanic
)
```

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 actually write this as a function so that, instead of us copying and pasting the code n times (supposing that we want to crate a plot similar to this n times). The next subsection does that.

4.4.2 Multiple plots

When you are repeating yourself over and over again, 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 changes 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. In the arguments part, we are declaring 4 specific arguments: `net`, `schoolid`, `mindgr`, and `vcol`. These are an `igraph` object, the school id, the minimum degree that a vertex must have to be included in the plot, and the color of the vertices. Notice that, as a difference from other programming languages, in R we don't need to declare the types that these objects are.
3. The elipsis object, `...`, is a special object in R that allows us passing other arguments without us specifying which. In our case, if you take a look at the `plot` bit of the body

of the function, you will see that we also added `...`; this means that whatever other arguments (different from the ones that we explicitly defined) are passed to the function, these will be passed to the function `plot`, moreover, to the `plot.gexf` function (since the subnet object is actually an igraph object). 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"
)
```

So what happend here?

- `oldpar <- par(no.readonly = TRUE)` This line stores the current parameters for plot-

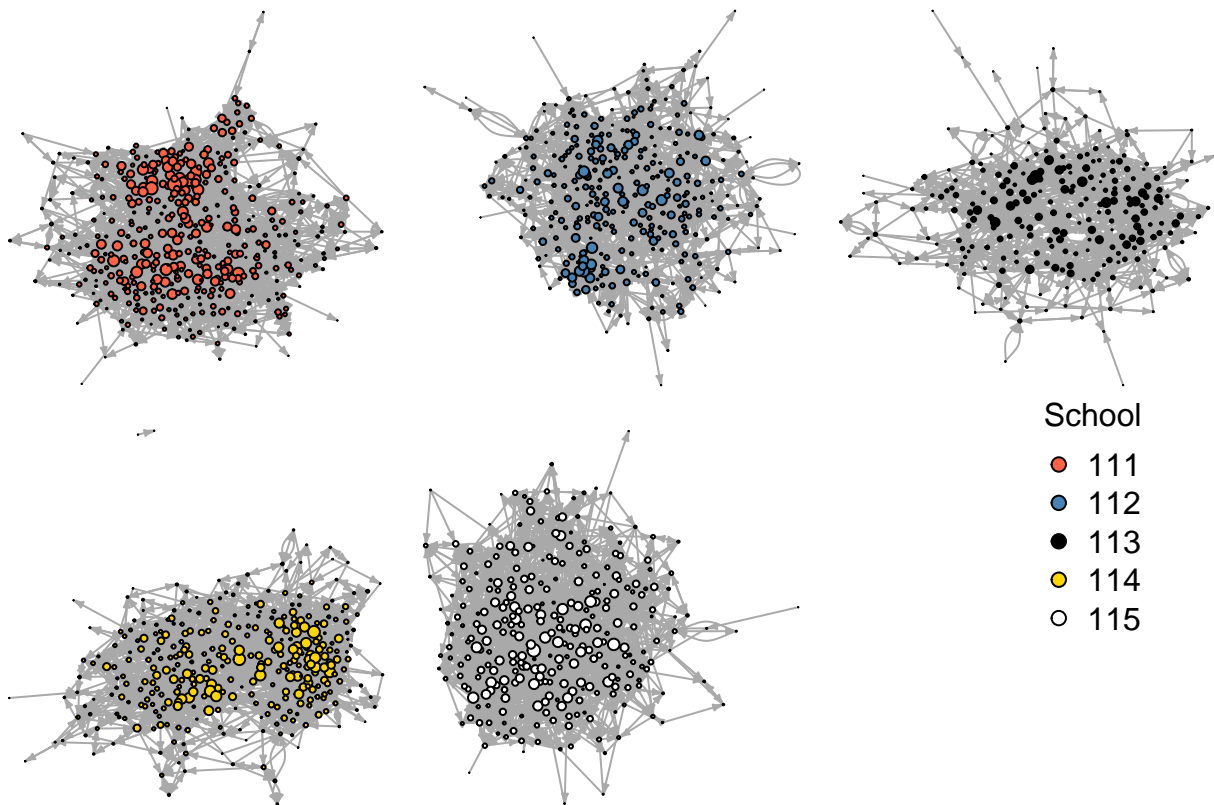


Figure 4.4: All 5 schools in time 1. Again, the graphs exclude isolates.

ting. 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 allow for $2 \times 3 = 6$ plots organized in 2 rows and 3 columns, and these will be 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 network itself. 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    = n(),
    nom_avg  = 1/mean(nnom)
  ) %>%
  rename(
    school = to_school
  )

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  0.150
```

```
## 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

SNS Exponential Random Graph Models

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

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

So what are ERGMs anyway. . .

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 like 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. — (Hunter et al. [2008](#))

The distribution of \mathbf{Y} can be parameterized in the form

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

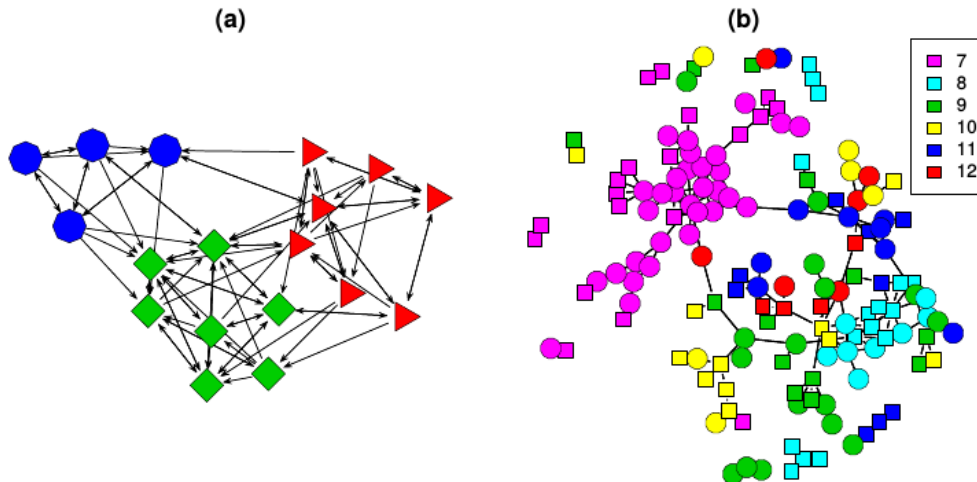


Figure 1: The (a) `samplike` and (b) `faux.mesa.high` networks described in Section 2. The values of nodal covariates may be indicated using various colors, shapes, and labels of nodes.

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

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 (5) may be expanded by replacing $\mathbf{g}(\mathbf{y})$ with $\mathbf{g}(\mathbf{y}, \mathbf{X})$ to allow for additional covariate information \mathbf{X} about the network. The denominator,

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

Is the normalizing factor that ensures that equation (5) 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 models hard to estimate as there are $N = 2^{n(n-1)}$ possible networks! (Hunter et al. 2008)

5.1 The `ergm` package

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

From the previous section:¹

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

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

load("03.rda")
```

In this section we will use the `ergm` package (from the `statnet`) suit, and the `intergraph` 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)
```

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"
## [1] "Warning: This network contains loops"
## [1] "Warning: This network contains loops"

## Evaluating log-likelihood at the estimate.

##
## MLE Coefficients:
## edges
## -4.732
```

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 on that with the `which_loop` function

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

```
E(ig_year1_111)[which_loop(ig_year1_111)]
```

```
## + 1/2638 edge from 2c3f50e (vertex names):
```

```
## [1] 1110111->1110111
```

We can get rid of these using the `igraph::` `.igraph`. Moreover, just to illustrate how it can be done, let's get rid of 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)
```

Let's rerun the model, now with a couple of extra terms

1. Estimate the simplest model, adding one variable at a time
2. After each estimation, run the `mcmc.diagnostics` function to see how good/bad behaved are the chains
3. Run the `gof` function to

`control.ergms`: Maximum number of iteration, seed for Pseudo-RNG, how many cores

`ergm.constraints`: Where to sample the network from. Gives stability as

`gof`

```
# Running a simple ergm (only fitting edge count)
```

```
ans0 <- ergm(
```

```
  network_111 ~
```

```
  edges +
```

```

    nodematch("hispanic") +
    nodematch("female1") +
    nodematch("eversmk1") +
    mutual
  ,
  constraints = ~bd(maxout = 19, maxin = 22),
  control = control.ergm(
    seed          = 1,
    MCMLE.maxit = 10,
    parallel      = 4,
    CD.maxit      = 10
  )
)

```

```
## Starting contrastive divergence estimation via CD-MCMLE:
```

```
## Iteration 1 of at most 10:
```

```
## Convergence test P-value:0e+00
```

```
## The log-likelihood improved by1.735
```

```
## Iteration 2 of at most 10:
```

```
## Convergence test P-value:0e+00
```

```
## The log-likelihood improved by1.867
```

```
## Iteration 3 of at most 10:
```

```
## Convergence test P-value:0e+00
```

```
## The log-likelihood improved by1.584
```

```
## Iteration 4 of at most 10:
```

```
## Convergence test P-value:5.4e-195
```

```
## The log-likelihood improved by0.7262
```

```
## Iteration 5 of at most 10:
```

```
## Convergence test P-value:6.4e-56
```

```
## The log-likelihood improved by0.1468
## Iteration 6 of at most 10:
## Convergence test P-value:4.4e-11
## The log-likelihood improved by0.02877
## Iteration 7 of at most 10:
## Convergence test P-value:8.9e-04
## The log-likelihood improved by0.01025
## Iteration 8 of at most 10:
## Convergence test P-value:1.1e-01
## The log-likelihood improved by0.004459
## Iteration 9 of at most 10:
## Convergence test P-value:8.1e-01
## Convergence detected. Stopping.
## The log-likelihood improved by0.001105
## Starting maximum likelihood estimation via MCMLE:
## Iteration 1 of at most 10:
## Optimizing with step length 0.927769132341318.
## The log-likelihood improved by 2.215.
## Iteration 2 of at most 10:
## Optimizing with step length 0.298934775640332.
## The log-likelihood improved by 2.493.
## Iteration 3 of at most 10:
## Optimizing with step length 0.314693064006812.
## The log-likelihood improved by 3.096.
## Iteration 4 of at most 10:
```



```

## Optimizing with step length 0.694716889523959.
## The log-likelihood improved by 1.537.
## Iteration 5 of at most 10:
## Optimizing with step length 1.
## The log-likelihood improved by 1.141.
## Step length converged once. Increasing MCMC sample size.
## Iteration 6 of at most 10:
## Optimizing with step length 1.
## The log-likelihood improved by 0.6937.
## Step length converged twice. Stopping.
## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
## Evaluating log-likelihood at the estimate. Using 20 bridges: 1 2 3 4 5 6 7 8 9 10 11 12
## This model was fit using MCMC. To examine model diagnostics and check for degeneracy,
# What did we get
summary(ans0)

## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
##
## =====
## Summary of model fit
## =====
##
## Formula:   network_111 ~ edges + nodematch("hispanic") + nodematch("female1") +
##           nodematch("eversmk1") + mutual
##
## Iterations: 6 out of 10
##
## Monte Carlo MLE Results:
##           Estimate Std. Error MCMC % p-value

```

```
## edges          -5.62242    0.05005    0 <1e-04 ***
## nodematch.hispanic 0.35938    0.03453    0 <1e-04 ***
## nodematch.female1 0.82255    0.04359    0 <1e-04 ***
## nodematch.eversmk1 0.33361    0.03844    0 <1e-04 ***
## mutual          4.09236    0.07035    1 <1e-04 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##      Null Deviance:      0 on 174306 degrees of freedom
## Residual Deviance: -37491 on 174301 degrees of freedom
##
## Note that the null model likelihood and deviance are defined to be 0.
##
## AIC: -37481    BIC: -37430    (Smaller is better.)
```

```
mcmc.diagnostics(ans0)
```

```
## Sample statistics summary:
##
## Iterations = 16384:1063936
## Thinning interval = 1024
## Number of chains = 4
## Sample size per chain = 1024
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##              Mean    SD Naive SE Time-series SE
## edges          5.2175 54.79    0.8562         4.504
## nodematch.hispanic -26.8933 46.16    0.7212         4.560
## nodematch.female1 16.6682 47.28    0.7388         4.372
## nodematch.eversmk1  6.1863 48.54    0.7584         5.370
## mutual          0.8477 21.24    0.3319         4.430
##
```

```
## 2. Quantiles for each variable:
```

```
##
```

| | 2.5% | 25% | 50% | 75% | 97.5% |
|--------------------|------|--------|-----|-----|--------|
| edges | -98 | -32.00 | 4 | 41 | 116.00 |
| nodematch.hispanic | -118 | -56.00 | -27 | 2 | 66.62 |
| nodematch.female1 | -75 | -14.25 | 15 | 48 | 113.00 |
| nodematch.eversmk1 | -84 | -27.00 | 4 | 37 | 109.00 |
| mutual | -46 | -12.00 | -1 | 14 | 43.00 |

```
##
```

```
##
```

```
## Sample statistics cross-correlations:
```

| | edges | nodematch.hispanic | nodematch.female1 |
|--------------------|-----------|--------------------|-------------------|
| edges | 1.0000000 | 0.7703930 | 0.8702355 |
| nodematch.hispanic | 0.7703930 | 1.0000000 | 0.6691023 |
| nodematch.female1 | 0.8702355 | 0.6691023 | 1.0000000 |
| nodematch.eversmk1 | 0.8321525 | 0.6704200 | 0.7319416 |
| mutual | 0.7012887 | 0.6083908 | 0.6661004 |

| | nodematch.eversmk1 | mutual |
|--------------------|--------------------|-----------|
| edges | 0.8321525 | 0.7012887 |
| nodematch.hispanic | 0.6704200 | 0.6083908 |
| nodematch.female1 | 0.7319416 | 0.6661004 |
| nodematch.eversmk1 | 1.0000000 | 0.6791293 |
| mutual | 0.6791293 | 1.0000000 |

```
##
```

```
## Sample statistics auto-correlation:
```

```
## Chain 1
```

| | edges | nodematch.hispanic | nodematch.female1 | nodematch.eversmk1 |
|----------|-----------|--------------------|-------------------|--------------------|
| Lag 0 | 1.0000000 | 1.0000000 | 1.0000000 | 1.0000000 |
| Lag 1024 | 0.9194094 | 0.9264455 | 0.9322242 | 0.9148191 |
| Lag 2048 | 0.8526321 | 0.8583709 | 0.8679994 | 0.8473601 |
| Lag 3072 | 0.7980086 | 0.8047307 | 0.8148158 | 0.7874674 |
| Lag 4096 | 0.7556384 | 0.7658891 | 0.7753939 | 0.7373127 |
| Lag 5120 | 0.7202209 | 0.7380723 | 0.7438670 | 0.6973020 |

```

##          mutual
## Lag 0      1.0000000
## Lag 1024  0.9922688
## Lag 2048  0.9850092
## Lag 3072  0.9782542
## Lag 4096  0.9718149
## Lag 5120  0.9651833
## Chain 2
##          edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0      1.0000000      1.0000000      1.0000000      1.0000000
## Lag 1024  0.8937698      0.9320595      0.9003587      0.9090676
## Lag 2048  0.8058768      0.8732285      0.8200779      0.8353461
## Lag 3072  0.7329964      0.8234520      0.7531759      0.7755789
## Lag 4096  0.6738326      0.7817764      0.6904470      0.7207250
## Lag 5120  0.6359449      0.7538172      0.6430627      0.6829884
##          mutual
## Lag 0      1.0000000
## Lag 1024  0.9883401
## Lag 2048  0.9764278
## Lag 3072  0.9637258
## Lag 4096  0.9517826
## Lag 5120  0.9404211
## Chain 3
##          edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0      1.0000000      1.0000000      1.0000000      1.0000000
## Lag 1024  0.8691963      0.8697551      0.8905401      0.8832045
## Lag 2048  0.7670783      0.7709648      0.8086326      0.7950309
## Lag 3072  0.6765077      0.6916824      0.7397413      0.7181674
## Lag 4096  0.5939888      0.6283922      0.6724809      0.6500649
## Lag 5120  0.5341970      0.5833965      0.6153834      0.5984216
##          mutual
## Lag 0      1.0000000
## Lag 1024  0.9742239

```

```

## Lag 2048 0.9505850
## Lag 3072 0.9285324
## Lag 4096 0.9067956
## Lag 5120 0.8879835
## Chain 4
##
##          edges nodematch.hispanic nodematch.female1 nodematch.eversmk1
## Lag 0      1.0000000          1.0000000          1.0000000          1.0000000
## Lag 1024 0.9083130          0.9144195          0.9124920          0.9072684
## Lag 2048 0.8301639          0.8566684          0.8337887          0.8298152
## Lag 3072 0.7625395          0.8035036          0.7719743          0.7678617
## Lag 4096 0.7030654          0.7581905          0.7172222          0.7157227
## Lag 5120 0.6528153          0.7165866          0.6732486          0.6688953
##
##          mutual
## Lag 0      1.0000000
## Lag 1024 0.9879414
## Lag 2048 0.9759230
## Lag 3072 0.9644667
## Lag 4096 0.9533075
## Lag 5120 0.9412684
##
## Sample statistics burn-in diagnostic (Geweke):
## Chain 4
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##          edges nodematch.hispanic  nodematch.female1
##          -4.505          -5.054          -6.423
## nodematch.eversmk1          mutual
##          -7.250          -7.454
##
## Individual P-values (lower = worse):
##          edges nodematch.hispanic  nodematch.female1

```

```

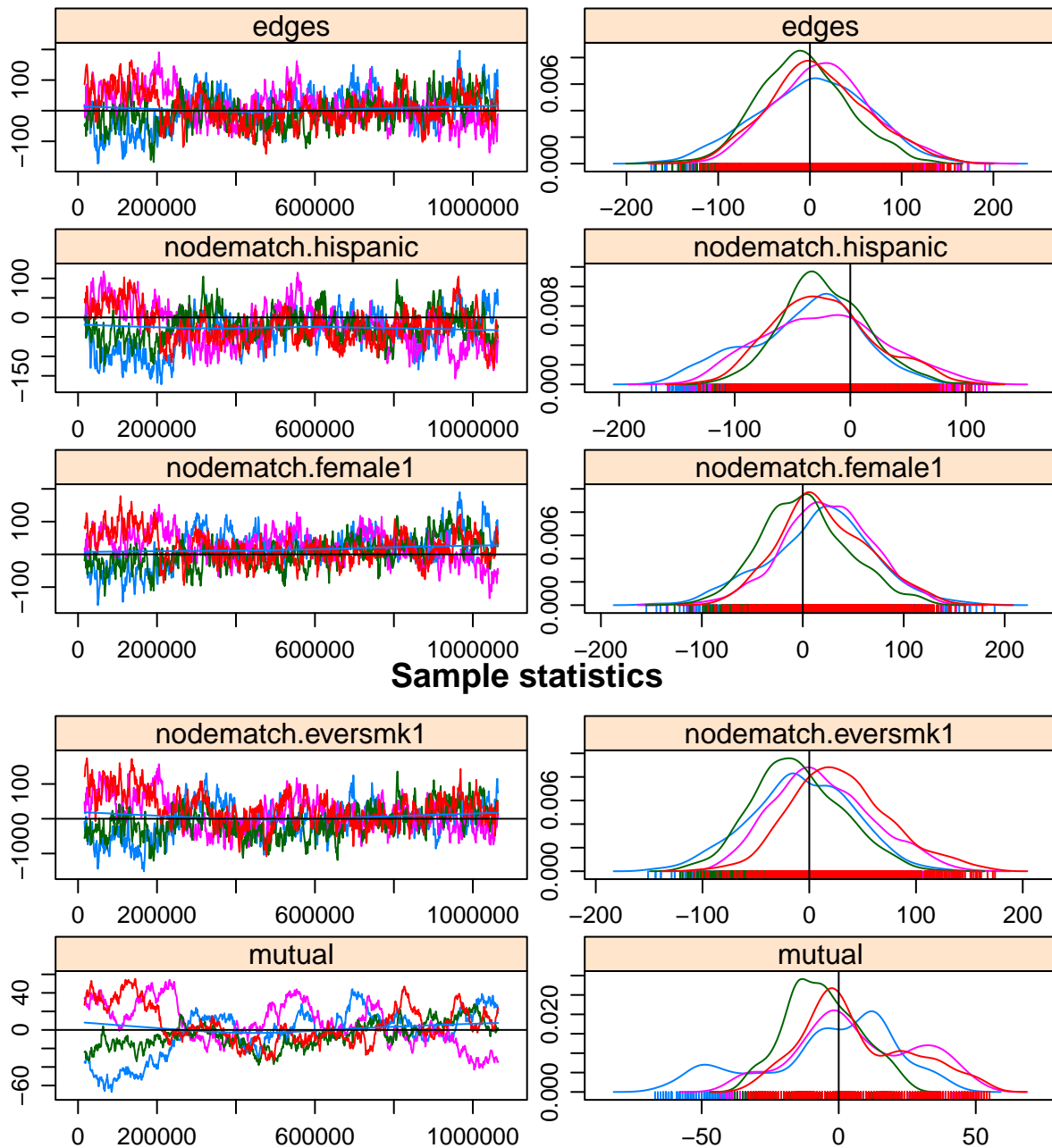
##          6.648201e-06      4.327581e-07      1.340410e-10
## nodematch.eversmk1      mutual
##          4.152969e-13      9.082950e-14
## Joint P-value (lower = worse):  0.0004497061 .
## Chain 4
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##          edges nodematch.hispanic  nodematch.female1
##          3.065          5.834          2.057
## nodematch.eversmk1      mutual
##          4.452          1.770
##
## Individual P-values (lower = worse):
##          edges nodematch.hispanic  nodematch.female1
##          2.179724e-03      5.396178e-09      3.967389e-02
## nodematch.eversmk1      mutual
##          8.487764e-06      7.678079e-02
## Joint P-value (lower = worse):  0.03093665 .
## Chain 4
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##          edges nodematch.hispanic  nodematch.female1
##          -3.482          -3.076          -6.079
## nodematch.eversmk1      mutual
##          -3.388          -3.348
##
## Individual P-values (lower = worse):
##          edges nodematch.hispanic  nodematch.female1
##          4.983811e-04      2.100660e-03      1.212941e-09

```

```
## nodematch.eversmk1          mutual
##      7.037519e-04      8.147796e-04
## Joint P-value (lower = worse):  0.1164457 .
## Chain 4
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##      edges nodematch.hispanic  nodematch.female1
##      4.702      7.223      5.269
## nodematch.eversmk1          mutual
##      6.900      4.079
##
## Individual P-values (lower = worse):
##      edges nodematch.hispanic  nodematch.female1
##      2.574907e-06      5.086754e-13      1.372144e-07
## nodematch.eversmk1          mutual
##      5.194108e-12      4.522375e-05
## Joint P-value (lower = worse):  0.002006982 .
```

```
## Warning in formals(fun): argument is not a function
```

Sample statistics



##

MCMC diagnostics shown here are from the last round of simulation, prior to computation

From the MCMC diagnostics we see that the chain still hasn't converged properly. How do we know this? Well, as a rule of thumb we should expect that at the end of the distribution all chains go at around the same value.


```
# Computing and printing GOF estatistics
```

```
ans_gof <- gof(ans0)
```

```
ans_gof
```

```
##
```

```
## Goodness-of-fit for in-degree
```

```
##
```

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

```
## 0    13   0  2.19   6      0.00
```

```
## 1    34   3  8.96  21      0.00
```

```
## 2    37  13 22.17  34      0.00
```

```
## 3    48  29 41.08  58      0.28
```

```
## 4    37  37 56.89  75      0.02
```

```
## 5    47  45 63.96  81      0.02
```

```
## 6    42  41 63.21  83      0.02
```

```
## 7    39  39 53.20  69      0.04
```

```
## 8    35  23 40.26  62      0.52
```

```
## 9    21  20 28.19  39      0.14
```

```
## 10   12   6 18.20  35      0.22
```

```
## 11   19   3  9.98  21      0.04
```

```
## 12    4   0  5.04  12      0.92
```

```
## 13    7   0  2.71   8      0.10
```

```
## 14    6   0  1.16   6      0.02
```

```
## 15    3   0  0.47   3      0.02
```

```
## 16    4   0  0.19   2      0.00
```

```
## 17    3   0  0.10   1      0.00
```

```
## 18    3   0  0.03   1      0.00
```

```
## 19    2   0  0.00   0      0.00
```

```
## 20    1   0  0.01   1      0.02
```

```
## 22    1   0  0.00   0      0.00
```

```
##
```

```
## Goodness-of-fit for out-degree
```

```
##
```

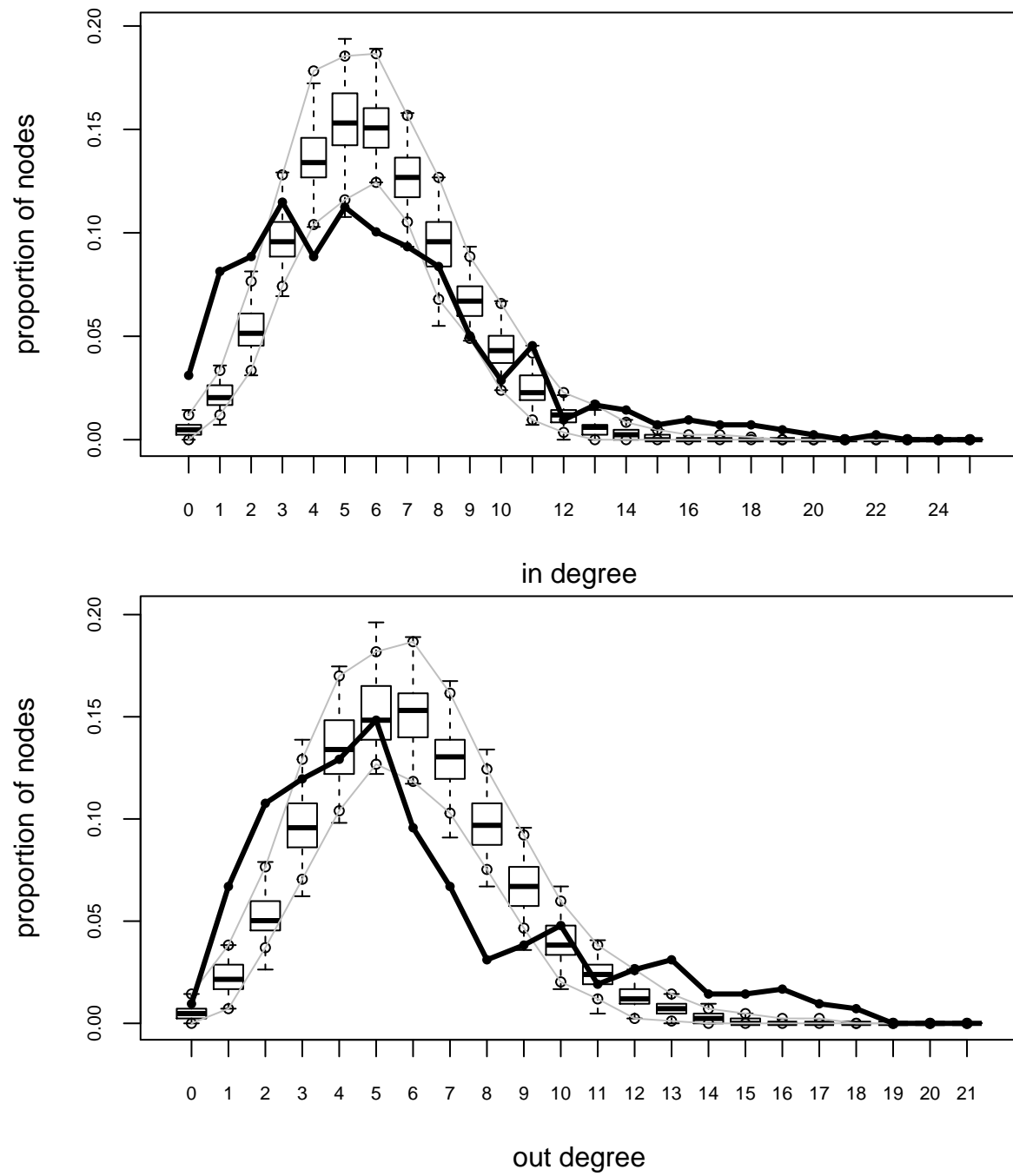
```

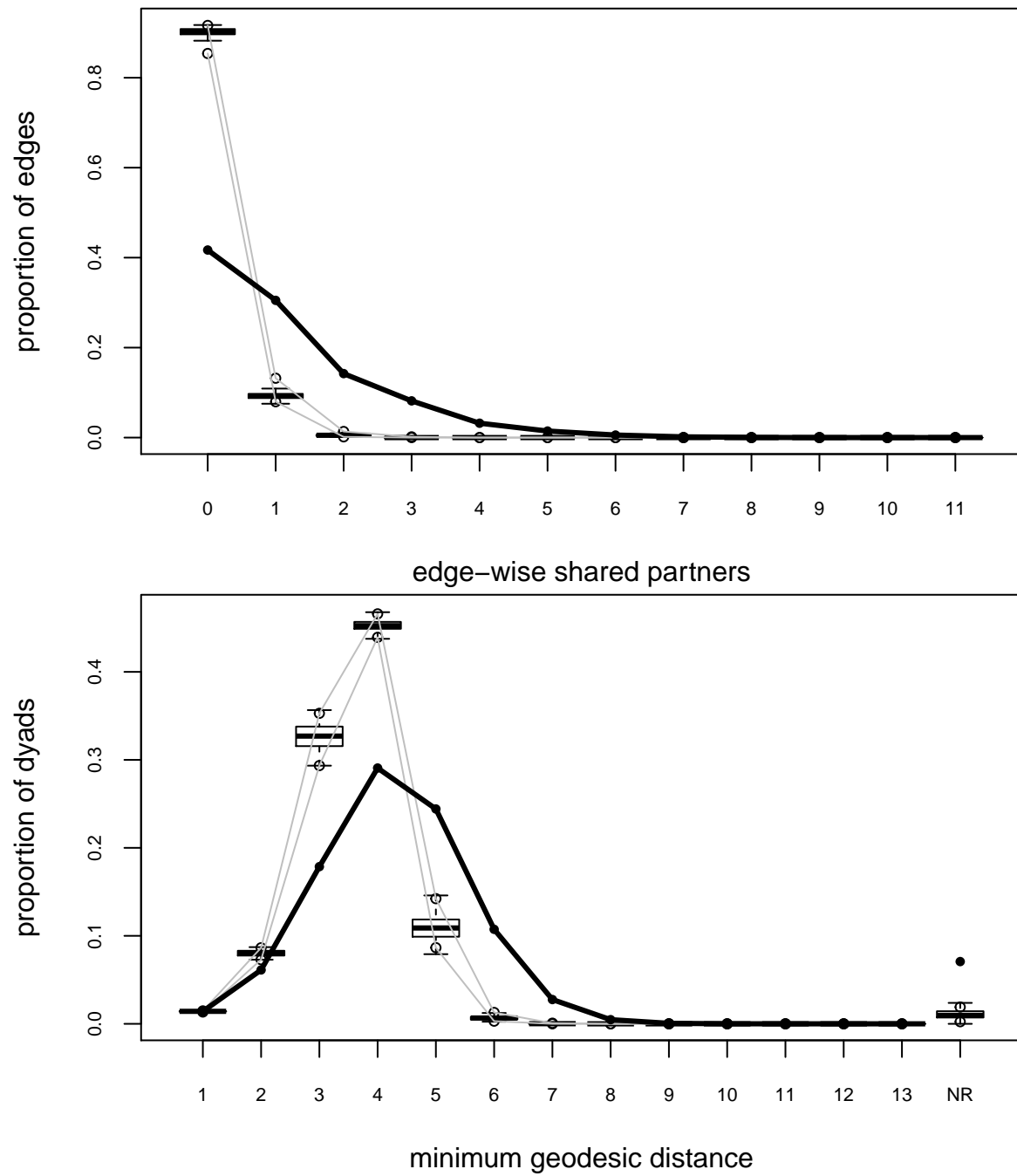
##      obs min  mean max MC p-value
## 0      4   0  2.10   6      0.24
## 1     28   3  9.45  20      0.00
## 2     45  11 22.31  35      0.00
## 3     50  26 40.55  61      0.20
## 4     54  41 56.40  73      0.90
## 5     62  51 63.56  82      1.00
## 6     40  44 63.59  84      0.00
## 7     28  38 54.11  70      0.00
## 8     13  28 40.71  56      0.00
## 9     16  15 28.00  45      0.02
## 10    20   7 16.90  28      0.64
## 11     8   2 10.14  17      0.54
## 12    11   1  5.35  15      0.08
## 13    13   0  2.95  11      0.00
## 14     6   0  1.09   4      0.00
## 15     6   0  0.56   2      0.00
## 16     7   0  0.17   2      0.00
## 17     4   0  0.05   1      0.00
## 18     3   0  0.01   1      0.00
##
## Goodness-of-fit for edgewise shared partner
##
##      obs  min   mean  max MC p-value
## esp0 1032 1941 2219.58 2352      0
## esp1  755  183  232.76  398      0
## esp2  352   1   14.39   98      0
## esp3  202   0    0.86   15      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

```

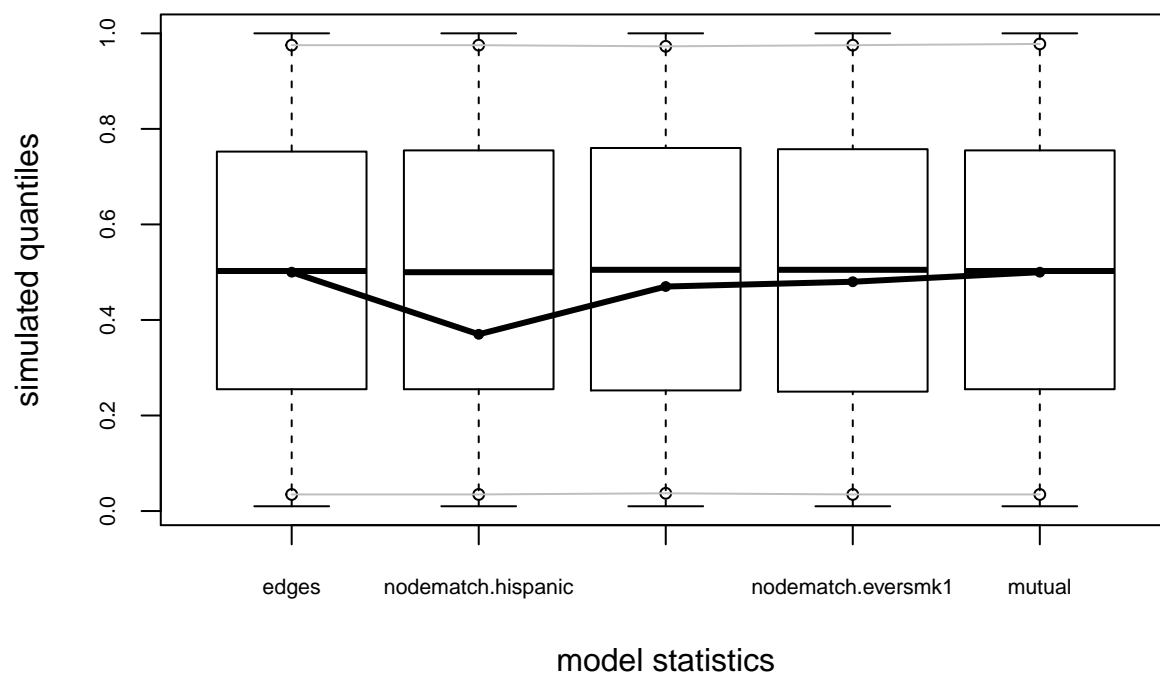
```
##
## Goodness-of-fit for minimum geodesic distance
##
##      obs   min    mean   max MC p-value
## 1    2475  2249  2467.61  2576      1
## 2   10672 11817 13956.56 15192      0
## 3   31134 47358 56761.50 62149      0
## 4   50673 75548 78943.57 81725      0
## 5   42563 13798 19146.59 28769      0
## 6   18719   421  1164.67  2750      0
## 7    4808     0    38.81   220      0
## 8     822     0     0.80    22      0
## 9     100     0     0.00     0      0
## 10      7     0     0.00     0      0
## Inf 12333     0  1825.89  4148      0
##
## Goodness-of-fit for model statistics
##
##              obs   min    mean   max MC p-value
## edges              2475 2249 2467.61 2576      1.00
## nodematch.hispanic 1615 1482 1599.23 1718      0.74
## nodematch.female1  1814 1628 1804.96 1909      0.94
## nodematch.eversmk1 1738 1524 1729.37 1840      0.96
## mutual              486  403  481.13  528      1.00

# Plotting GOF statistics
plot(ans_gof)
```





Goodness-of-fit diagnostics



Chapter 6

Final Words

We have finished a nice book.

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`.

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., 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). doi:[10.18637/jss.v024.i03](https://doi.org/10.18637/jss.v024.i03).