Applied SNA with R

George G. Vega Yon

2018-03-12

Contents

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]

[1] Some 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 extreamly useful, it is not necesary to use it with R.

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 plublished on https://github.com/USCCANA/netdiffuseR, which is usually called the development version.

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

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. You data can be Raw plain files like CSV, tab delimited or specified by column width, for which you can use the `readr` package (Wickham, Hester, and Francois 2017); or it can be binary files like dta (Stata), Octave, SPSS, for which `foreign` (R Core Team 2017) can be used; or it could be excel files in which case you should be using `readxl` (Wickham and Bryan 2017). In our case, the data for this session is in Stata format:

```r
library(foreign)

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

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

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

4.1.2 Creating a unique id for each participant

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:

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

```
## [1]    1 2074
```

As the variable spans up to 2074, we need to set the last 4 units of the variable to store the `photoid`. We will use `dplyr` (Wickham et al. 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 happend 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 nomitated 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, `tidyr` (Wickham and Henry 2017), which we will use to reshape the data, and `stringr` (Wickham 2017), which we will use to process 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 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
```

[1] Please 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.

```
## 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>, 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>,
## #   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:

```r
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 preceeded 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 preceeded 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 se this

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

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

```r
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 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:

```r
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:

```r
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 survery. 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:

```r
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 ba2bab7 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 ba2bab7 (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:

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

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

```r
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:

```r
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 measures[2] 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)$closeness  <- closeness(ig_year1_111, mode = "total")
V(ig_year1_111)$betweeness <- betweenness(ig_year1_111, normalized = TRUE)
```

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

```
# Extracting each vectex 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)),
```

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

```
   outdegree  = quantile(outdegree, c(.025, .5, .975)),
   closeness  = quantile(closeness, c(.025, .5, .975)),
   betweeness = quantile(betweeness, c(.025, .5, .975))
 )
})

stats_degree
```

```
##       indegree outdegree    closeness  betweeness
## 2.5%         0         0 3.526640e-06 0.000000000
## 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 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 a the raw counts, we can normalize the `triadic` object by its sum so we get proportions instead[3]

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

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

Pcent	code	description
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:

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 overalaping. So, instead of using the default size, we will size the vertices by indegree using the `degree` function, and

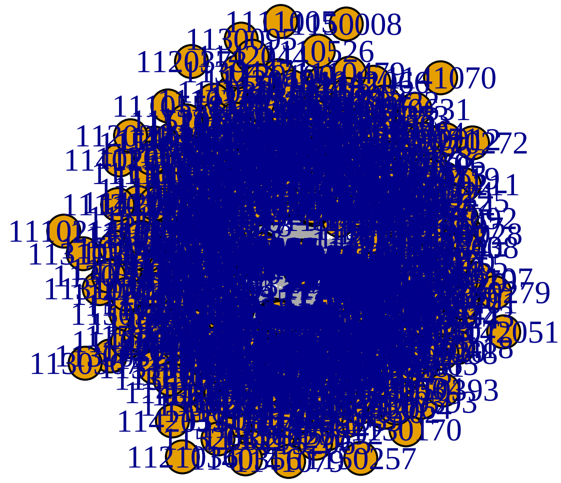


Figure 4.1: A not very nice network plot. This is what we get with the default parameters in igraph.

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 `NA`s with the number 3, so that our vector `col_hispanic` now ranges from 1 to 3 with no `NA`s in it.

3. In the last line we created a vector of colors. Essentially, what we are doing here is telling R to create a vector of length `length(col_hispanic)` by selecting elements by index

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

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

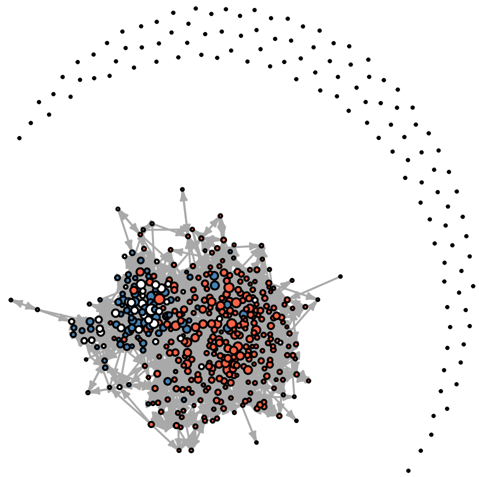


Figure 4.2: Friends network in time 1 for school 111.

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
  )
```

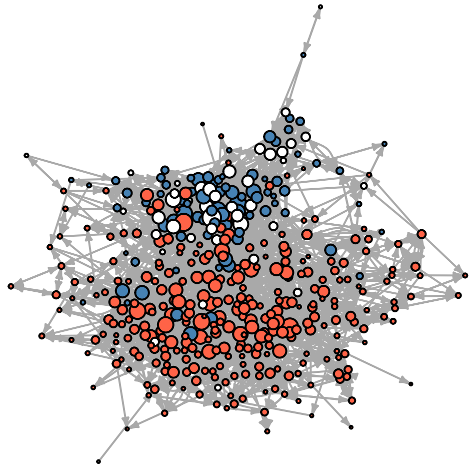


Figure 4.3: Friends network in time 1 for school 111. The graph excludes isolates.

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 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 included 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 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 allow for 2*3 = 6 plots organized in 2 rows and 3 columns, and these will be drawn by row.

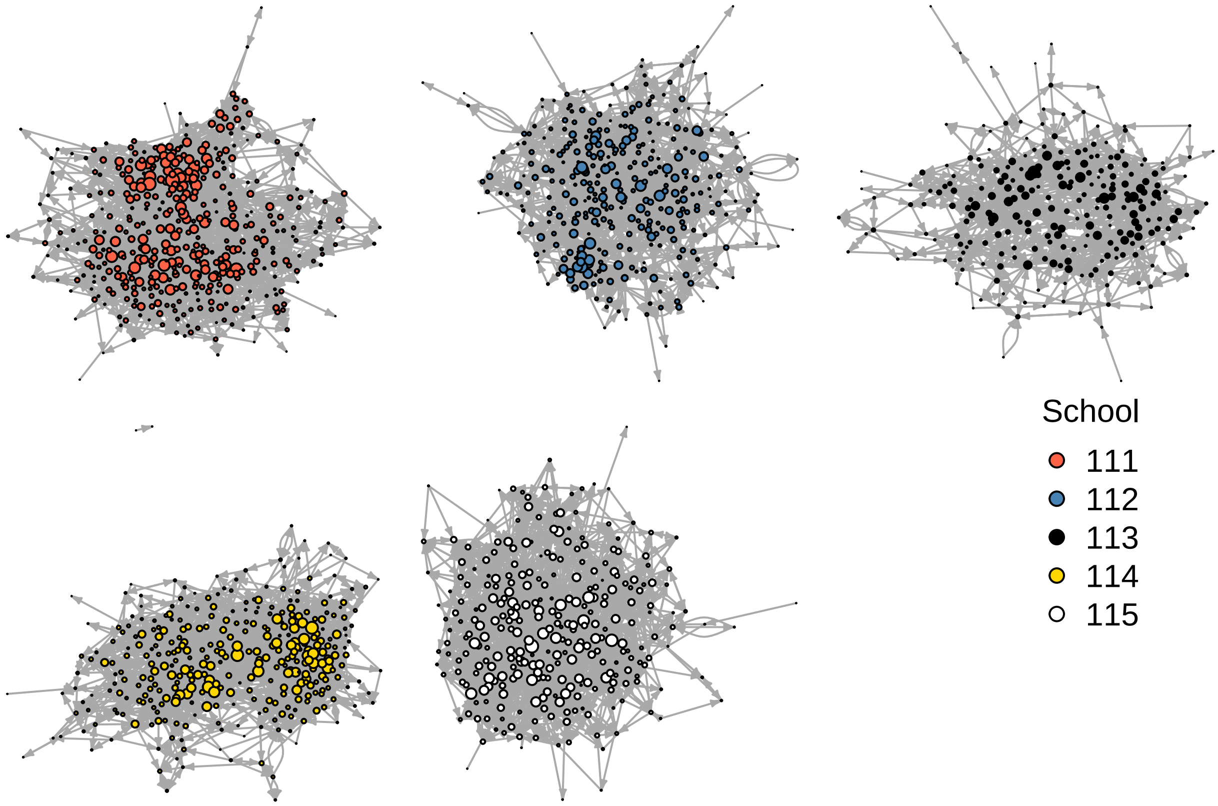


Figure 4.4: All 5 schools in time 1. Again, the graphs exclude isolates.

`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

```r
# 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
```

```r
# 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
```

```r
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

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

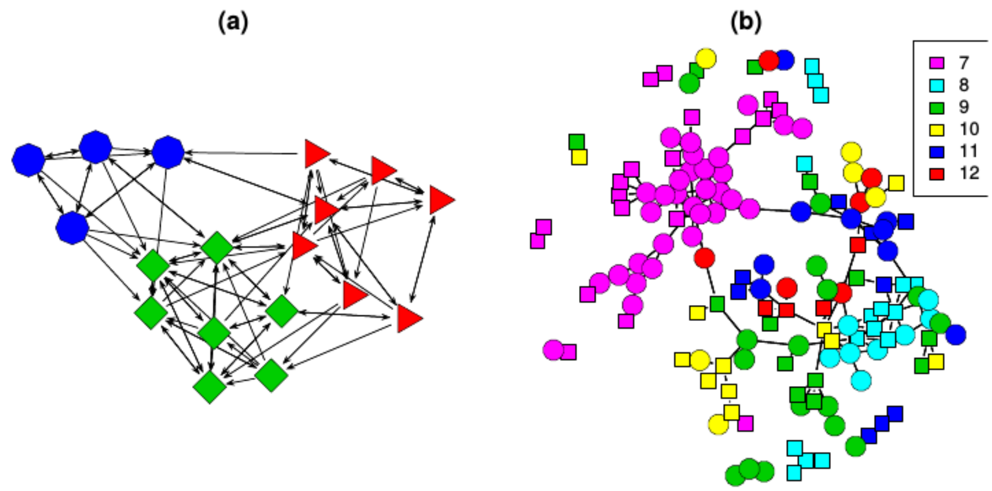


Figure 1: The **(a)** `samplike` and **(b)** `faux.mesa.high` networks described in Section 2. The values of nodal covariates may be indicated using various colors, shapes, and labels of nodes.

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

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

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

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

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]

[1] You 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 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 respectively[2]

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

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

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

```r
# 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

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

```
## + 1/2638 edge from 76b6f3b (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

```r
# 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)
```

5.2 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/bad behaved are the chains.
3. Run the `gof` function to see how good is the model at matching 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 compare[3]

```
ans0 <- ergm(
  network_111 ~
    edges +
    nodematch("hispanic") +
    nodematch("female1") +
    nodematch("eversmk1") +
    mutual
    ,
  constraints = ~bd(maxout = 19),
  control = control.ergm(
    seed        = 1,
    MCMLE.maxit = 10,
```

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

```
    parallel    = 4,
    CD.maxit    = 10
    )
  )
```

So what are we doing here: 1. The model is controling for: a. `edges` Number of edges in the network (as opposed to its density) b. `nodematch("some-variable-name-here")` Includes a term that controls for homophily/heterophily c. `mutual` Number of mutual connections between *i* and *j*. This can be related to, for example, triadic closure.

```
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 +
```

```r
    balance
    ,
  constraints = ~bd(maxout = 19),
  control = control.ergm(
    seed        = 1,
    MCMLE.maxit = 10,
    parallel    = 4,
    CD.maxit    = 10
    )
  )
```

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

```r
library(texreg)
```

```
## Version:  1.36.23
## Date:     2017-03-03
## Author:   Philip Leifeld (University of Glasgow)
##
## Please cite the JSS article in your publications -- see citation("texreg").
##
## Attaching package: 'texreg'
## The following object is masked from 'package:magrittr':
##
##     extract
```

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

```
## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
##
## ===============================================================
```

```
##                        Model 1        Model 2        Model 3
## ---------------------------------------------------------------
## edges                      -5.63 ***      -5.53 ***      -5.56 ***
##                            (0.06)         (0.06)         (0.05)
## nodematch.hispanic          0.37 ***       0.51 ***       0.38 ***
##                            (0.04)         (0.04)         (0.04)
## nodematch.female1           0.82 ***       1.10 ***       0.83 ***
##                            (0.04)         (0.05)         (0.04)
## nodematch.eversmk1          0.33 ***       0.47 ***       0.35 ***
##                            (0.04)         (0.04)         (0.04)
## mutual                      4.09 ***                     -4.76 ***
##                            (0.07)                        (0.25)
## balance                                                   0.02 ***
##                                                          (0.00)
## ---------------------------------------------------------------
## AIC                    -37835.55      -35862.16      -37902.94
## BIC                    -37785.21      -35821.88      -37842.53
## Log Likelihood          18922.78       17935.08       18957.47
## ===============================================================
## *** p < 0.001, ** p < 0.01, * p < 0.05
```

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

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

```
## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
## Note: The constraint on the sample space is not dyad-independent. Null model likelihood
```

	Model 1	Model 2	Model 3
edges	−5.63***	−5.53***	−5.56***
	(0.06)	(0.06)	(0.05)
nodematch.hispanic	0.37***	0.51***	0.38***
	(0.04)	(0.04)	(0.04)
nodematch.female1	0.82***	1.10***	0.83***
	(0.04)	(0.05)	(0.04)
nodematch.eversmk1	0.33***	0.47***	0.35***
	(0.04)	(0.04)	(0.04)
mutual	4.09***		−4.76***
	(0.07)		(0.25)
balance			0.02***
			(0.00)
AIC	-37835.55	-35862.16	-37902.94
BIC	-37785.21	-35821.88	-37842.53
Log Likelihood	18922.78	17935.08	18957.47

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 5.1: Statistical models

5.3 Model Goodness-of-Fit

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 code package (Plummer et al. 2006). This is what is called under the hood:

1. *Empirical means and sd, and quantiles*: The summary statistics of all chains

```
summary(ans0$sample)
```

```
##
## 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                 -32.32 51.14   0.7990          3.557
```

```
## nodematch.hispanic -26.81 39.08   0.6106          2.879
## nodematch.female1  -28.03 44.92   0.7018          3.678
## nodematch.eversmk1 -30.99 45.59   0.7123          3.420
## mutual             -14.35 20.47   0.3199          3.120
##
## 2. Quantiles for each variable:
##
##                     2.5% 25% 50% 75% 97.5%
## edges               -128 -68 -35   1    75
## nodematch.hispanic  -100 -54 -27  -2    53
## nodematch.female1   -115 -59 -29   2    65
## nodematch.eversmk1  -116 -63 -33   1    59
## mutual               -55 -27 -16  -1    29
```

2. *Cross correlation*:

```
coda::crosscorr(ans0$sample)
```

```
##                        edges nodematch.hispanic nodematch.female1
## edges              1.0000000          0.7842851         0.8454275
## nodematch.hispanic 0.7842851          1.0000000         0.6875136
## nodematch.female1  0.8454275          0.6875136         1.0000000
## nodematch.eversmk1 0.8278077          0.6009145         0.6802329
## mutual             0.6761018          0.5362456         0.6581912
##                    nodematch.eversmk1    mutual
## edges                       0.8278077 0.6761018
## nodematch.hispanic          0.6009145 0.5362456
## nodematch.female1           0.6802329 0.6581912
## nodematch.eversmk1          1.0000000 0.6111560
## mutual                      0.6111560 1.0000000
```

3. *Autocorrelation*:

```
coda::autocorr(ans0$sample)
```

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

```
## , , edges
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     1.0000000          0.8219687         0.8705328
## Lag 1024  0.8897580          0.7344981         0.7849176
## Lag 5120  0.6125344          0.5093458         0.5843733
## Lag 10240 0.4944537          0.4263616         0.5010519
## Lag 51200 0.2553223          0.2028499         0.2945153
##           nodematch.eversmk1    mutual
## Lag 0              0.8679762 0.6494130
## Lag 1024           0.7854503 0.6389852
## Lag 5120           0.5671411 0.5952745
## Lag 10240          0.4775676 0.5432361
## Lag 51200          0.2399202 0.3539473
##
## , , nodematch.hispanic
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.8219687          1.0000000         0.7213715
## Lag 1024  0.7273184          0.8790036         0.6501736
## Lag 5120  0.4679330          0.5328138         0.4685746
## Lag 10240 0.3548131          0.3719475         0.3735284
## Lag 51200 0.2275837          0.2027639         0.2450665
##           nodematch.eversmk1    mutual
## Lag 0              0.6933735 0.5332499
## Lag 1024           0.6242548 0.5214353
## Lag 5120           0.4278511 0.4686949
## Lag 10240          0.3486120 0.4199291
## Lag 51200          0.2193803 0.2047705
##
## , , nodematch.female1
##
##                edges nodematch.hispanic nodematch.female1
```

```
## Lag 0     0.8705328          0.7213715         1.0000000
## Lag 1024  0.7882288          0.6530082         0.9067042
## Lag 5120  0.5848867          0.4701031         0.6960912
## Lag 10240 0.5143759          0.4415248         0.6162009
## Lag 51200 0.2943576          0.2717081         0.3756573
##           nodematch.eversmk1    mutual
## Lag 0              0.7737254 0.6670677
## Lag 1024           0.7116355 0.6624944
## Lag 5120           0.5435839 0.6422426
## Lag 10240          0.4804301 0.6076076
## Lag 51200          0.2807997 0.3770320
##
## , , nodematch.eversmk1
##
##               edges nodematch.hispanic nodematch.female1
## Lag 0     0.8679762          0.6933735         0.7737254
## Lag 1024  0.7848037          0.6274016         0.7117444
## Lag 5120  0.5581929          0.4628434         0.5535105
## Lag 10240 0.4779814          0.3987335         0.4923237
## Lag 51200 0.2846086          0.2285842         0.3260756
##           nodematch.eversmk1    mutual
## Lag 0              1.0000000 0.5893853
## Lag 1024           0.9073188 0.5852995
## Lag 5120           0.6557095 0.5659845
## Lag 10240          0.5476628 0.5493667
## Lag 51200          0.2846972 0.4000402
##
## , , mutual
##
##               edges nodematch.hispanic nodematch.female1
## Lag 0     0.6494130          0.5332499         0.6670677
## Lag 1024  0.6513036          0.5378887         0.6639583
## Lag 5120  0.6380464          0.5446367         0.6412776
```

```
## Lag 10240 0.6214549          0.5353477         0.6214122
## Lag 51200 0.3647324          0.2297713         0.4240353
##           nodematch.eversmk1    mutual
## Lag 0              0.5893853 1.0000000
## Lag 1024           0.5878446 0.9884291
## Lag 5120           0.5696277 0.9396082
## Lag 10240          0.5458346 0.8806506
## Lag 51200          0.2897837 0.3673476
##
##
## [[2]]
## , , edges
##
##               edges nodematch.hispanic nodematch.female1
## Lag 0     1.0000000          0.7750707         0.8724304
## Lag 1024  0.8959790          0.6991308         0.7892434
## Lag 5120  0.6254356          0.4780257         0.5724076
## Lag 10240 0.4404273          0.3200836         0.4294502
## Lag 51200 0.2308864          0.1640476         0.2233667
##           nodematch.eversmk1    mutual
## Lag 0              0.7819239 0.6194790
## Lag 1024           0.7143139 0.6116771
## Lag 5120           0.5252397 0.5843330
## Lag 10240          0.3908589 0.5627992
## Lag 51200          0.3403279 0.3877780
##
## , , nodematch.hispanic
##
##               edges nodematch.hispanic nodematch.female1
## Lag 0     0.7750707          1.0000000         0.7208082
## Lag 1024  0.6933293          0.8964603         0.6584405
## Lag 5120  0.4852659          0.6185131         0.4865389
## Lag 10240 0.3607465          0.4570165         0.3876918
```

```
## Lag 51200 0.2184111          0.2426583         0.3035701
##           nodematch.eversmk1    mutual
## Lag 0              0.5456474 0.5213337
## Lag 1024           0.4984152 0.5145885
## Lag 5120           0.3617791 0.4935486
## Lag 10240          0.2595658 0.4717713
## Lag 51200          0.2699015 0.3286800
##
## , , nodematch.female1
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.8724304          0.7208082         1.0000000
## Lag 1024  0.8017653          0.6690313         0.9110176
## Lag 5120  0.6095160          0.5043152         0.6977288
## Lag 10240 0.4673743          0.3829748         0.5448081
## Lag 51200 0.3227987          0.2248235         0.3232075
##           nodematch.eversmk1    mutual
## Lag 0              0.6731106 0.6115803
## Lag 1024           0.6279781 0.6047936
## Lag 5120           0.4963274 0.5765856
## Lag 10240          0.3980467 0.5530858
## Lag 51200          0.4062673 0.4435604
##
## , , nodematch.eversmk1
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.7819239         0.54564737         0.6731106
## Lag 1024  0.7048310         0.48979884         0.6105604
## Lag 5120  0.4864786         0.30516884         0.4420611
## Lag 10240 0.3476654         0.18767747         0.3312933
## Lag 51200 0.1717941         0.03641337         0.1514472
##           nodematch.eversmk1    mutual
## Lag 0              1.0000000 0.5966418
```

```
## Lag 1024           0.9203159 0.5870069
## Lag 5120           0.6928700 0.5517682
## Lag 10240          0.5543427 0.5363205
## Lag 51200          0.4051086 0.2877537
##
## , , mutual
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.6194790         0.52133369         0.6115803
## Lag 1024  0.6181344         0.51940137         0.6086489
## Lag 5120  0.5850524         0.49321390         0.5680945
## Lag 10240 0.5267778         0.44682914         0.5069405
## Lag 51200 0.1282355         0.05773453         0.1568190
##           nodematch.eversmk1    mutual
## Lag 0              0.5966418 1.0000000
## Lag 1024           0.6009132 0.9826734
## Lag 5120           0.5895537 0.9151384
## Lag 10240          0.5528753 0.8372783
## Lag 51200          0.3110325 0.3198872
##
##
## [[3]]
## , , edges
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     1.0000000          0.7286851         0.7653768
## Lag 1024  0.8701425          0.6250434         0.6593788
## Lag 5120  0.5532752          0.3726507         0.4114312
## Lag 10240 0.3792709          0.2291407         0.2819342
## Lag 51200 0.1263857          0.1738039         0.1116028
##           nodematch.eversmk1    mutual
## Lag 0              0.8403781 0.6317173
## Lag 1024           0.7506730 0.6255161
```

```
## Lag 5120           0.5327467 0.5999073
## Lag 10240          0.4192655 0.5592668
## Lag 51200          0.1961950 0.2504959
##
## , , nodematch.hispanic
##
##                 edges nodematch.hispanic nodematch.female1
## Lag 0     0.72868515          1.0000000        0.50111552
## Lag 1024  0.62730728          0.8675990        0.42079182
## Lag 5120  0.40526678          0.5577184        0.24703166
## Lag 10240 0.28843483          0.3818085        0.18144815
## Lag 51200 0.04435087          0.1064335        0.01527911
##           nodematch.eversmk1    mutual
## Lag 0              0.6508997 0.4387059
## Lag 1024           0.5843063 0.4332236
## Lag 5120           0.4540849 0.4196131
## Lag 10240          0.3896065 0.3962962
## Lag 51200          0.1621884 0.1104146
##
## , , nodematch.female1
##
##                 edges nodematch.hispanic nodematch.female1
## Lag 0     0.76537675         0.50111552         1.0000000
## Lag 1024  0.65672139         0.41443939         0.8672803
## Lag 5120  0.40459965         0.20374156         0.5584743
## Lag 10240 0.28883468         0.09726804         0.3958318
## Lag 51200 0.07687284         0.07114294         0.1470283
##           nodematch.eversmk1    mutual
## Lag 0             0.58540195 0.6364623
## Lag 1024          0.50920947 0.6303865
## Lag 5120          0.33572992 0.6060308
## Lag 10240         0.25958351 0.5557550
## Lag 51200         0.06843075 0.1704989
```

```
##
## , , nodematch.eversmk1
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.8403781          0.6508997         0.5854020
## Lag 1024  0.7473610          0.5789305         0.5156466
## Lag 5120  0.5207861          0.4012611         0.3435639
## Lag 10240 0.3970581          0.3226432         0.2656331
## Lag 51200 0.2014317          0.2239727         0.1308759
##           nodematch.eversmk1    mutual
## Lag 0              1.0000000 0.6193419
## Lag 1024           0.8990832 0.6161748
## Lag 5120           0.6522335 0.5988452
## Lag 10240          0.5162238 0.5670618
## Lag 51200          0.2849473 0.3104405
##
## , , mutual
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.6317173          0.4387059         0.6364623
## Lag 1024  0.6273792          0.4354956         0.6314778
## Lag 5120  0.5785931          0.3856113         0.5851342
## Lag 10240 0.5246617          0.3376531         0.5338971
## Lag 51200 0.1809254          0.1913834         0.2329805
##           nodematch.eversmk1    mutual
## Lag 0              0.6193419 1.0000000
## Lag 1024           0.6144173 0.9873210
## Lag 5120           0.5695775 0.9341269
## Lag 10240          0.5190540 0.8654326
## Lag 51200          0.1982159 0.4177271
##
##
## [[4]]
```

```
## , , edges
##
##                  edges nodematch.hispanic nodematch.female1
## Lag 0     1.00000000         0.67290782      0.7963774235
## Lag 1024  0.84452232         0.54545325      0.6779810704
## Lag 5120  0.46245349         0.22234846      0.4044528924
## Lag 10240 0.27748812         0.05584143      0.3191300208
## Lag 51200 0.01398245        -0.05751708      0.0009401853
##           nodematch.eversmk1    mutual
## Lag 0             0.81109405 0.5043317
## Lag 1024          0.70588782 0.4956837
## Lag 5120          0.44860361 0.4434758
## Lag 10240         0.31412151 0.3869787
## Lag 51200         0.04395957 0.1453689
##
## , , nodematch.hispanic
##
##                   edges nodematch.hispanic nodematch.female1
## Lag 0      0.67290782          1.0000000        0.54020161
## Lag 1024   0.54361833          0.8254552        0.43849491
## Lag 5120   0.22063113          0.3544722        0.20155099
## Lag 10240  0.06487559          0.1292919        0.12350389
## Lag 51200 -0.04263724          0.1271527       -0.05216263
##           nodematch.eversmk1      mutual
## Lag 0             0.41421126  0.18938092
## Lag 1024          0.32881855  0.18303230
## Lag 5120          0.13666734  0.13951649
## Lag 10240         0.02975769  0.09319685
## Lag 51200        -0.08914960 -0.03052313
##
## , , nodematch.female1
##
##                  edges nodematch.hispanic nodematch.female1
```

```
## Lag 0      0.7963774         0.54020161         1.0000000
## Lag 1024   0.6866629         0.44782245         0.8676058
## Lag 5120   0.3805378         0.19571567         0.5413447
## Lag 10240  0.2135821         0.06128063         0.3860391
## Lag 51200 -0.1038172        -0.09361460        -0.0488699
##           nodematch.eversmk1     mutual
## Lag 0             0.63624875 0.43870157
## Lag 1024          0.56274254 0.42915171
## Lag 5120          0.36210585 0.36253061
## Lag 10240         0.22379296 0.30603104
## Lag 51200        -0.07368724 0.02328391
##
## , , nodematch.eversmk1
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.81109405          0.4142113        0.63624875
## Lag 1024  0.69818401          0.3236662        0.55441977
## Lag 5120  0.42356060          0.1052627        0.37561810
## Lag 10240 0.26505375         -0.0108905        0.29649077
## Lag 51200 0.04806457         -0.1041468        0.01866324
##           nodematch.eversmk1    mutual
## Lag 0              1.0000000 0.5527542
## Lag 1024           0.8833914 0.5451537
## Lag 5120           0.5922046 0.5103593
## Lag 10240          0.4325324 0.4749615
## Lag 51200          0.1453505 0.2511205
##
## , , mutual
##
##                edges nodematch.hispanic nodematch.female1
## Lag 0     0.50433172         0.18938092         0.4387016
## Lag 1024  0.49106344         0.17944168         0.4330371
## Lag 5120  0.43884568         0.11464054         0.4138038
```

```
## Lag 10240 0.35394592         0.02072383         0.3793707
## Lag 51200 0.05994241        -0.20559828         0.1145074
##           nodematch.eversmk1    mutual
## Lag 0              0.5527542 1.0000000
## Lag 1024           0.5440590 0.9744782
## Lag 5120           0.5057209 0.8660835
## Lag 10240          0.4416049 0.7510162
## Lag 51200          0.1838198 0.3309428
```

4. *Geweke Diagnostic*:

```r
coda::geweke.diag(ans0$sample)
```

```
## [[1]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##              edges nodematch.hispanic  nodematch.female1
##             1.2158             1.9078             1.0548
## nodematch.eversmk1             mutual
##             1.6781             0.4969
##
##
## [[2]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##              edges nodematch.hispanic  nodematch.female1
##              1.451              4.109              2.268
## nodematch.eversmk1             mutual
##              1.108              1.478
##
```

```
##
## [[3]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##              edges nodematch.hispanic  nodematch.female1
##             1.1445             0.3823            -1.4859
## nodematch.eversmk1             mutual
##             1.9638             0.3500
##
##
## [[4]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##              edges nodematch.hispanic  nodematch.female1
##           -0.07832            1.71294            0.62034
## nodematch.eversmk1             mutual
##           -1.35149           -1.05875
```

5. *(not included) Gelman Diagnostic*:

```r
coda::gelman.diag(ans0$sample)
```

```
## Potential scale reduction factors:
##
##                    Point est. Upper C.I.
## edges                    1.23       1.60
## nodematch.hispanic       1.13       1.36
## nodematch.female1        1.13       1.35
## nodematch.eversmk1       1.21       1.58
## mutual                   1.42       2.08
```

```
##
## Multivariate psrf
##
## 1.41
```

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! First, observe that in the plot we see 4 different lines, why is that? Well, since we were running in parallel using 4 cores the algorithm actually ran 4 different chains of the MCMC algorithm. An eyeball test is to see if all the chains moved at about the same place, if we have that we can start thinking about model convergence from the mcmc perspective.

What would be an indicator of no-convergence? Well, if you see something like this:

```r
# Computing and printing GOF estatistics
ans_gof <- gof(ans0)
ans_gof
```

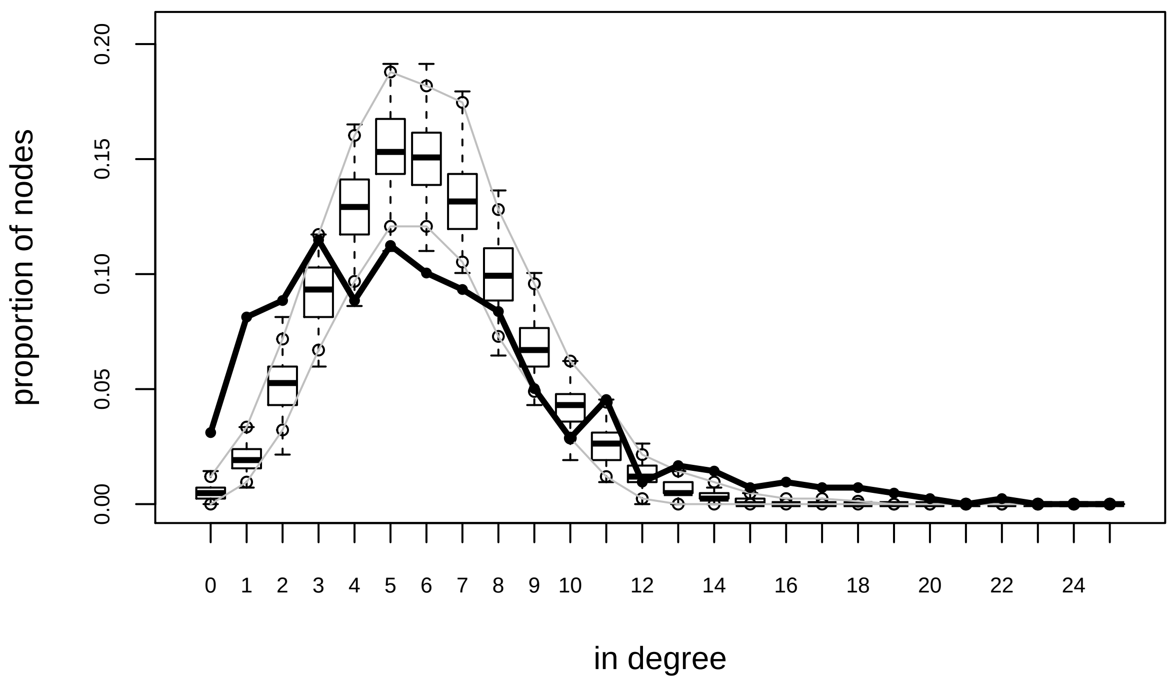
```
##
## Goodness-of-fit for in-degree
##
##    obs min  mean max MC p-value
## 0   13   0  2.05   6       0.00
## 1   34   1  8.51  17       0.00
## 2   37   9 21.64  34       0.00
## 3   48  25 38.73  58       0.16
## 4   37  36 53.98  69       0.02
## 5   47  46 64.97  86       0.02
## 6   42  46 62.93  80       0.00
```

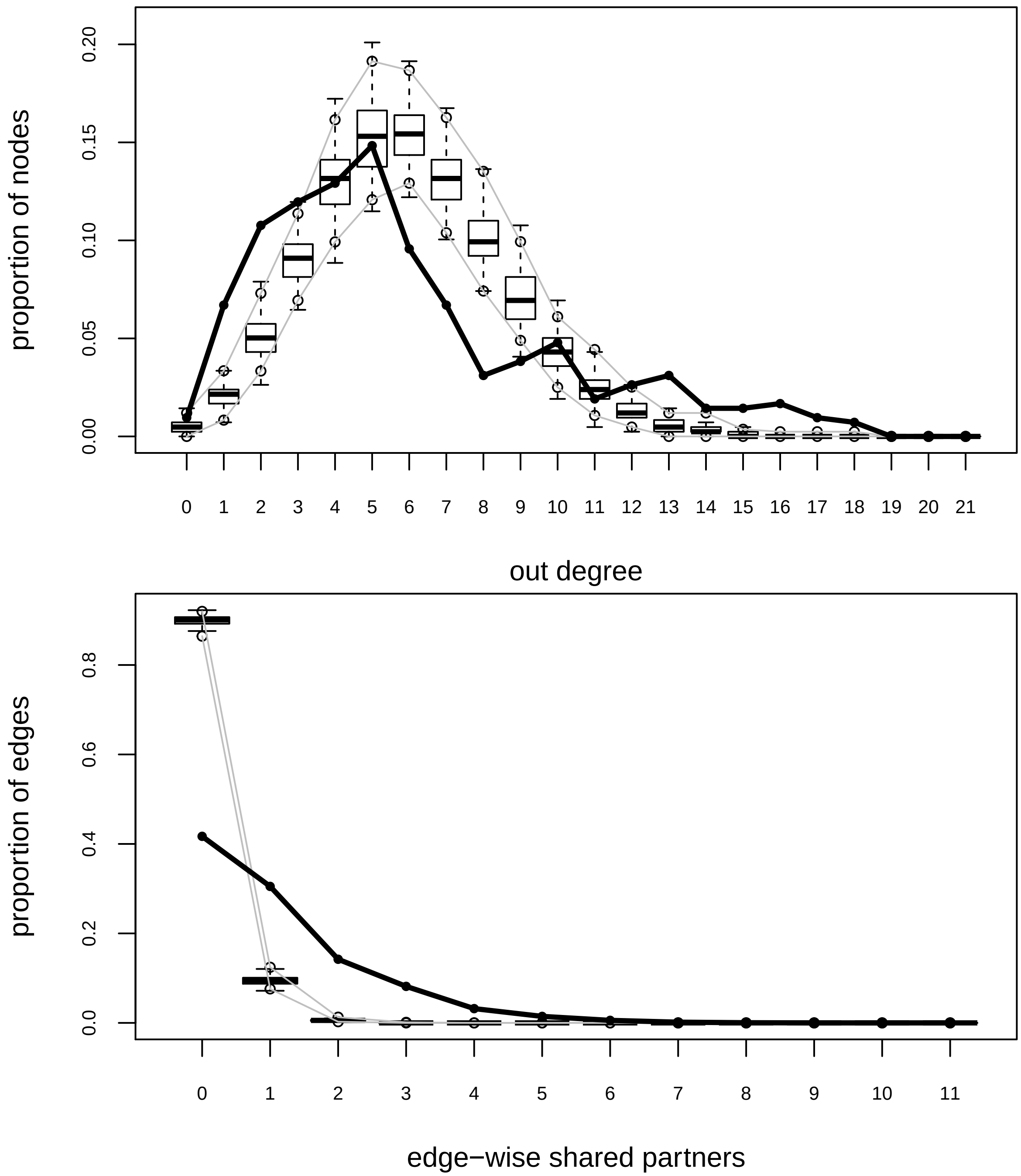
```
## 7   39  31 55.46  75       0.02
## 8   35  27 41.75  57       0.38
## 9   21  18 28.76  42       0.16
## 10  12   8 18.03  31       0.10
## 11  19   4 10.94  24       0.06
## 12   4   0  5.19  13       0.78
## 13   7   0  2.66   7       0.02
## 14   6   0  1.48   5       0.00
## 15   3   0  0.55   3       0.04
## 16   4   0  0.22   2       0.00
## 17   3   0  0.10   1       0.00
## 18   3   0  0.04   2       0.00
## 19   2   0  0.00   0       0.00
## 20   1   0  0.00   0       0.00
## 21   0   0  0.01   1       1.00
## 22   1   0  0.00   0       0.00
##
## Goodness-of-fit for out-degree
##
##    obs min  mean max MC p-value
## 0    4   0  2.12   6       0.26
## 1   28   2  8.84  18       0.00
## 2   45  11 21.15  33       0.00
## 3   50  27 37.98  50       0.02
## 4   54  37 54.49  72       0.92
## 5   62  48 63.90  84       0.78
## 6   40  51 64.48  88       0.00
## 7   28  42 54.98  70       0.00
## 8   13  27 42.19  62       0.00
## 9   16  17 29.46  45       0.00
## 10  20   8 18.07  29       0.72
## 11   8   2 10.20  21       0.56
## 12  11   1  5.32  16       0.06
```

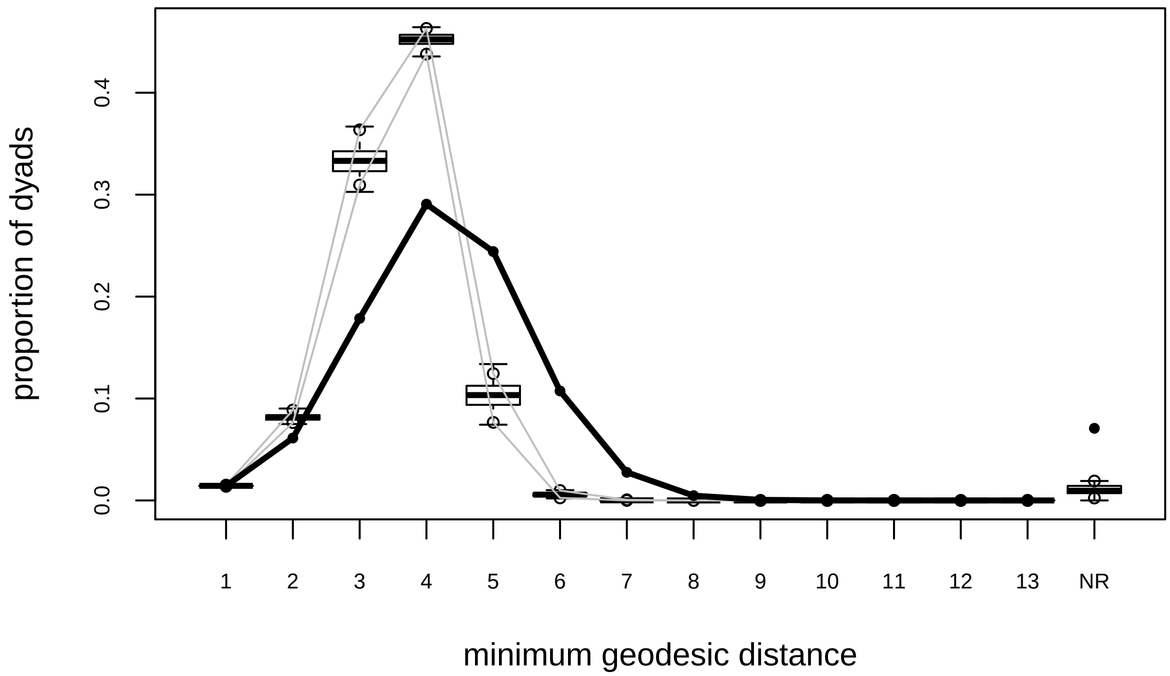
```
## 13  13   0  2.54   6       0.00
## 14   6   0  1.46   6       0.02
## 15   6   0  0.47   4       0.00
## 16   7   0  0.21   2       0.00
## 17   4   0  0.08   2       0.00
## 18   3   0  0.04   1       0.00
## 19   0   0  0.02   1       1.00
##
## Goodness-of-fit for edgewise shared partner
##
##       obs  min    mean  max MC p-value
## esp0 1032 2024 2244.16 2373          0
## esp1  755  174  240.27  395          0
## esp2  352    5   15.08   75          0
## esp3  202    0    0.50    6          0
## esp4   79    0    0.00    0          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  2301  2500.01  2625       0.68
## 2   10672 12158 14250.49 15716       0.00
## 3   31134 49068 58045.75 63937       0.00
## 4   50673 75648 78732.72 80928       0.00
## 5   42563 12954 17973.98 26706       0.00
## 6   18719   346  1021.26  2004       0.00
## 7    4808     1    30.06   182       0.00
## 8     822     0     0.41     7       0.00
## 9     100     0     0.01     1       0.00
```

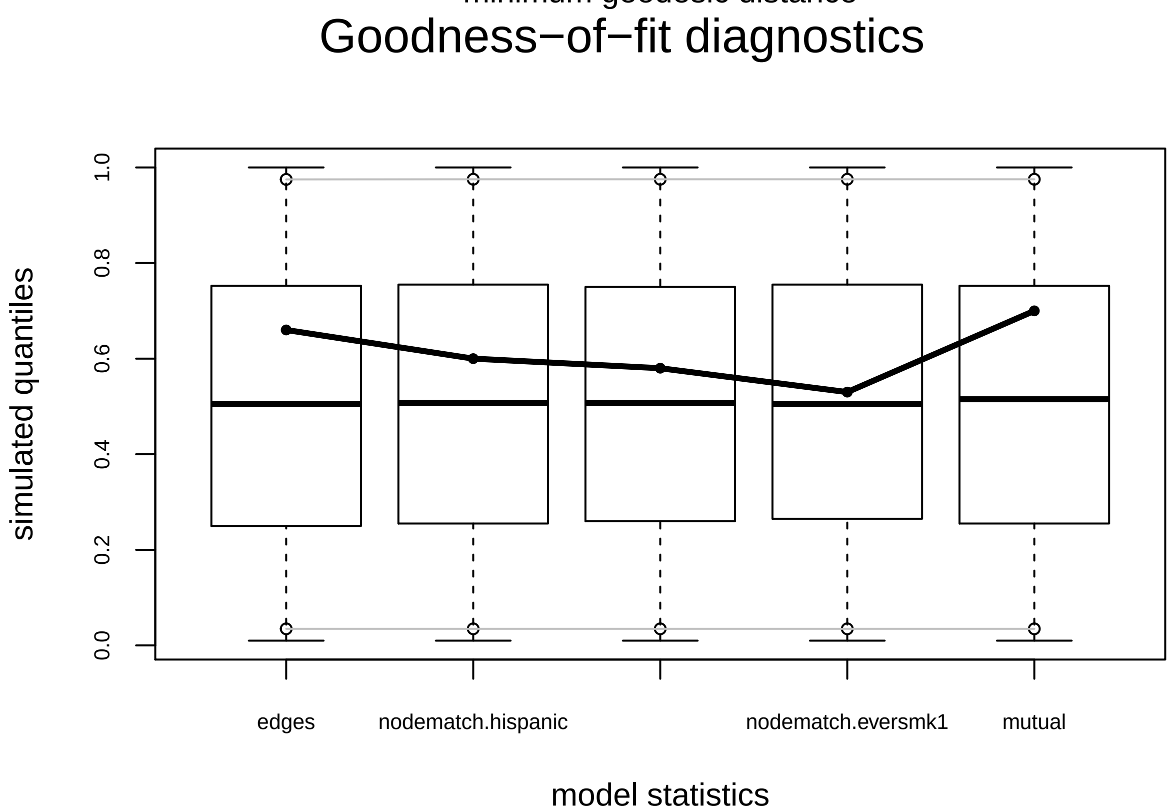
```
## 10       7     0     0.00     0       0.00
## Inf 12333     0  1751.31  3324       0.00
##
## Goodness-of-fit for model statistics
##
##                        obs  min    mean  max MC p-value
## edges                 2475 2301 2500.01 2625       0.68
## nodematch.hispanic 1615 1511 1627.76 1753       0.84
## nodematch.female1  1814 1690 1829.15 1959       0.88
## nodematch.eversmk1 1738 1638 1744.42 1842       0.98
## mutual                 486  449  495.08  554       0.62
```

```r
# Plotting GOF statistics
plot(ans_gof)
```




proportion of nodes
0.00
0.05
0.10
0.15
0.20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
out degree
proportion of edges
0.0
0.2
0.4
0.6
0.8
0 1 2 3 4 5 6 7 8 9 10 11
edge−wise shared partners


proportion of dyads
0.0
0.1
0.2
0.3
0.4
1 2 3 4 5 6 7 8 9 10 11 12 13 NR
minimum geodesic distance


Goodness−of−fit diagnostics
simulated quantiles
0.0
0.2
0.4
0.6
0.8
1.0
edges
nodematch.hispanic
nodematch.eversmk1
mutual
model statistics

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 indivual ever reported himself as hispanic.
- `female1`, `...`, `female4` Indicator variable that equals 1 if the individual reported to be female at the particular wave.

- `grades1`,..., `grades4` Academic grades by wave. Values from 1 to 5, with 5 been the best.
- `eversmk1`, ..., `eversmk4` Indicator variable of ever smoking by wave. A one indicated that the individual had smoked at the time of the survey.
- `everdrk1`, ..., `everdrk4` Indicator variable of ever drinking by wave. A one indicated that the individual had drink at the time of the survey.
- `home1`, ..., `home4` Factor variable for home status by wave. A one indicates home ownership, a 2 rent, and a 3 a “I don’t know”.

During the survey, participants were asked to name up to 19 of their school friends:

- `sch_friend11`, ..., `sch_friend119` School friends nominations (19 in total) for wave 1. The codes are mapped to the variable `photoid`.
- `sch_friend21`, ..., `sch_friend219` School friends nominations (19 in total) for wave 2. The codes are mapped to the variable `photoid`.
- `sch_friend31`, ..., `sch_friend319` School friends nominations (19 in total) for wave 3. The codes are mapped to the variable `photoid`.
- `sch_friend41`, ..., `sch_friend419` School friends nominations (19 in total) for wave 4. The codes are mapped to the variable `photoid`.

References

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.

Csardi, Gabor, and Tamas Nepusz. 2006. "The Igraph Software Package for Complex Network Research." *InterJournal* Complex Systems: 1695. http://igraph.org.

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.

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.

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.

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/.

Matloff, Norman. 2011. *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press.

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. 2017. *Foreign: Read Data Stored by 'Minitab', 'S', 'Sas', 'Spss', 'Stata', 'Systat', 'Weka', 'dBase', ...* https://CRAN.R-project.org/package=foreign.

Ushey, Kevin, Jim Hester, and Robert Krzyzanowski. 2017. *Rex: Friendly Regular Expressions*. https://CRAN.R-project.org/package=rex.

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, and Lionel Henry. 2017. *Tidyr: Easily Tidy Data with 'Spread()' and 'Gather()' Functions*. https://CRAN.R-project.org/package=tidyr.

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, Jim Hester, and Romain Francois. 2017. *Readr: Read Rectangular Text Data*. https://CRAN.R-project.org/package=readr.