Applied SNA with R

George G. Vega Yon

2018-02-02

Contents

1 Prerequisites **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 The Social Network Study . . . 11
4.2 Today's Goals . . . 11
4.3 Data preprocessing . . . 11
4.4 Creating a network . . . 12
4.5 Network descriptive stats . . . 17
4.6 Plotting the network in igraph . . . 20

5 Applications **25**
5.1 Example one . . . 25
5.2 Example two . . . 25

6 Final Words **27**

A Datasets **29**
A.1 SNS data . . . 29

Chapter 1

Prerequisites

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 2

Introduction

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.

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 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 plublished 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

4.1 The Social Network Study

The data can be downloaded from here.

The codebook for the data provided here is in the appendix.

4.2 Today's Goals

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

4.3 Data preprocessing

4.3.1 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; 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
```

4.3.2 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 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))
```

4.4 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.

4.4.1 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     1.00      NA      NA       0       0   NA      NA
##  2       2    111     1.00       0      NA      NA       0    3.00   NA
##  3       7    111     0          1       1       1       1    5.00    4.50
##  4      13    111     1.00       1       1       1       1    2.50    2.50
##  5      14    111     1.00       1       1       1      NA    3.00    3.50
##  6      15    111     1.00       0       0       0       0    2.50    2.50
##  7      20    111     1.00       1       1       1       1    2.50    2.50
##  8      22    111     1.00      NA      NA       0       0   NA      NA
##  9      25    111     0          1       1      NA       1    4.50    3.50
## 10      27    111     1.00       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>
```

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

   ```r
   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 `NA`s, so we filter the data:

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

```
## # A tibble: 39,561 x 4
##         id school varname      content
##      <dbl>  <int> <chr>          <int>
##  1 1110002    111 sch_friend11     424
##  2 1110007    111 sch_friend11     629
##  3 1110013    111 sch_friend11     232
##  4 1110014    111 sch_friend11     582
##  5 1110015    111 sch_friend11      26
##  6 1110020    111 sch_friend11     528
##  7 1110025    111 sch_friend11     135
```

```
##   8 1110027    111 sch_friend11     346
##   9 1110029    111 sch_friend11     369
##  10 1110030    111 sch_friend11     462
## # ... 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     = str_extract(varname, "(?<=[a-z])[0-9]"),
    nnom     = 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> <chr> <chr>
##   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

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

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

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

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

Now that we have this edgelist, we can create an igraph object

4.4.2 igraph network

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

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) %>%
  graph_from_data_frame(
    vertices = vertices
    )
```

```
## Error in graph_from_data_frame(., vertices = vertices): Some vertex names in edge list are not listed in v
```

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) %>%
  graph_from_data_frame(
    vertices = vertices
    )

ig_year1
```

```
## IGRAPH a863f3e 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)
## + edges from a863f3e (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
## [25] 1110096->1110065 1110109->1110330 1110114->1110172 1110115->1110039
## + ... 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.

4.5 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) # we have no edge attributes here
```

```
## character(0)
```

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:

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

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

The `which` function in R returns a vector of indices indicating which elements 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[1] for each vertex and store them in the igraph object itself:

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

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

```r
# Extracting each 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)),
   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
```

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

```
## 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
```

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

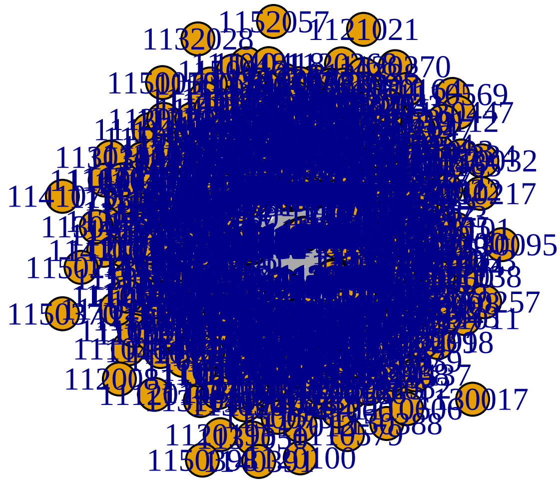

Figure 4.1: A not very nice network plot. This is what we get with the default parameters in igraph.

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.

4.6 Plotting the network in igraph

4.6.1 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.
2. All the vertices have the same size, and more over, are overalaping.
3. Given the number of vertices in these networks, the labels are not useful here.

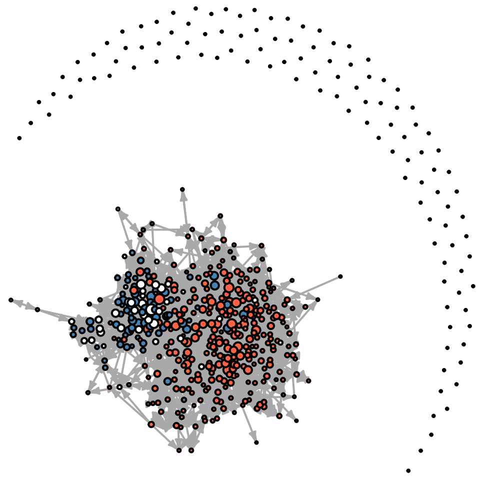

Figure 4.2: Friends network in time 1 for school 111. The graph excludes isolates.

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

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

```r
col_eversmk1 <- V(ig_year1_111)$eversmk1 + 1
col_eversmk1 <- coalesce(col_eversmk1, 3)
col_eversmk1 <- c("steelblue", "tomato", "white")[col_eversmk1]
```

```r
# Fancy graph
set.seed(1)
plot(
  ig_year1_111,
  vertex.size      = degree(ig_year1_111)/10 +1,
  vertex.label     = NA,
  edge.arrow.size = .25,
  layout           = layout_with_fr,
  vertex.color     = col_eversmk1
  )
```

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), we just need t

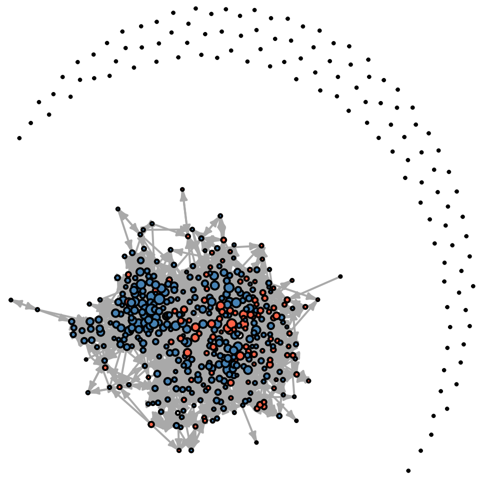

Figure 4.3: Friends network in time 1 for school 111. The graph excludes isolates.

4.6.2 Multiple plots

```
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)
  )

  # Computing colors


  # Fancy graph
  set.seed(1)
  plot(
    subnet,
    vertex.size     = degree(subnet)/10,
    vertex.label    = NA,
    edge.arrow.size = .25,
    vertex.color    = vcol,
    layout          = layout_with_fr,
    ...
    )
}
```

```
# 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")
```

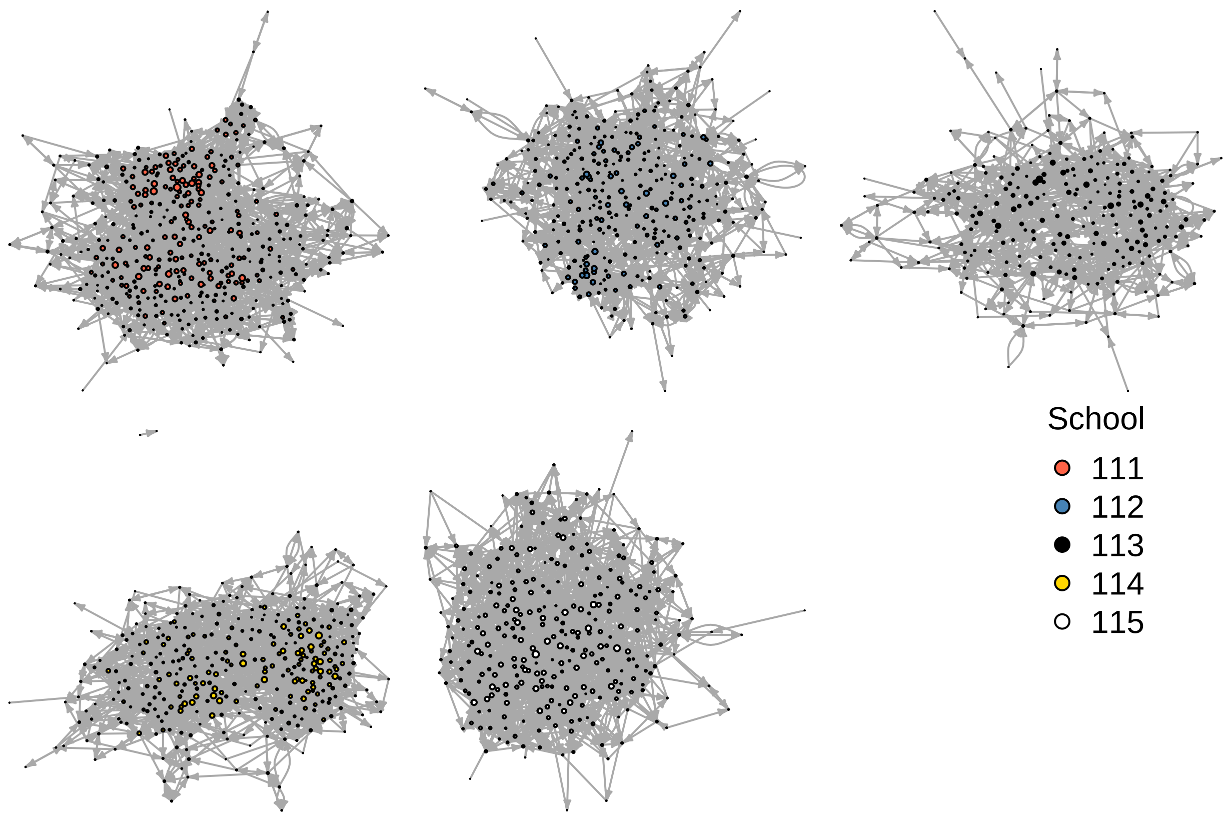

Figure 4.4: All 5 schools in time 1. Again, the graphs exclude isolates.

```
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"
  )
```

- `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.

  `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)` Finally, this line allows us to restore the plotting parameters.

Chapter 5

Applications

Some significant applications are demonstrated in this chapter.

5.1 Example one

5.2 Example two

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`.

Bibliography