Applied SNA with R

George G. Vega Yon

2018-02-28

Contents

Chapter 1

About this book

Chapter 2

Introduction

Chapter 3

R Basics

3.1 What is R

3.2 How to install packages

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; 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:

```r
library(dplyr)
library(magrittr)
library(foreign)

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

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

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

4.1.2 Creating a unique id for each participant

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

```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`. 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))
```

4.2 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.

```r
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*.

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

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

```r
dat
```

```
## # A tibble: 2,164 x 100
##    photoid school hispanic female1 female2 female3 female4 grades1 grades2
##      <int>  <int>    <dbl>   <int>   <int>   <int>   <int>   <dbl>   <dbl>
##  1       1    111       1.      NA      NA       0       0   NA      NA
##  2       2    111       1.       0      NA      NA       0    3.00   NA
##  3       7    111       0.       1       1       1       1    5.00    4.50
##  4      13    111       1.       1       1       1       1    2.50    2.50
##  5      14    111       1.       1       1       1      NA    3.00    3.50
##  6      15    111       1.       0       0       0       0    2.50    2.50
##  7      20    111       1.       1       1       1       1    2.50    2.50
##  8      22    111       1.      NA      NA       0       0   NA      NA
##  9      25    111       0.       1       1      NA       1    4.50    3.50
## 10      27    111       1.       0      NA       0       0    3.50   NA
## # ... with 2,154 more rows, and 91 more variables: grades3 <dbl>,
## #   grades4 <dbl>, 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     = 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 `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     = 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. 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:

```
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 7532eb5 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 7532eb5 (vertex names):
##  [1] 1110007->1110629 1110013->1110232 1110014->1110582 1110015->1110026
##  [5] 1110025->1110135 1110027->1110346 1110029->1110369 1110035->1110034
##  [9] 1110040->1110390 1110041->1110557 1110044->1110027 1110046->1110030
## [13] 1110050->1110086 1110057->1110263 1110069->1110544 1110071->1110167
## [17] 1110072->1110289 1110073->1110014 1110075->1110352 1110084->1110305
## [21] 1110086->1110206 1110093->1110040 1110094->1110483 1110095->1110043
## + ... omitted several edges
```

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

4.3 Network descriptive stats

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

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

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

```
list.edge.attributes(ig_year1)
```

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

Just like we would do with data frames, accessing vertex attributes is done via the dollar sign operator $ together with the V function, for example, accessing the first 10 elements of the variable `hispanic` can be done as follows:

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

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

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

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

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

The `which` function in R returns a vector of indices indicating which elements are true. In our case it will return a vector of indices of the vertices which have the attribute `school` equal to 111. Now that we have our subgraph, we can compute different centrality measures[1] 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)),
   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

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

with the dataset but without mentioning its name everytime that we ask for a variable. Without using the `with` function, the previous could have been done as follows:

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

Now we will compute some statistics at the graph level:

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

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

Triadic census

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

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

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

Pcent	code	description
95.88	003	A,B,C, the empty graph.
2.89	012	A->B, C, the graph with a single directed edge.
1.16	102	A<->B, C, the graph with a mutual connection between two vertices.
0.01	021D	A<-B->C, the out-star.
0.01	021U	A->B<-C, the in-star.
0.02	021C	A->B->C, directed line.
0.01	111D	A<->B<-C.
0.01	111U	A<->B->C.
0.00	030T	A->B<-C, A->C.
0.00	030C	A<-B<-C, A->C.
0.00	201	A<->B<->C.
0.00	120D	A<-B->C, A<->C.
0.00	120U	A->B<-C, A<->C.
0.00	120C	A->B->C, A<->C.
0.00	210	A->B<->C, A<->C.
0.00	300	A<->B<->C, A<->C, the complete graph.

4.4 Plotting the network in igraph

4.4.1 Single plot

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

```r
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

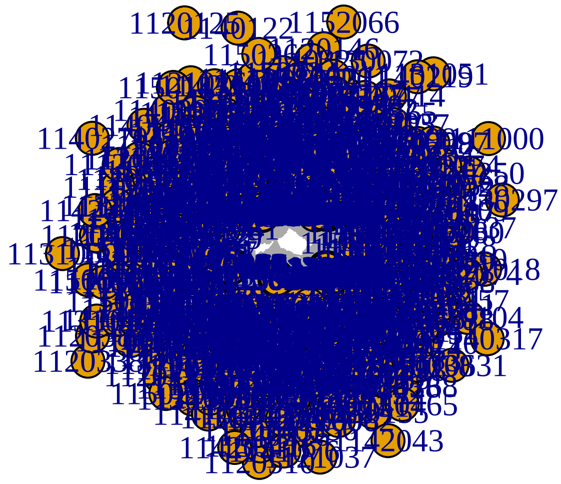

Figure 4.1: A not very nice network plot. This is what we get with the default parameters in igraph.

passing the vector of degrees to `vertex.size`.[2]

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

[2] 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
  )
```

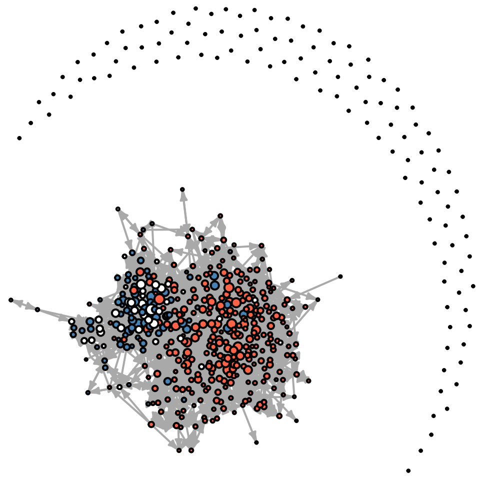

Figure 4.2: Friends network in time 1 for school 111.

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
  )
```

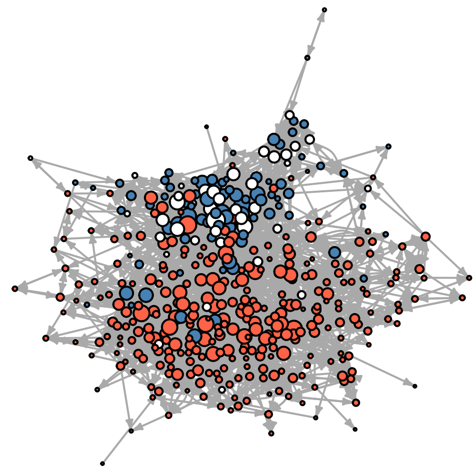

Figure 4.3: Friends network in time 1 for school 111. The graph excludes isolates.

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.

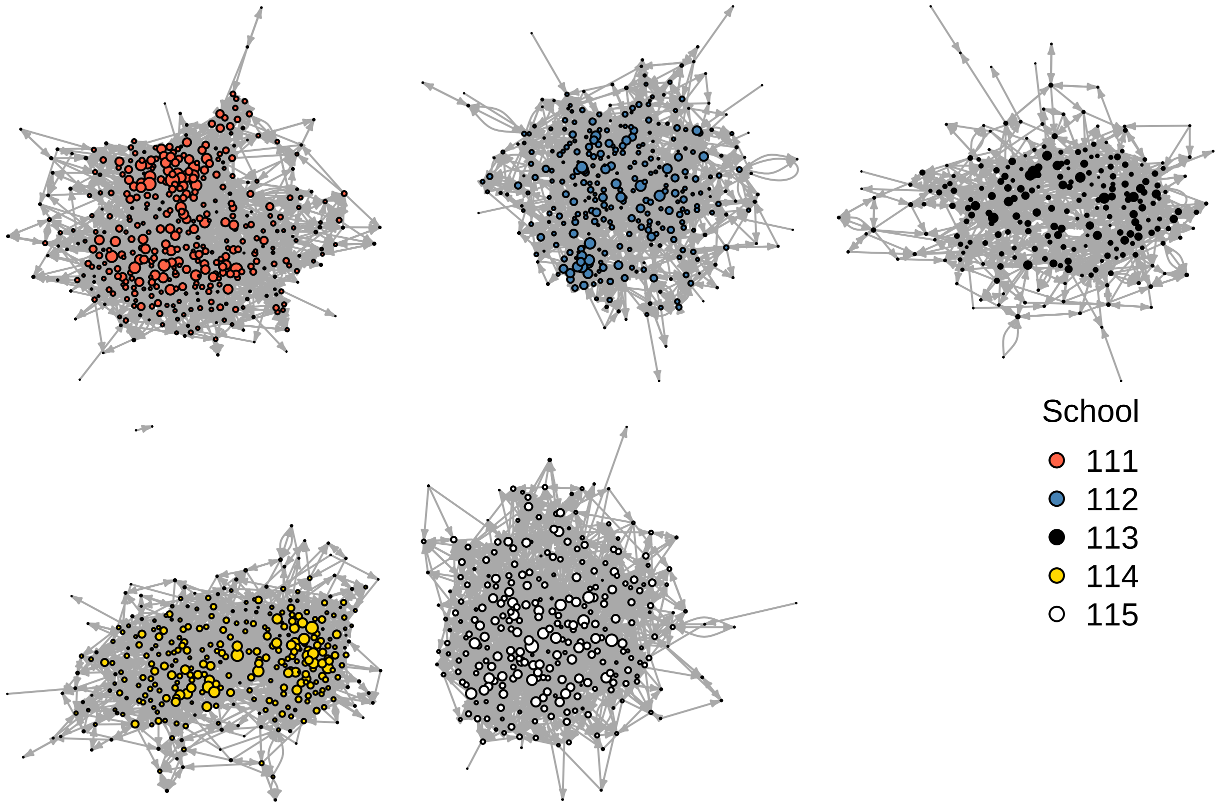

Figure 4.4: All 5 schools in time 1. Again, the graphs exclude isolates.

`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
indeg_nom_cor <- group_by(edgelist, to, to_name, to_school) %>%
  summarise(
    indeg   = n(),
    nom_avg = 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    4.50
##  2    3. 1110007    111     7    5.71
##  3    4. 1110013    111     6    5.83
##  4    5. 1110014    111    19    7.47
##  5    6. 1110015    111     3    6.67
##  6    7. 1110020    111     6    6.50
##  7    9. 1110025    111     6    4.67
##  8   10. 1110027    111    13    4.54
##  9   11. 1110029    111    14    7.64
## 10   12. 1110030    111     6    4.50
## # ... 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 = 5.0502, df = 1559, p-value = 4.933e-07
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.07774571 0.17538115
## sample estimates:
##       cor
## 0.1268707
```

4.6 Exponential Random Graph Models

```r
library(ergm)
```

Chapter 5

Applications

5.1 Example one

5.2 Example two

Chapter 6

Final Words

Chapter 7

Placeholder

Appendix A

Datasets

A.1 SNS data

Bibliography