Network Theory and Brain Imaging 2020: Workshop (R)

In this workshop, the idea is for you to get a feel of some network software, see how to process the network objects, and how to analyse and plot networks.

Before starting, you are going to want to install the network package.

```
install.packages("tidyverse")
install.packages('tidygraph')
install.packages('ggraph')
```

Let's begin

Loading the required packages.

```
library('tidyverse')
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library('tidygraph')
```

```
##
## Attaching package: 'tidygraph'
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
library('ggplot2')
library('ggraph')
```

Creating your first network

A network is a graph object that consists of a set of nodes and a set of edges. To define a graph object in *tidygraph*, we call `tbl_graph`. This function takes `nodes` and `edges` as input arguments.

Defining the nodes

Below we create a data frame of nodes. Each row represents a node. What the column names are called is up to you.

Below we will create a dataframe consisting of 6 nodes with the column name 'labels':

```
node_df <- data.frame(labels = c('Ashely', 'Blake',
                                 'Casey', 'Dylan',
                                 'Elliot', 'Finley'))
```

However, when referencing this list, we need to know the order of nodes (e.g. 1 implies Ashley, 2 implies Blake, and so on).

You can use the special column name `node_keys`, if you want to tell tidygraph how you will refer to the nodes:

```
node_df <- data.frame(node_key = c('A', 'B', 'C', 'D', 'E', 'F'),
                      labels = c('Ashely', 'Blake',
                                 'Casey', 'Dylan',
                                 'Elliot', 'Finley'))
```

The edges

Just like with the nodes, we are going to specify a data frame for the edges. Even if undirected, the column names need to be "from" and "to". These either reference the row indexes (if `node_key` does not exist) *or* they reference the `node_key` (if they exist).

You can add additional information to the edges. For example, if you have weights, you can add the weight column, as below. Other names can be voluntary as well if you have any additional information you want to add.

```
edge_df <- data.frame(from = c('A', 'A', 'B', 'C', 'D', 'D', 'E'),
                      to = c('B', 'C', 'C', 'D', 'E', 'F', 'F'),
                      weight = c(0.75, 1, 0.5, 1, 0.5, 1, 0.75))
```

Defining the graph

With these two data frames for the nodes and the edges, we can now create a network.

If the network is undirected, we have to specify this by setting `directed` option to be FALSE:

```
net1 = tbl_graph(nodes = node_df, edges = edge_df, directed=F)
```

Exploring the network object.

To get a summary of the network, we can type:

```
summary(net1)
```

```
## IGRAPH d703b9b U-W- 6 7 --
## + attr: node_key (v/x), labels (v/x), weight (e/n)
```

This shows quite a bit of information:

(1) tidygraph is a wrapper for igraph.
(2) The "U-W-" refer to the network being undirected (U) and weighted (W).
(3) the "6 7" implies 6 nodes and 7 edges.
(4) that there are three attributes: node_keys, labels, and weights (all our column names except for "from" and "to").
(5) The v and e in brackets indicate if it is an attribute of the nodes (v for vertices) or edges.
(6) x refers to a string, n to numeric/floats.

We can peak into the network object as well:

```
net1
```

```
## # A tbl_graph: 6 nodes and 7 edges
## #
## # An undirected simple graph with 1 component
## #
## # Node Data: 6 x 2 (active)
##   node_key labels
##   <fct>    <fct>
## 1 A        Ashely
## 2 B        Blake
## 3 C        Casey
## 4 D        Dylan
## 5 E        Elliot
## 6 F        Finley
## #
## # Edge Data: 7 x 3
##    from    to weight
##   <int> <int>  <dbl>
## 1     1     2   0.75
## 2     1     3   1
## 3     2     3   0.5
## # ... with 4 more rows
```

Where we get a little more information about the network.

dplyr: the tidyverse way with pipe operators and verbs

If you have experience in tidyverse (specifically the package dplyr), this will be obvious for you.

Pipe operators

Pipe operators try and make the code more readable.

The symbol %>% (Shortcut: ctrl + shift + M) connects statements.

Consider the following code:

```
tmp <- mean(c(1,2,3,4,5))
tmp
```

```
## [1] 3
```

With join operators, the innermost brackets come first, and you work outward.

```
tmp <- c(1,2,3,4,5) %>%
       mean()
tmp
```

```
## [1] 3
```

Thus the logic in base R is:

```
y <- A(B(C(x), arg))
```

And in tidyverse/dlpyr, it is:

```
y <- C(x) %>%`
     B(arg) %>%
     A() %>%
```

The advocated benefit for this is that it makes code easier to read.

Tidygraph is a wrapper for igraph that is compatible with tidyverse.

Verbs

There are multiple functions for dplyr which help interact with the data. For more information about data management in tidyverse/dplyr, see this cheatsheat. Here we will go through two examples.

Let us say that we want to look at all the edges in edge_df where an edge contains Casey (C). To do this, the verb to use is filter.

```
caseys_edges <- edge_df %>%
                filter(to == 'C' | from == 'C')
caseys_edges
```

```
##   from to weight
## 1    A  C    1.0
## 2    B  C    0.5
## 3    C  D    1.0
```

Next, let us just grab all the weights out of the dataframe.

```
weights <- edge_df %>%
           select(weight)
weights
```

```
##   weight
## 1   0.75
## 2   1.00
## 3   0.50
## 4   1.00
## 5   0.50
## 6   1.00
## 7   0.75
```

And these can be combined, so we can grab the weights of Casey's edges:

```
casey_weights <- edge_df %>%
                 filter(to == 'C' | from == 'C') %>%
                 select(weight)
casey_weights
```

```
##   weight
## 1    1.0
## 2    0.5
## 3    1.0
```

tidygraph verb: activate

In the example above, the network object was not used. We were only operating on the edge dataframe. In tidygraph, there is a particular verb `activate` which you call to state if you are applying your later queries to the nodes or edges. After activating the nodes or the edges, then everything works as expected. Below we do as above but using net1. We first activate the edges before filtering and selecting. This creates a new network object but with only three edges.

```
net_casey_edges <- net1 %>%
        activate(edges) %>%
        filter(to == '3' | from == '3') %>%
```

```
        select(weight)
net_casey_edges
```

```
## # A tbl_graph: 6 nodes and 3 edges
## #
## # An unrooted forest with 3 trees
## #
## # Edge Data: 3 x 3 (active)
##    from    to weight
##   <int> <int>  <dbl>
## 1     1     3    1
## 2     2     3    0.5
## 3     3     4    1
## #
## # Node Data: 6 x 2
##   node_key labels
##   <fct>    <fct>
## 1 A        Ashely
## 2 B        Blake
## 3 C        Casey
## # ... with 3 more rows
```

Sidenote: shortcut pipes for activate

Instead of always writing `activate(nodes)` and `activate(edges)`, there are two shortcut pipes `%N>%` and `%E>%` that you can use instead. Compare the below example to the example above:

```
net_casey_edges <- net1 %E>%
                   filter(to == '3' | from == '3') %>%
                   select(weight)
net_casey_edges
```

```
## # A tbl_graph: 6 nodes and 3 edges
## #
## # An unrooted forest with 3 trees
## #
## # Edge Data: 3 x 3 (active)
##    from    to weight
##   <int> <int>  <dbl>
## 1     1     3    1
## 2     2     3    0.5
## 3     3     4    1
## #
## # Node Data: 6 x 2
##   node_key labels
##   <fct>    <fct>
## 1 A        Ashely
## 2 B        Blake
## 3 C        Casey
## # ... with 3 more rows
```

The rest of the tutorial still writes out `activate()` to avoid confusion.

Adding and removing edges.

We have now defined a network and learned how to integrate querying the network with the rest of R/tidyverse.

To add a node, use `bind_vertices()`. To add edges, use `bind_edges()`. Below we add a node called Gabriel. Then we create an edge between Gabriel - Casey and Gabriel - Ashley with weights 1 and 0.5. Then we will remove this node and edge. Note the bind_nodes/bind_edges do not require activating nodes/edges first since they are changing the entire network.

```
net2 <- net1 %>%
        bind_nodes(data.frame(node_key=c('G'), labels='Gabriel')) %>%
        bind_edges(data.frame(from=c(1, 3), to=c(7, 7), weight=c(1, 0.5)))
```

The easiest way to remove nodes is to use the filter verb.

```
net3 <- net2 %>%
        activate(edges) %>%
        filter(to != 7) %>%
        activate(nodes) %>%
        filter(node_key != 'G')
```

The variables net1 and net3 are now identical (check for self if unsure).

Quantifying network properties

We now know how to create and manipulate the graph objects in tidygraph. It is important to understand the previous steps because, if they are understood, quantifying properties in the network are easy. As stated before, tidygraph is a wrapper for igraph, which contains lots of functions to analyse networks. Let us start with calculating the degree centrality of the network. The tidygraph logic is that any nodal property becomes a column/attribute in the node data frame. And any edge property becomes a column/attribute in the edge data frame.

We are going to calculate both the degree centrality and the strength. For both, we call `centrality_degree()` and assign it the column `degree` and `strength` (you can get these whatever you want). For strength, we need to specify that we want to use the column name "weight" to be used as the weights:

```
net1 <- net1 %>%
        activate(nodes) %>%
        mutate(degree = centrality_degree()) %>%
        mutate(strength = centrality_degree(weights = weight))
net1
```

```
## # A tbl_graph: 6 nodes and 7 edges
## #
## # An undirected simple graph with 1 component
## #
## # Node Data: 6 x 4 (active)
##   node_key labels degree strength
##   <fct>    <fct>   <dbl>    <dbl>
## 1 A        Ashely      2     1.75
## 2 B        Blake       2     1.25
## 3 C        Casey       3     2.5
## 4 D        Dylan       3     2.5
## 5 E        Elliot      2     1.25
## 6 F        Finley      2     1.75
## #
## # Edge Data: 7 x 3
##    from    to weight
```

```
##   <int> <int>  <dbl>
## 1     1     2   0.75
## 2     1     3   1
## 3     2     3   0.5
## # ... with 4 more rows
```

And we can see in the output that two new attributes have been added to the node data frame (degree and strength).

There are lots of measures that can be calculated in tidygraph, here we also calculate the betweennesss centrality and calculate the communities using the Louvain algorithm:

```
net1 <- net1 %>%
        activate(nodes) %>%
        mutate(betweenness = centrality_betweenness(weights = weight)) %>%
        mutate(community = group_louvain(weights = weight))
net1
```

```
## # A tbl_graph: 6 nodes and 7 edges
## #
## # An undirected simple graph with 1 component
## #
## # Node Data: 6 x 6 (active)
##   node_key labels degree strength betweenness community
##   <fct>    <fct>   <dbl>    <dbl>       <dbl>     <int>
## 1 A        Ashely      2     1.75           0         1
## 2 B        Blake       2     1.25           0         1
## 3 C        Casey       3     2.5            6         1
## 4 D        Dylan       3     2.5            6         2
## 5 E        Elliot      2     1.25           0         2
## 6 F        Finley      2     1.75           0         2
## #
## # Edge Data: 7 x 3
##    from    to weight
##   <int> <int>  <dbl>
## 1     1     2   0.75
## 2     1     3   1
## 3     2     3   0.5
## # ... with 4 more rows
```

To find out the other network measures, check out the tidygraph's documentation.

Plotting the network

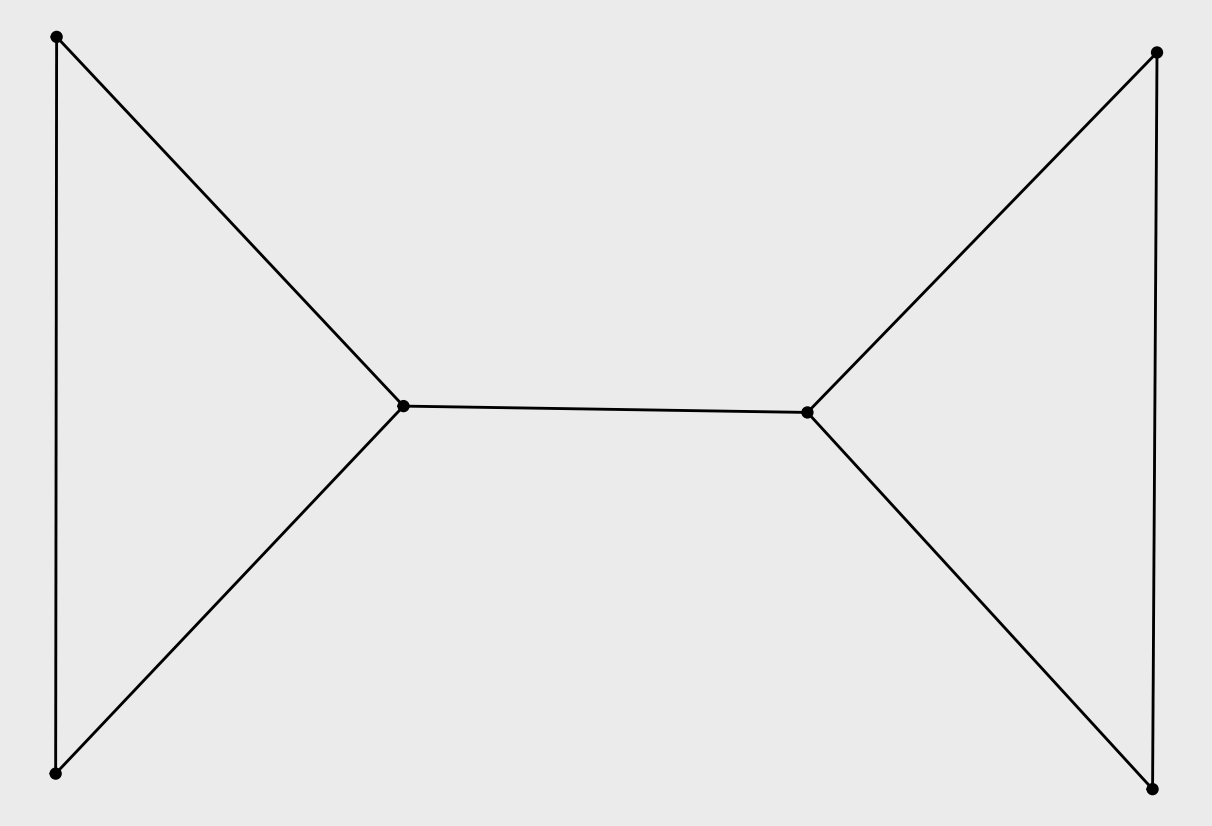
We have seen how to create and calculate measures on a network. Now it is time to plot a network. ggraph works a lot like ggplot. To create a network:

(1) call ggraph to say you want a graph, and specifying global properties such as layout.
(2) adding information about nodes and/or edges to the graph and specifying the properties of each.

Here is an example where we just plot the nodes, edges and an autoassigned layout:

```
ggraph(net1) +
  geom_node_point() +
  geom_edge_link()
```

```
## Using `stress` as default layout
```

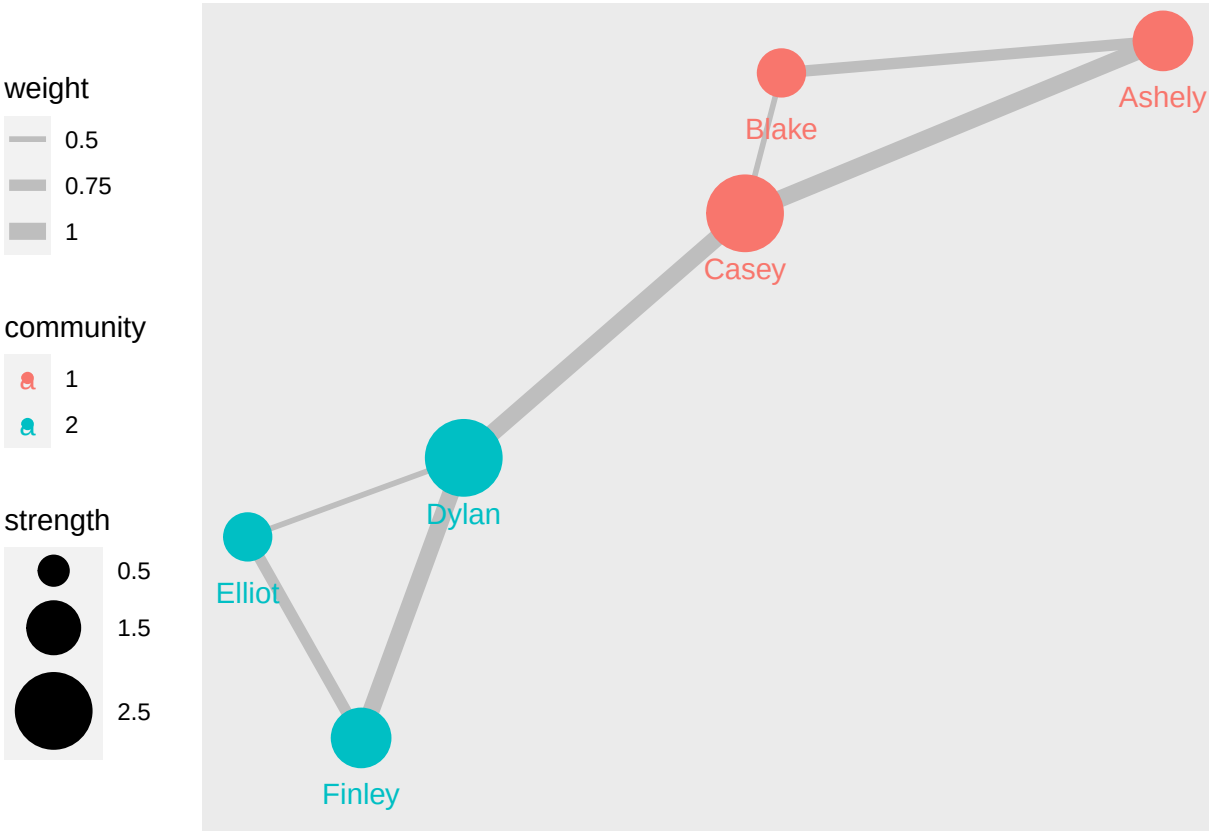


For the nodes and edges, multiple options start with geom_node_ and geom_edge_. The examples below shows multiple different options that can be added. The best way to learn what each of these do is to tweak them. Play around with the settings!

The first example colours the nodes by their community. The size is scaled by the strength. The width of the edges is the weights.

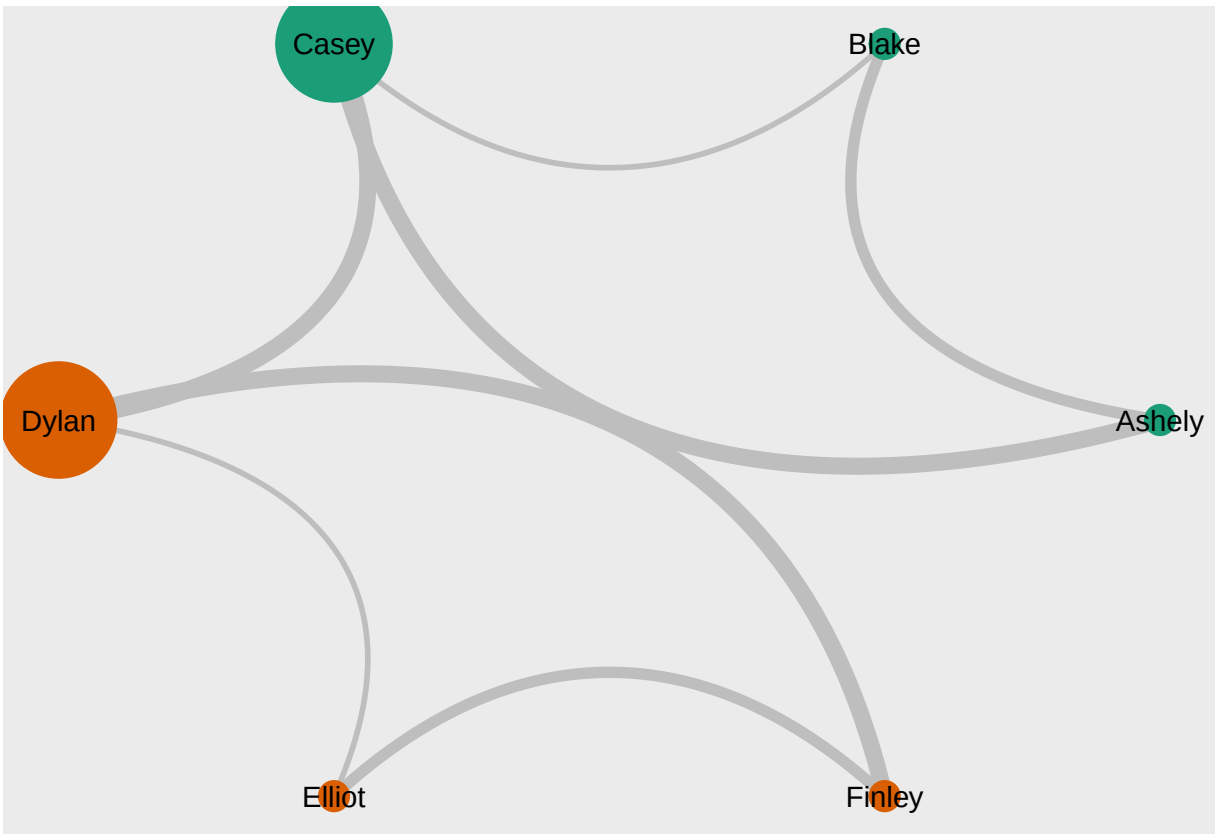
```
# Initiate the graph using the Kamada and Kawai algorithm (see help(ggraph) and help(layout_tbl_graph_i
ggraph(net1, layout='kk', weights=weight) +
  # Add edges
  geom_edge_link(aes(width=weight), color='gray') +
  # scale the edges so that they are not too big, set the legend ticks
  scale_edge_width_continuous(range=c(1,3), limits=c(0.5,1), breaks=c(0.5, 0.75, 1.0), label=c(0.5, 0.75
  # Add the nodes, and the size the strength and the color reflect the communities.
  geom_node_point(aes(size=strength, color=factor(community))) +
  # Add the labels for each node. Place name slighttly under each node
  geom_node_text(aes(label=labels, color=factor(community)), label_size=1, nudge_y=-0.16) +
  # Scale the size of the nodes to be a little bigger. specify which ticks are used in the legend
  scale_radius(range=c(5,15), limits=c(0.5,3), breaks=c(0.5, 1.5, 2.5), label=c(0.5, 1.5, 2.5)) +
  # Specify legend position
  theme(legend.position = 'left') +
  # Tranform name of legend factor(commumnity) to community
  labs(color='community')
```

```
## Warning: Ignoring unknown parameters: label_size
```



In the next example, we place the nodes in a circle and used curved edges. We also change the colourmap.

```
ggraph(net1, layout='circle') +
  geom_edge_arc(aes(width=weight), color='gray', strength=0.55, show.legend=F) +
  scale_edge_width(range=c(1,3)) +
  geom_node_point(aes(size=betweenness, color=factor(community)), show.legend=F) +
  geom_node_text(aes(label=labels)) +
  scale_radius(range=c(5,20)) +
  scale_color_brewer(palette="Dark2")
```



Play around

To get a greater understanding, change things in the notebook. Try and make a better figure than I made. Add different information etc.