Introduction

This is a tutorial for the SocialMediaLab R package. In this tutorial you will learn how to collect social media data, create networks, and perform basic social network analysis (SNA), including some natural language processing.

SocialMediaLab enables users to collect social media data and create different kinds of networks for analysis. It is a ‘Swiss army knife’ for this kind research, enabling a swift work flow from concept to data to fully-fledged network, ready for SNA and other analysis. It can handle large datasets and create very large networks, upwards of a million or more nodes (depending on your computer’s resources!). The following data sources are currently supported, and we will cover each of these:

1. Facebook
2. Instagram
3. YouTube
4. Twitter

This tutorial is roughly structured into the following sections:

1. Installation and setup
2. Facebook data collection and analysis
3. Twitter data collection and analysis
4. Data collection and network generation for Instagram and YouTube
5. Conclusions
Installation and setup

First ensure that the SocialMediaLab package is installed and loaded.

We also want to install the magrittr package, so we can simplify the work flow by using ‘verb’ functions that pipe together. We will also be using the igraph package for network analysis, and the gender package for gender analysis of usernames.

The following commands will check if the packages are installed and install them as necessary, then load them.

Note: SocialMediaLab is available as an official package on CRAN, but the latest development version is available on GitHub. We suggest installing the latest version from GitHub, using the code below.

Note: Recent changes in the httr package caused problems for the twitteR package. We resolve this using a quick-fix by installing an earlier version of httr. However, first we have to install the most recent version of httr package, before downgrading it to the earlier version.

```
install.packages("httr")
if (!"devtools" %in% installed.packages()) install.packages("devtools")
require(devtools)
devtools::install_version("httr", version="0.6.0", repos="http://cran.us.r-project.org")

if (!"SocialMediaLab" %in% installed.packages()) {
  devtools::install_github("voson-lab/SocialMediaLab/SocialMediaLab")
}
require(SocialMediaLab)

if (!"magrittr" %in% installed.packages()) install.packages("magrittr")
require(magrittr)

if (!"igraph" %in% installed.packages()) install.packages("igraph")
require(igraph)

if (!"gender" %in% installed.packages()) devtools::install_github("ropensci/genderdata")
require(gender)
```

You will also need to get API access for each data source (e.g. Facebook). You will not be able to collect any data until you have acquired API credentials. Step-by-step instructions for obtaining API access are available from the VOSON website.

Facebook data collection and analysis

In this section we will run through how to collect data from Facebook, create networks, and perform different kinds of analysis.

The process of authentication, data collection, and creating social networks can be expressed with the 3 verb functions: Authenticate(), Collect(), and Create(). This simplified workflow exploits the pipe interface of the Magrittr package, and provides better handling of API authentication between R sessions.

What we are doing is “piping” the data forward using the %>% operator, in a kind of functional programming approach. It means we can pipe together all the different elements of the work flow in a quick and easy manner.

This also provides the ability to save and load authentication tokens, so we don’t have to keep authenticating with APIs between sessions. Obviously, this opens up possibilities for automation and data mining projects.

Make sure we have our appId and appSecret values defined:
appID <- "xxxx"
appSecret <- "xxxx"

First, we will collect 2 days worth of activity from the Star Wars official page. This will collect all the posts posted between the `rangeFrom` and `rangeTo` dates, including all comments and likes, and other associated data including usernames, timestamps for comments, etc. Note: the date format is YYYY-MM-DD.

We will be using this data to create a bimodal network. This graph object is bimodal because edges represent relationships between nodes of two different types. For example, in our bimodal Facebook network, nodes represent Facebook users or Facebook posts, and edges represent whether a user has commented or ‘liked’ a post. Edges are directed and weighted (e.g. if user i has commented n times on post j, then the weight of this directed edge equals n).

Note: for Facebook, SocialMediaLab currently only supports creating bimodal and dynamic networks. More types of networks will be implemented soon.

```r

# authenticate, save credential, collect data

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret <- "xxxx"

appID <- "xxxx"
appSecret ...
```
This informs us that there are 1219 nodes and 1218 nodes in the network (this may differ somewhat for your own collected data). It tells us that our graph is Directed, Named, the edges are Weighted, and it also has the additional property of being a Bipartite graph.

Next we will do some more descriptive analysis:

```r
# list of nodes
V(g_bimodal_facebook_star_wars)
# list of edges
E(g_bimodal_facebook_star_wars)
# accessing particular node
V(g_bimodal_facebook_star_wars)[42]
# accessing particular edge
E(g_bimodal_facebook_star_wars)[1]

# list of "id" (node) attributes
V(g_bimodal_facebook_star_wars)$id
# list of "weight" (edge) attributes
E(g_bimodal_facebook_star_wars)$weight

# number of nodes in network
vcount(g_bimodal_facebook_star_wars)
# another way
length(V(g_bimodal_facebook_star_wars))

# number of edges
ecount(g_bimodal_facebook_star_wars)
# another way
length(E(g_bimodal_facebook_star_wars))

# list of the node attributes
list.node.attributes(g_bimodal_facebook_star_wars)
# list of the edge attributes
list.edge.attributes(g_bimodal_facebook_star_wars)

# test whether graph is "simple" (no loops or multiple edges)
is.simple(g_bimodal_facebook_star_wars)
```

Look at the connectivity of the graph:

```r
# who are the neighbours of node #42?
neighbors(g_bimodal_facebook_star_wars,42)

# this is not a weakly connected component
is.connected(g_bimodal_facebook_star_wars, mode="weak")

# information on connected components
cc <- clusters(g_bimodal_facebook_star_wars)
# which component node is assigned to
cc$membership
# size of each component
```
We will now look at node centrality:

```r
# node indegree
degree(g3, mode="in")
# node outdegree
degree(g3, mode="out")
# node indegree, using edge weights
ind <- strength(g3, mode="in")
# top-5 nodes, based on (weighted) indegree
V(g3)[order(ind, decreasing=T)[1:5]]
```

Network cohesion measures:

```r
# density
graph.density(g3)

# (global) clustering coefficient
# rel. frequency connected triples close to form triangles
transitivity(g3)

# number of dyads with reciprocated (mutual)
# edges/number of dyads with single edge
reciprocity(g3, mode="default")

# total number of reciprocated edges/total number of edges
reciprocity(g3, mode="ratio")
```

Another useful technique we can do is to perform a projection of the Facebook networks we just created. These networks are bipartite because nodes of the same type cannot share an edge (e.g. a user can only like/comment on a post, but not like/comment another user, and posts cannot perform directed actions either on users or other posts).

What we can do is induce two subgraphs from each network. More specifically, we can induce two *actor* networks, one for the users and one for the posts.

```r
## some data preparation
# coerce to factor
V(g_bimodal_facebook_star_trek)$type <-
```
```r
as.factor(V(g_bimodal_facebook_star_trek)$type)

# coerce all posts (i.e. "1") to logical (i.e. FALSE)
V(g_bimodal_facebook_star_trek)$type[
    which(V(g_bimodal_facebook_star_trek)$type=="1")]
<- as.logical(FALSE)

# coerce all users (i.e. "2") to logical (i.e. TRUE)
V(g_bimodal_facebook_star_trek)$type[
    which(V(g_bimodal_facebook_star_trek)$type=="2")]
<- as.logical(TRUE)

# now project the network
projection_g_bimodal_facebook_star_trek <-
    bipartite.projection(g_bimodal_facebook_star_trek)
```

Firstly, we will look at the induced graph for the “posts”. The induced “posts” actor network consists only of nodes that are of type “post”. An edge exists between post i and post j if they are both co-liked or co-commented by the same user (i.e. if they have any user in common). Not surprisingly, every post has at least one user in common, which results in the network being “complete”.

```r
str(projection_g_bimodal_facebook_star_trek[[1]])
plot(projection_g_bimodal_facebook_star_trek[[1]])
```

Secondly, we will look at the induced graph for the “users”. The induced “users” actor network consists only of nodes that are of type “user”. An edge exists between user i and user j if they both co-liked or co-commented the same post (i.e. they share an interaction with a post j). As you might expect, this create a network with a massive number of edges! A lot of users co-interact with the same posts. For this example, over 4.5 million edges (your results might be somewhat different).

```r
# warning - do not use `str` function because it will
# cause R to freeze up due to overloading the console output!
projection_g_bimodal_facebook_star_trek[[2]]
```

Maybe there is some community structure to this large network. There are several ways to find out. We will use the infomap algorithm implementation in igraph. Infomap uses an information theoretic, flow-based approach to calculating community structure in networks. It supports weighted and directed graphs, and also scales well.

The results show that there is definitely some interesting community structure to the user actor network (a handful of large communities and a tiny community). Although your results might differ, depending on the actual data collected.

```r
# limit the `trials` argument to a small number to save time
# (number of attempts to partition the network)
imc_starwars <- infomap.community(
    projection_g_bimodal_facebook_star_trek[[2]], nb.trials = 3)

# create a vector of users with their assigned community number
communityMembership_starwars <- membership(imc_starwars)
# summarise the distribution of users to communities
commDistribution_starwars <- summary(
```
Now we will do some analysis that involves combining natural language processing and SNA. The goal of this analysis is to find out if there is a gendered dimension to the Star Trek page versus the Star Wars page.

To achieve this we will use the `gender` package to analyse the usernames of participants in each network, and assign them a ‘gender’ attribute. This is a binary gender, i.e. male or female. There is also an ‘unknown’ category for instances where gender does not apply or cannot be predicted.

We will also take this opportunity to fire up *Gephi* and visualise our networks and their gender composition.

### Star Wars gender...

```r
# Star Wars
userNames <- V(g_bimodal_facebook_star_wars)$name
defirstNames <- sub(".*", "", userNames)
defirstNames <- gsub( "[^[:alnum:]-]", "", firstNames)
genderPredictions_star_wars <- gender(firstNames, method = "ssa")

summary(as.factor(genderPredictions_star_wars$gender))
```

```r
## Expressed as percentages:
paste0("Female: ", round(length(as.factor( which(genderPredictions_star_wars$gender=="female"))) / length(genderPredictions_star_wars$gender) * 100,1),"%")
paste0("Male: ", round(length(as.factor( which(genderPredictions_star_wars$gender=="male"))) / length(genderPredictions_star_wars$gender) * 100,1),"%")
```

```r
# now we apply these data to the network as a new attribute named `gender`
temp <- match(firstNames,genderPredictions_star_wars$name)
V(g_bimodal_facebook_star_wars)$gender <- genderPredictions_star_wars$gender[temp]
V(g_bimodal_facebook_star_wars)$gender[
is.na(V(g_bimodal_facebook_star_wars)$gender)] <- "unknown"
```

### Star Trek gender...

```r
# Star Trek
userNames <- V(g_bimodal_facebook_star_trek)$name
defirstNames <- sub(".*", "", userNames)
defirstNames <- gsub( "[^[:alnum:]-]", "", firstNames)
genderPredictions_star_trek <- gender(firstNames, method = "ssa")

summary(as.factor(genderPredictions_star_trek$gender))
```
# Expressed as percentages:

```r
paste0("Female: ", round(length(as.factor(
  which(genderPredictions_star_trek$gender=="female"))) / 
  length(genderPredictions_star_trek$gender) * 100,1),"%")
paste0("Male: ", round(length(as.factor(
  which(genderPredictions_star_trek$gender=="male"))) / 
  length(genderPredictions_star_trek$gender) * 100,1),"%")
```

# now we apply these data to the network as a new attribute named `gender`

temp <- match(firstNames,genderPredictions_star_trek$name)

```r
V(g_bimodal_facebook_star_trek)$gender <- genderPredictions_star_trek$gender[temp]
V(g_bimodal_facebook_star_trek)$gender[
  is.na(V(g_bimodal_facebook_star_trek)$gender)] <- "unknown"
```

Our little study suggests that Star Wars is quite male dominated, while Star Trek has a more equal distribution of gender.

Now we will quickly visualise the Star Wars network in Gephi, and we can really provide a powerful illustration of the gender composition. Tim will take us through this quickly in Gephi. First, we need to export the network to file:

```r
write.graph(g_bimodal_facebook_star_wars, 
  "FacebookBimodalNetwork_Star_Wars_GENDER.graphml", format="graphml")
```

We will end up with a Star Wars network (coloured by gender) that looks like this:

![Star Wars network](image)

For comparison, here is the Star Trek network coloured by gender:
Twitter data collection and analysis

In this section we will run through how to collect data from Twitter, create networks, and perform different kinds of analysis.

It is currently possible to create 3 different types of networks using Twitter data collected with SocialMediaLab. These are (1) actor networks; (2) bimodal networks; and (3) semantic networks. In this session we will create an actor and a semantic network (we created a bimodal Facebook network in the previous section).

First, define the API credentials. Due to the Twitter API specifications, it is not possible to save authentication token between sessions. The Authenticate() function is called only for its side effect, which provides access to the Twitter API for the current session.

```
# REPLACE WITH YOUR API KEY
myapikey <- "xxxx"
# REPLACE WITH YOUR API SECRET
myapirequest <- "xxxx"
# REPLACE WITH YOUR ACCESS TOKEN
myaccessstoken <- "xxxx"
# REPLACE WITH YOUR ACCESS TOKEN SECRET
myaccessstokensecret <- "xxxx"
```

Given that we are going to be creating two different types of Twitter networks (actor and semantic), we will
Collect() the data, but not pipe it directly through to Network() straight away. This means we can reuse the data multiple times to create two different kinds of networks for analysis.

We will collect 150 recent tweets that have used the #auspol hashtag. This is the dominant hashtag for Australian politics.

```r
myTwitterData <- Authenticate("twitter", apiKey=myapikey, apiSecret=myapisecret, accessToken=myaccesstoken, accessTokenSecret=myaccesstokensecret) %>%
  Collect(searchTerm="#auspol", numTweets=150, writeToFile=FALSE,verbose=TRUE)
```

We can have a quick look at the data we just collected:

```r
View(myTwitterData)
```

Note the class of the dataframe, which lets SocialMediaLab know that this is an object of class dataSource, which we can then pass to the Create() function to generate different kinds of networks:

```r
class(myTwitterData)
```

If you find that you are encountering errors possibly related to the text of the tweets, you can try converting the tweet text to UTF-8 character encoding:

```r
myTwitterData$text <- iconv(myTwitterData$text, to = "utf-8")
```

**Mac users only** may also wish to try the following if they are encountering errors that may be due to character encoding issues:

```r
myTwitterData$text <- iconv(myTwitterData$text, to = "utf-8-mac")
```

First, we will create an actor network. In this actor network, edges represent interactions between Twitter users. An interaction is defined as a ‘mention’ or ‘reply’ or ‘retweet’ from user i to user j, given ‘tweet’ m. In a nutshell, a Twitter actor network shows us who is interacting with who in relation to a particular hashtag or search term.

```r
g_twitter_actor <- myTwitterData %>% Create("Actor")
```

We can now examine the description of our network:

```r
g_twitter_actor
```

Here is a visualisation (in Gephi). There are quite a number of isolated clusters of inter-activity between users.
We have done some core SNA in the previous section, so here we will tackle a few different techniques.

Who are the top 3 important users in our #auspol actor network? There are several ways to do this. We will use the PageRank algorithm implementation in `igraph` to calculate this:

```r
pageRank_auspol_actor <- sort(page.rank(g_twitter_actor)$vector,decreasing=TRUE)
head(pageRank_auspol_actor,n=3)
```

What about the 3 least important users (with all due respect...):

```r
tail(pageRank_auspol_actor,n=3)
```

Is there any kind of community structure within the user network? As per the previous Facebook analysis we will use the infomap algorithm implementation in `igraph`.

```r
imc <- infomap.community(g_twitter_actor, nb.trials = 100)
# create a vector of users with their assigned community number
communityMembership_auspol <- membership(imc)
# summarise the distribution of users to communities
commDistribution <- summary(as.factor(communityMembership_auspol))
# which community has the max number of users
tail(sort(commDistribution),n=1)

# create a list of communities that includes the users assigned to each community
communities_auspol <- communities(imc)
# look at the members of the most populated community
communities_auspol[names(tail(sort(commDistribution),n=1))]
# same as doing it manually
 communities_auspol[5]
```
Next, we will create a *semantic* network. In this network nodes represent unique concepts (in this case unique terms/words extracted from a set of 150 tweets), and edges represent the co-occurrence of terms for all observations in the data set. For example, for this Twitter semantic network, nodes represent either hashtags (e.g. “#auspol”) or single terms (“politics”). If there are 150 tweets in the data set (i.e. 150 observations), and the term #auspol and the term politics appear together in every tweet, then this would be represented by an edge with weight equal to 150.

```
g_twitter_semantic <- myTwitterData %>% Create("Semantic")
```

Let’s have a look at the network description:

```
g_twitter_semantic
```

Here is a visualisation (in Gephi):

![Gephi visualization of the semantic network](image)

What are the top 10 important terms in our #auspol actor network? There is no reason why we can’t reuse the PageRank algorithm implementation in *igraph*:

```
pageRank_auspol_semantic <- sort(page.rank(g_twitter_semantic)$vector, decreasing=TRUE)
head(pageRank_auspol_semantic, n=10)
```

Obviously the #auspol hashtag is going to be the most important because it occurs at least once in every tweet. We can actually avoid this by using the `removeTermsOrHashtags` argument when we `Create()` the network. This argument specifies which terms or hashtags (i.e. nodes with a name that matches one or more terms) should be removed from the semantic network. This is useful to remove the search term or hashtag that was used to collect the data (i.e. remove the corresponding node in the graph). For example, a value of “#auspol” means that the node with the name “#auspol” will be removed. Note: you could also just delete the #auspol node manually.

Another key aspect of semantic networks is how many terms to include in the network. By default, *SocialMediaLab* does not include every unique term that it finds in the tweets, but only the 5 percent most
frequently occurring terms. You can change this when calling the `Create()` network function, for example by specifying a value 50 (meaning that the 50 percent most frequently occurring terms will be included in the semantic network).

We can actually try this out now. We will create another semantic network, but we will exclude the #auspol hashtag, and we will include every single term available in the tweets.

```r
g_twitter_semantic_auspol_allTerms <- myTwitterData Create("Semantic", termFreq=100, removeTermsOrHashtags=c("#auspol"))
```

The size of the network will increase a lot, even in the absence of the #auspol term!

```r
g_twitter_semantic_auspol_allTerms
```

It will also change the nature of the network, for example, by reconfiguring which terms are the most important. Basically, the #auspol term won’t hog all the pagerank. Let’s run the PageRank algorithm again and calculate the top 10 terms:

```r
pageRank_auspol_semantic_allTerms <- sort(page.rank(g_twitter_semantic_auspol_allTerms)$vector, decreasing=TRUE)
head(pageRank_auspol_semantic_allTerms, n=10)
```

### Data collection and network generation for Instagram and YouTube

In this section we will briefly cover how to collect data from Instagram and YouTube, and create networks. We will not be doing much analysis in this section, but going through the workflow to become familiarised with it.

#### Instagram

Instagram is a bit different to Facebook and Twitter, mainly because it supports the creation of **ego** networks. That is, networks of follower/followee relationships that extend out from one or more ‘ego’ nodes of interest. Currently, SocialMediaLab also supports creating **bimodal** networks. Additionally, Instagram data has a strong focus on images (as well as text), so we are able to also collect the images and store them in a local directory.

As usual, we define our API credential values:

```r
myAppId <- "xxxx"
myAppSecret <- "xxxx"
```

First, we will construct an ego network that includes two US Senators who each have a humble amount of followers (so we don’t sit around all day waiting for the data to collect). These are Adam Kinzinger (Republican) and Harry Reid (Democrat).

There are several key aspects to mention here, before we dive in.

1. We are using the `ego = TRUE` argument in the `Collect()` function, which means that we are not collecting *posts* data, but are only collecting *followers* data.

2. The default `degree` (aka *order*) of the network is 1, meaning that it collects data about ego + alters (ego and followers). Changing the `degreeEgoNet` argument in the `Collect()` function to 2 means that it would collect data from ego + alters + alters of alters (ego and followers and followers of followers of ego).
3. By default, *SocialMediaLab* will also collect ‘follows’ for the ego nodes. So it will collect who each ego node itself *follows* (i.e. not just the *followers* of ego), and integrate these data into the ego network.

!! Caution / advice: depending on the above arguments, you can create *extremely* large networks very quickly, with only a small number of ego nodes! If you really want to create super massive networks, use the `waitForRateLimit = TRUE` argument when you `Collect()` data. If `TRUE` then it will try to observe the Instagram API rate limit by ensuring that no more than 5000 API calls are made per hour (the current rate limit). If more than 5000 calls are made within a 60 minute window, then all operates will suspend for 60 minutes, and resume afterwards.

```r

Next, we will construct an Instagram network based on posts from... Newport Beach, California! The types of nodes are ‘users’ and ‘posts’. An edge between user i and post j means that user i has liked and/or commented on post j at least once. Thus, the social network shows co-interactions of users to posts made in Newport Beach (within a radius of 5km from its latitude and longitude).

There are five key aspects in this example:

1. We need to find the latitude and longitude of Newport Beach, and supply these values as the `lat` and `lng` arguments to `Collect()`.

I have gone ahead and visualised the network using Gephi. The ego network for these two senators appears to be extremely modular! The vast majority of users only follow one senator or the other, but not both.
2. We provide the maximum value of 5000 for the distance argument. This collects Instagram posts within a radius of 5000m (5km) of the latitude and longitude coordinates.

3. We want to collect pictures/photos and save these to a directory named ‘newport_beach_pics’. We use the folder argument to achieve this.

4. We want to collect 50 posts, which we define as the value of the n argument. For each post, SocialMediaLab will collect all the likes and comments for that post.

5. We define verbose as TRUE. This means that SocialMediaLab will keep us informed of how things are progressing as the data collection occurs. It will tell us how many likes and comments it is collecting, for each post that we collected.

```
g_bimodal_instagram_DEBUG <- LoadCredential("InstagramCredential.RDS") %>%
  Collect(lat=33.6189, lng=-117.9289, distance=5000, n=50, verbose=TRUE, folder='newport_beach_pics') %>%
  Create("Bimodal")
```

Description of network:

```
g_bimodal_instagram
```

There are quite a lot of node attributes in the network, including useful information such as when the post was created (post_created_time_UNIX_epoch) and the post text (i.e. captions - post_caption).

Let’s have a quick look at the first 10 post captions. Nodes in the network that are not posts (i.e. nodes that are users) will not have any captions, so there will be a lot of NA values as a result. We need to filter these out.

Note: The text from comments is not stored in the network (it tends to create very large/bloated networks in terms of RAM usage and file size). To keep this data, you will need to Collect() the data into its own object before passing or “piping” it to the Create() function.

```
V(g_bimodal_instagram)$post_caption[which(!is.na(V(g_bimodal_instagram)$post_caption))]
```

I have gone ahead and visualised the network using Gephi. There are many different little clusters of activity.
YouTube

In this section we will collect data from YouTube and create an actor network from the comments section of particular videos of interest. Such a network maps the relationships of YouTube users who have interacted with each other in the comments section for particular videos (i.e. user i has replied to user j or mentioned user j in a comment).

First, set up the API credentials:

```r
apiKey <- "xxxx"
```

Next, we specify which videos we want to collect data from, using a character vector specifying one or more YouTube video IDs. For example, if the video URL is ‘https://www.youtube.com/watch?v=W2GZFeYGU3s’, then use `videoIDs="W2GZFeYGU3s"`.

For speed, we will collect data from two videos (about R programming) that have a small number of comments.

```r
videoIDs <- c("W2GZFeYGU3s","mL27TAJG1Wc")
```

The workflow is fairly straightforward - we just pipe together the ‘verb’ functions. A couple of comments.

1. By default, all the available comments are collected. If desired, the ‘maxComments’ argument can be used to limit the number of comments (but as noted in the documentation, this is not always perfect, due to the YouTube API specifications).

2. Often, we will wish to retain the comment text for further analysis. There are two approaches (as discussed previously). First option is to leave out `Create()` function from the pipeline, so we are just creating a dataframe object with our data (which we can later pipe through to `Create()` an actor network). The second option, which we use in this example, is to specify `writeToFile=TRUE`, so we write the data to disk before piping it through `Create()` the network.
g_youtube_actor <- Authenticate("youtube", apiKey= apiKey) %>%
  Collect(videoIDs = videoIDs, writeToFile=TRUE) %>%
  Create("Actor")

A description of the network:

g_youtube_actor

Read in the YouTube data that we saved to disk, for example:

```r
# make sure you change the filename:
myYouTubeData <- read.csv("Apr_02_12_19_24_2016_AEST_YoutubeData.csv")
View(myYouTubeData)
```

You could then do some interesting text analysis, or combine it with SNA, e.g. performing text analysis and then applying the results as attributes to nodes or edges. For example, you could do sentiment analysis of comments, and then categorise users into “negative” or “positive” sentiment. Then you could create a node attribute such as `user_mood`.

**Conclusion**

We hope that you enjoyed this tutorial and that you find SocialMediaLab useful in your research. Please feel free to contact us should you have any questions or comments (or find any errors in the document). We would love to hear your ideas and feedback.

All the best,

Robert Ackland and Timothy Graham