Absolute Beginner’s Guide to the SocialMediaLab package in R

(updated for SocialMediaLab version 0.20.1)

Timothy Graham & Robert Ackland

6 April 2016

1. Introduction

This guide is aimed at people with little or no experience in programming who wish to start collecting social media data and generating networks for analysis using the SocialMediaLab package. It does not cover every detail of SocialMediaLab, but aims to provide a quick and gentle introduction to working with SocialMediaLab in RStudio, starting from minimal or even “zero” knowledge of the subject area.

What is R?

R is a programming language that has exploded into popularity in recent years. It is particularly suited to statistics and ‘data science’ problems. It is free, as in open source.

One of the key advantages of R is that it has a diverse and well-established ecology of packages that provide additional functionality, making it very easy to do all sorts of tasks and analyses with little programming. CRAN is the official repository for R packages, although you can find packages also on GitHub and elsewhere.

What is SocialMediaLab?

SocialMediaLab is an R package that is designed to make it very easy to collect data from social media platforms and construct networks for analysis. Currently SocialMediaLab has support for 4 social media platforms: Facebook, Twitter, YouTube, and Instagram. It enables users to generate different kinds of networks from the data.

SocialMediaLab draws heavily on other R packages, but extends the functionality considerably by enabling users to construct networks out of the data. For example, one might wish to construct a network that depicts all the activity (comments and likes on posts) on a public Facebook page over a few weeks, a month, or even years. This opens up many possibilities for analysis and research.

In this guide you will learn how to:

1. Install RStudio, which is a graphical interface to ‘vanilla’ R;
2. Install SocialMediaLab, which is an R package for generating networks out of social media data;
3. Do basic programming in the R language (optional section);
4. Collect social media data from Instagram and create a bimodal network;
5. Collect social media data from Twitter and create an actor network; and
6. Export a network as a graph file and import it into Gephi to visualise it.
2. Installation and setup

This section provides detailed step-by-step information for how to download and setup the required software.

2.1 Installing R and RStudio

Head to the RStudio download page and download the installer that is relevant to your system (e.g. if you are on Windows versus Mac).

Once you have installed RStudio, click to launch it. You will be presented with a screen with three panes (probably), including the “Console” (left-hand-side), the “Environment” (upper-right), and “Files” (bottom-right). The console is where you do the work in R. It may look frightening, but it will soon become your best friend and constant companion. You type commands into the console and press enter, and the program does its thing. You can try it now by typing into the console the following command, and pressing enter to run it.

```r
print("Hello, world!")
```

Congratulations, you have just written your first R program! We will return to a brief introduction to R programming in the next section (this is optional depending on your time and requirements).

2.2 Installing the SocialMediaLab package

You will now create a new project for this tutorial. Click ‘File’–>‘Create New Project’–>‘New Directory’–>‘Empty Project’. Select an appropriate directory to store the project in. This will be the ‘base’ for your project, which is known as the ‘working directory’ in R. When you have selected a directory click ‘Create Project’.

To install SocialMediaLab, click ‘Tools’–>‘Install Packages’. Ensure that the ‘Install from’ dropdown box is set to ‘Repository (CRAN)’. In the ‘Packages’ text field, type in ‘SocialMediaLab’ (without the inverted commas).

If you have a problem installing via CRAN, you can also install SocialMediaLab from GitHub, where the latest version is hosted. Simply copy and paste the following commands into the R console (and hit enter, as per the “Hello, world!” example in the previous sub-section):

```r
install.packages("httr")
if (!"devtools" %in% installed.packages()) install.packages("devtools")
require(devtools)
# we have to install an earlier version of 'httr' package
# due to incompatibility problems with the 'twitterR' package
devtools::install_version("httr", version="0.6.0", repos="http://cran.us.r-project.org")
devtools::install_github("voson-lab/SocialMediaLab/SocialMediaLab")
require(SocialMediaLab)
if (!"magrittr" %in% installed.packages()) install.packages("magrittr")
require(magrittr)
```

To do network analysis we also need to download and install the igraph package created by Gábor Csárdi. We can do this by entering the following command into the R console, as in the previous step:

```r
install.packages("igraph")
```
2.3 Installing Gephi

In the final part of this guide you will have the opportunity to export the network and import it into Gephi, which is a fantastic network visualisation program. If you wish to do this, then you need to download and install Gephi.

3. Quick and gentle introduction to R programming (OPTIONAL)

In this optional section we will cover some very basic concepts for programming in the R language. This knowledge will provide the absolute minimum to get you on your feet in R, and hopefully not become completely baffled.

This section is aimed for people with minimal or no programming experience. You can skip this section and still get a lot out of this guide, but if you are new to programming it is highly recommended to complete it. At the least, it is recommended that you read the sub-section about data frames because SocialMediaLab uses this data structure extensively.

The basic concept in computer programming is that you provide instructions to the computer about how to solve some problem, which you provide using code, resulting in some sort of outcome or output. It is like a cake recipe. You specify the instructions in a language such as English (1 cup of baking soda, 2 cups of flour, stir for 1 minute, etc), and the cook executes the instructions, and at the end the oven outputs a cake (hopefully).

3.1 Variables

With R you write code and execute it in the console, resulting in some output. Here is the “Hello, world!” example again, but with a key difference. Instead of printing “Hello, world!” straight to the console, we are going to store this sentence into a ‘variable’ so we can access it easily without having to type it out each time. Run the following command in your console (note that R is case sensitive!).

```r
myVariable <- "Hello, world!"
```

Despite its simplicity, this little program demonstrates several important concepts for R programming. Let’s break down what is happening in plain English:

Create a variable (which is kind of like a container) that stores the sentence “Hello, world!”, and also give this variable the name `myVariable`.

Variables are a key concept in programming. They allow you to store many types of data, for example: character strings (e.g. “Hello, world!”), or different kinds numbers (e.g. 42 or 3.14), or whether something is true or false. The main thing to remember is that generally speaking everything in R is an object. Variables are a type of object.

But what is that strange notation in the middle: `<-`? This is known as an assignment operator. For now you can think about it as basically the equals sign. So `<-` is quite similar to `=`. (However, there are important differences, which is why we use the `<-` notation rather than `=`).

The `<-` assignment operator basically says “take what is on the right-hand side of me and assign it to the left-hand side of me”. In the example above we are basically saying: take the sentence “Hello, world!” and assign it to the variable `myVariable`, which we code as `myVariable <- "Hello, world!"`, meaning `myVariable` is assigned (i.e. `<-`) the value "Hello, world!".
3.2 Functions

Now that we have `myVariable`, we can put this variable to use. Run the following command:

```
print(myVariable)
```

What happened here? This line of code is similar to the “Hello, world!” example we saw earlier, but there is an important difference. Once again let’s proceed slowly and interpret this line of code in plain English:

Use the `print` function to ‘print’ the contents of `myVariable` to the screen. In other words, we are ‘feeding’ the contents of `myVariable` into the `print` function, and the `print` function will then do it’s one job, which is to print the contents to the console output.

This example provided an introduction to the concepts of ‘functions’. Functions are an extremely important aspect of R programming (and indeed most programming languages). In fact, generally speaking we can say that in R everything that happens is a function call. In other words, just about every time you type a command into the console you are running some kind of function.

So what are functions? Functions can be thought about in a very rudimentary sense as recipes. In this example `myVariable` is like the ingredients (raw egg, sugar, flour) and `print` is the recipe: you input or ‘pour’ the ingredients (variable) into the recipe (function) and execute or ‘cook’ it, and you get some result (the words “Hello, world!” appear on the screen). That’s the basic idea. In R you have the function name, e.g. `print`, and then in between the round brackets you provide the ‘input’ for the function operate on, in this case `myVariable` (which we know contains the characters “Hello, world!”). To use the analogy of a recipe, it would basically take the form: `recipe(ingredients)`.

Let’s work through some more examples of functions. In the next line of code we will define our own function, basically a simple little ‘recipe’ for adding two numbers together. There are three lines of code, so you can copy and paste them into the R console, and it will run them one after the other (or you could put them in one line, if you want to try and do that).

```
myFunction <- function(x,y) {
  result <- x + y
}
```

Here is a basic overview of what is happening in this example:

1. We create a function named `myFunction` that takes as input two numbers, which we specify as `x` and `y`.
2. We assign `myFunction` some instructions for what to do with the input, which we put in between the curly brackets.
3. Inside the curly brackets we provide instructions for what to do with the two numbers `x` and `y` we provided as input.
4. The instructions are `results <- x + y`. This means that we add together `x` and `y` and store the result in a variable called `result`. Unlike the previous “Hello, world” example, our variable `result` is numeric (not character), i.e. it stores a number.
5. The function will ‘spit out’ or, more formally, ‘return’ the value of `result`.

Now we can put `myFunction` to use, by typing in the following command:

```
output <- myFunction(4,5)
output
```
What we are doing is running `myFunction` and providing the numbers 4 and 5, and asking it to add them together, and return the result. Additionally, we are storing the result into a variable named `output`. If we simply type in `output` it will display the contents of `output` on the screen, which is of course 9.

The cool thing about functions is that we don’t need to supply values as input, because we can also supply variables as input. Try the following commands:

```r
someNumber <- 20
anotherNumber <- 22
output <- myFunction(someNumber, anotherNumber)
output
```

What we are doing here is defining two new variable, `someNumber` and `anotherNumber` and assigning them values of 20 and 22 respectively. Then we use our trusty old `myFunction` to add them together, and store the result in the variable `output`. Lo and behold, we have 42.

### 3.3 Further material on data structures and data types

We have so far covered only the very basics of R programming. In this section we will move fairly quickly through other data structures and data types. You can skip this section if you want to get straight into working with `SocialMediaLab`, because by now you have enough knowledge that you won’t be completely baffled by what is happening later on. But it is advised to work through this section if you have the time, even if just to get a sense of the wider world of R.

#### 3.3.1 Vectors

Probably the most important/common data structure in R are vectors. In fact, we have been already working with vectors. If you have been working through the previous examples, try typing in the following:

```r
is.vector(output)
```

It turns out that our number “9” is actually a vector of length one. This means that it is just one number by itself, no more or less.

In R, a vector is a set of elements that are most commonly character, logical, integer or numeric.

Here is a simple numeric vector:

```r
x <- 5
x
```

Here is a character vector:

```r
myName <- "John"
myName
```

We can look at attributes of vectors, e.g. finding out how many characters `myName` is. In the following code, we are using the `nchar` function and providing as input our `myName` variable.

```r
nchar(myName)
```

We can also assign ‘logical’ or ‘boolean’ vectors, which are either TRUE or FALSE:
skyIsBlue <- TRUE
skyIsBlue

Then we can ask R to tell us whether skyIsBlue is TRUE or not, by feeding it into the `isTRUE` function:

`isTRUE(skyIsBlue)`

We don’t have to have only one element in a vector. Often we want to use multiple elements, e.g. creating a numeric vector with the numbers 1 through 5.

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

We can also use shorthand notation to do the same thing, but for 1 through 10:

`countToTen <- 1:10`
`countToTen`

We can find out how many elements are in the numeric vector:

`length(countToTen)`

We can access the fifth element of `countToTen` using the square brackets notation. This indexes `countToTen` and looks for the fifth element, and returns the value to us.

`countToTen[5]`

We can also access the first 3 elements:

`countToTen[1:3]`

We can find out information about our `countToTen` vector. The `typeof` function gives us basic information about the type of object. The `str` function is useful for finding out more detail what an object is. This tells us that it is of type ‘int’ (integer).

`typeof(countToTen)`
`str(countToTen)`

We can also examine the `myName` character vector that we created earlier. Note the difference, namely that is of type ‘chr’ (character).

`str(myName)`

### 3.3.2 Matrices

Matrices are special vectors in R. They are basically a multi-dimensional vector, i.e. with rows and columns. You can think about them as similar to tables in a spreadsheet document. However, they are ‘atomic’, so they can only contain data of one type (e.g. you can’t have a column with integer data and another column with character data).

Matrices are filled column-wise, for example:
myMatrix <- matrix(1:6, nrow = 2, ncol = 3)

You can see that the matrix has 2 rows and 3 columns, and we have filled the ‘cells’ of the matrix with the numbers 1 through 6 (i.e. 1:6). Again, the matrix is filled column-wise, so the numbers 1 through 6 are filled starting in the first row of the first column, and then second row of the first column, then first row of second column, and so forth.

You can make a matrix out of two vector objects, for example:

vector1 <- 1:5
vector2 <- 6:10
myMatrix2 <- cbind(vector1, vector2)

There are some new things here. We defined two numeric vectors vector1 and vector2. Then we used the cbind function to ‘stick’ these together column-wise (i.e. vertically next to each other). The result is stored in a new variable called myMatrix2, which is a matrix, because it is a multi-dimensional numeric vector. The matrix also has names for the columns, which it gets automatically from the names of the variables vector1 and vector2.

We can access the column names of myMatrix2:

colnames(myMatrix2)

We access the element in the first row and second column using the square bracket notation for accessing particular elements (i.e. fishing out the values of particular cells of the matrix). Within the square brackets we have the row number on the left hand side of the comma, and the column number of the right hand side of the comma. For example, if we want to access the value of the cell in the first row and second column we would use:

myMatrix2[1,2]

We can access all of the second column of our matrix by simply not providing a row number (i.e. leaving it empty on the left hand side of the comma):

myMatrix2[,2]

3.3.3 Lists

Lists in R act like ‘containers’. They are similar to vectors but very different too, because each element can be a different type. In this way, you can mix integers and characters quite happily together.

myList <- list("Hello", 1, TRUE, "Goodbye")

We access elements in lists slightly differently to vectors, but still use the square brackets notation.

myList[1]
What on earth is going on here? Actually it is quite similar to indexing vectors using square brackets notation. The key difference is that we are accessing the first ‘slice’ of the myList, which is a list which itself also contains one element, namely the character vector “Hello”. The list contains nested elements. It’s like the movie ‘Inception’, which has different realities within realities, and you can move from one level to the next. We can get more specific and access the “Hello” character vector directly, using double square brackets notation. The next line of code accesses the first slice of myList, accessed using the subset [1], and then the first element within that slice, accessed using the subset [[1]]. We are subsetting myList twice, once with [1] and secondly with [[1]].

myList[1][[1]]

OK, so what about the first element of the second slice of myList?

myList[2][[1]]

And what about the first element of the third slice of myList? You get the idea...

myList[3][[1]]

### 3.3.4 Factors

In R, factors are special vectors that represent categorical data. Here we create a factor myFactor that provides data on 5 students and whether each one passed or failed their assignments. We can see that there are two ‘levels’ to this factor, namely “pass” or “fail”. (This is a very badly performing cohort of students!)

myFactor <- factor(c("pass", "fail", "fail", "pass", "fail"))

myFactor

We can subset the factor to find which students are “fail”. This returns the indexes of the elements in myFactor that equal “fail”. Notice that the equals sign is == (not =). This double equals sign is used to test for equality. Here we are asking the question: which elements of myFactor are equal to “fail”? If there are any elements that match, then it returns the indexes of these elements. Sure as eggs, they match up with what we expect (the 2nd, 3rd, and 5th elements are fail).

which(myFactor=="fail")

### 3.3.5 Data frames

Data frames are very important in R, and we will use them a lot in the next sections of this tutorial using SocialMediaLab. So it is important to understand the basics of how they work and how to handle them.

Data frames are similar to matrices, in that they are often two-dimensional with rows and columns. Roughly speaking, the biggest difference between data frames and matrices is that data frames can contain columns with different types of data.

In this way, each column in the data frame can have a different data type, for example:

df <- data.frame(names = c("John", "Jane", "Sally"),
                 testScores=c(99,84,30), failingGrade=c(FALSE, FALSE, TRUE))

df
We can access the data in the third column in two ways. First, we can use the brackets notation:

```
df[,3]
```

We can also use the dollar sign notation to do the same thing. We know that the third column of `df` has name, ‘failingGrade’, so we can subset the data frame by name using the dollar sign notation:

```
df$failingGrade
```

We can find the number of rows in data frame by calling the `nrow` function and supplying `df` as input to it:

```
nrow(df)
```

We can also find out the structure of each column of `df` using the `str` function:

```
str(df)
```

Finally, we can view the data frame quite nicely using the `View` function. This is especially useful for working with large data frames that we will be working with in the remainder of this tutorial.

```
View(df)
```

4. Collecting Instagram data and creating bimodal networks

In this section we will learn how to collect Instagram data and generate ‘bimodal’ networks from the data for further analysis using the igraph package, or visualisation using Gephi.

Social Network Analysis (SNA) is the broad field that we are working with when it comes to the types of networks that `SocialMediaLab` generates. For more information the interested reader is referred to Wikipedia or to the book Web Social Science by Robert Ackland.

We first ensure that the `SocialMediaLab` package is loaded into R, as well as the igraph package. We use the `require` function to achieve this.

```
require(SocialMediaLab)
require(igraph)
require(magrittr)
```

Generally, there are three steps involved for collecting data and creating social networks using `SocialMediaLab`. These are:

1. **Authenticating** with the platform API (to gain access to the free data they provide);
2. **Collecting** data from the API and storing it for later use; and
3. **Creating** networks from the data collected in the previous step.

The process of authentication, data collection, and creating social networks can be expressed with the 3 verb functions provided by `SocialMediaLab`. These are: `Authenticate()`, `Collect()`, and `Create()`. This simplified workflow uses the pipe interface of the `Magrittr` package, and provides better handling of API authentication between R sessions. This 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.
4.1 Authenticating with the Instagram API

What is an API?

From Wikipedia: an API (Application Programming Interface) is a set of [routines], protocols, and tools for developing applications for a particular system. It may be a web based system, operating system, or database system. The API provides facilities to develop applications for that system using a programming language.

In a nutshell: social media platforms often provide an API that enables anyone with an account to get data for free. Bonza!

An Instagram app (with id and secret key) is required for authenticating with the Instagram API (otherwise we cannot collect data). This requires an Instagram account. Instructions for obtaining API access from Instagram are available from the VOSON website.

In the following code you would replace the values with your own credentials. You need a valid “client ID” and a “client secret” in order to proceed with collecting data from the Instagram API.

```r
myAppId <- "xxxxxx"
appSecret <- "xxxxxx"
```

We then run the following function, which creates an authorised token that we can use to access the API. Note: it will open up a browser window - just click OK and close.

```r
instaToken <- Authenticate("instagram", appID = myAppId, appSecret = myAppSecret)
```

4.2 Data collection

Suppose we want to collect Instagram data from an area of interest. In this example we will collect recent Instagram from Brisbane, the capital city of Queensland (Australia). To do this we will be using the `Collect()` function.

The `Collect()` function accepts various arguments that control how data are collected. In this example we will use a selection, and provide some explanation of what is going on. This function draws heavily on the `searchInstagram()` function from the `instaR` package, and includes all the arguments for collecting Instagram data using that function.

1. We need to find the latitude and longitude of Brisbane, and supply these values as the ‘lat’ and ‘lng’ arguments.
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 the ‘brisbane_pics’ directory. We use the `folder` argument to achieve this.
4. We want to collect 100 posts, which we define as the value of the `n` argument.
5. We define `verbose` as TRUE. This means that `SocialMediaLab` will keep us informed of how things are progressing as the data collection occurs.
6. The Instagram API has a ‘rate limit’ for how much data you can access per hour. The current maximum is 5000 calls to the API per hour. If we don’t abide by the rules, the API temporarily (and sometimes permanently) denies access. To get around this, we can set the `waitForRateLimit` argument to TRUE. In this way `SocialMediaLab` will put itself to sleep if the rate limit is maxed out. Note: it will go to sleep for 60 minutes (until the rate limit refreshes), so keep this in mind!
We collect the data and store in the ‘myInstagramData’ object (of class data.table):

```r
myInstagramData <- Collect(lat=-27.4701, lng=153.0220, distance=5000, n=100, folder="brisbane_pics", verbose=TRUE, credential=instaToken)
```

We can view the data to see its makeup:

```r
View(myInstagramData)
```

4.3 Network generation

Now that we have our Instagram data we can generate a bimodal network. This kind of network provides many possibilities for analysis and generating insights from our data.

In this bimodal network there are two types of nodes: users and posts. A user can either comment or like a post, therefore there are two types of edges (i.e. ties between nodes): comments and likes. If a user i has both liked and commented on a post j, then there are two edges e1 and e2 directed from i to j.

The bimodal network is therefore:

- directed (users can like or comment on posts, but posts can’t like or comment back)
- weighted (users can comment multiple times on a post)
- bipartite (users can like or comment on posts, but posts can’t like or comment back)
- multiple edges or parallel edges (we have one edge for each interaction from user i to post j)

We now run the `Create()` function, which creates an igraph object called `g_bimodal_instagram`. Creating networks in SocialMediaLab is straightforward. We simply pass the `myInstagramData` object to the `Create()` function, and it takes care of the rest. We specify what kind of network we want to create (i.e. a bimodal network) by specifying this as an argument to the `Create()` function.

To use a crude analogy, this workflow is the equivalent of passing the ingredients for a cake into a machine that mixes them and does the baking, and outputs the finished cake. Note also that there is a tricky operator introduced here, the ‘pipe’ operator `%>%`, which we have not covered yet. This operator comes from the `Magrittr` package, and it is used to ‘pipe’ together commands in a chain, passing the values along the pipeline until it reaches the final command, which returns the output (i.e. the network we wish to create). In this instance we are passing (or “piping”) the data we collected using `Collect()` through to the `Create()` function.

```r
g_bimodal_instagram <- myInstagramData %>% Create("Bimodal")
```

We can now view basic information about the network:

```r
g_bimodal_instagram
```

Next we will do some more descriptive analysis.

How many nodes are in the network?

```r
vcount(g_bimodal_instagram)
```

How many edges in the network?
Get a list of the nodes in the network:

\( V(g_{bimodal\_instagram}) \)

List of edges in the network:

\( E(g_{bimodal\_instagram}) \)

Access a particular node in the network (node #42):

\( V(g_{bimodal\_instagram})[42] \)

Access a particular edges:

\( E(g_{bimodal\_instagram})[1] \)

Who are the neighbours of node #42? In other words, which posts does this user like and/or comment on?

\( neighbors(g_{bimodal\_instagram}, 42) \)

We can look at some network cohesion measures.

How dense is the graph? In other words, of all the possible connections between nodes, how many are actually observed? (For this kind of Facebook network it will be tiny…)

\( \text{graph.density}(g_{bimodal\_instagram}) \)

Who are the top 3 most important posts in the network? There are several ways to do this. For fun we will use the PageRank algorithm implementation in \texttt{igraph} to calculate this. PageRank is made famous by the Google co-founders, who invented this method to determine the importance of webpages, revolutionising the search engine industry. The following code calculates PageRank for nodes in the network, and returns the 3 ‘top’ nodes (which have the highest share of PageRank), providing the ID.

\[
pagerank_{instagram} <- \text{sort}(\text{page.rank}(g_{bimodal\_instagram})[\text{vector}], \text{decreasing=TRUE}) \]

\[
\text{head}(pagerank_{instagram}, n=3) \]

Finally, it is also possible to simplify the entire workflow using the “pipe” interface approach. We can \texttt{Authenticate()} with the API and \texttt{save} the token credentials to our computer (in the current R working directory), so we can just \texttt{load} it up any time we wish to authenticate with the API (rather than doing the browser dance). We can then load the credential, \texttt{Collect()} our data, and \texttt{Create()} our bimodal network all in one go.

We will not collect pictures this time around (i.e. using the \texttt{folder} argument to \texttt{Collect()} function). We will collect 100 recent Instagram posts from Newport Beach, California.

Importantly, we will write the network to disk (into our current R working directory), by specifying \texttt{writeToFile=TRUE} in the \texttt{Create()} function. This will write the network to disk in ‘graphml’ format, which we will then import into \texttt{Gephi} at the end of this tutorial. The filename uses the following format: “Month\_Day\_Hour\_Minute\_Second\_Year\_Timezone\_Instagram\_Bimodal\_Network”. For example: “Apr\_05\_21\_34\_23\_2016\_PDT\_Instagram\_Bimodal\_Network.graphml”.

12
5. Collecting Twitter data and creating actor networks

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 tokens between sessions. The `Authenticate()` function is called only for its side effect, which provides access to the Twitter API for the current session.

### 5.1 Authenticating with the Twitter API

```r
myapikey <- "xxxx"
myapisecret <- "xxxx"
myaccesstoken <- "xxxx" # avoids the browser authentication dance
myaccesstokensecret <- "xxxx" # avoids the browser authentication dance
```

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.

The first step in the work flow is to authorise access the Twitter API. Instructions for obtaining Twitter API access are available from the VOSON website. See the previous section for a brief explanation of APIs.

```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:
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. Roughly speaking, this command will help to deal with ‘odd’ characters in the text.

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:

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.

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

We can now examine the description of our network:

g_twitter_actor

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

What are the top 10 important terms in our #auspol actor network? There is no reason why we can’t use the PageRank algorithm to calculate this (as per the Instagram analysis previously):

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
```

### 5.4 Visualise the network using Gephi

Gephi is the ideal software for visualising networks created in *SocialMediaLab*. In this section we will import into Gephi the Instagram network we generated earlier, in order to visualise it. If you have not installed Gephi, please refer back to Section 2.3.

First, open Gephi. You should be presented with a dialogue box. Click “Open Graph File...” (or select from menu ‘File’–>'Open') and navigate to the working directory of your R project for this tutorial. The working directory should contain a graphml file that was generated in the Instagram section. Select this file and click open.

You will be presented with an ‘Import report’ dialogue box providing information about the network. Simply click OK. Your network will then be presented in a very awful format - possibly looking like a gray-coloured hairball. Let’s fix that.

In the top-left hand box under the ‘Appearance’ tab, click the little icon that says ‘Size’ when you hover over it. Select “Indegree” from the dropdown menu. Set the min size to 5 and the max size to 50, and click Apply. Now the nodes in your network are sized based on how many inlinks they receive from other nodes.
In the bottom-left hand box named ‘Layout’, click the ‘Choose a layout’ dropdown menu. Select “ForceAtlas 2”. In the options for this layout, click the ‘Prevent Overlap’ checkbox. Set the ‘Scaling’ to about 7, and set the Gravity to about 50. You may need to adjust these settings depending on the size and density of your network.
network. Finally, click ‘Run’. You will see the network explode into action, rearranging the nodes and edges into something (hopefully) much more visually appealing and interpretable.

Third, in the right-hand ‘Statistics’ box, click Run on the ‘Modularity’ option, and click OK when the dialogue box pops up. This will detect ‘communities’ or modularity classes of nodes in your network and label the nodes automatically based on their assigned modularity class. A report will pop up and you can just close it.
Now go back to the ‘Appearance’ tab. Click on ‘Nodes’, click on the ‘Color’ icon, click ‘Attribute’, and select ‘Modularity Class’. Now click ‘Palette’ and select a colour scheme you prefer, then click ‘Apply’. Now we have a colourful network, where the nodes are coloured according to which community cluster they are assigned to.

You can now play around with the network and interact with it. Test out the different buttons and features that are available in the left-hand menu. There are many tutorials and guides on the web for how to manipulate networks in Gephi.
Finally, you can create really attractive network visualisations by clicking on the ‘Preview’ button at the top of the screen.

In the Preview section, ensure ‘Show Labels’ is toggled on. You can try toggling ‘Proportional Size’ to off. Now hit ‘Refresh’ at the bottom, and voila! You have a nice network with labels on the nodes.

6. Conclusion

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

All the best,

Timothy Graham and Robert Ackland.