txtXploR

Jabber Cruncher

Thursday, September 25, 2014

Contents

Text Mining

Text mining is the process of deducing information from text. This typically involves parsing sentences and deducing patterns. All language processing used in this package entail tockenisation; breaking text into words. This practice is often criticised for *treating text as bags of words*. Thereby overlooking complex information conveyed by associations of words rather than one term on its own. However it should be ackowledged that this package focuses on processing social media conversations, particularly those taking place on Twitter. Tweets are 140-character long maximum and tend to convey simple and concise ideas. Tockenisation only truly becomes a problem when processing text that communicate complex concepts in which case looking at terms individually might not give an accurate representation of the whole item. I have also discussed this very issue when running a sentiment analysis.

txtXploR

This document looks into the functionalities of the txtXploR package. The package does not provide any original function but rather attempts to neatly wrap functions from other packages, mainly that of the tm package.

Functions included in the package:

- `clean_corpus`
- `term_bar`
- `hier_clust`
- `pam_k`
- `plot_topic`
- `term_network`
- `tag_cloud`

The aim of the package is to ease the application of various commonly used text mining functions and models. It does so by making all the functions work at a term-document matrix level. Thereby curbing the amount of text processing and data manipulation that has to be made.

Below we attempt to demonstrate the ease with which the package can be used. We do not assume that the audience has knowledge of text mining as we try to explain the various functions and steps undertaken as comprehensively as possible. Note that facilitating text mining comes at a cost; reduced flexibility. However it is an ideal stepping stone to discover text mining and mastering the tm package and other text mining tools.

Pulling Data

Here we show two ways of mining twitter data. One collecting data via the Twitter REST API using the twitteR package and another that obtains near real-time data from the Twitter Streaming API using the streamR package. Though both methods are demonstrated below the example will be using data obtained via the Twitter REST API.

Twitter APIs require authentication by using API Keys which can be acquried by creating an app. The application enables generating the requried tokens (API Keys tab). Take note of the API key, API secret, access token and token secret which you are going to need to register - different OAuth systems for each package, same information and/or app needed.

Using streamR:

```
#load or install required packages
if("streamR" %in% rownames(installed.packages()) == FALSE) {
  install.package("streamR")
  library(streamR)
} else if ("ROAuth" %in% rownames(installed.packages()) == FALSE){
  install.packages("ROAuth")
  library(ROAuth)
} else {
  libs <- c("ROAuth", "streamR")
  lapply(libs, library, character.only=TRUE)
}

#OAuth
requestURL <- "https://api.twitter.com/oauth/request_token"
```

```
accessURL <- "https://api.twitter.com/oauth/access_token"
authURL <- "https://api.twitter.com/oauth/authorize"
my_oauth <- OAuthFactory$new(consumerKey = "your_consumer_key", consumerSecret = "your_consumer_secret"
    requestURL = requestURL, accessURL = accessURL, authURL = authURL)

#download cacert.pem
download.file(url="http://curl.haxx.se/ca/cacert.pem", destfile="cacert.pem")

#Register OAuth
my_oauth$handshake(cainfo = system.file("CurlSSL", "cacert.pem", package = "RCurl"))

#Save OAuth for future sessions (use load)
save(my_oauth, file = "my_oauth.Rdata")

## date when loop will be stopped
end.date <- as.Date("2014-10-20")

## continue running until current date is end.date
while (Sys.Date() < end.date){
  current.time <- format(Sys.time(), "%Y_%m_%d_%H_%M")  #Save file
  file.name <- paste("scotland_", current.time, ".json", sep="")
  filterStream( file=file.name, track="scotland", locations = c(49, -11, 61, 1),
                oauth=my_oauth, timeout=3600) ## capture tweets for 3600 seconds = 3 hours
}

#Parse results, your_json_file = file.name in while loop above
df <- parseTweets(your_json_file, simplify = TRUE)
```

Using twitteR:

```
#load twitteR, download if not already installed
if("twitteR" %in% rownames(installed.packages()) == FALSE) {
  devtools::install_github("geoffjentry/twitteR")
  library("twitteR")
} else {
  library("twitteR")
}
```

```
setup_twitter_oauth("API key", "API secret", "Access token", "Access secret")
```

```
## [1] "Using direct authentication"
```

Once registered we can pull some tweets from the REST API. It will return a list of past tweets; up to seven days old depending on the amount of tweets posted.

```
#Mine 2000 tweets on rstats in english
tweets <- searchTwitter("rstats", n=2000, lang="en")

class(tweets)
```

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

In this example we mined 2000 tweets that include the keyword "rstats", this include the hashtag *"#rstats"*. The tweets are returned as a list, we run `twListToDF` to unlist them and obtain a data.frame.

```
#Unlist the tweets to a data.frame
tw_df <- twListToDF(tweets)

#Print names
names(tw_df)
```

```
##  [1] "text"          "favorited"     "favoriteCount" "replyToSN"
##  [5] "created"       "truncated"     "replyToSID"    "id"
##  [9] "replyToUID"    "statusSource"  "screenName"    "retweetCount"
## [13] "isRetweet"     "retweeted"     "longitude"     "latitude"
```

Corpus

With data we can start looking at the functions of the txtXploR package. We start by transforming the text column to a corpora; a collection of documents. In this case the documents are tweets. Text is inherently dirty so it will have to be cleaned. The `clean_corpus` functions will remove punctuaion, urls, numbers, etc.

The function also allows stemming terms. In most cases you will want to stem the documents. Stemmming brings the terms to their root (radicals - stem) so that various forms of a stem are counted as identical, i.e.: *cat* and *cats* (same lemma). The stemmer removes suffixes, affixes, plurals and the like. However the stemmer is dependent on language and at this point the function only supports the English language. Hence specifying `lang="en"` in the `searchTwitter` function earlier on.

By default the function removes English stop words. These are short function words which are not essential in uncovering the meaning of text documents. Terms such as "the", "which", "at", etc. You can remove other terms using the rem_words parameter. For instance, it is wise to remove the very search term you used to pull the tweets, it's very likely to be the most frequent term and will be associated to numerous other terms if not all as it is present in every tweet. Moreover the function might miss some stop words; this will enable you to remove them.

Check the documentation for more information `?clean_corpus`.

```
#transform tweets to corpus
corpus <- Corpus(VectorSource(tw_df$text))

#clean corpus, stem and remove some terms including search term
corpus <- clean_corpus(corpus=corpus, stem=TRUE, rem_words=c("rstats", "data", "lol", "omg", "just"))
```

Term-document matrix

Once the corpus is clean we can transform it into as term-document matrix. It follows the popular *tf-idf* scheme in which the occurences of a defined set of terms are counted in each of the documents comprising the corpus. The count of occurrences is then compared to the inverse document frequency count which effectively measures the occurences of a term in the whole corpus. The term-document matrix is generally a pivotal component of text mining hence making **all** the following functions work on that level.

```
tdm <- TermDocumentMatrix(corpus, control=list(minWordLength=1))
```

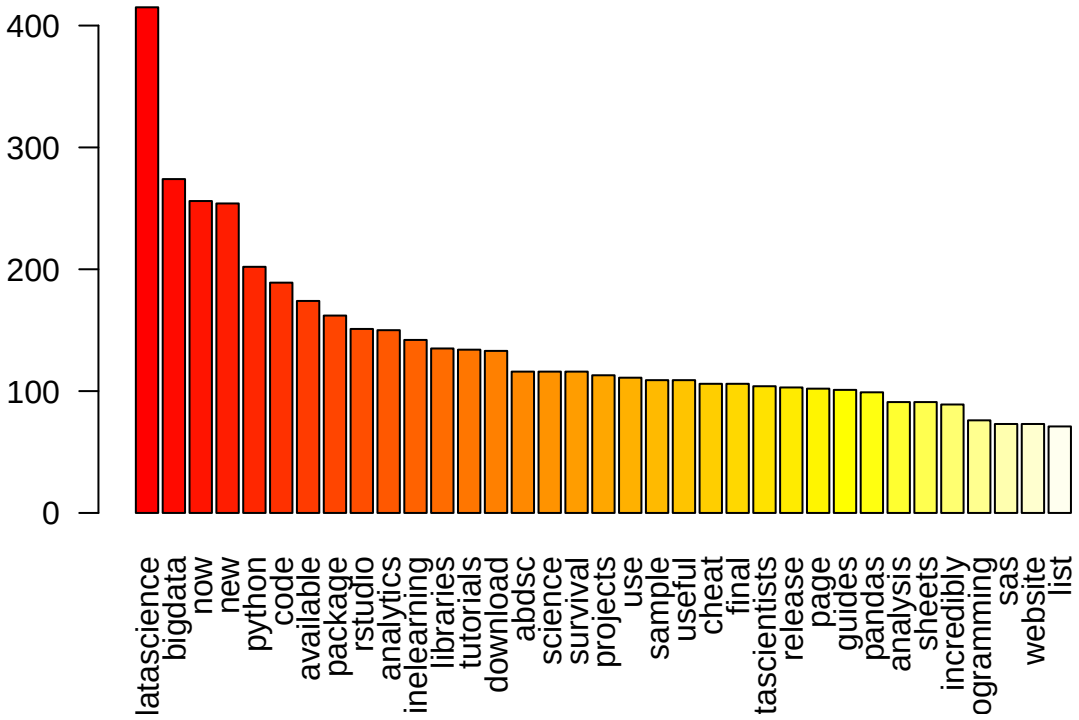
Term Frequency

Barplot

At this point we can start exploring the content of the tweets. We can fist have a look at the most popular terms by creating a barplot of the most popular terms. `term_freq` indicates the minimum frequency of terms to be plotted; the higher the fewer terms on our barplot. The function defaults to `colour = "heat"`.

- x axis: terms with frequency > `term_freq`
- y axis: number of occurences

```
term_bar(tdm, term_freq = 70, colour = "heat")
```



tag cloud

You can also use a word cloud to plot term frequency. Although more fancy it is *less accurate* as in it makes it more difficult to see the ranking of terms. On the word cloud the color as well as the size are a function of the occurences of terms. Then again, `term_freq` indicates the minimum frequency of terms to appear on the cloud. `scale` is a vector of length 2 indicating the range of the size of the words. While `order` is a boolean to choose whether you want the words to be spread on the wordcloud at random or not, defaults to FALSE. When FALSE the cloud will show the most popular terms at the center and the least on the edges in a circular fashion.

```
library(wordcloud)
library(RColorBrewer)
tag_cloud(tdm, min_freq = 20, scale=c(3,.2), order = FALSE)
```

This might give you a bit of an idea what is being discussed online. However we can be a bit more thorough.

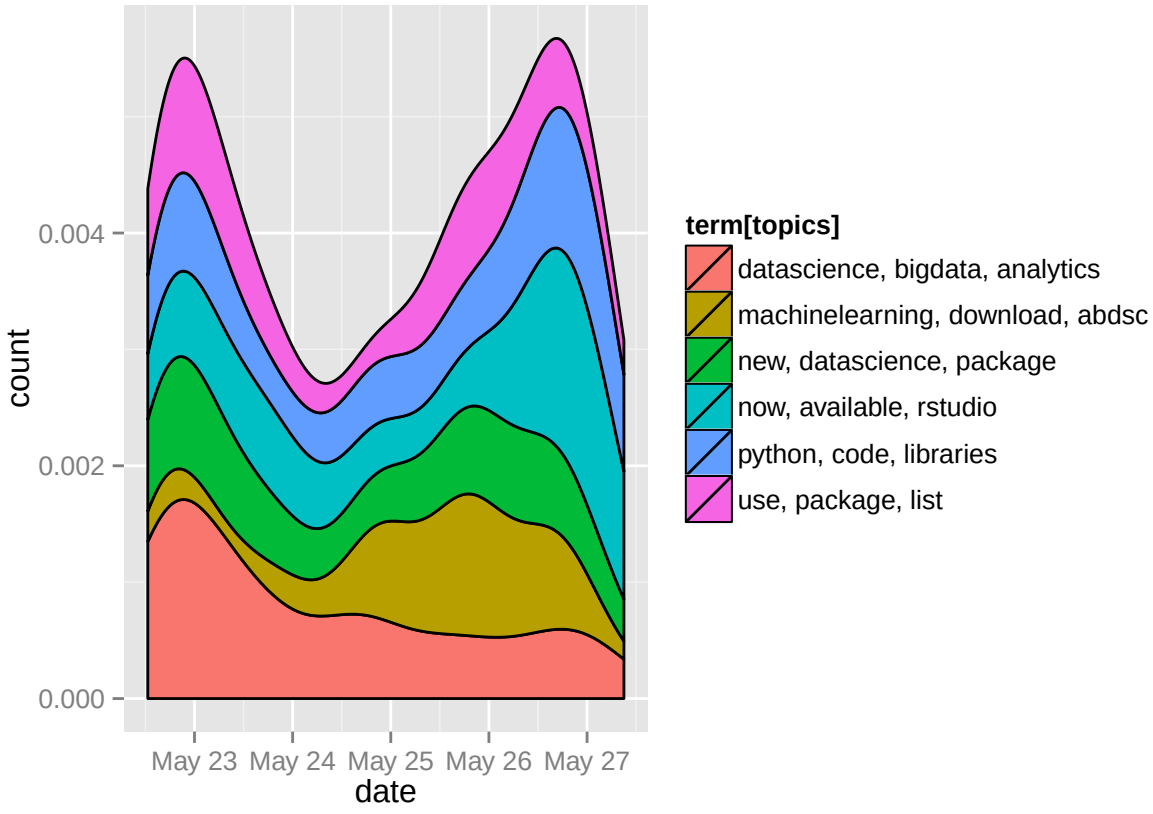
Uncover topics

We can use `plot_topic` to plot the topics discussed and their evolution over time. This uses the Latent Dirichlet Allocation generative probabilistic model built in the topicmodels package. The bayesian model assumes each document is a mixture of underlying topics and that each term is associated to a topic.

Since the output of the function is a graph depicting the evolution of the topics over-time we have to specify a vector of dates to be used. We also specify the number of topics we want to obtain, this defaults to 5. Meanwhile `terms` is an integer of the number of terms to be shown for each topic, defaults to 4.

```
plot_topic(tdm, tw_df$created, topics=6, terms=3)
```

```
##
## Attaching package: 'ggplot2'
##
## The following object is masked from 'package:NLP':
##
##     annotate
```

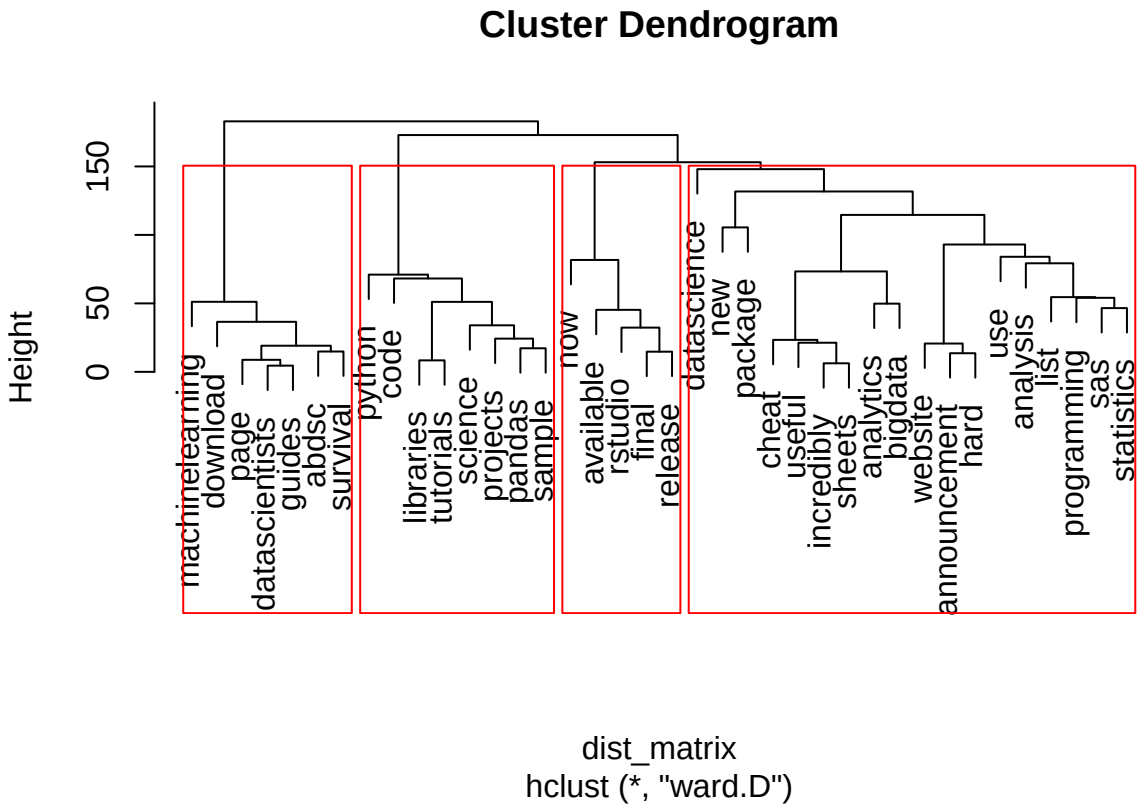


Cluster terms

Hierarchical clustering

We can also use various clustering methods to cluster words together. txtXploR provides two types of clustering methods partitioning around medoids (`pam_k`) and hierarchical clustering - `hier_clust` which removes sparse terms, calculates the distances between terms then clusters words. The function returns a dendrogram.

```
hier_clust(tdm, sparsity=0.97, clusters=4)
```

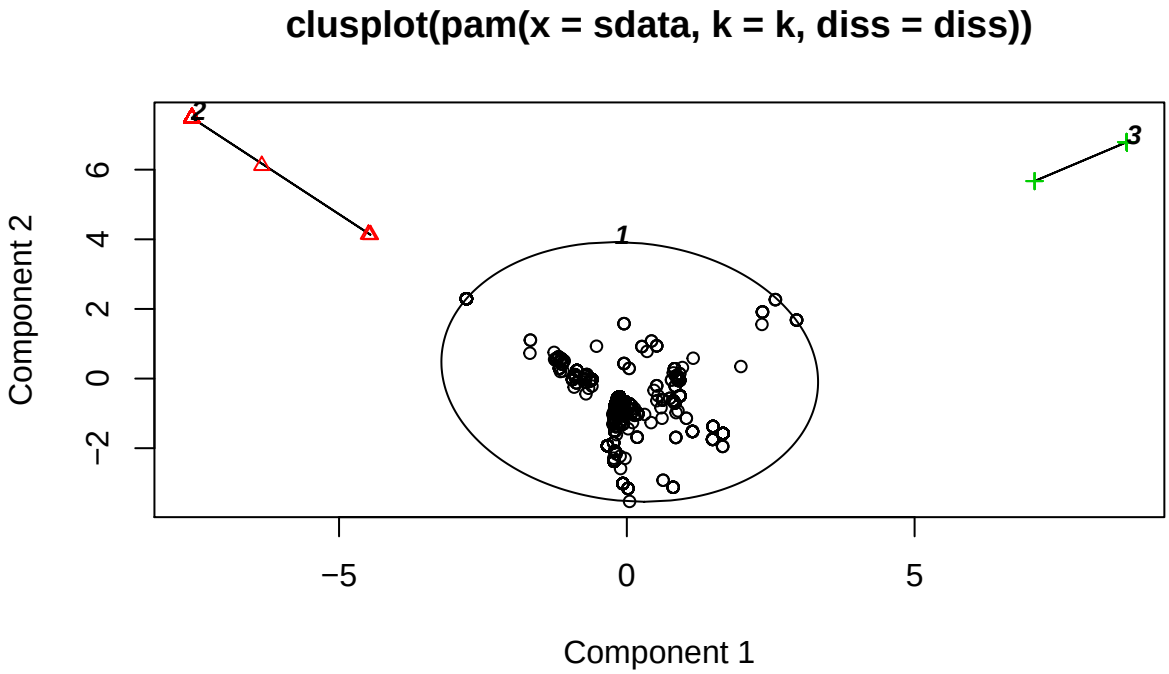


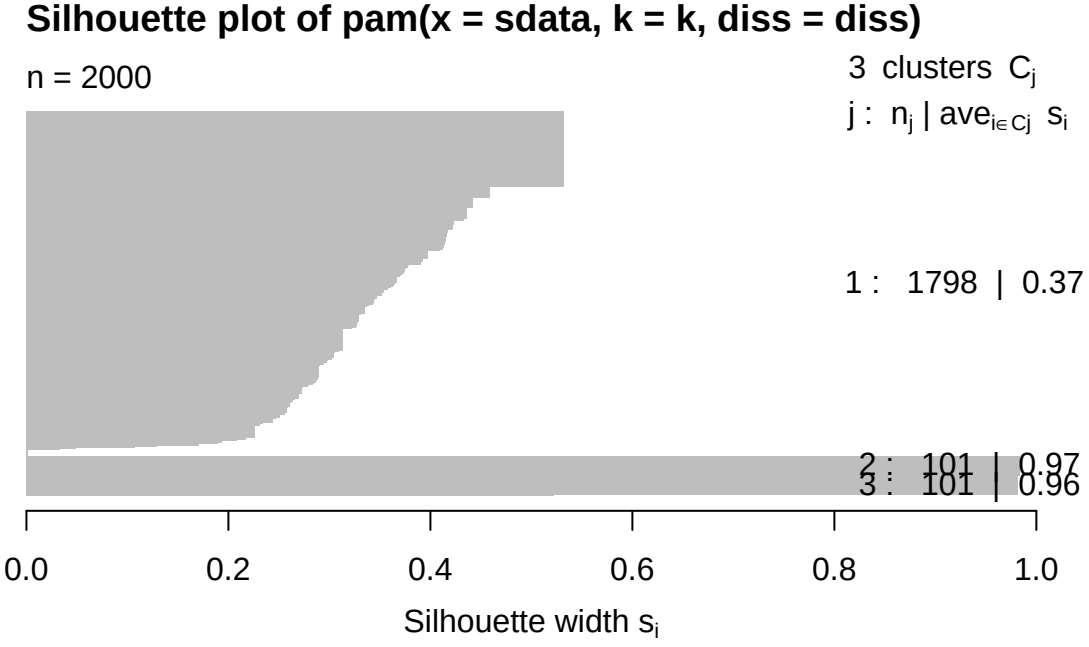

```
##
## Call:
## hclust(d = dist_matrix, method = "ward.D")
##
## Cluster method   : ward.D
## Distance         : euclidean
## Number of objects: 38
```

Partitioning around medoids

The other clustering method - `pamk_k` - uses the Partitioning Around Medoids (PAM) algorithm. It is similar to k-means but uses medoids (center) rather than means. The higher the silhouette width the better the clusters fit.

```
pam_k(tdm, sparsity=0.98)
```





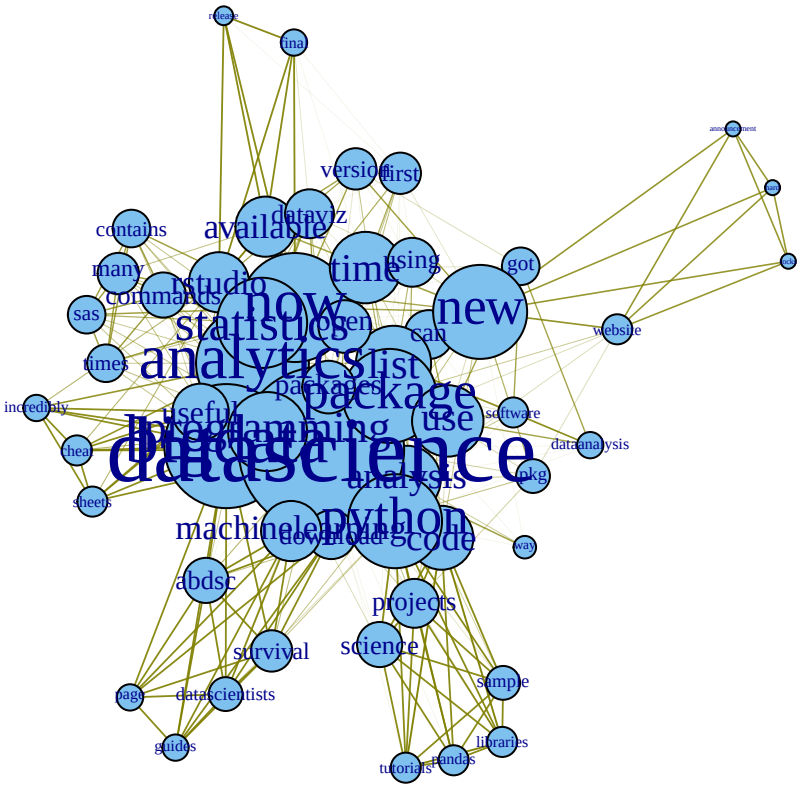
Network of terms

Finally we can use `term_network` to explore associations between terms. You can save the graph as a graphml file by specifying `graphml=TRUE`. The size of the nodes

```
graph <- term_network(tdm)
```

```
## [1] "saving graph as object"
```

```
plot(graph, vertex.size=V(graph)$degree, layout=layout.fruchterman.reingold,
      vertex.label.cex=V(graph)$degree/15)
```



Social Network Analysis