Chapter 12

Empirical Analysis of the Stack Overflow Tags Network

Christos Iraklis Tsatsoulis

Nodalpoint Systems, Athens, Greece

12.1 Introduction 265
12.2 Data Acquisition and Summary Statistics 267
12.3 Full Graph — Construction and Limited Analysis 268
Clusters and Connected Components 269
Node Degrees 270
Clustering Coefficient, Path Length, and Small-world Characteristics 271
12.4 Reduced Graph — Construction and Macroscopic Analysis 272
General Statistics 272
12.5 Node Importance: Centrality Measures 273
Betweenness Centrality 273
Closeness Centrality 275
Eigenvector-based Centrality 276
12.6 Community Detection 277
Fast Greedy Algorithm 277
Infomap Algorithm 279
12.7 Visualization 282
Visualizing the Communities Graph 283
Egocentric Visualizations 284
12.8 Discussion 285
12.9 Appendix: Data Acquisition & Parsing 290
Data Acquisition 290
Parsing the Tag List & Frequencies 290
Parsing the Tag Co-occurences 291
Building the Adjacency Matrix 291
12.9 Bibliography 292

12.1 Introduction

Stack Exchange is self-described as "*a fast-growing network of 129 question and answer sites on diverse topics from software programming to cooking to photography and gaming*", with a current base of more than 5.5 million users and over 82 million monthly unique visitors[1]. By far, the greatest among these 129 sites is Stack Overflow, devoted to questions about computer programming and related topics.

[1] http://stackexchange.com/.

Question tags are an essential part of the submitted questions in Stack Overflow, allowing answering users to monitor questions relevant to their field of expertise and to answer promptly to submitted questions. Only users holding an "advanced" status in Stack Overflow are allowed to generate new tags, with the rest of the community limiting themselves in using the existing tags for their questions. Tagging the submitted questions is mandatory (i.e., there is a minimum of one tag per question), and each question can contain up to five tags.

In this chapter, we depart slightly from text analysis proper in order to undertake a task inspired by the recent advances in what is usually termed as "network science" : hence, we carry out an analysis of the Stack Overflow tags viewed as a *network,* or a *graph.* Specifically, we aim to get some insight about the user communities by representing tags and their co-occurrences as a graph, where the graph nodes are the tags themselves, and an edge between two nodes exists if the corresponding tags are found together in the same Stack Overflow question. The resulting network edges are *weighted*, i.e., the more questions exist with two tags co-occurring, the higher the weight of the relevant edge between them will be; for example, if the tag `python` is found to co-exist with the tag `android` in 500 questions, the weight of the edge between the nodes `python` and `android` in our graph will be 500. In graph terminology, the resulting graph is a *weighted undirected* one.

What is the motivation behind our approach? The recent development of network science [1], largely triggered by the publication of two seminal papers in the late '90s [2, 3], has since made clear that the graph representation of data and processes is certainly of benefit, and can lead to insights not directly or easily achievable otherwise. Our scope is to demonstrate the merit of such a representation in a text-related area and to possibly alert the interested reader for the availability of the relevant principles, tools, and techniques.

There are several introductory textbooks available on network science: Easley and Kleinberg [4] aim for an advanced undergraduate audience and have made their book freely available online;[2] the books by Jackson [5] and Newman [6] are standard graduate-level introductions to the subject, from the perspectives of economics and statistical physics respectively; Kolaczyk [7] provides another graduate-level introduction, with emphasis to the statistical analysis of network data; and Barabási [1] is gradually making available an online introductory book as a work in progress. We also highly recommend the popular book by Duncan Watts [8] for a smooth, non-mathematical introduction to the relevant concepts, notions, and ideas. We will not provide here a formal introduction to graph theory and network science, something that would require a separate chapter by itself; instead, we will introduce rather informally and intuitively the relevant concepts as we go along. For formal definitions, the interested reader can always consult some of the free online resources mentioned above. For the purposes of our discussion, we will use the terms "network" and "graph" loosely and interchangeably, as if they were synonyms (although they are not). The other interchangeable pairs of terms we will use are "node" – "vertex" and "edge" – "link".

We have chosen R [9] for our investigation, as it has become the de facto standard tool for the statistical analysis of data. Among the several available packages for graph and network analysis, we utilize the `igraph` library [10], a powerful and flexible tool which is already used by at least a hundred other R packages [11].

The rest of this chapter is organized as follows: in the next section, we describe briefly the data acquisition process and present some summary statistics of the raw data; in Section 3 we expose the construction of our first graph based on the whole data set, along with some limited analysis; Section 4 describes the meaningful reduction of our raw data set, which is used in the rest of the chapter; Section 5 deals with various measures of node importance (centrality), and Section 6 we demonstrate the application of some community

[2] http://www.cs.cornell.edu/home/kleinber/networks-book/.

Number of records (questions)	7,214,697
Number of unique tags	36,942
No. of tag occurrences	21,294,348
Average no. of tags/question	2.95

TABLE 12.1: Statistics of raw data.

detection algorithms; some attempted visualizations are presented in Section 7, utilizing the communities detected in Section 6; Section 8 concludes the chapter with some general discussion.

12.2 Data Acquisition and Summary Statistics

Stack Exchange makes its updated "data dump" periodically available roughly twice a year (January & September), under a Creative Commons BY-SA 3.0 License.[3] The version we are going to work with is of January 2014. The questions data are available as a large (~ 25GB) XML file, which was downloaded and parsed using BaseX, an open-source lightweight XML database[4]. A separate XML file with the tag frequencies is also provided. The details of the raw data parsing and processing can be found in the Appendix; the results are:

- A list of the used tags, ordered by decreasing frequency
- An adjacency matrix keeping track of the tag co-occurrences, as described in the Introduction.

These results are available as R workspace files (`tag_freq.RData` and `adj_matrix.RData` respectively) from the author's Github repository[5], and they constitute the departure point for our analysis; as said, readers interested in the parsing procedure itself and the construction of these results from the raw data can consult the Appendix; the R scripts used for the parsing are also available at Github, so the whole procedure is entirely reproducible, and even extensible as new versions of the Stack Overflow data dump become available in the future. The summary statistics of the raw data are shown in Table 12.1.

The `tag_freq` dataframe contains two variables: the name of the tag and the corresponding frequency (counts) in the data set. We will load it and add a third variable, to indicate the relative percentage frequency of each tag, and then view the 10 most frequent tags:

```
> load("tag_freq.RData") # needs to be in the current working directory
> tag_freq$rel <- tag_freq$freq/sum(tag_freq$freq)*100
> head(tag_freq,10)
          tag   freq      rel
1          c# 635338 2.983599
2        java 632575 2.970624
3  javascript 605552 2.843722
4         php 573279 2.692165
```

[3] http://blog.stackexchange.com/category/cc-wiki-dump/.
[4] http://basex.org/.
[5] https://github.com/desertnaut.

```
5     android 503255 2.363327
6      jquery 468723 2.201162
7      python 297952 1.399207
8         c++ 287483 1.350044
9        html 284234 1.334786
10      mysql 243701 1.144440
```

We can thus see the 10 most frequent tags, along with their absolute and relevant frequencies. Going a little further,

```
> sum(tag_freq$rel[1:10])
[1] 21.28307
```

we can easily see that the 10 most frequent tags account for about 21% of all tag occurrences in the raw data.

We can similarly explore the other end of the list, that is how many tags appear very infrequently in the data set:

```
> length(which(tag_freq$freq < 100))
[1] 26723
> length(which(tag_freq$freq < 10))
[1] 9614
```

We can thus see that the majority of our 36,942 tags appear fewer than 100 times, whereas about 26% of the tags have a rather negligible presence (fewer than 10 times) in our data.

These results suggest a highly skewed distribution, with relatively few frequent tags dominating the data set. Plotting the tag frequencies for the first 100 tags (recall that tags are indexed in decreasing frequency) confirms this intuition, as shown in Figure 12.1 :

```
> plot(tag_freq$freq, xlim=c(1,100),
+ xlab="Tag index", ylab="Frequency (counts)")
```

12.3 Full Graph — Construction and Limited Analysis

Having performed the initial exploration of our tag frequency distribution, we now proceed to construct the full graph using the adjacency matrix already built as shown in Appendix A (file `adj_matrix.RData`). The `igraph` library, from version 0.5.1 on, utilizes fully the sparse matrix infrastructure provided by the `Matrix` package [12], something extremely convenient, since working with dense matrices of size 40,000 x 40,000 can be really cumbersome for most common desktop computers.

```
> library(Matrix)
> library(igraph)
> load("adj_matrix.RData") # contains sparse matrix G
> g <- graph.adjacency(G, weighted=TRUE, mode=c("plus"), diag=FALSE)
> summary(g)
IGRAPH U-W- 36942 2699465 --
attr: weight (e/n)
```

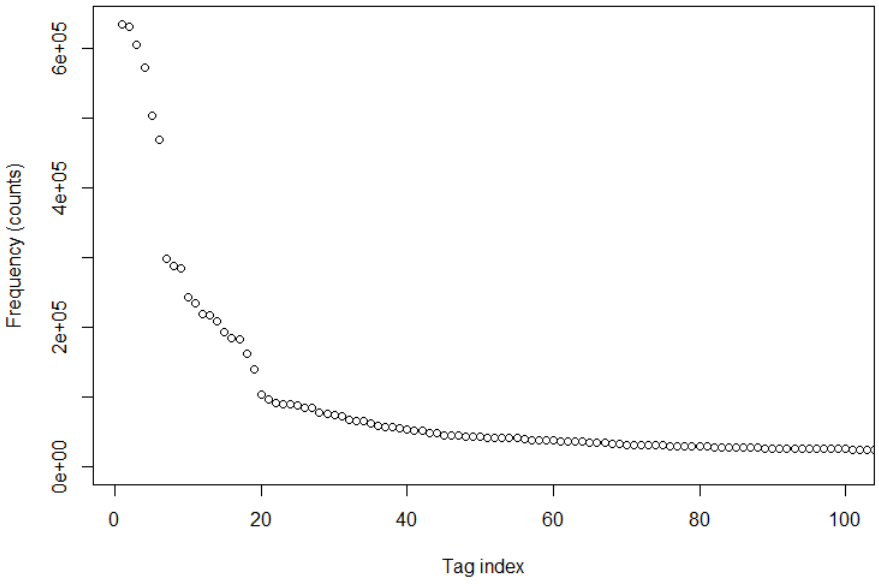

FIGURE 12.1: Tag frequency distribution for the first 100 tags.

The `summary()` function provides some general information about the graph `g`, namely, that it is undirected ('`U`') and weighted ('`W`'), with 36,942 vertices (nodes) and about 2.7 million edges, and only one attribute (`weight`), which refers to the edges ('`e`') and it is a numeric one ('`n`'). We can decorate the graph with more attributes, such as a name for it as well as names for each vertex (tag):

```
> # set the graph and vertices name attributes:
> g$name <- "FULL TAG GRAPH"
> V(g)$name <- tag_freq$tag  # set tags as node names
> summary(g)
IGRAPH UNW- 36942 2699465 -- FULL TAG GRAPH
attr: name (g/c), name (v/c), weight (e/n)
```

We can see that our graph is now named ('`N`'), and it contains two more character attributes ('`c`'), one for the whole graph ('`g`') and one for the individual vertices ('`v`').

Clusters and Connected Components

One of the first things in network diagnostics is to check if the network is globally connected:

```
> is.connected(g)
[1] FALSE
```

Since our graph is not globally connected, the next step is to check for its connected components (clusters):

```
> cl <- clusters(g)
> str(cl)
```

```
List of 3
 $ membership: num [1:36942] 1 1 1 1 1 1 1 1 1 1 ...
 $ csize     : num [1:53] 36888 1 1 1 1 ...
 $ no        : int 53
> cl$csize
[1] 36888     1     1     1     1     2     1     1
[9]     1     1     1     1     1     1     1     1
[17]    1     1     1     1     1     1     1     1
[25]    1     1     1     1     1     1     1     1
[33]    2     1     1     1     1     1     1     1
[41]    1     1     1     1     1     1     1     1
[49]    1     1     1     1     1
```

From the `csize` attribute of our clusters variable, it is apparent that our graph consists of one giant connected component (36,888 nodes) and 52 clusters with only one to two nodes each. It is trivial to write a short loop, in order to check the tag names in these small isolated clusters (only the first 10 shown here, for brevity):

```
# print isolated tags ("graph periphery")
l <- cl$no # number of clusters
for (i in 2:l) {
    label <- V(g)$name[which(cl$membership==i)]
    print(label)
}
```

```
[1] "untagged"
[1] "beaker-testing"
[1] "arden-syntax"
[1] "zzt-oop"
[1] "ontopia" "tolog"
[1] "ebuild"
[1] "nikeplus-api"
[1] "infinity.js"
[1] "jquery-ias"
[1] "paperless"
```

Even without specific domain knowledge, it is apparent that all the tags in the "graph periphery" listed already are far from common terms expected to be found in a programming-related forum.

Node Degrees

The *degree* of a node is simply the number of its direct neighbors, or equivalently the number of the links in which the node is involved [5, 6]. The degree is very frequently used as a proxy for the importance or the significance of individual nodes in a graph. `igraph` provides a relevant function `degree(g)`, which can be used to compute the degrees of all the nodes in the graph `g`; nevertheless, it is of no use in our case here, as it does not take into account the weights of the graph edges. The correct `igraph` function for the computation of the *weighted* degree [13] is `graph.strength(g)`. Again, it is straightforward to see the nodes with the highest weighted degrees in our graph:

```
> s <- graph.strength(g)
> V(g)$name[which.max(s)]
[1] "c#"
> s <- sort(s, decreasing=TRUE)
```

```
> s <- as.data.frame(s) # for better display
> head(s, 10)
                 s
c#         1485191
java       1446078
javascript 1359147
php        1192764
jquery     1034841
android     965297
html        705233
c++         639678
ios         621107
python      608211
```

Not unsurprisingly, the above list is similar (but not identical) to the 10 most frequent tags already shown. If we plot the weighted degree, the results are qualitatively very similar with Figure 12.1 (not shown).

Clustering Coefficient, Path Length, and Small-world Characteristics

The *local clustering coefficient* of a node is the fraction of its direct neighbors that are themselves direct neighbors (i.e., directly connected); in a social network framework, for example, it would be the fraction of one's friends that are also friends themselves [4]. The *average* clustering coefficient of the whole graph is simply the average of the nodes' local clustering coefficients [5]. By definition, both coefficients lie between 0 and 1. `igraph` provides the function `transitivity()` for the computation of these quantities, as well as a simple function for the computation of the average path length in a graph. For these calculations, we ignore the edge weights, treating the edges between nodes simply as binary entities (present/absent):

```
# Clustering coef & average path length
# Use type="average" argument for average cluster coef
> cc <- transitivity(g, type="average", isolates="zero")
> cc
[1] 0.6099061
> path.length <- average.path.length(g, unconnected=TRUE)
> path.length
[1] 2.341709
```

Thus, on average, the "distance" between any two nodes in our graph is between 2 and 3 hops, making our graph a small-world network [4, 5, 6]. Before commenting on the clustering coefficient, we take a moment to compute the same quantities for a random (Erdös-Renyi - ER) graph with the same number of nodes and edges:

```
# Compare with an equivalent random (ER) graph
> set.seed(42) # for reproducibility
> g.ER <- erdos.renyi.game(length(V(g)), length(E(g)), type="gnm")
> cc.ER <- transitivity(g.ER, type="average", isolates="zero")
> cc.ER
[1] 0.00395176
> path.length.ER <- average.path.length(g.ER, unconnected=TRUE)
> path.length.ER
[1] 2.554753
```

We can see that, compared with a random graph with the same number of nodes and edges, our graph possesses roughly the same average path length between two nodes, with a

hugely higher (two orders of magnitude) clustering coefficient. This is a recurrent motif for several real-world networks, including social ones [6], first captured in an abstract network model by Watts and Strogatz in a seminal paper published in *Nature* [2], that partially triggered the subsequent birth of the whole network science field.

12.4 Reduced Graph — Construction and Macroscopic Analysis

We have presented some general analysis for our full graph, but the information revealed so far is rather trivial. To go deeper, we will need to revert to a reduced graph, due to the prohibitively high computational power demanded for the analysis of the full tag graph; due to the highly skewed distribution of the tag occurrences (see Section 12.2), we claim that this reduction still captures almost all of the interesting characteristics of our data set.

We suggest that we proceed with a graph containing only those tags which occur a minimum of 500 times in our data (recall that we have more than 7 million questions and 21 million tag occurrences in total; see Table 12.1). We have taken special care when building our adjacency matrix, so that its columns and rows extend in a decreasing tag frequency; hence, it is a trivial operation to discard the following tags a given frequency in the adjacency matrix:

```
> # select only tags with freq >= 500
> ind <- which(tag_freq$freq >= 500)
> max(ind)
[1] 3711
> tag_freq <- tag_freq[ind,]
> sum(tag_freq$freq)
[1] 19166670
> sum(tag_freq$rel)
[1] 90.00825
> # keep only these entries in the adjacency matrix
> G <- G[1:max(ind), 1:max(ind)]
```

There are thus 3711 tags in our reduced list, or about 10% of all tags used; the `sum()` functions in the code above show that our reduced tag list contains more than 19 million tag occurrences out of the 21 million in total, i.e., about 90%. Proceeding with a manner absolutely similar to the one presented for the full graph in Section 12.3, we end up with our reduced graph `gr`, containing about 1 million edges:

```
> summary(gr)
IGRAPH UNW- 3711 1033957 -- REDUCED TAG GRAPH
attr: name (g/c), name (v/c), weight (e/n)
```

We have thus now a much more "lightweight" graph, nevertheless capturing the vast majority of the most frequently used tags. We are ready now to apply some more sophisticated techniques from the toolbox of graph theory and network science in order to reveal possible unexpected information and insights, hidden in the network structure.

General Statistics

```
> is.connected(gr)
[1] TRUE
```

```
> cc <- transitivity(gr, type="average", isolates="zero")  # clustering coef.
> cc
[1] 0.4992685
> path.length <- average.path.length(gr)
> path.length
[1] 1.849807
> # Compare with the equivalent ER graph
> set.seed(42) # for reproducibility
> gr.ER <- erdos.renyi.game(length(V(gr)), length(E(gr)), type="gnm")
> is.connected(gr.ER)
[1] TRUE
> cc.ER <- transitivity(gr.ER, type="average", isolates="zero")
> cc.ER
[1] 0.1501704
> path.length.ER <- average.path.length(gr.ER)
> path.length.ER
[1] 1.849801
```

We can see that our reduced graph is now globally connected (i.e., there are no isolated nodes); it has an even lower average path length between two nodes (1-2 hops), and its clustering coefficient is still higher than the one of the equivalent random graph, although this difference is not as marked as with our full graph.

In what follows, we will limit the discussion to the reduced tag network we have just constructed.

12.5 Node Importance: Centrality Measures

We recall from the previous sections that the information revealed so far is rather trivial: taking the weighted node degree as a rough importance measure, the most important nodes essentially coincide with the most frequent tags; clearly, we did not need to go to a graph representation in order to discover something like that.

The graph representation starts paying off when we consider other measures of node importance, which in the networks literature usually go under the label of *centrality* measures (actually, the *degree centrality*, which we have already demonstrated, is just one of them). Furthermore, these measures, in contrast with the *macro* measures considered so far (connected components, average clustering coefficients and path lengths), are in essence *micro* measures, since they do not refer to the graph as a whole, but rather to individual nodes and how they relate to the overall network structure [5].

There are several measures of node centrality in the literature, and most of them are implemented in `igraph`. We now proceed to have a look to some of them.

Betweenness Centrality

The betweenness centrality, first proposed in the context of social network analysis by Freeman [14], measures the extent to which a node is located in the shortest paths between other pairs of nodes (i.e., "between" them) [5, 11]. As with other centrality measures, `igraph` provides two ways of calculation: the straightforward `betweenness()` function, which can be computationally demanding, especially for large graphs; and the `betweenness.estimate()` function, with a cutoff argument for providing the maximum

path length to consider in the calculation. Although the direct calculation of betweenness is not prohibitive for our graph, we will hereby demonstrate the usage of the approximate function, which will most probably be of use in a range of other situations, where the graphs are prohibitively large for the direct application of the `betweenness()` function. In our case, the reader can confirm that the approximate calculation gives results identical to the exact one. Both functions take into account the edge weights [13].

A reasonable value to use as a cutoff value for the path length is the diameter of the graph, i.e., the maximum (shortest) path length between two nodes [5]:

```
# calculate (approximate) betweenness
> diam <- diameter(gr)
> diam
[1] 3
> bc <- betweenness.estimate(gr, cutoff=diam)
> which.max(bc)
[1] 111
> V(gr)$name[which.max(bc)]
[1] "debugging"
```

Keeping always in mind that our node indices are sorted as per their occurrence frequency it the data set, the meaning of the `which.max(bc)` above should be apparent: the node with the highest betweenness centrality in our graph corresponds to the tag `debugging`, which is ranked only 111th in our tag list.

In hindsight, it seems appropriate that the tag linking most of the questions in a computer programming forum should be something that has to do with errors or bugs, but we wonder how apparent this fact could be beforehand.

With a little data manipulation, it is easy to see more of the nodes with high betweenness centrality, and compare their betweenness ranking to the frequency ranking of the corresponding tags:

```
> # see betweenness ranking VS frequency ranking:
> bc.df <- data.frame(bc=bc, freq.rank = 1:length(V(gr)))
> bc.df <- bc.df[order(bc.df$bc, decreasing=TRUE),] # order in decreasing bc
> head(bc.df, 20)  # show the top-20
                       bc freq.rank
debugging        20363.39       111
design           19011.70       198
caching          17694.07       162
exception        17655.67       142
testing          16669.95       183
error-handling   16482.48       370
dynamic          15932.37       223
api              14840.20        78
data             14178.36       261
unit-testing     14144.10       101
user-interface   14009.76       185
parsing          13927.16       105
filter           13884.12       390
web-applications 13834.88       210
optimization     13730.19       201
memory-leaks     13653.39       305
logging          13646.06       207
object           13535.99       144
web              13496.93       254
text             13487.15       230
```

Thus, not only the highest betweenness node is nowhere to be seen in the 100 most frequent tags, but also none of the 10 most frequent tags scores high in betweenness centrality; we can see that, in fact, with the exception of the tag `api`, none of the 20 tags with the highest betweenness centrality is in the top 100 frequency list. We can go a little further, and check in particular how exactly the 10 most frequent tags score and rank in betweenness centrality:

```
> bc.df$bc.rank <- 1:length(V(gr)) # add bc.rank field
> ind <- which(bc.df$freq.rank <= 10) # find the 10 most frequent tags
> bc.df[ind,]  # display them
                bc freq.rank bc.rank
mysql      7035.926        10     178
html       6295.522         9     226
jquery     5764.672         6     263
android    5645.326         5     269
javascript 4691.252         3     387
c++        4147.752         8     440
python     3967.835         7     465
c#         3665.712         1     511
php        3539.976         4     538
java       2007.629         2     914
```

Hence, we can explicitly see that all 10 most frequent tags rank lower than 100 in betweenness centrality, with the 2nd most frequent one (`java`) barely making it to the top-1000 of betweenness centrality.

Interestingly enough, all the high betweenness node-tags shown above are somewhat "general" ones, in the sense that they are not related with specific software or hardware platforms. This makes perfect sense intuitively, and it demonstrates how the network structure can capture relations and characteristics in the data that are not easily or directly expressed in simple aggregate measures, such as the frequency of occurrence.

As said earlier, the reader can easily confirm that, for our graph, the approximate betweenness coincides with the exact one (which nevertheless would be computationally prohibitive for larger graphs):

```
> bc.exact <- betweenness(gr)
> length(bc==bc.exact)
[1] 3711
```

The last statement means that the condition `bc==bc.exact` is true for all our 3711 nodes.

Closeness Centrality

The *closeness centrality* of a node is a measure of how 'close' a node is to all the other nodes in the network [5, 11]. Since ours is a graph with a rather low diameter (3), we do not expect this measure to be of particular usefulness, but we include it here for illustrative purposes. Following a procedure completely analogous with the one already demonstrated for the betweenness centrality, but using instead the `igraph` function `closeness.estimate()`, we arrive at very similar results:

```
> head(clc.df, 20)
                         clc freq.rank clc.rank
design          0.0001497230       198        1
error-handling  0.0001496110       370        2
```

```
debugging       0.0001485884       111        3
filter          0.0001484120       390        4
caching         0.0001483900       162        5
testing         0.0001483680       183        6
documentation   0.0001483239       593        7
exception       0.0001480604       142        8
logging         0.0001478415       207        9
dynamic         0.0001478197       223       10
web             0.0001477541       254       11
optimization    0.0001476887       201       12
runtime-error   0.0001474926      1021       13
parameters      0.0001474709       339       14
memory-leaks    0.0001474491       305       15
data            0.0001474274       261       16
user-interface  0.0001473188       185       17
frameworks      0.0001473188       429       18
crash           0.0001473188       506       19
design-patterns 0.0001472971       179       20
> clc.df[ind,] # the 10 most frequent tags:
                    clc freq.rank clc.rank
mysql      0.0001414027        10      778
html       0.0001408848         9      915
jquery     0.0001405284         6     1031
c++        0.0001394117         8     1488
android    0.0001385425         5     1944
python     0.0001382361         7     2128
php        0.0001372684         4     2810
javascript 0.0001366680         3     3222
c#         0.0001355381         1     3601
java       0.0001308729         2     3710
```

Eigenvector-based Centrality

Eigenvector-based centrality measures express the importance of a node based on the importance of its neighbors. Since this definition is somewhat self-referential, these measures turn out to be the solutions of some appropriately defined eigenvalue problem (hence the name) [5, 11]. The relevant `igraph` function is `evcent()`, which has a somewhat different interface than the `betweenness()` and `closeness()` centrality functions already used:

```
> eigc <- evcent(gr) # returns a list, not an array
> eigc <- eigc$vector # coerce to array
> V(gr)$name[which.max(eigc)]
[1] "javascript"
```

Following a similar procedure to the one already described for the betweenness, we check the 20 nodes with the highest eigenvector centrality (notice that this measure is normalized to lie between 0 and 1):

```
> head(eigc.df,20)
                eigc freq.rank
javascript 1.00000000         3
jquery     0.94831551         6
html       0.72171495         9
css        0.51204736        13
```

```
php         0.44673099         4
ajax        0.28340051        22
mysql       0.19656924        10
c#          0.18316147         1
asp.net     0.18129922        12
html5       0.13735764        39
json        0.13219053        28
java        0.10853696         2
forms       0.09822225        46
jquery-ui   0.09133711        84
sql         0.08945679        14
arrays      0.08822174        24
.net        0.08759265        16
android     0.08142784         5
regex       0.07787668        25
css3        0.07391215        83
```

We notice that the results are very different compared to the betweenness and closeness centralities: `javascript` and `jquery` are pronounced as the most important nodes, and the leaderboard is populated by a mix of the most frequent and the less so tags.

12.6 Community Detection

Community detection is both a huge topic and the subject of intense current research: a recent (2010) survey paper in a high-caliber physics journal runs for no less than 100 pages [15], where we read that "*[t]his problem is very hard and not yet satisfactorily solved, despite the huge effort of a large interdisciplinary community of scientists working on it over the past few years*". The subject also goes under some alternative labels, such as *clustering* and *graph partitioning* [11, 15], although it is advised that the former term be avoided in order to prevent confusion with the clustering coefficient measures mentioned already [6].

Intuitively speaking, the problem of community detection is to uncover a somehow "natural" division of a network into groups of nodes, such that there are many edges *within* groups and relatively few edges *between* groups [6]. Like the measures introduced in the first sections of this chapter, it is a *macro* one, aiming at discovering and understanding the large-scale structure of graphs and networks. Although the first algorithms for community detection and graph partitioning in computer science were proposed in the early 1970's, recent research on the topic was greatly triggered by the publication of a seminal paper by Girvan and Newman in 2002 [16], considered as "*historically important, because it marked the beginning of a new era in the field of community detection and opened this topic to physicists*" [15]. Their method focuses on the concept of betweenness centrality introduced already, and it remains highly popular today. It is implemented in `igraph` by the function `edge.betweenness.community()`, but unfortunately it is computationally very demanding, and we will not demonstrate it here. We will focus on two other popular algorithms instead.

Fast Greedy Algorithm

The fast greedy algorithm was proposed in 2004 by Clauset, Newman, and Moore [17]; hence, it is sometimes referred to as the CNM algorithm. It tries to optimize a modularity

measure for the graph [18] by applying, as its name implies, a fast greedy approach. It is available in `igraph` via the `fastgreedy.community()` function:

```
> fc <- fastgreedy.community(gr)
```

All the community detection functions in `igraph` return their results as objects of the dedicated class `communities`, which provides its own special functions and operations for this kind of objects

```
> length(fc) # how many communities
[1] 10
> sizes(fc)
Community sizes
 1   2   3   4   5   6   7   8   9  10
32 514 625 506 976 316 521  36 183   2
```

We can see that the fast greedy algorithm returns 10 communities, with the sizes shown above. Before getting for a closer inspection, we can see that most of the detected communities consist of a rather high number of nodes. To get an idea of the quality of the returned communities, we proceed to visually inspect some of the smaller ones:

```
> V(gr)$name[membership(fc)==1]
[1] "actionscript-3" "flash"          "flex"
[4] "actionscript"   "air"            "flex4"
[7] "salesforce"     "adobe"          "flash-builder"
[10] "flex3"          "components"     "swf"
[13] "actionscript-2" "textfield"      "flash-cs5"
[16] "flex4.5"        "apex-code"      "mxml"
[19] "flash-player"   "flexbuilder"    "movieclip"
[22] "visualforce"    "spark"          "red5"
[25] "flv"            "builder"        "loader"
[28] "flash-cs4"      "blazeds"        "flashdevelop"
[31] "swfobject"      "flash-cs6"

> V(gr)$name[membership(fc)==8]
[1] "excel"                  "vba"
[3] "excel-vba"              "macros"
[5] "google-apps-script"     "ms-word"
[7] "automation"             "export"
[9] "google-drive-sdk"       "ms-office"
[11] "google-spreadsheet"     "range"
[13] "excel-formula"          "excel-2007"
[15] "word"                   "excel-2010"
[17] "powerpoint"             "cell"
[19] "google-drive"           "spreadsheet"
[21] "formula"                "google-docs"
[23] "copy-paste"             "word-vba"
[25] "pivot-table"            "google-apps"
[27] "xls"                    "google-docs-api"
[29] "openoffice.org"         "ole"
[31] "outlook-vba"            "excel-2003"
[33] "shapes"                 "powerpoint-vba"
[35] "google-spreadsheet-api" "vlookup"

> V(gr)$name[membership(fc)==10]
[1] "tridion"      "tridion-2011"
```

All three communities exposed above look meaningful: for example, community #1 seems to be about Adobe Flash and related technologies, whereas community #8 contains mostly tags related to "office" applications (by Microsoft, Google, and Apache). Nevertheless, we find it difficult to imagine that the bigger communities, i.e., of size 300, 500, or even higher, may contain such kind of coherent information that could possibly be labeled by a single term (like "Adobe Flash", or "office applications" for our two previous examples).

Could we possibly do better? The answer is not clear beforehand; recall from the discussion so far that, edge weights aside, our graph is densely connected, with a diameter of only 3. It could very reasonably be the case that, given this "tightness" of our graph, we cannot uncover any finer details regarding distinct coherent communities.

To see that this is not the case, we now turn to a powerful, state of the art algorithm, that at the same time is not computationally prohibitive for a common desktop or laptop computer, at least for the size of our graph.

Infomap Algorithm

The Infomap algorithm for community detection was suggested in 2008 by Rosvall and Bergstrom [19, 20]; according to comparative tests, it *"appears to be the best"*, and it is *"also very fast, with a complexity which is essentially linear in the system size, so [it] can be applied to large systems"* [15]. It is available in `igraph` via the `infomap.community()` function. Since it has a random element, we first set the random seed for exact reproducibility of the results shown here:

```
> set.seed(1234) # for reproducibility
> imc <- infomap.community(gr)
> length(imc)
[1] 80
> plot(sizes(imc), xlab="Community no.", ylab="Community size")
```

Infomap reveals 80 communities, with the corresponding sizes as shown in Figure 12.2.

Despite the fact that here also we have some large communities detected, we have also a rather high number of smaller ones, with sizes ranging from 2 to about 40 nodes each.

In our case, the only way for validating the communities is by "manual inspection", that is, by going through them and examining if the communities are meaningful. We claim that the communities discovered by the Infomap algorithm are indeed highly meaningful; we leave the complete set of this exercise to the reader, and we will illustrate here only some typical cases.

Before examining the large communities, it makes sense to first examine the smaller ones, as the possible (in)coherences will be much easier to spot. Let us begin with the communities #17-19:

```
> V(gr)$name[membership(imc)==17]
[1] "facebook"                "facebook-graph-api"
[3] "comments"                "facebook-like"
[5] "facebook-javascript-sdk" "facebook-fql"
[7] "facebook-php-sdk"        "share"
[9] "facebook-opengraph"      "facebook-c#-sdk"
[11] "opengraph"               "social-networking"
[13] "like"                    "photo"
[15] "facebook-apps"           "facebook-login"
[17] "facebook-android-sdk"    "facebook-ios-sdk"
[19] "access-token"            "facebook-comments"
[21] "sharing"                 "social"
```

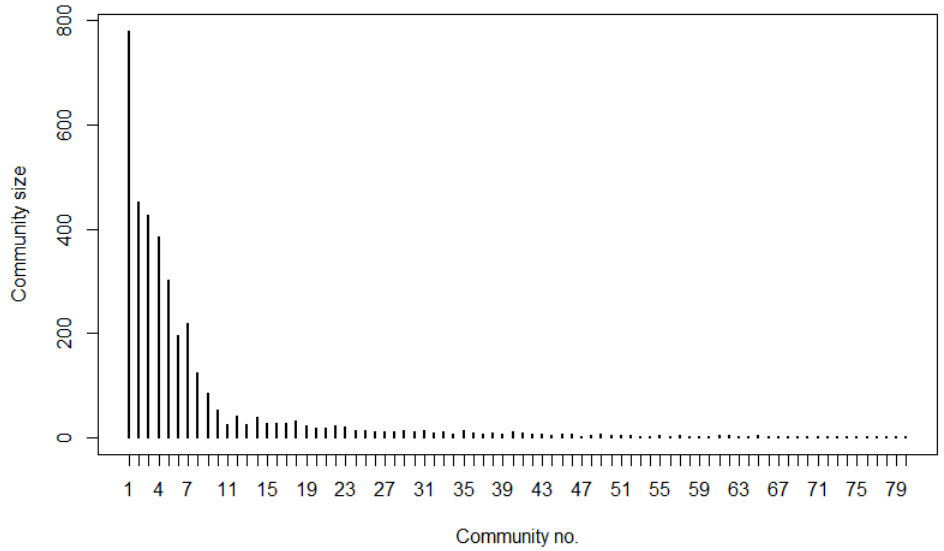

FIGURE 12.2: Communities revealed by Infomap, with corresponding sizes.

```
[23] "facebook-page"           "fbml"
[25] "facebook-oauth"          "facebook-social-plugins"
[27] "facebook-access-token"   "facebook-wall"

> V(gr)$name[membership(imc)==18]
[1] "r"                  "plot"              "ggplot2"
[4] "statistics"         "latex"             "data.frame"
[7] "gnuplot"            "time-series"       "markdown"
[10] "visualization"      "aggregate"         "histogram"
[13] "data.table"         "bar-chart"         "data-visualization"
[16] "regression"         "subset"            "plyr"
[19] "legend"             "knitr"             "stata"
[22] "apply"              "rstudio"           "xts"
[25] "heatmap"            "correlation"       "shiny"
[28] "reshape"            "finance"           "igraph"
[31] "labels"             "lattice"

> V(gr)$name[membership(imc)==19]
[1] "google-maps"          "google-maps-api-3"    "geolocation"
[4] "maps"                 "coordinates"          "google-maps-markers"
[7] "openlayers"           "gis"                  "geocoding"
[10] "kml"                  "polygon"              "latitude-longitude"
[13] "distance"             "geospatial"           "google-places-api"
[16] "openstreetmap"        "postgis"              "google-fusion-tables"
[19] "google-maps-api-2"    "spatial"              "arcgis"
[22] "google-earth"         "infowindow"
```

Community #18 could very easily be labeled as "R": indeed, `plyr`, `ggplot2`, `lattice`, `xts`, `igraph`, `reshape`, and `data.table` are `R` packages; `rstudio` is the dominant `R`

IDE, while `statistics`, `regression`, `visualization`, and `correlation` are perhaps the most typical tasks for which `R` is used; `latex` could be a possible outlier, but we suspect that its presence is due to the increasing use of the LaTex output option, as facilitated by the `knitr` and `shiny` packages in a `markdown` framework; a similar argument could be brought forward also for `stata` (questions regarding porting of Stata tasks from and to `R`); `apply` is an `R` family of functions...

Similarly, it is rather straightforward to assign the labels "Facebook" and "GIS" to the communities # 17 and #19, respectively.

How about discovering "new" knowledge from this community structure? We include an illustrative case, as it occurred to us while writing. Community #33 clearly has to do with install/setup tasks in a Windows environment:

```
> V(gr)$name[membership(imc)==33]
[1] "wiz"               "installation"      "installer"
[4] "windows-installer" "setup"             "inno-setup"
[7] "msi"               "installshield"     "nsis"
[10] "uninstall"         "wix3.5"            "custom-action"
```

Having never heard of `wiz`, we Googled it; it proved to be *"The most powerful set of tools available to create your Windows installation experience"* (see wixtoolset.org).

The list can go on; communities #28 and #35 are devoted to "big data" and "machine learning" respectively, with virtually no outliers at all:

```
> V(gr)$name[membership(imc)==28]
[1] "hadoop"     "mapreduce"  "cassandra"  "hive"       "hbase"
[6] "apache-pig" "hdfs"       "bigdata"    "thrift"     "piglatin"
[11] "cloudera"   "zookeeper"

> V(gr)$name[membership(imc)==35]
[1] "machine-learning"        "nlp"
[3] "artificial-intelligence" "neural-network"
[5] "classification"          "cluster-analysis"
[7] "data-mining"             "weka"
[9] "scikit-learn"            "svm"
[11] "mahout"                  "libsvm"
[13] "k-means"
```

We encourage the reader to google possible unknown terms (`mahout`, perhaps?), in order to confirm that they are indeed relevant with the other members of their respective community.

What about our very small communities? Let us try communities #65 and #69:

```
> V(gr)$name[membership(imc)==65]
[1] "sip"      "asterisk" "voip"     "skype"

> V(gr)$name[membership(imc)==69]
[1] "neo4j"  "cypher"
```

In community #65, `sip` stands for the Session Initiation Protocol (SIP) in `voip` networks (of which `skype` is the most popular example), while `asterisk` is *"an open-source telephony switching and private branch exchange service"* (see Asterisk.org). Regarding community #69, `neo4j` is perhaps the most widely used graph database, and `cypher` is its dedicated query language [21].

As expected, the situation is not so clear-cut with the larger communities; nevertheless, even in those cases, there is certainly some nontrivial coherence in the communities —

for example, there are different and distinct communities for iOS (#6) and Android (#7); almost each and every one of them are associated with a major programming language or framework. We can easily take a compact glimpse at the first 4 node tags of each of the first 10 communities with the following `for` loop:

```
for (i in 1:10) {
     print(V(gr)$name[membership(imc)==i][1:4])
}

[1] "c++"  "c"  "arrays"  "linux"
[1] "javascript"  "jquery"  "html"  "css"
[1] "php"  "mysql"  "sql"  "sql-server"
[1] "c#"  "asp.net"  ".net"  "asp.net-mvc"
[1] "java"  "eclipse"  "spring"  "swing"
[1] "ios"  "iphone"  "objective-c"  "xcode"
[1] "android"  "sqlite"  "android-layout"  "listview"
[1] "python"  "django"  "list"  "google-app-engine"
[1] "ruby-on-rails"  "ruby"  "ruby-on-rails-3"  "activerecord"
[1] "git"  "svn"  "version-control"  "github"
```

We think the reader will agree that a certain coherence is already visible from the above list. And as we gradually move towards smaller communities, the picture becomes even more coherent. Here is a similar glimpse of the communities #11-20:

```
for (i in 11:20) {
     print(V(gr)$name[membership(imc)==i][1:4])
}

[1] "xml"  "parsing"  "xslt"  "xpath"
[1] "excel"  "vba"  "ms-access"  "excel-vba"
[1] "actionscript-3"  "flash"  "flex"  "actionscript"
[1] "opengl"  "graphics"  "opengl-es"  "3d"
[1] "node.js"  "mongodb"  "express"  "websocket"
[1] "unit-testing"  "testing"  "selenium"  "automation"
[1] "facebook"  "facebook-graph-api"  "comments"   "facebook-like"
[1] "r"  "plot"  "ggplot2"  "statistics"
[1] "google-maps"  "google-maps-api-3"  "geolocation"  "maps"
[1] "unicode"  "encoding"  "utf-8"  "character-encoding"
```

The specific setting of the random seed has no effect on the qualitative aspects of the result: the reader is encouraged to perform the previous experiment with several different random seeds, to confirm that the results are indeed robust and stable, although the exact number of the uncovered communities may differ slightly.

12.7 Visualization

It is a common truth between graph researchers and practitioners that straightforward visualizations for graphs with more than 100 nodes are of very little use, and when we move to even a few thousand nodes, as with our reduced graph, visualization ceases to have any meaningful value [11]. Nevertheless, there are still useful and meaningful ways to visually explore such graphs. Two common approaches are [11]:

- To "coarsen" the graph, by merging several nodes together, possibly exploiting the results of an already existing graph partitioning (macro level)
- To highlight the structure local to one or more given nodes, resulting in the so called *egocentric* visualizations, commonly used in social networks analysis (micro level)

We are now going to demonstrate both these approaches with our graph.

Visualizing the Communities Graph

For the first approach, we are going to further exploit here the partitioning into communities provided by the Infomap algorithm from the previous Section, effectively resulting in a dimensionality reduction for our graph, in order to produce a useful visualization. As we will see, `igraph` provides several convenient functions for such purposes. We demonstrate first the use of the `contract.vertices()` function, which will merge the nodes according to the community to which they belong:

```
> # first, add "community" attribute to each node
> gr <- set.vertex.attribute(gr, "community",
+   value = imc$membership)
> grc <- contract.vertices(gr, V(gr)$community)
> grc <- remove.vertex.attribute(grc, "name")
> for (i in 1:length(V(grc))) {
+   grc <- set.vertex.attribute(grc, "name", index=i,
+      V(gr)$name[membership(imc)==i][1])
+ }
> grc <- set.graph.attribute(grc, "name",
+        value = "Contracted (communities) graph")
> summary(grc)
IGRAPH UNW- 80 1033957 -- Contracted (communities) graph
attr: name (g/c), name (v/c), weight (e/n)
```

The `for` loop aims to give to each node in our contracted (communities) graph the name of the first member of the respective community (usually the community "label"), which will be subsequently used for the graph visualization. From the summary, we can see that we now have only 80 nodes (recall from the previous Section that this is the number of the communities uncovered by the Infomap algorithm), but we still carry all the 1 million edges from the `gr` graph. That is because, as its name may imply, the `contract.vertices()` function does not affect the edges of the graph. We can further simplify the contracted graph by merging also the edges, summing up the corresponding weights, utilizing the `simplify()` function. This is an important point for visualizing, as one can discover by trying to plot the graph as it is so far, with more than 1 million edges.

```
> grc <- simplify(grc, edge.attr.comb = "sum") # sum edge weights
> grc <- set.graph.attribute(grc, "name",
+    value = "Contracted & simplified (communities) graph")
> summary(grc)
IGRAPH UNW- 80 2391 -- Contracted & simplified (communities) graph
attr: name (g/c), name (v/c), weight (e/n)
```

Thus, we have ended up with a rather simple graph of 80 vertices and only 2391 edges, which should not be hard to visualize. We invoke the `tkplot()` command (the `R` package `tcltk` needs to be installed, but once installed `igraph` will load it automatically). The results are shown in Figure 12.3 (for best results, maximize the screen and choose some layout other than "Random" or "Circle" from the figure menu).

```
> tkplot(grc)
Loading required package: tcltk
[1] 1
```

Although our communities graph is still very dense, we can visually distinguish the central nodes from the peripheral ones, and the visualization is in line with what we might expect: nodes representing highly used tools, environments, and tasks are positioned in the central bulk, while nodes representing less widely used tools are pushed towards the graph periphery. We notice that the visualization provided by the `tkplot()` function is interactive, for example, one can move and highlight nodes and edges or change their display properties etc.

We now turn to the second approach mentioned in the beginning of this Section, i.e., the so-called egocentric visualizations focused on individual nodes.

Egocentric Visualizations

We can use egocentric visualizations in order to focus closer on our uncovered communities, keeping in mind the informal visualization rule of thumb mentioned before, i.e., that we should try to keep the number of our visualized nodes under about 100. The following code produces the subgraph of the "R" community (#18) discussed already:

```
> k.name <- "r" # the name of the node of which the community we want to examine
> k.community <- imc$membership[which(V(gr)$name==k.name)]
> k.community.graph <- induced.subgraph(gr, (V(gr)$community==k.community))
> k.community.graph$name <- paste(k.name, "community graph")
> k.community.graph
IGRAPH UNW- 32 311 -- r community graph
+ attr: name (g/c), name (v/c), community (v/n), weight (e/n)
```

As expected, we end up with a 32-node graph, which should not be hard to visualize. Although not clearly documented in the `igraph` package manual, it turns out that we can indeed use an `edge.width` argument in the `tkplot()` function, in order to visualize the graph edges proportionally to their weight. Experimenting with the corresponding coefficient (invoking the edge weight as-is produces a graph completely "shadowed" by the edges), we get:

```
> tkplot(k.community.graph, edge.width=0.05*E(k.community.graph)$weight)
```

with the results shown in Figure 12.4.

From Figure 12.4, we can immediately conclude that the vast majority of R-related questions in our data have to do with plotting (and the `ggplot2` package in particular), as well as with the specificities of the `data.frame` structure. Also of notice is the rather strong presence of the relatively new `data.table` package.

We can go a step further, and remove the `r` node itself from the plot; that way we expect the edge width visualization to be less dominated by the presence of the "central" node (`r` in our case), and possibly uncover finer details regarding the structure of the particular community. To do that, we only need to modify the `induced.subgraph()` line in the previous code as follows:

```
> k.community.graph <- induced.subgraph(gr,
+ (V(gr)$community==k.community & V(gr)$name != k.name))
> tkplot(k.community.graph, edge.width=0.05*E(k.community.graph)$weight)
```

The results are shown in Figure 12.5. We can confirm, for example, that the presence of `latex` in the "R" community is indeed not spurious, as the subject node is strongly connected with both `knitr` and `markdown`, although our initial speculation for its relation also to `shiny` turns out to be incorrect.

We can easily extend the above rationale in order to examine closer the relationship between two or more communities, as long as we limit the investigation to relatively small communities (remember our 100-node max rule of thumb for visualizations). Say we would like to see how the "Big Data" and "Machine Learning" communities are connected, excluding these two terms themselves:

```
> k.names <- c("bigdata", "machine-learning")
> k.communities <- imc$membership[which(V(gr)$name %in% k.names)]
> k.communities.graph <- induced.subgraph(gr,
+ (V(gr)$community %in% k.communities & !(V(gr)$name %in% k.names)))
> tkplot(k.communities.graph, edge.width=0.05*E(k.communities.graph)$weight)
```

The results in Figure 12.6 show that, apart from the mildly strong connections of `mahout` with both `cluster-analysis` and `k-means`, the two communities are practically strangers.

12.8 Discussion

We have provided a brief demonstration of how graph and network approaches and tools can be applied for the analysis of semi-structured text in the form of tags. In summary, the steps we have undertaken in the present chapter are as follows:

- Starting from a raw XML file of about 26 GB in size, we parsed the data and built a graph structure of about 2.7 million edges for the co-occurrences of all 36,942 tags.
- We made an informed reduction of our data, in order to end up with a graph that can be feasibly analyzed using only commodity hardware; in doing so, we retained 90% of our 21 million tag occurrences while ending up with a much more manageable reduced graph of 3711 nodes and about 1 million edges.
- Despite the fact that the relevant metrics (diameter = 3, average path length = 1.85) suggested a very dense and tightly packed structure for our reduced graph, we successfully employed the state of the art Infomap algorithm for community detection, in order to uncover 80 distinct communities that, upon close inspection, look indeed meaningful and coherent.
- Utilizing the detected communities in what stands effectively as a **dimensionality reduction** operation on our reduced graph, we managed to end up with a coarsened graph of only 80 nodes and about 2,500 edges, which could subsequently be visualized and used for further exploratory data analysis.
- Finally, we exploited further the uncovered community structure of our graph, in order to demonstrate how further micro-analysis in the neighborhood of selected nodes is possible and potentially insightful.

There are certainly different choices than can be made in several key points of our analysis; for example, we could have pruned our initial graph not only according to the frequency of

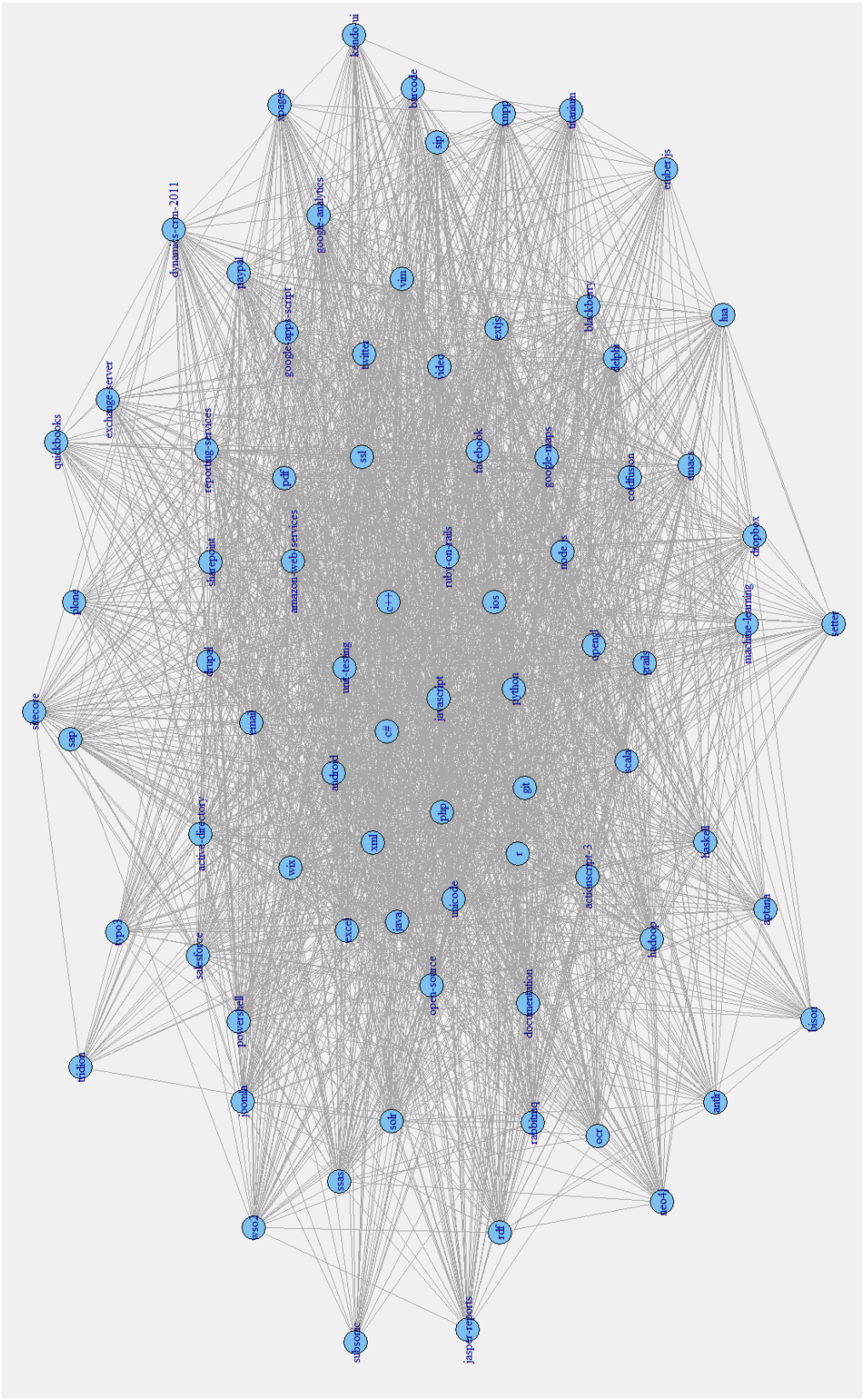

FIGURE 12.3: Visualization of our communities graph.

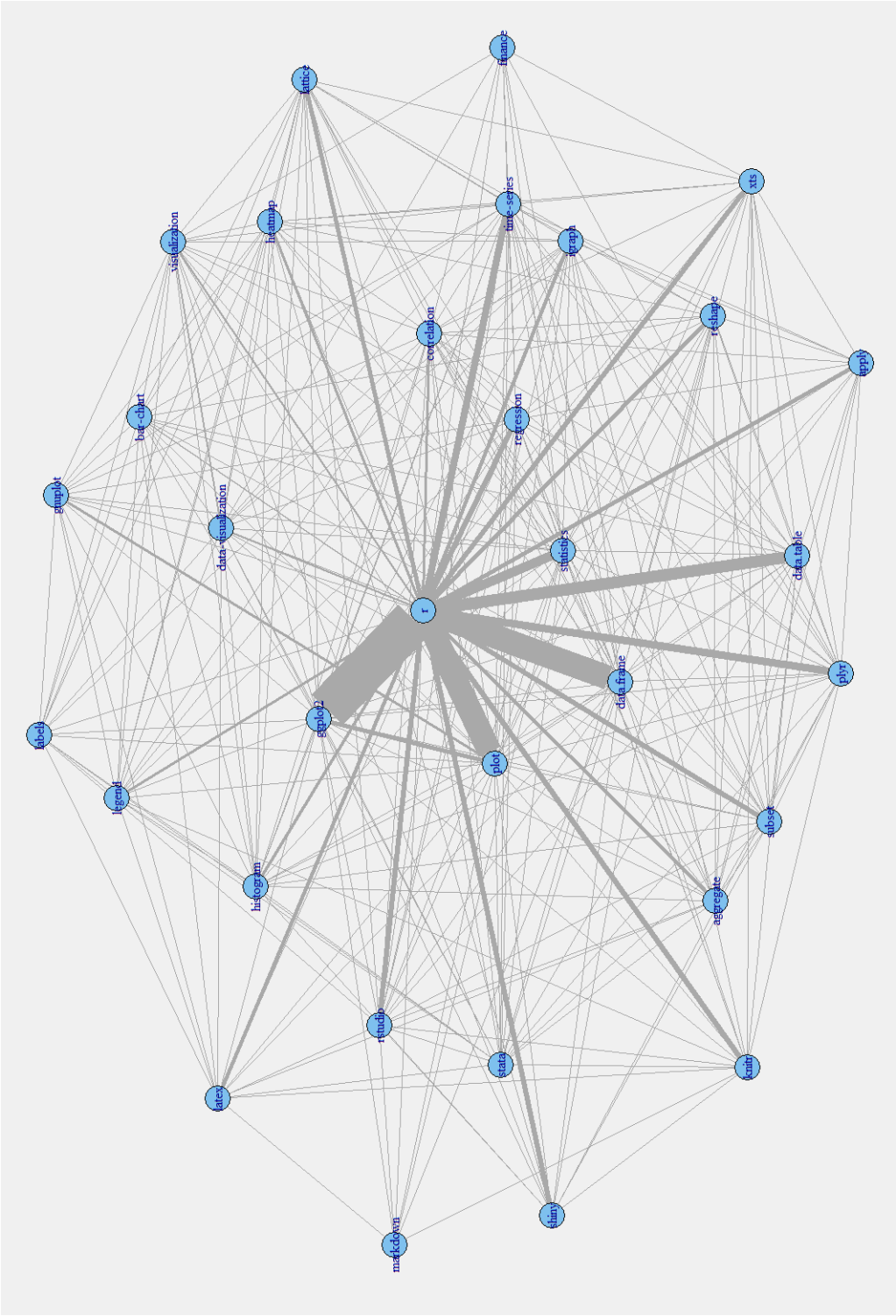

FIGURE 12.4: Visualization of the "R" community.

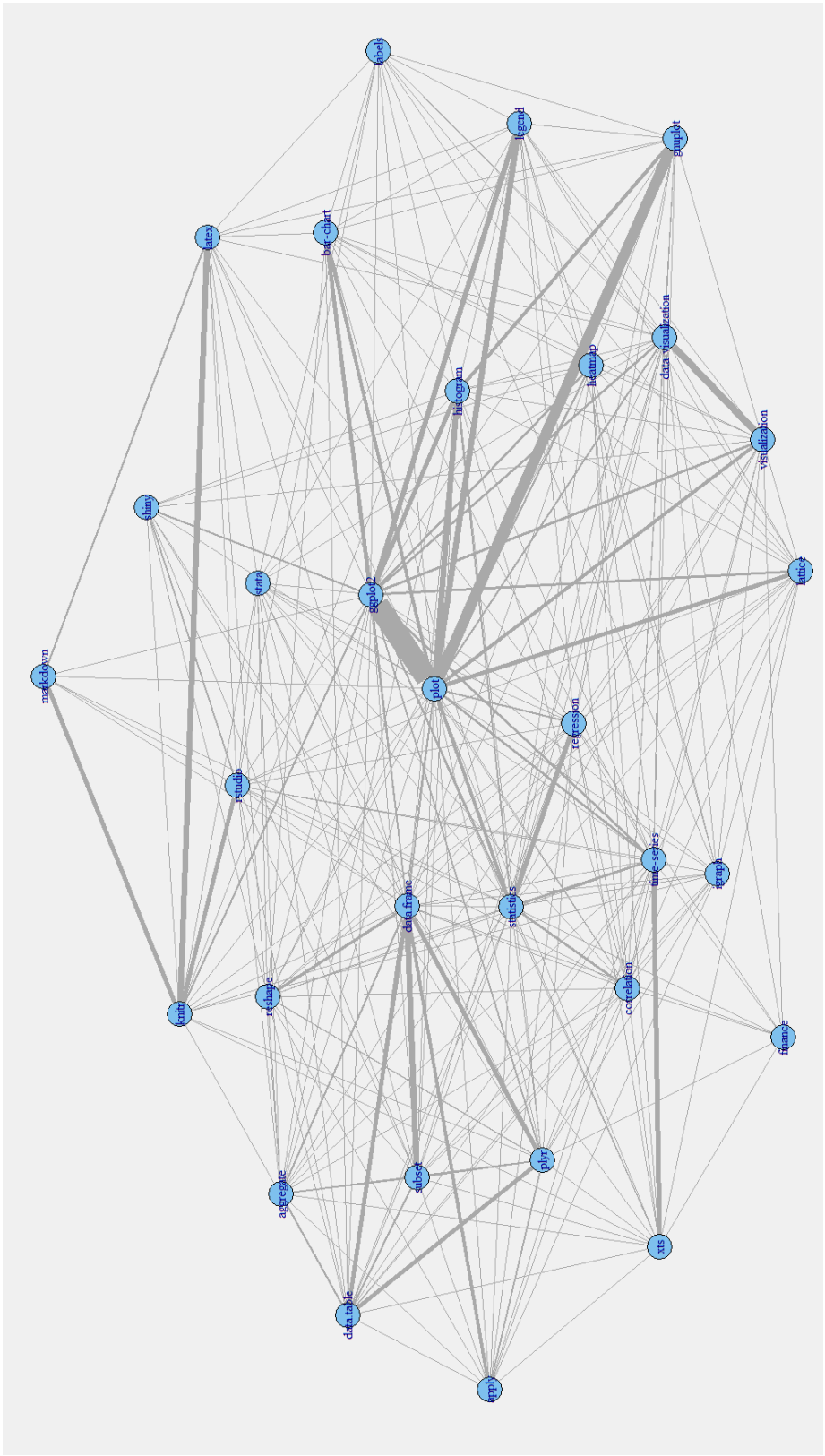

FIGURE 12.5: The "R" community, with the `r` node itself removed.

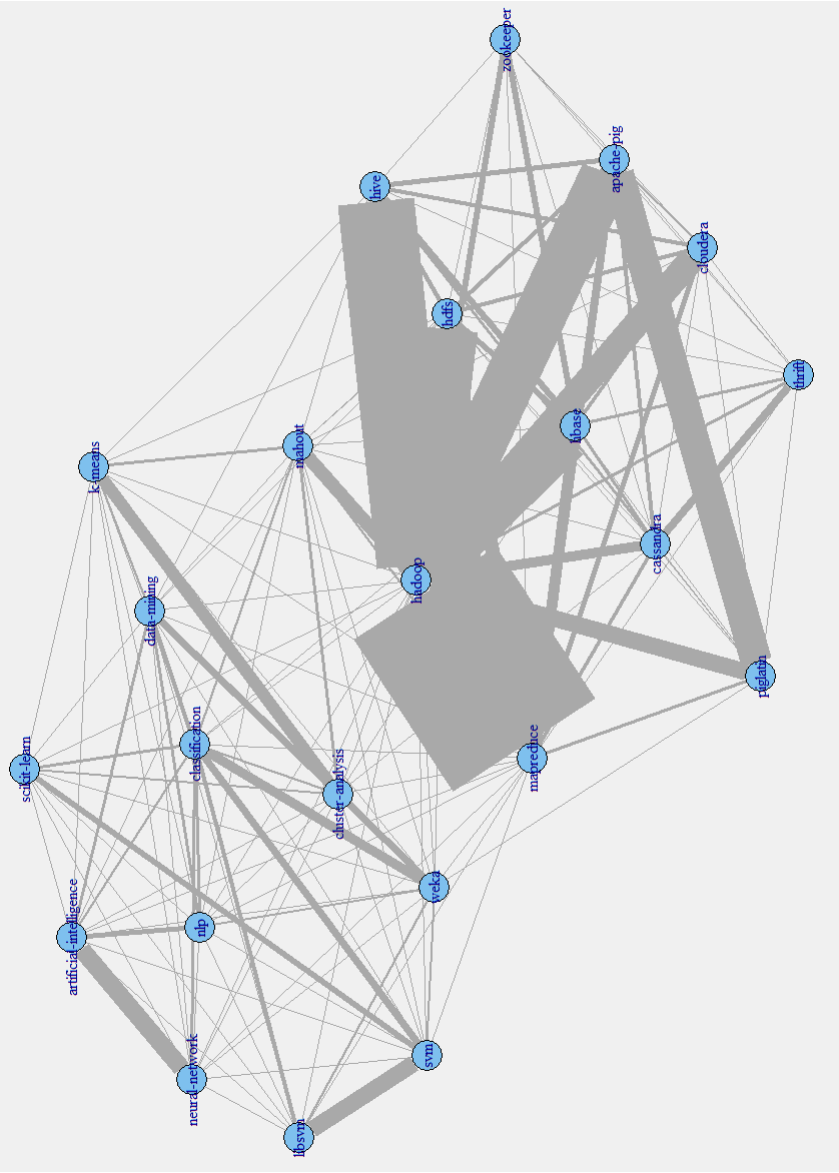

FIGURE 12.6: The "Big Data" and "Machine Learning" communities (excluding these terms themselves).

the corresponding tags, but also according to the edge weights (which are also based on a frequency measure). That way, we would have ended with a more clustered and less densely connected graph, with a higher enough diameter and average path length. We chose not to do so mainly in order to fully demonstrate the capabilities of the Infomap algorithm even in such a dense graph, since the community detection is definitely easier for a graph that possesses already some clustering on its own. With the step-by-step instructions and code provided, the reader can easily explore this direction by herself.

We hope that we have presented a convincing case regarding the possible merits of the graph representation, whenever appropriate, and of the handy tools provided by the R language and the `igraph` library. Given the increasing presence of such representations in modern data analysis, we certainly consider that a basic familiarity with the relevant tools and concepts is a must-have skill for the practitioner.

12.9 Appendix: Data Acquisition & Parsing

Here we describe how to obtain the original data and how to parse them, in order to end up with the two R workspace files, `tag_freq.RData` and `adj_matrix.RData`, which are necessary for the analysis described in the chapter's main body.

Data Acquisition

The whole Stack Exchange dump is available from the Internet Archive[6] in `7z` format. The files necessary for our analysis are:

- `stackoverflow.com-Posts.7z` (about 5 GB, which contains the 26 GB uncompressed file `Posts.xml` for Stack Overflow)
- `stackoverflow.com-Tags.7z` (contains the `Tags.xml` file, with the tags and their frequencies)
- `readme.txt` (with various information useful for parsing the data)

Parsing the Tag List & Frequencies

Nowadays, R offers extensive functionality for working with XML files [22], including the `XML` package. After a quick inspection of the XML structure in `Tags.xml`, it is almost trivial to parse the file and get the individual tag names with their corresponding frequencies (notice that the `Tags.xml` file needs to be in the current R working directory, and the `XML` package needs to be installed):

```
> library(XML)
> f <- 'Tags.xml'
> doc <- xmlParse(f)
> tag <- xpathSApply(doc, '//row', xpathSApply, '@TagName')
> freq <- as.integer(xpathSApply(doc, '//row', xpathSApply, '@Count'))
```

Subsequently, we just combine the variables `tag` and `freq` in a data frame, we sort it in decreasing frequency order, and we save the resulting data frame in an R workspace file:

[6] https://archive.org/details/stackexchange.

```
> tag_freq <- data.frame(tag, freq,
  + stringsAsFactors=FALSE, row.names = NULL)
> # sort in descending frequency:
> tag_freq <- tag_freq[order(tag_freq$freq, decreasing=TRUE),]
> # save in R workspace file:
> save(tag_freq, file='tag_freq.RData')
```

File `tag_freq.RData` should now be in R's current working directory.

Parsing the Tag Co-occurences

In order to parse the tag co-occurrences present in the file `Posts.xml`, we need to download and install BaseX[7], a free open-source XML database. Installation is straightforward, and so is the creation of a new database from `Posts.xml` (although it can take a while); the resulting database is about 19 GB in size.

`Posts.xml` contains all the information about posts, including the answers. From the provided `readme` file, we can see that `PostTypeID=1` refers to questions, while `PostTypeID=2` refers to answers. Since tags are included only in the questions, we are not interested in the latter. After some experimenting in order to arrive to the desired output format (one line of tags per post, with proper separation), we end up with the following simple query (file `get_tags.xqy`):

```
for $x in //row[@PostTypeId="1"]
return ($x/@Tags/string(),'&#xa;')
```

Pasting the above code to the BaseX query window (double-clicking the provided script file also opens the query to a new instance of the BaseX GUI) and running it, we get a sample of the desired output displayed in the relevant window of the BaseX GUI. Although the report window indicates that only 250,000 hits are returned, saving the output results indeed to a text file with the tag co-occurrences of all 7,214,697 posts included in `Posts.xml`.

Building the Adjacency Matrix

Once we have the tag co-occurrences in a text file, we can proceed to build the graph adjacency matrix. It is not exactly a straightforward job, and we will not describe it here in detail - interested readers can always consult the provided script `parse_posts.R`. Here we highlight some challenges that needed to be addressed, using them also as a demonstration of good programming approaches in R.

In theory, with enough memory available (at least 8 GB), it is possible to load the whole `post_tags.txt` file produced in the previous step, and parse it to directly build the (sparse) graph adjacency matrix `G`; unfortunately, it turns out that in practice one encounters the following problem: the sparse matrix `G`, as it gets populated, it gets increasingly "heavier" for incremental updates, leading to unacceptably high execution times, even in machines with powerful processors and lots of memory.

The solution we applied to overcome this issue was twofold: first, we read the `post_tags.txt` file via a file connection, taking in only a bunch of rows at a time in an external `while` loop; second, in each loop iteration, we initialized an all-zeros temporary co-location matrix `G_temp`, which was the one being incrementally updated, and was added to the adjacency matrix `G` only at the end of each loop iteration. This approach speeded up things significantly, since `G_temp` never grew up to sizes that slowed down its

[7] http://basex.org/.

incremental updates, while the matrix addition of `G` and `G_temp` proved to be of practically no computational cost.

In our setting, it turned out that a value of 1000 rows per iteration was optimal, leading to 7215 `while` loop iterations; users with slower hard discs may wish to experiment with these settings, since there is a disk read operation in each loop iteration. The resulting solution also has the merit that it is usable to machines with low memory, since the number of rows read at each loop iteration can be as low as desired if the value of 1000 we used turns out to be prohibitive with the memory available.

Been forced to use an external `while` loop for the reasons just mentioned, we aimed to completely avoid any `for` loops in the parsing procedure, since they are notoriously slow in R (see "The Dreaded for Loop" in [23]). We thus made extensive use of the `lapply` family of R functions. Novice R users usually find `lapply` and its relatives rather hard to master initially, but once comprehended, they provide an excellent set of tools for writing efficient R code.

Utilization of the `lapply` functions is also important for one more reason, leading naturally the discussion towards the third ingredient we used for speeding up our code: parallelization, which is rather trivial nowadays in R, even in Windows systems (notice that the `parallel` package is now included by default in any R distribution). Indeed, incorporating parallelization in a multicore machine is almost as easy as the following code snippet indicates:

```
library(parallel)
load("tag_freq.RData")
# initialize parallelization:
num_cores <- 4
cl <- makeCluster(num_cores)
clusterExport(cl, varlist="tag_freq")
```

Now, if we have already put the effort to use `lapply` functions in our code instead of `for` loops, converting them in order to effectively use the existing parallelizatiuon backend is trivial: just compare the single-threaded version shown below, where we build a temporary variable by applying our function `getTags` to the raw file input `doc`

```
temp <- apply(as.matrix(doc), 1, getTags)
```

to its parallel version

```
temp <- parApply(cl, as.matrix(doc), 1, getTags)
```

On the contrary, and despite the functionality available in the `foreach` package, parallelizing `for` loops is by no means that easy and straightforward.

Bibliography

[1] A.-L. Barabási. *Network Science*. 2014. `http://barabasilab.neu.edu/networksciencebook/`.

[2] D.J. Watts and S.H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684):440–442, 1998.

[3] A.-L. Barabási and Réka A. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] Easley D. and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, New York, NY, 2010.

[5] M.O. Jackson. *Social and Economic Networks*. Princeton University Press, Princeton, NJ, 2008.

[6] M. Newman. *Networks: An Introduction*. Oxford University Press, Inc., New York, NY, 2010.

[7] E.D. Kolaczyk. *Statistical Analysis of Network Data: Methods and Models*. Springer, 2009.

[8] D.J. Watts. *Six Degrees: The Science of a Connected Age*. W.W. Norton & Company, 2004.

[9] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[10] G. Csárdi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[11] E.D. Kolaczyk and G. Csárdi. *Statistical Analysis of Network Data with R*. Springer, 2014.

[12] D. Bates and M. Maechler. *Matrix: Sparse and Dense Matrix Classes and Methods*, 2014. R package version 1.1-4.

[13] A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani. The architecture of complex weighted networks. *Proceedings of the National Academy of Sciences of the USA*, 101(11):3747–3752, 2004.

[14] L.C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):3–415, 1977.

[15] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[16] M. Girvan and M.E.J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the USA*, 99(12):7821–7826, 2002.

[17] A. Clauset, M.E.J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70:066111, 2004.

[18] M.E.J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[19] M. Rosvall and C.T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences of the USA*, 105(4):1118–1123, 2008.

[20] M. Rosvall, D. Axelsson, and C.T. Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, 2009.

[21] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

[22] D. Nolan and D.T. Lang. *XML and Web Technologies for Data Sciences with R.* Springer, 2014.

[23] Norman Matloff. *The Art of R Programming.* No Starch Press, 2011.