The Journal of Systems & Software 170 (2020) 110744

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A complex network analysis of the Comprehensive R Archive Network (CRAN) package ecosystem

Marçal Mora-Cantallops [*], Salvador Sánchez-Alonso, Elena García-Barriocanal

Universidad de Alcalá, Spain

ARTICLE INFO

Article history:
Received 8 June 2020
Accepted 13 July 2020
Available online 18 July 2020

Keywords:
CRAN
Complex network analysis
Package ecosystems
R

ABSTRACT

Free and open source software package ecosystems have existed for a long time and are among the most sophisticated human-made systems. One of the oldest and most popular software package ecosystems is CRAN, the repository of packages of the statistical language R, which is also one of the most popular environments for statistical computing nowadays. CRAN stores a large number of packages that are updated regularly and depend on a number of other packages in a complex graph of relations; such graph is empirically studied from the perspective of complex network analysis (CNA) in the current article, showing how network theory and measures proposed by previous work can help profiling the ecosystem and detecting strengths, good practices and potential risks in three perspectives: macroscopic properties of the ecosystem (structure and complexity of the network), microscopic properties of individual packages (represented as nodes), and modular properties (community detection). Results show how complex network analysis tools can be used to assess a package ecosystem and, in particular, that of CRAN.

1. Introduction

The surge in open source software (OSS) development has resulted in abundant available software packages that, in each particular software ecosystem, can be used by developers as building blocks for new projects, reducing development costs and time (Mohagheghi and Conradi, 2007) and which can contribute with a positive and significant value-added return (Nagle, 2019). In a recent report, the European Commission report estimated that using free/libre and open source software (FLOSS) saves the European economy roughly €114 billion per year directly and up to €399 billion per year overall (Harutyunyan, 2020). But, on the other hand, such third-party libraries introduce both direct dependencies and transitive dependencies that need to be kept updated to prevent vulnerabilities and bug propagation that might endanger the whole ecosystem (Cox et al., 2015). Although developers can have a clear vision of the direct dependencies they add to their packages, transitive dependencies might be less clear as they are not included by them, becoming hidden one or multiple levels below the direct dependency. Even the common action of updating packages entails risks, as changes might break existing functionalities on other packages (Raemaekers et al., 2014).

One of the oldest and most popular software package ecosystems is CRAN, the repository of packages of the statistical language R. The R programming language is widely used among statisticians and data miners for developing statistical and data analysis libraries, while also being one of the most popular languages among data scientists thanks to its flexibility and expansion capabilities, as R can be extended through user-created packages. As of March 2020, it ranks 11th in the TIOBE index (https://www.tiobe.com/tiobe-index/), a measure of popularity of general purpose programming languages. The Comprehensive R Archive Network (CRAN) (https://cran.r-project.org/) is a network of web servers around the world where R source code, R manuals, documentation, and contributed packages can be found, and it can be considered as the official repository, containing the largest collection of available R packages. At the end of 2019, it hosted a total of 15.368 packages.

As is common in these environments, developers of many software applications or packages rely on using other OSS packages; such dependencies manifest in different forms. In some cases, packages or applications might need the source code of another package or class to compile correctly. In other cases, such as CRAN, source-code dependencies do not exist; binary-level library sharing is required for many package to function properly. Such dependencies might be shared among many projects and repositories, although in CRAN this is limited to its own repository except for a few and specific packages that are stored in Bioconductor. A package management system serves the purpose

[*] Corresponding author.
E-mail addresses: marcal.mora@uah.es (M. Mora-Cantallops), salvador.sanchez@uah.es (S. Sánchez-Alonso), elena.garciab@uah.es (E. García-Barriocanal).

https://doi.org/10.1016/j.jss.2020.110744
0164-1212/© 2020 Elsevier Inc. All rights reserved.

of managing such dependencies, which is important for both functioning and maintenance (e.g., automated updating) of software packages. German et al. (2013) conducted an exploratory empirical study on the evolution of the R software ecosystem, and showed how R was "a flourishing ecosystem of user-contributed packages" that was growing and contained a "strong set of core packages". Among their observations, they found packages to be typically well-maintained.

Open-source software ecosystems such as CRAN could be considered as very complex networks of artifacts, due to the increase in collaborative development under the open source software paradigm in the last two decades. This enables us to study software package ecosystems from the perspective of complex network analysis (CNA) to analyze its structural, individual and modular characteristics, but also to detect potential risks and vulnerabilities in the network formed by such packages.

Overall, the current analysis aims to demonstrate how complex network analysis techniques can be applied to a OSS package ecosystems (such as CRAN) after building its dependency network, and how the results reflect its scale-free and small-world behavior, the potential vulnerability of some of its packages and the modular structure that is hidden behind the dependency network.

The remainder of this article is structured as follows. Section 2 discusses related work. Section 3 includes details on the data source and the network construction. Section 4 addresses the results of the complex network analysis on CRAN. Section 5 adds perspective to the results, linking them to the related work and discussing this work's contributions. Section 6 concludes, with an additional mention to threats to validity and future lines of work in Section 7.

2. Background

The networks formed by real-world systems in many disciplines (e.g. sociology, biology or computer science, among many others) have been proven to exhibit complex network properties, such as being scale-free (Barabási, 2009), following a power law distribution and obeying the small world principles (Watts and Strogatz, 1998). In the past decade, the growth in the body of the literature studying complex networks is remarkable in a wide variety of fields. Some examples include politics (Halberstam and Knight, 2016; Mora-Cantallops et al., 2019), medicine (Rubinov and Sporns, 2010), economics (Cerina et al., 2015), trade networks (Kasakawa et al., 2016), transport (Feng et al., 2017), or even online gaming (Mora-Cantallops and Sicilia, 2018). Software systems, represent another important subset of systems that can benefit from complex network analysis (Myers, 2003).

Earlier work has revealed that class dependency graphs of individual open source software systems exhibit complex network characteristics, not only in their scale-free degree distributions and the existence of small-world phenomena, but also in their community structure, as Šubelj and Bajec (2011) empirically confirmed in their study of several networks constructed from Java and various third party libraries. To do so, they built class dependency networks, where nodes represent software classes and edges represent dependencies among them. Pan et al. (2011) used complex network theory as a tool to analyze the evolution of object-oriented software from a multi-granularity perspective. Besides other relevant findings, they also highlighted how complex network techniques provided "a different dimension to our understanding of software evolution and also are very useful for the design and development of object-oriented software systems". In the same line of work, Chong and Lee (2015) enhanced this technique by assigning weights to the edges of the network "to denote the strength of communicational cohesion between a pair of related software components", all in order to capture its structural characteristics and to enable a maintainability and reliability analysis. They found that this technique made pattern identification easier and that it was also possible to identify software components that violated common software design principles. This idea of using complex network analysis (via dependency graphs) was also applied by Zimmermann and Nagappan (2008) to evaluate Windows Server 2003; their models improved the performance of models build only by complexity metrics (without network metrics) by ten points and were able to identify twice as many critical binaries. Li et al. (2018) recently noted how "traditional software reliability evaluation approaches lack the analysis inter-component interactions of component-based software systems" and proposed a reliability evaluation model for such systems based on complex network analysis.

This software systems approach has been extended to software package networks over the last decade. Zheng et al. (2008), for example, recognized how "understanding the structure of software systems can provide useful insights into software engineering efforts and can potentially help the development of complex system models applicable to other domains". To prove their hypothesis, they empirically analyzed the package ecosystem of the Gentoo Linux distribution, modeling software packages as nodes and their dependencies as nodes, and developed two growth models for the network. In their future work, they stated that to "study a number of open-source software systems beyond Gentoo Linux [...] could lead to fruitful research contributions". One of such contributions, by Fortuna et al. (2011), compiled all packages and dependencies/conflicts from the Debian/GNU operating system per each major stable release and discussed the parallelisms between its evolution and dynamics over the first 10 releases with that of ecological webs of interacting species, demonstrating the interdisciplinary nature of the CNA toolset. Abate et al. (2009) introduced novel notions on dependencies and sensitivity (related to how critical a component is); the main applications for these metrics were "tools for quality assurance in large component ecosystems and upgrade risk evaluation" that they applied to the Debian package ecosystem; among other findings, they found Debian to be "a small world".

Cataldo et al. (2014), in the editorial of a topical issue devoted to the complex network perspective on software engineering, stated that network-based methods can be utilized to study research questions relevant to empirical software engineering. Zheng et al. (2008) wrote, a few years prior, that the reasons behind the lack of CNA studies on software engineering were "the difficulties with data collection and the lack of applicable models". Cataldo et al. (2014) recognized that, with more massive data sets from platforms such as SourceForge or GitHub (to which many other repositories could also be added), it would mark "the beginning of a fruitful field of research".

R and CRAN have also been the objects of research by a few studies from the perspective of their packages. Decan et al. (2016) made use of the dependency network to compute dependencies in their study and comparison of three different ecosystems (R's CRAN archive network, Python's PyPI distribution, and JavaScript's NPM package manager), but did not follow a complex network analysis perspective. In the future work of a later article, however, Decan et al. (2019) hinted at the how dependency networks of open source packaging ecosystems also reveal complex network behavior and that "it would be worthwhile to study [...] the complex network properties of ecosystem package dependency network". It is also worth adding, although this perspective will not be used in the current article, that complex (and social) network analysis can also be useful to assess the contributor networks and communities that take part in the ecosystem, as modeled by Korkmaz et al. (2018).

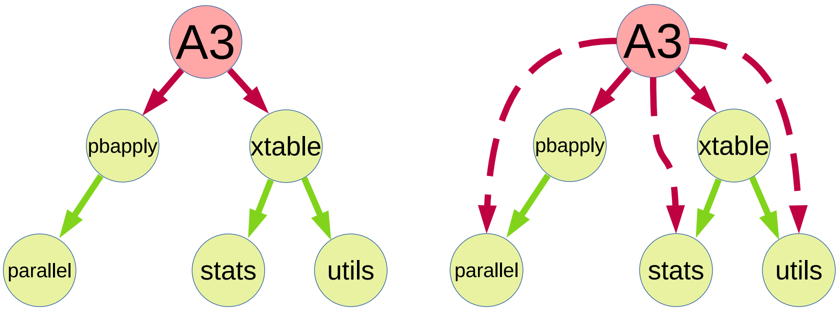


Fig. 1. Dependency graph of the 'A3' package (left) compared to its transitive closure (right).

This work, thus, attempts to understand whether CNA can be applied to analyze a software ecosystem such as CRAN, pointing at additional analytical tools and opening new possibilities for developers and software engineers in general when assessing package dependency networks, their structure or their quality.

3. Materials and methods

3.1. Data extraction

The extraction was executed using R and the "pkgsearch" package (Csárdi and Salmon, 2019), which uses the 'R-hub' search server (see https://r-pkg.org) and the CRAN metadata database to provide detailed information about CRAN packages. The extracted metadata per package includes the following key elements among others:

- Descriptive features, such as name of the package, description or version.
- Author(s).
- Imports: dependencies that are required for the package to work.
- Suggests: packages that can be used by the package but that are not required.
- Depends: currently states the version of R required by the package, but it is relevant as before R 2.14.0 this field contained the dependencies to other packages (therefore, it was equivalent to imports).
- Date/Publication: time stamp with the date of publication of the particular version of the package.

The extraction results in a total of 15.368 unique packages as of the 31st of December 2019. 148 additional external packages come from dependencies that are stored in another related repository, Bioconductor (an open source and open development software project for the analysis and comprehension of genomic data). Thus, a total 15.516 packages are considered.

Although the extraction using the "pkgsearch" package covers most of the features needed for the following analysis, data was manually cross-checked to ensure the reliability of the package and the obtained information was complemented with the information directly scraped from the CRAN web repository at https://cran.r-project.org. The number of packages obtained using this method is exactly the same and the results were positive, so we can depart from the assumption that the information obtained from "pkgsearch" is reliable.

3.2. Network construction

Inspired by the same principles as previous works on class dependency networks (Šubelj and Bajec, 2011, 2012; Chong and Lee, 2015) and following the approach used in similar research on package ecosystems or open-source software systems, the CRAN network will be represented by the packages as nodes and their dependencies as edges (Zheng et al., 2008; Kikas et al., 2017). Note that prior to the rollout of namespaces in R 2.14.0, the metadata field "Depends" was the only way to reflect dependencies on another package. After that, developers are expected to use the field "Imports" instead; to both account for older and newer packages, and possible inappropriate labeling in the metadata, both fields are combined to obtain the complete dependencies.

Table 1
Properties of the CRAN package dependency network.

Network	Nodes (N)	Edges (E)	Avg. Degree (k)
Full CRAN (FC)	15516	66594	8.584
Giant Component (GC)	13838	66574	9.622
Transitive Closure (TC)	13838	381998	55.210
TC except base packages (TCNB)	12686	274449	43.26

Formally, CRAN can be considered to be a set of packages $P = P_1, P_2, \ldots$. The package dependency network is, thus, a directed graph $DG(N, E)$ with nodes N and edges E where node i corresponds to package P_i and directed links $(i, j) \in E$ represent a dependency between packages P_i and P_j; P_i imports or depends on P_j. The average number of edges directed towards the network nodes is their average in-degree (k^{in}), while the average number of edges leaving them become their average out-degree (k^{out}). The average degree in the network, therefore, can be represented as $k = k^{in} + k^{out}$. Note how k_i^{in} corresponds to the number of classes that use (import or depend on) P_i, while k_i^{out} corresponds to the number of other packages that are required for P_i in order to function.

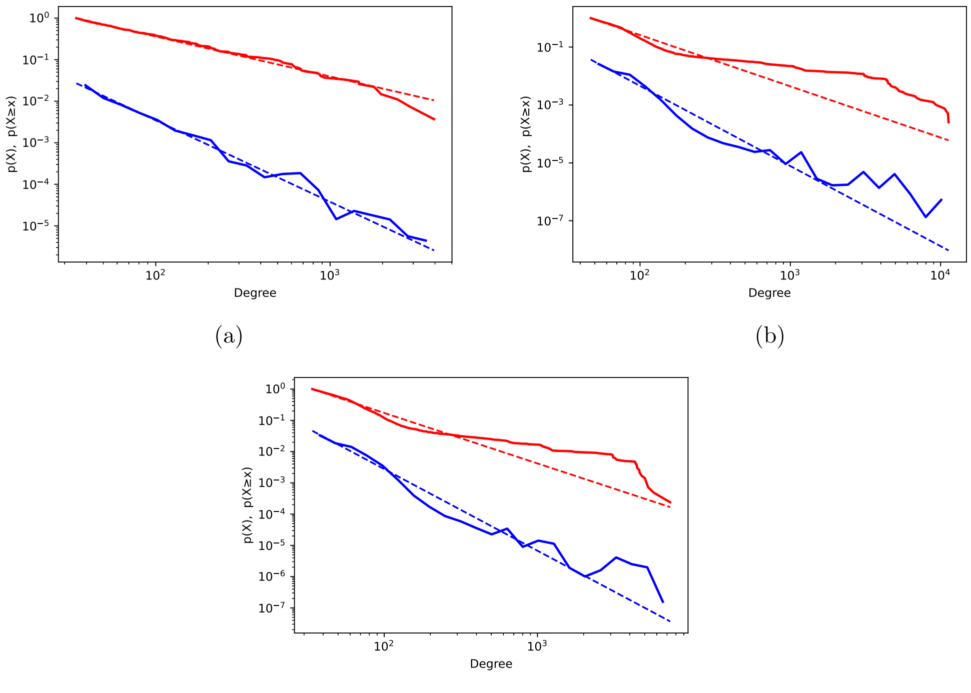
The full CRAN network, constructed using packages as nodes and direct dependencies as edges, is not connected. A number of packages can be found in the periphery of the network that are either standalone (with no dependencies) or depend on a handful of other peripheral packages. Following the approach in previous works, such disconnected packages are discarded by reducing the CRAN network to its largest connected component (also known as the giant component). Additionally, to reflect the transitive dependencies (the recursive dependencies of dependent packages) in the network, the transitive closure of the network is considered.

Definition 3.1. Transitive closure. The transitive closure of $G = (N, E)$ is a graph $G+ = (N, E+)$ such that for all i, j in N there is an edge (i, j) in $E+$ if and only if there is a path from i to j in G.

An example of such transformation is represented in Fig. 1, where the ego network of the package 'A3' (Fortmann-Roe, 2015) (the first in alphabetical order in CRAN) is used for reference.

Finally, there are two special sets of packages to be considered:

- Base packages that are included with the R distribution. This list is formed by a total of 14 packages, namely: 'base',



(a)

(b)

(c)

Fig. 2. Probability density function ($p(X)$, blue, corresponding to the lower lines in each sub-graph) and complementary cumulative distribution function ($p(X \geq x)$, red, corresponding to the upper lines in each sub-graph) of degrees in (a) Giant Component (GC); (b) Transitive Closure (TC); (c) TC except base packages (TCNB). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

'compiler', 'datasets', 'grDevices', 'graphics', 'grid', 'methods', 'parallel', 'splines', 'stats', 'stats4', 'tcltk', 'tools' and 'utils'.

- CRAN-recommended add-on packages, included in all binary distributions of R. These are a total of 15 packages: 'KernSmooth', 'MASS', 'Matrix', 'boot', 'class', 'cluster', 'codetools', 'foreign', 'lattice', 'mgcv', 'nlme', 'nnet', 'rpart', 'spatial' and 'survival'.

As this set of 29 packages is highly relevant and close to the core (there is no binary distribution of R without them), a disproportional large number of packages depend on them. Although this is indeed relevant to assess the modular structure of the network, for instance, it can distort the analysis of the vulnerabilities. Base packages could hardly be considered third-party risks to the R package ecosystem when they are inseparable from the R base distribution. Therefore, the transitive closure network without these two sets of packages will also be considered for a complete analysis and to obtain insights beyond the core packages of R.

Table 1 shows the properties of the CRAN package dependency network in the four iterations (full network, giant component, transitive closure and transitive closure without base packages) considered above. The latter three will also be compared to their equivalent Erdös–Renyi random graphs (Erdős and Rényi, 1960), where a link is placed between a certain pair of nodes with probability $k/(n-1)$, where $k = 2e/n$ for a given number of nodes (n) and edges (e).

4. Analysis and results

In this section, the CRAN package dependency network will be analyzed in three dimensions in order to show how complex network analysis can be applied to package ecosystems while benefiting developers, maintainers and contributors. These three dimensions result in the following research questions:

Table 2
Network statistics for the CRAN package dependency network.

Network	γ	C	C_{ER}	l	l_{ER}	D	n_d
Giant Comp. (GC)	1.931	0.21	0.0006	3.10	4.47	0.0003	78.9
Trans. Closure (TC)	2.769	0.31	0.0040	2.21	2.79	0.0020	5.6
TC except base (TCNB)	2.613	0.26	0.0034	2.94	2.860	0.0017	15.5

1. (RQ1) In regard to its structure and complexity, does the CRAN package dependency network exhibit scale-free and small-world behaviors?
2. (RQ2) Concerning the individual packages in the ecosystem, what are the most vulnerable?
3. (RQ3) Is the CRAN package dependency network modular? Is it possible to infer an underlying structure using the relationships between packages?

Each subsequent subsection aims to answer one of these research questions.

4.1. RQ1: Structure and complexity

Degree distribution experienced by simple random graphs is either binomial or Poisson when the size of the graph is large (Albert and Barabási, 2002). However, many real-world networks have been found to follow different patterns. For instance, many networks' degree distribution follows the power-law property, while others' exhibits non-power-law features such as exponential cutoffs (Amaral et al., 2000). Software networks have been found to follow a power-law degree distribution (Potanin et al., 2005; Šubelj and Bajec, 2012):

$$p_k \sim k^{-\gamma} \tag{1}$$

with p_k as the probability of a certain degree k and γ as the scale-free exponent, with $\gamma > 1$. The power-law relationship can be directly observed in a log–log plot with an straight line

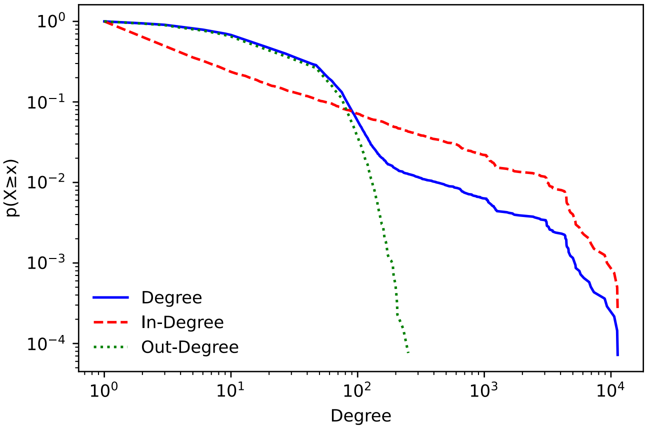


Fig. 3. Comparison between the degree, in-degree (reverse dependencies) and out-degree (dependencies) distributions. It can be seen how the out-degree distribution (green, dotted) is heavily truncated versus the in-degree (red, dashed) distribution.

of slope $-\gamma$ (Alstott and Bullmore, 2014). The values for γ in each network can be found in Table 2 and their corresponding log–log diagrams in Fig. 2, where the complementary cumulative distribution function is also added for reference. All three networks exhibit power-law degree distributions, in line with the hypothesis of their scale-free property. In scale-free networks, the probability of two nodes being linked is not a constant as in random graphs; instead, it depends on the number of links that a node already has. In other words, the more popular a node is, the more likely it is to increase its number of links when new nodes are added. In the case of the CRAN ecosystem (and the dependency networks that are being analyzed here), this implies that packages with a high number of reverse dependencies (meaning high in-degree) are more likely to become dependencies of newly developed packages too, a phenomenon that is noticeable, for example, looking at the base packages.

Nevertheless, this power-law probability distribution would then be expected only in the in-degree distribution, as it represents the number of other packages that require a given package to function. In-degree distribution, thus, is analogue to the degree of package reusability. Besides, out-degree distribution takes another approach; as it represents the number of packages required for a given package to work, it reflects software complexity. The ideal software project should exhibit scale-free behavior on in-degree (high reusability) and a highly truncated out-degree distribution (avoiding high complexity in a single package dependencies) (Šubelj and Bajec, 2012). In the case of CRAN, these distributions are represented in Fig. 3, showing how both properties are fairly obeyed by its distribution.

However, while high reusability decreases the probability of fault propagation through the system, it also increases its vulnerability in case of a bug in any of the highly reused packages, as even a very small fraction of faulty nodes can already render the entire system inoperable. Both packages with high in-degree and out-degree need to be particularly monitored and carefully maintained; the details per node will be analyzed in sub Section 4.2.

On the other hand, small-world (Watts and Strogatz, 1998) behavior usually refers to high clustering (C) and a short average distance (l) between the nodes. Clustering measures transitivity in the network; for unweighted graphs, the clustering of a node u is the fraction of possible triangles through that node that exist. It can also be understood as the probability of any two neighbors of a given network being also linked. From their definition, small-world graphs should exhibit $C \gg C_{ER}$ and $l \sim l_{ER}$, where both C_{ER} and l_{ER} are the respective properties for an equivalent Erdös–Renyi random graph. All these figures for the CRAN networks can be found in Table 2 and, from them, it can be derived that the ecosystem also behaves as a small-world network, which is desirable in well designed software projects (Šubelj and Bajec, 2012), as it indicates a good relationship between packages that share similar functions (measured by C) while avoiding a balkanization of the ecosystem (where parts become independent and unaware of each other) (measured by l). It should be noted, though, that these measures only make sense with the dependency graph converted to undirected (Kohring, 2009); the opposite would actually imply that there would be cyclic dependencies among packages, which are undesired.

Network densities (D), which are the ratio between the actual number of edges and those of a complete graph, were also computed for all three networks in Table 2; as expected for real-world and software networks (Zheng et al., 2008), the CRAN network is sparse in general.

Therefore, in answer to the first research question, the CRAN package dependency network adheres to a power-law, which follows the principles of the scale-free networks, while also reflecting a small-world behavior.

4.2. RQ2: Individual packages (nodes)

As developers work with previous packages to develop newer ones, quality, maintenance and trustworthiness of the existing packages is key for the stability of the ecosystem. These properties, however, are not particularly visible and they are not under the developer's control, either. A famous incident happened in 2016 when a single JavaScript package, called *left-pad* (https://github.com/stevemao/left-pad/issues/4) was removed from the central JavaScript package repository *npm*. This removal caused issues not only for the projects that depended on it, but also for those that depended transitively on the package. In the case of CRAN, previous work has shown that up to 41% of the errors in CRAN packages were caused by incompatible changes in one of its dependencies (direct or transitive) (Claes et al., 2014). Thus, issues or bugs with packages propagate through any number of levels of dependency, not only on direct ones. It is therefore possible to measure the vulnerability (v) of the ecosystem to an issue of a given package as the fraction of packages in the whole ecosystem that would be impacted by the propagation of that issue through its dependencies. Such information could be incorporated in measuring package importance with regards to vulnerability in an ecosystem, as a high vulnerability score should alert developers and maintainers to ensure a fast response to bugs ans issues, as they could both raise a chain reaction and raise the interest of any attacker interested in finding an opportunity to exploit the project (Kikas et al., 2017).

To find the most critical nodes in regards to vulnerability, one can make use of the centrality metrics, whose main purpose is measuring nodes influence. Many centrality measures are available, and each one defines "relevancy" differently. For instance, nodes with high betweenness centrality influence the flow around a system, while closeness centrality aims to measure how well placed a node is in the network. Degree centrality, even though it could be considered as the simplest measure of node connectivity, is also the most appropriate metric to find very connected or popular nodes that, in case of failure, would impact a larger number of other nodes (or packages). In the present case, the normalized degree centrality (DC_i) will be computed for each package in the network as

$$DC_i = \frac{k_i}{n-1} \tag{2}$$

Table 3
Top 20 influential nodes in CRAN.

All packages (TC)					Excluding base packages (TCNB)				
Package	*DD*	*TD*	DC_i	$\upsilon(\%)$	Package	*DD*	*TD*	DC_i	$\upsilon(\%)$
methods	2876	11298	.8165	81.7	Rcpp	1786	7333	.5781	57.8
utils	2436	11197	.8092	80.9	magrittr	930	5754	.4536	45.4
stats	3953	10607	.7666	76.7	glue	175	5271	.4155	41.6
grDevices	1210	9302	.6723	67.2	digest	215	5163	.4070	40.7
graphics	1962	8923	.6449	64.5	R6	262	5075	.4000	40.0
Rcpp	1786	7333	.5301	53.0	rlang	612	5005	.3946	39.5
grid	440	6930	.5008	50.0	pkgconfig	7	4776	.3765	37.7
lattice	400	6728	.4861	48.6	crayon	172	4702	.3707	37.1
tools	261	6113	.4418	44.2	assertthat	205	4628	.3648	36.5
magrittr	930	5754	.4158	41.6	stringi	177	4623	.3644	36.4
Matrix	874	5628	.4072	81.6	backports	29	4599	.3626	36.3
glue	175	5271	.3810	38.1	ellipsis	12	4493	.3543	35.4
MASS	1461	5242	.3788	37.9	vctrs	14	4452	.3514	35.1
digest	215	5163	.3731	37.3	zeallot	6	4457	.3514	35.1
R6	262	5075	.3667	36.7	cli	78	4436	.3500	35.0
rlang	612	5005	.3617	36.2	fansi	5	4437	.3498	35.0
pkgconfig	7	4776	.3451	34.5	pillar	19	4353	.3431	34.3
crayon	172	4702	.3398	34.0	utf8	5	4357	.3434	34.3
assertthat	205	4628	.3344	33.4	tibble	673	4339	.3420	34.2
stringi	177	4623	.3340	33.4	stringr	841	4293	.3384	33.8

Table 4
Packages with out-degree larger than 200 in CRAN.

All packages (TC)			
Package	Imports	Transitive	Inverse υ (%)
smartdata	24	251	1.81
ggstatsplot	27	229	1.65
psychNET	23	206	1.49
KNNShiny	9	205	1.48
STAT	9	203	1.47
CLUSTShiny	8	201	1.45

with k_i being the degree of node i, n the total number of nodes in the network and $DC_i \in [0, 1]$. We can, thus, assimilate the degree centrality for a node i to the fraction of nodes it is connected to (independently of the direction of the link). For each node, both the degree centrality DC and the corresponding υ is represented in Table 3, combined with their direct dependencies (*DD*) and transitive dependencies (*TD*). The table contains the top 20 influential nodes for both the TC and TCNB cases, as it can be noticed that in the TC case base packages take most of the slots. For the TCNB case, there are a total of 20 packages that, if exposed, would individually impact more than 30% of the whole network through their dependencies. In this list, it is worth highlighting how a number of packages have quite limited numbers of direct dependencies but their transitive ones are up to three orders of magnitude larger. On the other hand, if the base packages are included, one can realize how issues in ‘methods’, ‘utils’ or ‘stats’ would basically take the whole ecosystem down.

Packages with out-degree larger than 200 (so, packages that are transitively dependent on more than 200 other packages) can also be found in Table 4. Again, this table shows how packages can show a limited number of imports but are indirectly (and probably, unknowingly) importing hundreds of transitive dependencies. The *smartdata* (Cordón et al., 2019) package, for instance, might import a total of 24 packages (a large number already) but it ends up depending on 251 packages, ten times more than that, which means an inverse vulnerability (proportion of packages in the ecosystem that could break it) of 1.8%. Extra care should be put in development of packages that either have high vulnerability or relatively large inverse one; developer efforts seem to reflect this tendency, as an statistically relevant relationship (p-value ≈ 0) is found between the centrality of a package and the number of times it has been updated over time.

Controllability of complex networks (Liu et al., 2011) is another concept that could be useful to understand and characterize software package ecosystems. For scale-free networks with exponent γ and average degree $\langle k\rangle$ we can use the following equation to compute the fraction of nodes that would be required to “control” the system (known as driver nodes).

$$\frac{n_d}{n} \sim exp\left[-\frac{1}{2}\left(1-\frac{1}{\gamma-1}\right)\langle k\rangle\right] \tag{3}$$

The results are shown in the last column of Table 2. Note that, although the giant component network (GC) requires almost 79 packages to be controlled, it is misleading; once the transitive relationships are considered the number is notably lower. In summary, when the base packages are included, six packages would suffice to take control of the whole network; 16 in case these are not considered. This is consistent with the vulnerabilities found in Table 3.

With regards to the second research question, thus, the present analysis of the packages available in CRAN using CNA quickly reveals how a few packages could expose the whole ecosystem in case of failure, bug or malicious attack.

4.3. RQ3: Modules

The links between packages in dependency networks are a product of the underlying patterns and structures. It could be expected, for instance, that packages that have similar functions are aggregated into package communities of relatively densely connected nodes. Finding a clear package community structure would mean that the software package ecosystem is highly modular (meaning that functions are basically independent from each other), while under-structured or disorganized projects would have a quasi-random structure.

In general, one of the many challenges proposed in the field of complex network analysis consists of community detection, and multiple community detection algorithms have been described (Gadek et al., 2018). One of the most popular and widely used algorithm is the Louvain method (Blondel et al., 2008), which maximizes a modularity score for each community. For reference, the modularity of a partition is a scalar value between −1 and 1 that measures the density of links inside communities as compared to links between communities (Newman, 2006).

The Louvain algorithm is thus applied to the dependency network; the number of communities that emerge is stable at 17 and

Table 5
Summary of package communities found in the CRAN ecosystem, from largest to smallest (containing over 1% of packages).

%	Sample pkgs	Critical pkgs	Keywords in description	Summary
26.7	mvtnorm nlme lme4	stats utils graphics	Analysis Methods Regression	Popular packages for general statistical analysis.
19.9	jsonlite tibble tidyr	dplyr magrittr stringr	API Tables Dataset	Packages for managing and tidying data.
11.9	cluster nnet caret	parallel Matrix foreach	Selection Regression Classification	Classification, regression and clustering models.
6.94	rgdal fields maptools	sp lattice raster	Spatial Raster Map	Classes and methods for spatial data.
6.68	ggplot2 colorspace reshape2	ggplot2 grid plyr	ggplot2 Plot Tools	Visualization tools and data arrangement.
6.35	Rcpp rstan rjags	Rcpp coda Rdpack	C++ Bayesian MCMC	Integration languages into R, plus Bayesian and MCMC models.
5.90	gsl rrcov leaps	methods robustbase stats4	Methods Distribution Multivariate	Tools allowing to analyze data with robust methods.
2.88	rjson tm nlp	XML RCurl rjava	Text Web Java	Text mining applications plus web/java/json interfaces.
2.58	expm ade4 seqinr	ape gtools vegan	Phylogenetic Species Trait	Analysis of ecological/biological data in environmental sciences.
2.55	tseries timeData timeSeries	zoo xts forecast	Time Series Financial	Time series analysis and computational finance.
2.18	Biobase limma Biostrings	matrixStats R.utils future	Gene Genomic RNA	Bioconductor (bioinformatics) subcommunity.
2.12	tkrplot tcltk2 gWidgets	rgl tcltk Rcmdr	GUI Graphical 3D	Tools for both interactive GUI and (3D) graphics.
1.42	sna ergm intergraph	igraph network GGally	Network analysis Graph Clustering	Tools for Social/Complex Network Analysis

the resulting graph is shown in Fig. 4, where the most relevant nodes (in regards to their in-degree) are labeled, obtaining a modularity of 0.4. The summary of the 13 communities that contain more than 1% of the total packages can also be found in Table 5, which reflects the portion of the total packages that each package community represents, three sample relevant packages (avoiding base packages that might distort the results) and three critical packages (understood as the ones with highest in-degree–highest vulnerabilities).

To infer a meaning for each partition, natural language processing techniques were used; all the available textual descriptions for the packages in each set is aggregated and analyzed using *spaCy* (https://spacy.io/), a Python library. After removing the common standard stopwords, the 30 top unique words found in the package's descriptions were annotated manually by three independent annotators, one with statistical and two with computer science background, that also analyzed each of the top packages in each package community, initially agreeing in 11 out of the 13 groups (84.6%). The remaining two were discussed afterwards and a final agreement was reached with the identification found in Table 5, which produces a small summary of the structural reasons hidden behind the clustering produced by the algorithm.

Among them, the largest package community contains slightly over one quarter of the total packages in the ecosystem (26.7%) and could be considered the functional core of the R package ecosystem, with the most popular packages for general statistical analysis. The rest of the communities are, overall, more specific of particular functionalities, disciplines or environments. Functionally, for instance, the second largest package community is formed by a large number of packages that are devoted to managing and cleaning data (such as the *tidyverse* set) while the fifth largest group (6.68%) is formed by ggplot2 (Wickham, 2016) and the visualization ecosystem (including all the "gg" family) around this highly relevant package. In regards to disciplines, a few package communities are found that, for example, are focused in social/complex network analysis (1.42%) or time series analysis (2.55%). With the environmental perspective, there are at least two communities that are focused in environmental sciences, distinguished by whether their common packages are in the CRAN (2.58%) or in the Bioconductor (2.18%) repository.

Thus, and in response to the third research question, the modular analysis (using the community detection approach in the network of packages) reveals how this approach can detect and highlight the functional or environmental modules in a software package ecosystem such as CRAN.

5. Discussion

In this article, we empirically studied the CRAN software package ecosystem through complex network analysis tools, a method common in other fields but not as widely adopted for software

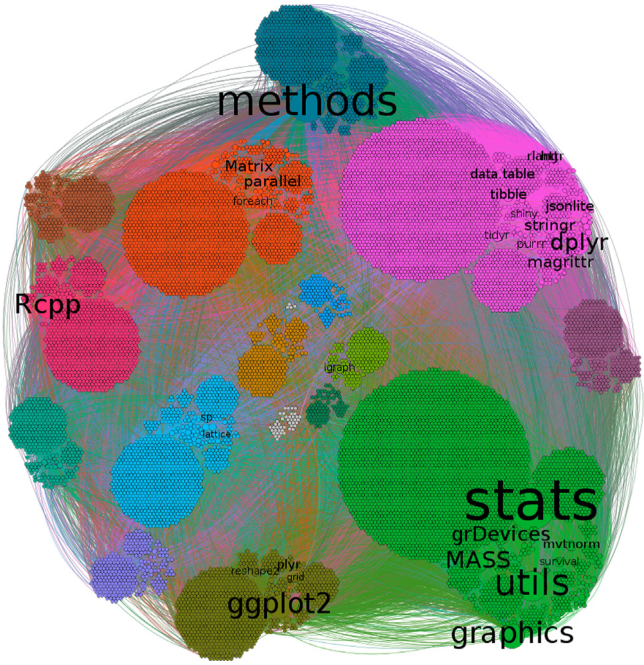


Fig. 4. Package communities obtained using Louvain's method for detecting community structure based on modularity. A total of 17 package communities are identified.

engineering. Using the metadata from CRAN, the network of packages was built and its properties have been analyzed.

Structure and complexity. Previous research already pointed out at the power-law (or near power-law) nature of dependency networks, although most of them was based in classes instead of packages (Šubelj and Bajec, 2012). In the case of CRAN, the degree distribution adheres to a power-law, both in degree and in in-degree, implying that packages with a high number of reverse dependencies tend to have a higher probability of receiving further incoming links, following the principles of the scale-free networks. Additionally, the small-world behavior was tested for the CRAN case, noticing how the actual clustering of the network is orders of magnitude higher than the simulated clustering for a random network with the same characteristics. Average shortest patch between packages, on the other hand, is in the same order of the average shortest path in a random graph, which combined with the previous results in clustering determines that the behavior of the CRAN package dependency network is in line with what is expected from a small-world network in software engineering: good relationship between packages that share similar functions while avoiding creating separate components far from each other. The small-world result for software package ecosystems has also been found, for instance, in the Debian repository (Abate et al., 2009).

Packages. Our analysis of the packages available in CRAN (represented by the nodes of our network) reveals how a few packages could expose the whole ecosystem in case of failure, bug or malicious attack. When taking base packages into account, some of them (e.g. 'methods', 'utils' or 'stats') could affect around 80% of the packages in the repository if an update went wrong, as their transitive reverse dependencies are huge. However, it could be understood that base packages sit at the core of R and, therefore, the base packages and R could be considered as one. Additionally, it could be assumed that such base packages are both taken with more care and updated less often, so they might be less prone to a fault. In a deeper analysis, we found a total of 20 packages (besides the core 29 base packages) whose removal could impact more than 30% of the other packages, which is a higher figure than found in other systems (Šubelj and Bajec, 2012). Some of them might seem harmless when only direct dependencies are considered, with less than 10 of them, but once transitive dependencies are considered, the potential impact could render the ecosystem unusable. In summary, we showed how CRAN has a few central packages (and a few highly imported ones) that are critical; high vulnerabilities, as shown in other ecosystems, should alert developers but, specially, maintainers, to keep a close look on potential bugs or security issues and their fixes for these packages, as being able to control a few of them would effectively give control over the (Kikas et al., 2017). Moreover, from the opposite perspective, packages with large number of dependencies should also be monitored as their potential to fail is multiple times larger.

CRAN has strict policies on maintainers and contributions (https://cran.r-project.org/web/packages/policies.html). Among other policies, CRAN runs a periodic check on compatibility among packages; should any package fail the test, its maintainers would be notified and asked to resolve the issue before the following major R release, at the risk of having their package archived otherwise. CRAN also forces dependencies to be kept within itself or Bioconductor (to avoid external dependencies). Additionally, back-compatibility versions of current packages is not allowed, and any changes to CRAN packages that could cause significant disruption to other packages must be agreed with the CRAN maintainers before releasing it. These policies have a direct impact in mitigating most of the risks highlighted previously, although it is at the cost of the CRAN's maintainers efforts; developing tools based in CNA metrics could contribute towards minimizing CRAN volunteers lost time.

Modules. We also explored the structure of the CRAN package network from the perspective of community structure or modularity, running the Louvain algorithm and adding insights on the meaning of the resulting communities using NLP techniques on the available descriptions of packages. The main contribution here is to show how the dependency network obtained from CRAN reveals a significant package community structure and how such structure can be explained using the functionality or other relevant contextual aspects of the clustered packages. This is, thus, the proof of another property that is true for other networks as for software networks; hidden structure in software can be brought to surface using community detection algorithms (Šubelj and Bajec, 2011). In the case of CRAN, communities appear to be relatively balanced and, besides a core set of packages that cover the statistical analysis tools R is known for, there are multiple communities of packages devoted to common tasks (such as data wrangling) or to particular disciplines (such as bioinformatics). A modular approach has been shown to enhance functionality and evolvability (Fortuna et al., 2011). This results help drawing a map of CRAN, which, as a large software ecosystem, represents one of the most complex human made systems.

6. Conclusion

Our analysis of the CRAN software package ecosystem from the perspective of complex network analysis shows how CRAN follows a scale-free and small-world behavior, as found previously in other OSS package ecosystems, and that relates to good practices of software engineering. CRAN, however, presents a large number of packages that are critical for the correct function of the ecosystem and that, in case of any bug or issue, it could render the whole system unusable. CRAN's policies, in any case, are in place to prevent such event from happening, but it is something that has a maintenance cost. Finally, we also shown how the CRAN package network presents a significant modular structure, which is also a positive aspect of software engineering and enhances its functionality and evolvability, allowing such complex product of many contributors to go further into the future. Our findings show how package networks such as CRAN could benefit from complex network analysis as a tool to assess many aspects of software engineering, such as quality assurance or update risk evaluation. In particular, it is also worth highlighting how making dependency relationships more visible in package networks could help developers visualize the relevance of some packages and the critical value of others, so they could for example balance the number of dependencies or inverse dependencies, or aim to group together existing functionalities in order to reduce the number of critical packages to be maintained. In summary, CNA provides insights into relationships between components in package ecosystems that may be useful for a number of stakeholders, including core development teams, project managers and contributors (both individuals and organizations) who might want to understand how to contribute to the ecosystem in a way that best fits their audiences and interests.

7. Threats to validity and future work

The current work is limited, however, as package versions are not considered. CRAN's policies periodically checks compatibility among packages so only the packages need to be considered. The generalization of the results found previously would need to add package versioning (and, therefore, vulnerability per version) as an additional layer. In the same line, it would be interesting to extend the CNA analysis to study how the CRAN network has changed over time and how the maintainers' mitigation activities have developed over time in response to changes in the networks of dependencies.

Another limitation is the use of manual analysis in Section 4.3, where NLP techniques were combined with manual annotation to infer a meaning for the clustering produced by the algorithm. Although the use of three independent reviewers mitigates the impact partially, it is acknowledged that experts from other fields (e.g. bioinformatics) might identify further nuances that were not captured in our analysis.

These findings show how CNA can be a valuable tool to study package ecosystems from the perspective of their dependency networks. Future work should follow at least three lines of research. First of all, the relationship between the CNA metrics of the packages and their quality metrics (e.g. open issues, user downloads, number of developers, binary sizes, among others) should be explored deeper. On one hand, to provide specific tools that developers could use to be aware of the most balanced approach for their packages, whether it is a leaner package with fewer dependencies or a more "feature-full" package directed to a particular audience, and to distinguish between those packages that are safe or those that should be avoided. On the other hand, the awareness of the developers should also be studied: what is their approach in regards to dependencies? Do they use any quantitative approaches when deciding on the dependencies they are going to include in their software? Second, the analysis could be extended with additional measures and network metrics, that could also be further combined with techniques from other disciplines such as NLP. Finally, CRAN is a popular ecosystem, but there are many more that could either be analyzed individually or compared from a complex network perspective to bring additional and potentially valuable findings to the field.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S., 2009. Strong dependencies between software components. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 89–99.

Albert, R., Barabási, A.-L., 2002. Statistical mechanics of complex networks. Rev. Modern Phys. 74 (1), 47.

Alstott, J., Bullmore, D.P., 2014. Powerlaw: a Python package for analysis of heavy-tailed distributions. PLoS One 9 (1).

Amaral, L.A.N., Scala, A., Barthelemy, M., Stanley, H.E., 2000. Classes of small-world networks. Proc. Natl. Acad. Sci. 97 (21), 11149–11152.

Barabási, A.-L., 2009. Scale-free networks: a decade and beyond. Science 325 (5939), 412–413.

Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E., 2008. Fast unfolding of communities in large networks. J. Stat. Mech.: Theory Exp. 2008 (10), P10008.

Cataldo, M., Scholtes, I., Valetto, G., 2014. A complex networks perspective on collaborative software engineering. Adv. Complex Syst. 17 (07n08), 1430001.

Cerina, F., Zhu, Z., Chessa, A., Riccaboni, M., 2015. World input-output network. PLoS One 10 (7).

Chong, C.Y., Lee, S.P., 2015. Analyzing maintainability and reliability of object-oriented software using weighted complex network. J. Syst. Softw. 110, 28–53.

Claes, M., Mens, T., Grosjean, P., 2014. On the maintainability of CRAN packages. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. CSMR-WCRE, IEEE, pp. 308–312.

Cordón, I., Luengo, J., García, S., Herrera, F., Charte, F., 2019. Smartdata: Data preprocessing to achieve smart data in R. Neurocomputing URL: https://doi.org/10.1016/j.neucom.2019.06.006.

Cox, J., Bouwers, E., Van Eekelen, M., Visser, J., 2015. Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 109–118.

Csárdi, G., Salmon, M., 2019. Pkgsearch: Search and query CRAN R packages. URL: https://CRAN.R-project.org/package=pkgsearch. R package version 3.0.1.

Decan, A., Mens, T., Claes, M., 2016. On the topology of package dependency networks: A comparison of three programming language ecosystems. In: Proceedings of the 10th European Conference on Software Architecture Workshops, pp. 1–4.

Decan, A., Mens, T., Grosjean, P., 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir. Softw. Eng. 24 (1), 381–416.

Erdős, P., Rényi, A., 1960. On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci. 5 (1), 17–60.

Feng, J., Li, X., Mao, B., Xu, Q., Bai, Y., 2017. Weighted complex network analysis of the Beijing subway system: Train and passenger flows. Physica A 474, 213–223.

Fortmann-Roe, S., 2015. Consistent and clear reporting of results from diverse modeling techniques: The A3 method. J. Stat. Softw. 66 (7), 1–23, URL: http://www.jstatsoft.org/v66/i07/.

Fortuna, M.A., Bonachela, J.A., Levin, S.A., 2011. Evolution of a modular software network. Proc. Natl. Acad. Sci. 108 (50), 19985–19989.

Gadek, G., Pauchet, A., Malandain, N., Vercouter, L., Khelif, K., Brunessaux, S., Grilhères, B., 2018. Topological and topical characterisation of twitter user communities. Data Technol. Appl.

German, D.M., Adams, B., Hassan, A.E., 2013. The evolution of the R software ecosystem. In: 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, pp. 243–252.

Halberstam, Y., Knight, B., 2016. Homophily, group size, and the diffusion of political information in social networks: Evidence from Twitter. J. Public Econ. 143, 73–88.

Harutyunyan, N., 2020. Managing your open source supply chain-why and how?. Computer 53, 77–81. http://dx.doi.org/10.1109/MC.2020.2983530.

Kasakawa, S., Yamanishi, T., Takahashi, T., Ueno, K., Kikuchi, M., Nishimura, H., 2016. Approaches of phase lag index to EEG signals in Alzheimer's disease from complex network analysis. In: Innovation in Medicine and Healthcare 2015. Springer, pp. 459–468.

Kikas, R., Gousios, G., Dumas, M., Pfahl, D., 2017. Structure and evolution of package dependency networks. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. MSR, IEEE, pp. 102–112.

Kohring, G., 2009. Complex dependencies in large software systems. Adv. Complex Syst. 12 (06), 565–581.

Korkmaz, G., Kelling, C., Robbins, C., Keller, S.A., 2018. Modeling the impact of R packages using dependency and contributor networks. In: 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. ASONAM, IEEE, pp. 511–514.

Li, K., Yu, M., Liu, L., Zhai, J., Liu, W., 2018. A novel reliability analysis approach for component-based software based on the complex network theory. Softw. Test. Verif. Reliab. 28 (6), e1674.

Liu, Y.-Y., Slotine, J.-J., Barabási, A.-L., 2011. Controllability of complex networks. Nature 473 (7346), 167–173.

Mohagheghi, P., Conradi, R., 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empir. Softw. Eng. 12 (5), 471–516.

Mora-Cantallops, M., Sánchez-Alonso, S., Visvizi, A., 2019. The influence of external political events on social networks: The case of the Brexit Twitter Network. J. Ambient Intell. Human. Comput. 1–13.

Mora-Cantallops, M., Sicilia, M.-Á., 2018. Player-centric networks in League of Legends. Social Networks 55, 149–159.

Myers, C.R., 2003. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68 (4), 046116.

Nagle, F., 2019. Open source software and firm productivity. Manage. Sci. 65, 1191–1215.

Newman, M.E., 2006. Modularity and community structure in networks. Proc. Natl. Acad. Sci. 103 (23), 8577–8582.

Pan, W., Li, B., Ma, Y., Liu, J., 2011. Multi-granularity evolution analysis of software using complex network theory. J. Syst. Sci. Complex. 24 (6), 1068–1082.

Potanin, A., Noble, J., Frean, M., Biddle, R., 2005. Scale-free geometry in OO programs. Commun. ACM 48 (5), 99–103.

Raemaekers, S., Van Deursen, A., Visser, J., 2014. Semantic versioning versus breaking changes: A study of the maven repository. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 215–224.

Rubinov, M., Sporns, O., 2010. Complex network measures of brain connectivity: uses and interpretations. Neuroimage 52 (3), 1059–1069.

Šubelj, L., Bajec, M., 2011. Community structure of complex software systems: Analysis and applications. Physica A 390 (16), 2968–2975.

Šubelj, L., Bajec, M., 2012. Software systems through complex networks science: Review, analysis and applications. In: Proceedings of the First International Workshop on Software Mining, pp. 9–16.

Watts, D.J., Strogatz, S.H., 1998. Collective dynamics of 'small-world' networks. Nature 393 (6684), 440.

Wickham, H., 2016. Ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, URL: https://ggplot2.tidyverse.org.

Zheng, X., Zeng, D., Li, H., Wang, F., 2008. Analyzing open-source software systems as complex networks. Physica A 387 (24), 6190–6200.

Zimmermann, T., Nagappan, N., 2008. Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th International Conference on Software Engineering, pp. 531–540.