npm Packages as Ingredients: a Recipe-based Approach

Kyriakos C. Chatzidimitriou, Michail D. Papamichail, Themistoklis Diamantopoulos,
Napoleon-Christos Oikonomou, and Andreas L. Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, Greece
{*kyrcha, mpapamic, thdiaman, noikon*}*@issel.ee.auth.gr, asymeon@eng.auth.gr*

Keywords: Dependency Networks, Software Reuse, JavaScript, npm, node.

Abstract: The sharing and growth of open source software packages in the npm JavaScript (JS) ecosystem has been exponential, not only in numbers but also in terms of interconnectivity, to the extend that often the size of dependencies has become more than the size of the written code. This reuse-oriented paradigm, often attributed to the lack of a standard library in node and/or in the micropackaging culture of the ecosystem, yields interesting insights on the way developers build their packages. In this work we view the dependency network of the npm ecosystem from a "culinary" perspective. We assume that dependencies are the ingredients in a recipe, which corresponds to the produced software package. We employ network analysis and information retrieval techniques in order to capture the dependencies that tend to co-occur in the development of npm packages and identify the communities that have been evolved as the main drivers for npm's exponential growth.

1 INTRODUCTION

The popularity of JS is constantly increasing, and along is increasing the popularity of frameworks for building server (e.g. Node.js), web (e.g. React, Vue.js, Angular, etc.), desktop (e.g. Electron) or mobile applications (e.g. React Native, NativeScript, etc.), even IoT solutions (e.g. Node-RED). A common denominator to this explosive growth has been the launch of the npm registry (i.e. the package manager of JS) in 2010. The npm ecosystem is often seen as one of the JS revolutions[1] that have transformed JavaScript from "a language that was adding programming capabilities to HTML" into a full-blown ecosystem. In fact, the growth is so rapid that terms like "JS framework fatigue" have become common among developers. Indicatively, the June 2018 Redmonk survey[2], the GitHub status report[3], and the 2018 Stack Overflow survey[4] position JS as the most popular programming language, while Module Counts[5] depicts an exponential growth of npm modules against repositories of other languages.

Given that dependencies and reusability have become very important in today's software development process, npm registry has become a "must" place for developers to share packages, defining code reuse as a state-of-the-practice development paradigm (Chatzidimitriou et al., 2018). A white paper by Contrast Security (Williams and Dabirsiaghi, 2014) mentions that up to 80% of the code in today's software applications comes from libraries and frameworks. This is evident in the npm ecosystem, where the number of dependencies for a package has been shown to grow with time (Wittern et al., 2016). There are even extreme cases, where one-liner libraries have more than 70 dependencies (Haney, 2016). Such extreme reusability is usually attributed to the lack of a standard library in node.js and to the micropackaging culture of the npm ecosystem.

Against this background, in this work we extract the collective knowledge and preferences when creating JavaScript (node.js) packages through mining the most proliferate module repository, the npm registry. Inspired by (Teng et al., 2012), we treat packages as recipes and dependencies as ingredients. Just like a recipe comprises a list of ingredients along with a process on how to combine them, one can consider an npm package as a recipe: a list of core dependencies and a list of development dependencies, also known as devDependencies in the npm lingo, that are combined together with some code that uses them. And

[1] https://youtu.be/L-fx2xXSVso

[2] https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/

[3] https://octoverse.github.com/

[4] https://insights.stackoverflow.com/survey/2018/

[5] http://www.modulecounts.com/

just like online recipes receive reviews and comments, npm packages have repository stars, forks, watchers and package downloads, GitHub issues, Stack Overflow posts, and so on. This type of information can provide insights not only about package popularity or quality but also about user preferences, development tendencies, and which dependencies go well with others. We focus on the following research questions:

- RQ_1: Are there any interesting communities (clusters) developed by dependencies that tend to co-occur together in "recipes"?
- RQ_2: Is the same true for devDependencies?
- RQ_3: Can we identify the scope these communities are used for?
- RQ_4: What are the similarities or differences between package recipes (node.js packages of the npm registry) and application recipes (node.js applications that are not part of the npm registry)?

By providing answers to these questions we aim to get a better understanding of how packages work when used together as dependencies. We can also answer questions as to which packages mix well together and for what purpose.

2 BACKGROUND

The npm registry[6] is the largest and fastest growing module (or packages in the JS terminology) repository of all programming languages. At the moment of writing it includes more than $800K$ packages, while more than 500 are added every day. Packages published to the registry must contain a `package.json` file. The main goals of the `package.json`[7] file are:

1. to list the packages that the project depends on.
2. to allow specifying the versions of a package that the project can use via semantic versioning rules.
3. to make one's build reproducible, and therefore much easier to share with other developers.

Among other things, this file holds information, such as the package name and version, its description, the authors, keywords, license, repository, and more.

Besides the dependencies, the `package.json` file contains a devDependencies list that accounts for packages used while developing the software project and not included in the "production" build of the product. Most often, the npm packages are associated with a GitHub repository, which contains the source code under development and other information (issues, commits, stars, forks, watchers etc.). All the above hold true for node.js applications as well. That is, end-user applications that too have dependencies and devDependencies and allow us to see package usage from an application developer viewpoint.

3 METHODOLOGY

The target of our methodology is to distill the knowledge found in the `package.json` files of the npm registry, with respect to their declared dependencies. To do so, we parse them and extract the dependencies and devDependencies, the description and keywords that denote their functionality, along with certain meta-data regarding package popularity.

We construct four networks in order to capture developers' practices about how they combine them: one that reflects the relationships between dependencies and one for devDependencies. This is performed twice one for npm packages and one for node.js applications. Upon creating the networks, we employ cluster detection algorithms in order to find communities of packages that are often used together and apply information retrieval techniques to the keywords and description fields in order to assign keywords of functionality to the communities. Last but not least, we try to identify if employing frequently used dependencies has any impact to the popularity of the produced package in terms of GitHub stars or package downloads.

3.1 The Dataset

In such big open databases like npm, the quality and usefulness of the packages usually follows a Power Law[8] or the Pareto Principle[9]. So in order to mine useful, quality packages, we augmented each package with its monthly download count gathered from npms.io[10] and the GitHub stars from the repository mentioned in the `package.json` file.

Using popularity thresholds of $5,000$ monthly downloads and 70 GitHub stars, we extracted $8,732$ packages. The aforementioned thresholds were not arbitrarily selected, but were investigated using the elbow method based on the number of packages that satisfy them. This number is exponentially increasing when we further decrease those thresholds. For each package (recipe) we kept the name, its dependencies (ingredients), its devDependencies (another type of

[6] https://www.npmjs.com/

[7] https://docs.npmjs.com/getting-started/using-a-package.json

[8] https://en.wikipedia.org/wiki/Power_law

[9] https://en.wikipedia.org/wiki/Pareto_principle

[10] https://api-docs.npms.io/

ingredients), the keywords and the description fields found in the `package.json` file, augmented with its monthly downloads and the GitHub stars of the repository declared in its `package.json` file. We only kept the name of the package and the names of its dependencies and didn't consider any declared semantic version. As for the applications, by crawling GitHub, we downloaded `package.json` files that were contained in the root folder of repositories with more than 70 stars and their package names did not exist in the npm registry. Upon applying the aforementioned methodology, we gathered 13,884 application `package.json` files. The value of the threshold in the number of stars was chosen again using the elbow methods as shown in Figure 1[11].

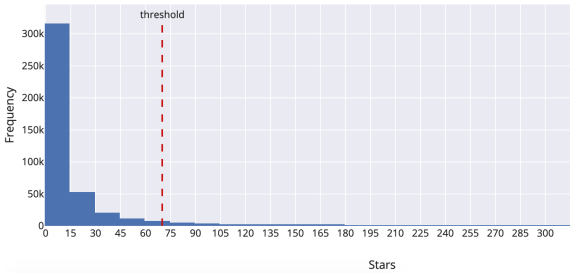

Figure 1: Choosing the star threshold visually.

We then generated a list sorted by frequency of dependency occurrence in packages, and selected the top 1000 most frequent dependencies as our final dependency list. In Figure 2 one can see the top 10 most frequent dependencies, with `lodash` making an appearance in 8.1% of the 8686 packages. These dependencies also accounted for 16.8% of dependency entries in the dataset. For devDependencies, dependency `mocha` appeared in 2558 packages (29.4%).

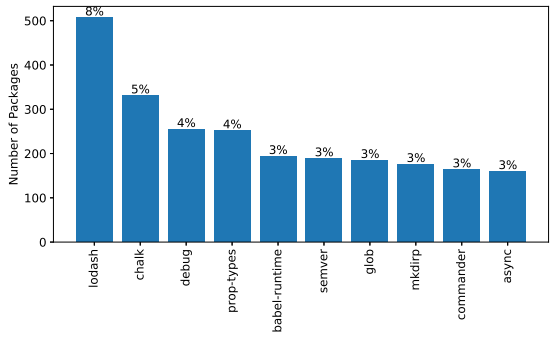

Figure 2: Top dependencies of high quality packages.

[11]The dataset with the `package.json` information of the retrieved packages and applications can be found in https://github.com/AuthEceSoftEng/icsoft-2019-npm-recipes.

3.2 Package Dependencies Complement Network

We constructed a dependency complement network based on the Pointwise Mutual Information (PMI) criterion, defined on pairs of dependencies (a,b) as follows:

$$PMI(a,b) = log\frac{p(a,b)}{p(a)p(b)} \quad (1)$$

where $p(a)$ and $p(b)$ denote the frequency of packages containing a and b, respectively, and $p(a,b)$ denotes the frequency of packages where a and b co-occur. More specifically:

$$p(a,b) = \frac{\#\ of\ packages\ containing\ a\ and\ b}{\#\ of\ packages} \quad (2)$$

$$p(a) = \frac{\#\ of\ packages\ containing\ a}{\#\ of\ packages} \quad (3)$$

$$p(b) = \frac{\#\ of\ packages\ containing\ b}{\#\ of\ packages} \quad (4)$$

The PMI is the probability that two dependencies occur together against the probability that the same two dependencies occur separately. A high PMI expects dependencies to occur together far more often than by chance. After calculating PMI for the top 1000 dependencies in the dataset, we kept only dependencies with high PMI, i.e. a value of more than 6 (maximum PMI was 10.77 between `d` and `es5-ext` and minimum PMI was -3.76 between `prop-types` and `mkdirp` packages). Figure 3 depicts the dependency complementary network[12].

Upon examining Figure 3 without the complementary colors, it is obvious that the communities are not clearly visible through plain visualization. In order to make our approach systematic, we have employed the Girvan-Newman network clustering algorithm (Girvan and Newman, 2002), a hierarchical method used to detect communities, by progressively removing edges from the original network. The top-7 in size (number of packages belonging to a community), distinct communities are colored in Figure 3. Each one of them represents a population of more than 2% of the nodes.

We use the members of the formulated communities in order to extract their domain of use. Towards this direction, we apply information retrieval techniques on the meta-data of `package.json` files.

[12]Graph visualizations and network analyses were performed using the Gephi open source graph visualization platform (Bastian et al., 2009).

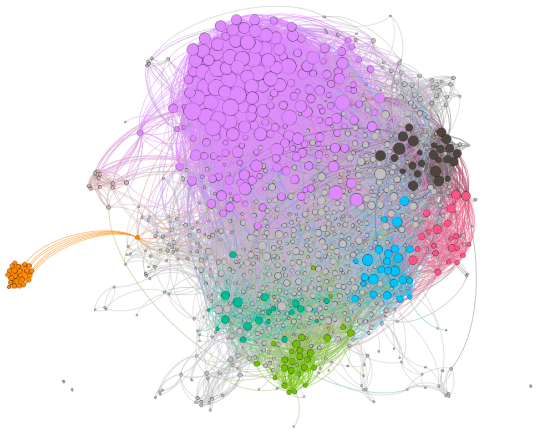

Figure 3: Dependencies complementary network. Larger nodes correspond to nodes with higher degree (nodes with more connections to other nodes).

In specific, we use the tfidf (term frequency − inverse document frequency) algorithm, which quantifies how important a word is to a document in a corpus (Salton and Buckley, 1988).

Every cluster corresponds to a corpus, while its members (packages) correspond to documents. Each document contains the name (split in keywords on dashes and slashes), description and keywords of a `package.json` file. Upon removing all stop words and the keywords `npm`, `node`, `js`, `javascript`, `package`, `plugin`, we employ tf-idf for each cluster and compute the significance of each word as the average of the corresponding values included in the term-document matrix. Finally, we apply the elbow method on the calculated significance values to extract the top keywords that describe the domain. Figure 4 visualizes the elbow method towards deriving the top 5 keywords for one of the communities, while Table 1 depicts the keywords for each community.

Table 1: Keywords of function of complement dependency communities with more than 2% support

Com.	Keywords
C1	webpack, babel, loader, css, module
C2	http, methods, content, parse, utility
C3	stack, line, string, cli, ansi
C4	opcua, sdk, iot, internet, module
C5	lodash, exported, method, module modularized
C6	simple, authentication, session, web, zip
C7	d3, time, format, module, data

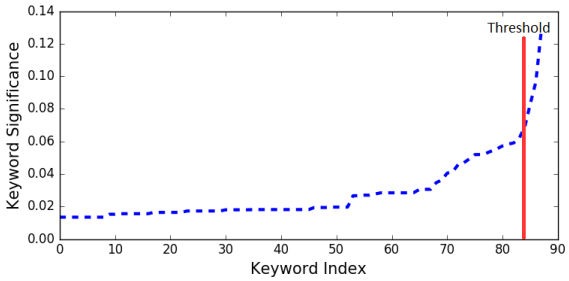

Figure 4: Decision boundary for the keywords to keep.

The main communities that accounted for more than 2% of ingredients are those of: general node.js packages related to webpack and babel (C1), packages related to web development (C2), cli related recipes that involve string manipulation (C3), a specialized community revolving around a node.js IoT framework, `opcua` (C4), the lodash community (C5), the community of tools about web plugins for authentication of session manipulation (C6) and the community around `d3` development (C7).

In addition, for each package, we calculated the minimum, average and maximum pairwise PMI between dependencies, in order to check if complementary dependencies would yield a better downloads count or GitHub star count. We have only found statistically significant correlation (statistical significance 0.043 with a p-value of 0.003) between the monthly downloads count and the minimum PMI. This may suggest that having at least two complementary dependencies could boost the package's popularity prospects. On the other hand, uncommon dependencies used together do not cause any harm.

3.3 Package Development Dependencies Complement Network

We repeated the procedure for development dependencies of packages. Again, after calculating the PMI for the top 1000 devDependencies, we kept only dependencies with a PMI of 6. The maximum PMI $= 9.92$ was found between the `@babel/plugin-proposal-do-expressions` and the `@babel/plugin-proposal-logical-assignment-operators` packages, while the minimum PMI $= -5.34$ was found between the `@babel/preset-env` and the `babel-cli` packages). A high PMI value accounts for packages that probably have been are separated to increase modularity. Figure 5 shows the devDependencies complementary network for packages. Table 2 presents the keywords for the top-10 development dependency communities identified.

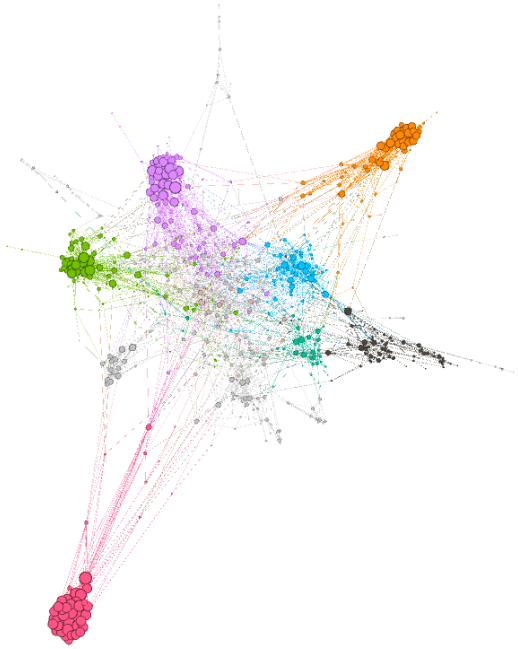

Figure 5: DevDependencies complementary network.

As for statistical correlations, minimum PMI was found positively correlated with downloads, maximum PMI was found positively correlated with stars and negatively correlated with downloads, while average PMI was found positively correlated with stars.

Table 2: Keywords of function of complement development dependency communities with more than 5% support.

Com.	Keywords
C1	loader, vue, webpack, css, eslint
C2	eslint, babel, flow, cli, react
C3	gulp, gulpplugin, stream, git, coverage
C4	grunt, gruntplugin, karma, css, files
C5	typescript, karma, loader, reporter, copy
C6	ember, cli, addon, eslint, sass
C7	tslint, react, prettier, rules, file

In devDependencies the communities seem easier to distinguish, both in terms of the visualization and in terms of keywords: the vue development related dependencies (C1), the react ones (C2), the gulp automation ecosystem (C3), the grunt automation ecosystem (C4), development packages related to typescript (C5), the ember web development platform ecosystem (C6), and a community related to typescript, linting and react (C7). This is probably due to the fact that devDependencies are usually added for the same reasons: task automation, transpiling, compiling, bundling, testing, linting etc., whereas production dependencies follow the application domain of the developed package.

3.4 Application Complement Networks

We repeated the same procedure for dependencies found in node.js applications that too have a `package.json` file, although not uploaded in the npm registry. We crawled 13822 `package.json` files using the GitHub search API. Dependency `react` was the top package in appearances found in 2414 applications and `eslint` was the top appearing package in devDependencies found in 3141 applications. Furthermore, the top PMI for dependencies was achieved by pair `grunt-sails-linker` and `include-all` with 9.67, and pair `@babel/plugin-proposal-nullish-coalescing-operator` and `@babel/plugin-proposal-pipeline-operator` with 9.36 for devDependencies. Top pairs exhibit the micropackaging approach, where developers split features into smaller and smaller packages performing simpler and simpler tasks. The lowest PMI were achieved by the pairs `angular-forms-react-dom` with -5.72 for dependencies and `eslint-plugin-import-grunt-contrib-jshint` with -6.32 for devDependencies. These values are explainable as one would use `angular` or `react` for web development, and `eslint` of `jshint` for linting.

Figure 6 and Figure 7 depict the complementary networks for node.js application dependencies and devDependencies, respectively. From the figures one can observe that communities are easier to distinguish in applications than in package complementary networks, since applications are usually developed for a specific sector and use specific plugins, while packages are as broadly applicable as possible in order for the to be as reusable as possible.

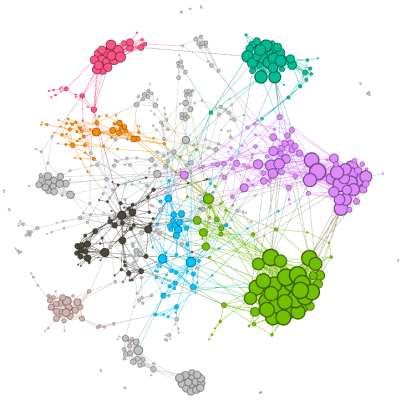

Figure 6: Dependencies complementary network for applications.

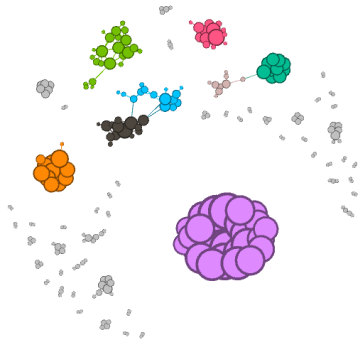

Figure 7: DevDependencies complementary network for applications.

Table 3 depicts the 7 largest communities identified for application dependencies and Table 4 for application devDependencies. For the first case we have: general web development utilities like `webpack` and `babel` (C1), react-ethereum applications (C2), react web applications with graphql (C3), desktop applications developed with electron (C4), grunt automation ecosystem dependencies (C5), gulp automation ecosystem dependencies (C6) and mobile applications developed with the cordova framework (C7). For the devDependencies case we have: ember related application development (C1), grunt automation for angular development (C2), babel helper packages (C3), react development utilities (C4), gulp automation (C5), angular related application development (C6) and even more application development utilities for loading modules dynamically (C7).

Table 3: Function keywords of application dependencies communities with more than 5% support.

Com.	Keywords
C1	webpack, loader, babel, css, eslint
C2	react, ethereum, component, library, utility
C3	react, graphql, apollo, json, link
C4	electron, file, windows, module, app
C5	gruntplugin, contrib, files, grunt, express
C6	gulp, browserify, gulpplugin, require transform
C7	cordova, android, animations, whitelist ionic

3.5 Graph Analysis

Last but not least, in an effort to further validate our results we also examine the networks from a pure graph analysis perspective. Towards this direction, we compute the degree, the closeness centrality and the betweenness centrality metrics.

Table 4: Function keywords of application devDependencies communities with more than 4% support.

Com.	Keywords
C1	ember, cli, addon, loader, eslint
C2	karma, grunt, angularjs, gruntplugin angular
C3	babel, es2015, preset, eslint, markdown
C4	react, create, webpack, fetch, preset
C5	bower, html, gulp, gulpplugin, inject
C6	zones, angular (absence of more keywords)
C7	static, css, module, systemjs, build

The closeness centrality quantifies the degree to which a node is able to spread information through a graph, while betweenness centrality measures the importance of nodes in terms of maintaining the integrity of the formulated network. From a software engineering perspective, the nodes (packages) that exhibit high closeness centrality are the ones that despite being direct dependencies of a small number of packages, these packages appear to be of great importance and thus indirectly influence a large number of packages/applications. As for the betweenness centrality, the packages that exhibit high values can be considered the ones that are integral for certain combination of operations. For instance, the high betweenness centrality in a middleware package that is required for back-end testing denotes that whenever a developer wants to test the back-end of an application, it is vital to use that certain package. The graph metrics regarding the degree and the betweenness centrality for all four networks are presented in Table 5. The presented values refer to the 3 packages with the highest values for each case. Given that the values for closeness centrality are normalized, the top packages in all four networks have a value of 1 and as a result they are not included.

Upon examining the packages that have the highest values regarding both degree and betweenness centrality, the results that occur are reasonable and expected from a software engineering perspective. For instance, in the case of the package dependencies, the package `object.values` appears to exhibit the highest betweenness centrality. This is expected as this package provides an ES2017 spec-compliant `Object.values` shim (code that enables two not entirely matching components to work together) and thus is used by major leading packages. This is also evident by its number of monthly downloads, which is more than 15 million. The same applies for the findings based on the degree of packages in all four networks. In the case of package dependencies, the packages `webpack-dev-server` and `react-hot-loader`

Table 5: Packages with high valued graph analysis metrics.

Package	Degree	Package	Betweeness
Package dependencies			
webpack-dev-server	115	object.values	22865.477
react-hot-loader	115	http-proxy-middleware	8864.721
raw-loader	112	react-hot-loader	7513.229
Package devDependencies			
ember-cli-blueprint-test-helpers	43	karma-phantomjs-shim	12521.219
ember-load-initializers	41	karma-ng-html2js-preprocessor	11042.628
ember-resolver	41	jasmine-jquery	10723.319
Application dependencies			
gatsby-plugin-google-analytics	37	resize-observer-polyfill	32193.66
gatsby-plugin-catch-links	36	babel-plugin-add-module-exports	29276.75
gatsby-plugin-feed	36	@fortawesome/free-brands-svg-icons	25307.09
Application devDependencies			
broccoli-assert-rev	29	detect-port	124.16
ember-cli-dependency-checker	29	filesize	104.85
ember-cli-htmlbars-inline-precompile	29	karma-ng-scenario	95.13

appear to have the highest degree, which is reasonable as those two packages provide functionality that is vital in web development such as live reloading and real time changes in components. Consequently, given the aforementioned results, the findings based on graph analysis metrics come in harmony with the ones expected from a software engineering perspective.

4 RELATED WORK

There are several works that mine dependency networks of different languages and development platforms to understand their properties and evolution. However, as different ecosystems do not share the same properties (Decan et al., 2016) and since our focus is on the npm registry, we review here the related work only on the npm dependency network.

Wittern et al. (2016) analyzed the evolution of packages in the npm registry, their dependencies, their popularity, and the creation and adoption of new packages. What was particularly interesting and connects to our case is that at that time, 81% of the packages depended on at least one dependency, while 32.5% of them depended on 6 or more packages with a steady increase. For our snapshot, at the time of writing, we found that 61% depended on at least one dependency and 14% on 6 or more. These may be fewer packages in terms of percentages but far more in terms of absolute numbers.

In (Abdalkareem et al., 2017), authors analyzed the use of trivial npm packages (less that 35 lines of code and less than 10 McCabe's cyclomatic complexity) as dependencies in packages and applications. They found that trivial packages are increasing, while using them as dependencies is not considered a bad practice by the majority of developers, especially if they are well implemented and increase productivity.

Decan et al. (2016) try to identify the differences in software package ecosystems (CRAN, PyPI, NPM), though package dependency graphs. Based on their results, NPM is the one ecosystem that supports the extreme re-usability and micropackaging culture by following the single-responsibility principle to the package level. In (Kikas et al., 2017) authors exhibited that indeed the JS ecosystem is the fastest growing and has high inter-connectivity between packages. Finally, Bogart et al. (2016) identified through interviews that developers, in order to make sure their packages do not break and since they are not always aware of the status of their dependencies, try to limit their exposure to them and adopt "best-practice" packages.

In the context of our work, we view network analysis of npm dependencies from a completely different perspective, that of being ingredients to recipes (other packages). Moreover, our work focuses on identifying packages that work well together as dependencies in popular packages and thus can help package maintainers make their choices, through "trusted ingredients".

5 CONCLUSIONS

In this work, we have analyzed dependencies as ingredients and packages/applications as recipes, which is not far from the truth considering the high utilization of software reuse in npm packages, and the trivial packaging phenomenon. For each dependency complement network we have identified 7 big clusters used for various purposes. Moreover, we have found that using development dependencies with high average complementarity has a positive correlation (0.06) with the popularity in star counts of npm packages as a statement of "you know what you are doing".

From the applications' analysis, one can observe as main communities those of web development, mobile development and desktop development. In addition, no community with respect to data science and data processing projects was identified as those are mainly supported by the Python and R ecosystems. Other applications of such an approach could be useful in recommender software systems, for instance:

- Through information retrieval techniques developers can identify packages that work well together in a domain (for example in linting or testing) using keywords in a search engine.
- If a developer saves *packageA* as a dependency in the `package.json` file, the system could suggest to "use packageB along with packageA".
- We could use the complementary networks to calculate a metric of how well-together our dependencies fit together by summing up the PMI of all combinations between our "ingredients".

Last but not least, a common question among developer forums is which development platform to use, for example for web application development (e.g. React?, Vue?, Angular? Ember?). Such an analysis could reveal which platforms are more popular or the ones that create a closed community that can only use platform-specific packages, or even ones that are more open to connections with third-party libraries. An idea for future work would be to mine Stack Overflow in order to find packages that can substitute other packages and build package substitute networks.

ACKNOWLEDGEMENTS

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-02347).

REFERENCES

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *Proc. of the 11th Joint Meeting on Foundations of Software Engineering*, pages 385–395, NY, USA. ACM.

Bastian, M., Heymann, S., and Jacomy, M. (2009). Gephi: An Open Source Software for Exploring and Manipulating Networks. In *Proc. of the Third International AAAI Conference on Weblogs and Social Media*, ICWSM 2009, pages 361–362, Menlo Park, CA, USA. AAAI Press.

Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. (2016). How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 109–120, New York, NY, USA. ACM.

Chatzidimitriou, K. C., Papamichail, M. D., Diamantopoulos, T., Tsapanos, M., and Symeonidis, A. L. (2018). Npm-miner: An infrastructure for measuring the quality of the npm registry. In *Proc. of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 42–45, New York, NY, USA. ACM.

Decan, A., Mens, T., and Claes, M. (2016). On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, pages 21:1–21:4, New York, NY, USA. ACM.

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

Haney, D. (2016). NPM & left-pad: Have we forgotten how to program? https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/. Accessed: 2019-01-16.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 102–112, Piscataway, NJ, USA. IEEE Press.

Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523.

Teng, C.-Y., Lin, Y.-R., and Adamic, L. A. (2012). Recipe recommendation using ingredient networks. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pages 298–307, New York, NY, USA. ACM.

Williams, J. and Dabirsiaghi, A. (2014). The unfortunate reality of insecure libraries. Technical report, Contrast Security.

Wittern, E., Suter, P., and Rajagopalan, S. (2016). A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 351–361, New York, NY, USA. ACM.