A Look at the Dynamics of the JavaScript Package Ecosystem

Erik Wittern Philippe Suter Shriram Rajagopalan

IBM T. J. Watson Research Center

{witternj, psuter, shriram}@us.ibm.com

Abstract

The *node package manager* (`npm`) serves as the frontend to a large repository of JavaScript-based software packages, which foster the development of currently huge amounts of server-side `Node.js` and client-side JavaScript applications. In a span of 6 years since its inception, `npm` has grown to become one of the largest software ecosystems, hosting more than 230,000 packages, with hundreds of millions of package installations every week. In this paper, we examine the `npm` ecosystem from two complementary perspectives: 1) we look at package descriptions, the dependencies among them, and download metrics, and 2) we look at the use of `npm` packages in publicly available applications hosted on `GitHub`. In both perspectives, we consider historical data, providing us with a unique view on the evolution of the ecosystem. We present analyses that provide insights into the ecosystem's growth and activity, into conflicting measures of package popularity, and into the adoption of package versions over time. These insights help understand the evolution of `npm`, design better package recommendation engines, and can help developers understand how their packages are being used.

1 Introduction

Software ecosystems consist of software projects that are developed and evolve together in a shared environment [13]. Research on such ecosystems only recently emerged within software engineering [15]. It addresses, among other things, the analysis of the characteristics and evolution of software ecosystems, which some researchers consider to be a central subject in this field of research [26]. To that regard, some software ecosystems have been scientifically assessed, including the Maven [23], Apache [2], Gentoo [4], Ruby [11], and R [8] ecosystems. Nonetheless, a systematic literature study from 2013 found that research regarding real-world software ecosystems is lacking [14].

The study of characteristics and the evolution of software ecosystems is an end in itself, allowing to understand how certain technologies spread or why others fail. In addition, this research can also inform the design of new ecosystems and associated tooling, including technical as well as social aspects [26]. Furthermore, as software projects are seldom created in isolation, studying individual software projects often requires studying the ecosystem around them [3].

The *node package manager* (`npm`) combines a set of open source tools that developers use to describe their JavaScript packages, and specifically Node.js packages. These tools include, for example, a command line interface to create and maintain `package.json` files, which declare, among other things, the name, description, version, and dependencies of a package. `npm` furthermore features a registry that developers can use to publish their packages, making them available for others to use. Packages within `npm` may depend on each other, and software projects outside of `npm`, for example applications, may specify dependencies to packages hosted on `npm`. Since its creation in 2009, `npm` has grown rapidly to now feature over 230,000 packages (as of January 28th, 2016).

`npm` provides a complete set of various historic data points on the packages in the ecosystem. This data thus not only provides insights into the current state of the ecosystem, but also into how the ecosystem evolved over time. The diversity of available data points furthermore allows us to assess the ecosystem from multiple perspectives and to compare these perspectives. Understanding `npm` provides valuable insights into the rapid growth JavaScript and Node.js experienced within the last years. It also helps to understand how individual packages rose in popularity, prevailed, and sometimes were eventually replaced or disregarded.

This paper presents an extensive analysis of the `npm` ecosystem. We make the following contributions:

- We study the evolution of the `npm` ecosystem re-

garding growth and development activities. Our findings indicate a highly active developer community. We look at the relationships across packages and find that package dependencies have increased from 23.4% in 2011 to 81.3% in 2015, with 32.5% of packages having 6 or more dependencies. On the other hand, only 27.5% of packages in the npm ecosystem are being depended upon, indicating that developers largely depend on a core set of packages.

- We assess the notion of package popularity in npm. Our analysis considers three different popularity measures and finds that they are not substitutes for another. Rather, they can be used to depict the popularity of specific types of packages. These findings impact the design of package recommendation tools. We further assess the evolution of package popularity for one selected measure, focusing on top ranking packages as well as on sets of functional equivalent ones. We further find that new packages continuously enter the top ranks for the first time, while there are also selected packages that manage to remain popular over time.

- We study the creation of new package versions in npm, and the adoption of package versions by application developers. We find that package maintainers adopt a variety of numbering conventions and that version numbers in themselves are therefore not good predictors of package maturity. We observe that application developers are flexible when declaring dependencies, and often automatically accept minor updates. A detailed analysis of usage data for the popular express package shows that, as a consequence, up to half of all users automatically depend on the latest version when it is released.

The rest of this paper is organized as follows: we present the data at the foundation of our analyses, and how we collected it (Section 2). We then present our findings regarding the evolution npm (Section 3), package popularity (Section 4), and version adoption (Section 5). We give an overview of related work (Section 6) before concluding (Section 7).

2 Data Sources

We confine our observation period from October 1st 2010 to September 1st 2015, and all data collected for our analyses was pruned to cover this range only.[1] We collected three data sets that provide indicators about the evolution and popularity of packages, from different sources such as the npm registry, GHTorrent [9] and the GitHub project hosting platform. Our final dataset includes 185,005 npm packages and 114,995 applications. The following subsections provide further details on our collection process.

2.1 Package Metadata

The metadata associated with a package contains information such as the dependencies on other packages, version information, search keywords associated with the package, as well as download (i.e. installation) counts. We use this data in Section 3 to analyze the relationship between packages, in Section 4 to analyze the popularity of packages, and in Section 5 to analyze the evolution of versions over a package's lifetime.

```
1  {
2    "name": "myexample",
3    "version": "1.3.1",
4    "maintainers": [
5       {"name": "Some Name",
6         "email": "me@example.org"}
7    ],
8    "repository": {
9      "type": "git",
10     "url": "https://github.com/x/myexample"
11   },
12   "main" : "myexample.js",
13   "keywords": ["Web", "REST"],
14   "dependencies": {
15     "async": "~0.8.0",
16     "express": "4.2.x"
17   },
18   "devDependencies": {
19     "vows": "0.7.0",
20     "assume": ">=2.5.2 <3.0.0"
21   }
22 }
```

Listing 1: npm metadata file package.json for a fictitious package myexample. Some fields are omitted for brevity.

Metadata files. Each npm package has an associated metadata file called package.json as exemplary shown in Listing 1. We obtained the metadata files corresponding to every version of every package during the observation period. This data is publicly available from the npm registry[2]. Every package is uniquely identified by the name field (line 2). The package.json file contains version information (line 3) specified using semantic versioning [22], the source repository associated with the package (line 8), keywords associated with the package (line 13), and information about developers maintaining the package (line 4). npm distinguishes between two types of dependencies: *dependencies* (line 14) specifies the set of runtime dependencies, and *devDependencies* (line 18) specifies the set of modules required by the

[1] Despite npm being created in 2009, we were able to obtain reliable data about packages only starting from October 1st 2010.

[2] source: https://registry.npmjs.org/-/all

package developer for her development and testing purposes.

Download counts. Every time a package is installed from npm (whether it is for production, testing or development), its download count is incremented. npm publishes the download data on a daily basis through its web API[3]. We obtained the download figures for each package in npm (irrespective of its version) for every day within the observation period.

2.2 Applications using npm Packages

To obtain a sample of Node.js-based applications that use npm packages, we turned to open source project hosting platforms. Specifically, for the purposes of this study, we targeted various types of applications and tools hosted on the popular GitHub platform. We use the information collected from these applications in Section 4 to assess the popularity of packages as determined by their usage, and in Section 5, to measure adoption ratios of specific package versions.

Since GitHub hosts software projects in different languages, we first needed to obtain a list of GitHub projects written in JavaScript. For this purpose, we analyzed the data set from the GHTorrent [9] project, obtained in March 2015. The dataset contained information on 245,389 JavaScript projects from GitHub.[4]

We then eliminated to the extent possible npm packages that are themselves hosted on GitHub and that may thus be contained in the GHTorrent dataset. The package.json file described earlier optionally includes a repository field (Listing 1, line 8) with a link to the source repository. We used this information to filter out npm packages from our application dataset, leaving us with a list of 237,349 JavaScript projects.

We further refined our application dataset to include only software projects specifying dependencies to npm (typically but not exclusively Node.js projects); using GitHub's web API, we identified such projects by looking for the existence of a package.json file. Our final list of GitHub applications using npm packages consisted of 114,995 software projects. For each project in this final list, we cloned its repository and analyzed the commit history pertaining to the package.json file, to retrieve every different version together with the timestamp at which it was committed. We obtained a total of 4,222,864 versioned files, indicating that on average a project had 36.7 commits affecting package.json (the median was 12, the maximum 1,203).

[3] source: https://api.npmjs.org/downloads/range/2010-01-01:2015-09-01/packageName

[4] Note that the date at which this dataset was obtained has no bearing on our observation window. We used this dataset to simply obtain a sample set of JavaScript-based projects in GitHub.

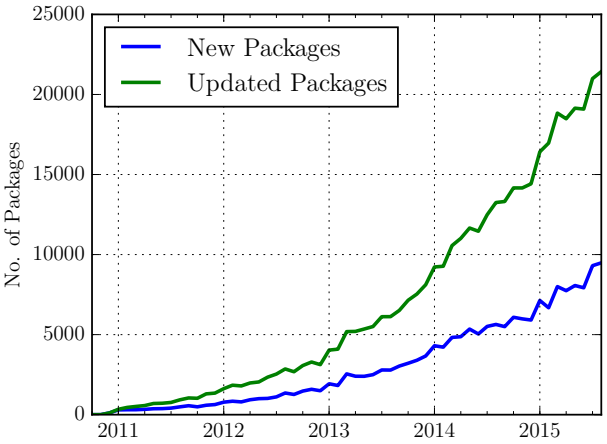

Figure 1: Packages created per month and packages updated per month. Packages with multiple updates in a month are counted only once.

While the number of projects in our GitHub data set may be a small subset of the total number of JavaScript-based projects in GitHub that use npm packages, we believe that we have a representative sample set for our analyses. Looking at this sample set, we found that the oldest date at which projects in this set were either created or first updated dated as far back as March 2010, even earlier than our observation window.

3 Ecosystem Evolution

The npm package repository was created in 2009. Over the last six years, the software repository has evolved rapidly and currently hosts over 230,000 packages. We investigate the evolution of this ecosystem over this period and look for signs of stagnation. Stagnation indicates that the community involvement has slowed down, while continued signs of growth and activity indicates that there is increasing adoption and contributions by the developer community. To characterize growth and activity, we look at the number of new packages added to the repository over the observation period, the number of packages that were updated, and the dependencies among packages.

In Figure 1, we show the growth in number of packages that are being added to npm every month, and the number of packages that are being updated per month. Broadly speaking, we find that the developer community around Node.js has been steadily increasing over the last 6 years, as evidenced by the increasing number of packages being created every month in the npm repository. In addition, the community is also quite active in terms of maintaining the packages hosted in the repository, as indicated by the two-fold increase in the number of packages being updated every month.

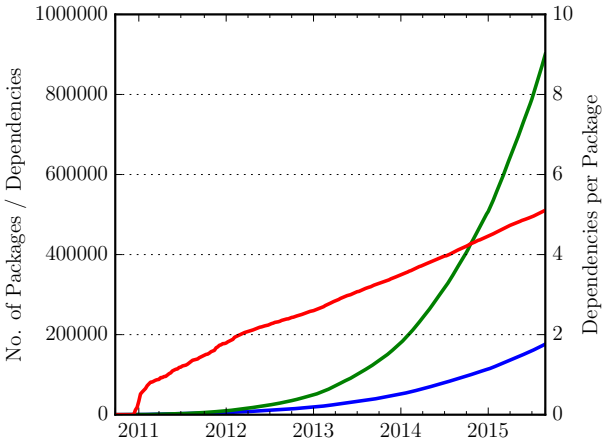

Figure 2: The `y1-axis` on the left shows the growth in the number of packages in `npm` (blue line) and the dependencies among packages (green line) over time. The `y2-axis` on the right shows the average number of dependencies per package (red line) over time.

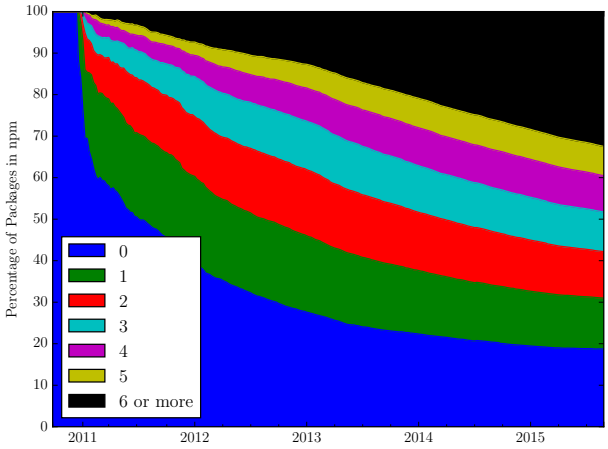

Figure 3: Characterizing packages in `npm` by their dependencies on other packages.

We then compare the growth in the total number of packages over time with the dependencies, i.e., relationships, between packages. The dependency measure per package is the sum of number of its dependencies mentioned in `dependencies` and `devDependencies` fields in the `package.json` file corresponding to each package. As shown in Figure 2, not only is the `npm` ecosystem growing superlinearly in terms of the number of packages and dependencies, the relationship among packages is also growing at a much higher rate, indicating that the packages are depending more and more on each other. We confirm our observation by plotting the average number of dependencies per package (red line) in the same figure. On average, a package in `npm` had approximately 4-6 dependencies on other packages in late 2015, compared to just one dependency in early 2011.

To further understand the dependency relationship between packages, we constructed a directed graph, where packages form the vertices and directed edges between vertices represent the dependency between the two packages. The out degree of a vertex indicates the number of dependencies of the package represented by the vertex, while the in degree represents the number of packages that depend on the given package.

Figure 3 shows the distribution of the out degrees across all packages over time. The number of packages having one or more dependencies has increased from 23.4% in January 2011 to 81.3% by end of August 2015. Specifically, there has been a steady increase in the number of packages with 6 or more dependencies, starting with 0% in January 2011 and reaching 32.5% by end of August 2015.

Given the increasing number of external dependencies per package, we tried to understand whether such dependencies were spread equally across all packages or if they were confined to just a limited set of packages. To answer this question, we look at Figure 4, showing the distribution of in degrees across all packages. At the end of August 2015, 72.5% of packages had no incoming dependency, i.e., they had no dependent packages, while only 4.9% of packages had 6 or more dependents, up from 1.1% in January 2011. Such uneven distribution of package dependencies has been previously observed in other software ecosystems as well [18]. In Section 4, we investigate the ranking of packages within and outside the ecosystem to shed light on the dynamics of package popularity in the `npm` ecosystem.

Takeaways. We find that the `npm` ecosystem continues to grow in terms of the number of packages it hosts. At the same time, the number of packages being updated monthly has also grown two-fold, indicating that the developer community remains quite active in terms of maintaining their packages. Looking at the relationships between packages, we find there is an increasing amount of dependency across packages, with 81.3% of them depending on at least one package and 32.5% of them depending on 6 or more packages. However, the proportion of packages that are being depended upon is only 27.5% of overall packages in `npm`, indicating that package dependencies exhibit a power law distribution, as has been observed by prior research on software-related artifacts and ecosystems [18, 12].

4 Package popularity

In this section, we analyze the popularity of packages in the `npm` ecosystem and the evolution of package popularity over time. Popularity may be a function of different

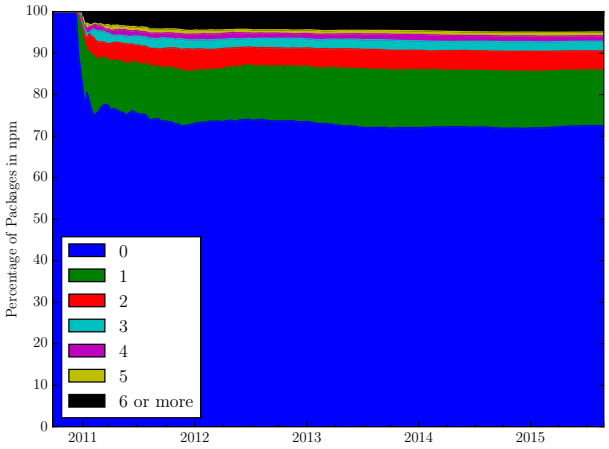

Figure 4: Characterizing packages in npm by the number of packages depending on them.

measures, either individual ones or combinations. In our analysis we focus on the following three measures:

1. The npm **rank** reflects the PageRank [6] of a package within the npm dependency graph. Packages specify dependency relationships to each other as described in Section 2.1. By applying the iterative PageRank algorithm on the resulting dependency graph, a package obtains a high *PageRank value* if it is depended upon by many packages that themselves have high PageRank values. Ordering packages by their PageRank value, we assign the resulting npm rank to them. The npm rank thus denotes the relative importance of every package within npm. PageRank is commonly used to rank software artifacts, for example Java components [10] or JavaScript packages [21]. In our measure for the npm rank, we consider both the dependencies and development dependencies that a package specifies. When performing the PageRank algorithm, we applied a damping factor of 0.85 and stopped iterations once the total cumulative change in the values of all vertices was below 10^{-6}.

2. The **download rank** reflects the number of times a package was downloaded within one month leading to the considered date. As described in Section 2.1, the download figures are published on a daily basis per package by npm and were thus aggregated by us. By ordering packages by their download figures, we derive the download rank.

3. The GitHub **rank** reflects, for a given day, the number of dependencies on a package as stated in the GitHub projects we collected (c.f. Section 2.2). As in the case of download numbers, we derive a ranking of packages from ordering the counts.

	npm pagerank	Downloads	GitHub
npm pagerank	-	0.385	0.445
Downloads	0.385	-	0.567
GitHub	0.445	0.567	-

Table 1: Spearman rank correlation coefficients between package popularity measures.

Unless specified explicitly, all measures of popularity were computed based on data as of September 1st 2015.

Popularity measures can be used for package recommendation, or source code recommendation more generally, which is a common goal of recommendation systems in software engineering [7]. In many existing systems, like npm's own search interface [19], the "npm Discover" tool [20], or the "npm packages PageRank" tool [21], users enter search terms to specify requirements and thus narrow down the packages to consider. The remaining packages are then ranked for users based on a single popularity measure or a combination of them. For example, "npm packages PageRank" relies on PageRank values, while "npm Discover" considers usage from GitHub projects.

4.1 Relationships between Measures

A first question to answer is whether the considered measures report the popularity of a package in a consistent way. To this end, we calculated the Spearman's rank correlation coefficients between them as illustrated in Table 1. The input data used to calculate the correlations only covers a subset of all packages: The npm rank for packages that are not depended upon at all cannot be determined because they all share the same, minimum PageRank value. Similarly, the GitHub rank of packages that are never depended upon cannot be determined. Thus, when calculating the correlation coefficients in Table 1, we consider only packages with assigned ranks for both of these measures. All packages featured at least one download, so we did not have to dismiss any package based on this measure. The low correlation values presented in Table 1 show that the three popularity measures do not generally depict popularity in the same way and can thus not necessarily be substituted for another.

To assess the relationship between the measures in more detail, Figure 5 plots the differences in ranks of packages for every combination of popularity measures. Again, for every comparison, we consider only packages with assigned ranks in both measures. As a consequence, each comparison is done over a different number of packages. The *y* axis in each of the top three graphs in Figure 5 ranges from minus to plus the number of compared packages. Limiting the axes this way allows the three

graphs to be compared to each other with regards to the shape of the distribution. The three histograms at the bottom of Figure 5 illustrate the distribution of the differences in popularity measures of packages.

As can be seen, all three comparisons result in a relatively normal distribution of differences in popularity ranks. On the one hand, this result may seem predictable, given the large size of data points we considered. On the other hand, all comparisons exhibit extreme cases where packages rank considerably higher in one measure as compared to the other and vice versa.

Takeaways. Different package popularity measures produce different outcomes. All comparisons of the three measures considered in this work reveal packages that perform strongly in the first measure and poorly in the second as well as packages for which the opposite is true. This finding has implications on package recommendation tools making use of PageRank within npm, e.g., [21]. While their recommendations may be useful for package developers, they might not be suited for application developers.

4.2 Differentiating Package Types

Section 4.1 revealed that there are packages with significantly different ranks regarding the different popularity measures. To gain insight into the nature of these packages, we now focus on two measures, npm rank and the GitHub rank. We dismiss download ranks because we cannot with certainty explain their origin or exclude influences, for example, through web miners or crawlers.

Focusing on the npm rank and GitHub rank, we propose to explain their differences by defining the following *types* of packages:

- **End user packages** are used commonly in applications, but not necessarily by other packages. Examples are database drivers like bookshelf (GitHub rank: 399, npm rank: 2950), or authentication libraries like passport (GitHub rank: 65, npm rank: 718). We expect end-user packages have high GitHub ranks, but a comparatively low npm ranks. Given that many recommendation systems filter down packages based on user-input, these exemplary differences in rank can make the difference between a package being displayed in the top results or not. For example, among all packages in npm with the keyword "authentication" assigned, passport ranks 1st based on the GitHub rank, but only ranks 3rd based on the npm rank.

- **Core utility packages** are mostly used by other packages but seldom by applications outside of npm. Examples are packages providing low-level functionalities like ieee754 (GitHub rank: 37287, npm rank: 2258) for reading/writing floating point numbers to buffers or is-relative (GitHub rank: 20299, npm rank: 434) for detecting relative package dependencies. We expect core utility packages have high npm ranks, but low GitHub ranks.

In order to assess whether we find evidence for this classification of packages, we look further into the nature of packages with highly different ranks. Packages in npm can be categorized by any number of *keywords*, which package developers may assign, as shown in Listing 1, line 13. We assess the keywords assigned to the 1000 packages with the highest npm and GitHub rank. We count the appearances of every observed keyword and calculate the Pearson correlation coefficient between these counts. The resulting correlation coefficient of 0.823 is relatively strong.

Thus, to look into more detail, we focus our analysis on those packages that reveal the highest difference in npm rank as compared to the GitHub rank. Table 2 shows the keywords with the highest difference in count in "npm strong" packages as compared to "GitHub strong" packages. "npm strong" denotes the set of the 1000 packages that perform comparatively the best in npm while performing the worst in GitHub. On the other hand, "GitHub strong" denotes the set of 1000 packages that perform comparatively the best in GitHub while performing the worst in npm. As we can see, the keywords most unilaterally used to describe "npm strong" packages relate to low-level capabilities such as dealing with arrays, buffers, or strings. These keywords are assigned to core utility packages, as introduced above.

In contrast, Table 3 shows the opposite, that is, the keywords with the highest different in count in "GitHub strong" packages as compared to "npm strong" packages. As we can see, the keywords most unilaterally used to describe "GitHub strong" packages are related to capabilities typically used in application development. grunt and gulp are plug-in-supporting tools to build applications. express is a server-side web application framework, and react is a library used to render views. These keywords are assigned to user packages, as introduced above.

Takeaways. We assumed that there are qualitative differences between packages with either high npm ranks and low GitHub ranks or vice versa. Our analysis of the keywords used uniquely to describe these packages confirms this suspicion. We find indications for both core utility packages and end user packages. This finding strengthens our above takeaway that package recommendation requires choosing an appropriate popularity measure depending on the intended outcome.

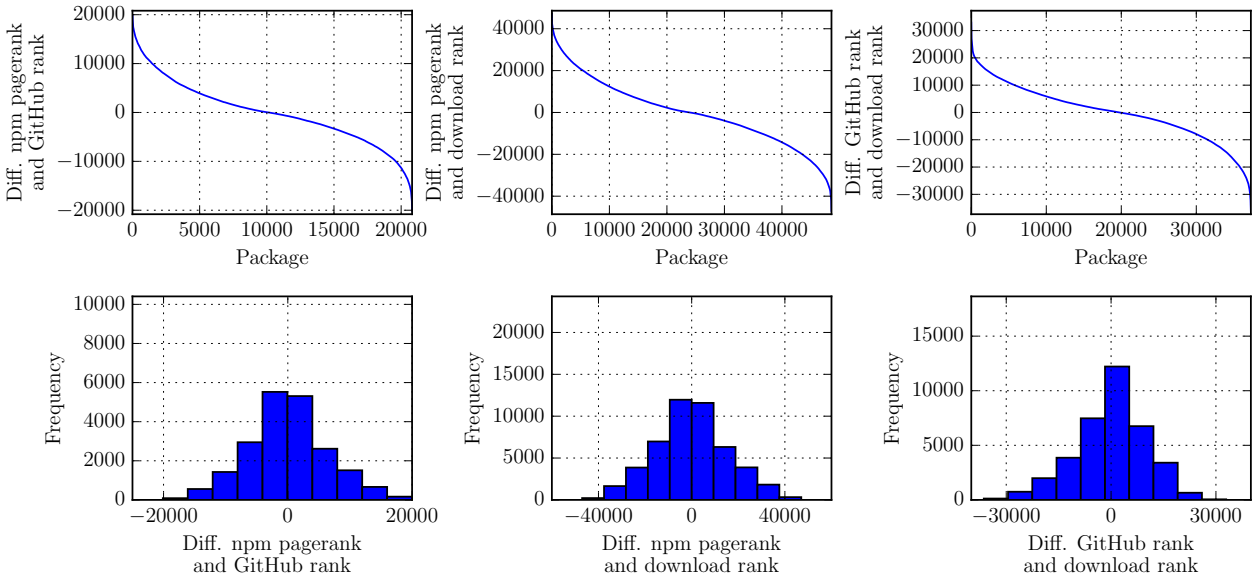

Figure 5: Differences in ranks between popularity measures.

Keyword	"npm strong"	"GitHub strong"	Diff.
util	35	3	32
array	18	3	15
buffer	16	2	14
string	20	6	14
file	21	7	14

Table 2: Count of keywords ordered by their difference in count between describing "npm strong" and "GitHub strong" packages.

Keyword	"GitHub strong"	"npm strong"	Diff.
gruntplugin	92	24	68
gulpplugin	54	9	45
express	34	5	29
react	31	2	29
authentication	22	1	21

Table 3: Count of keywords ordered by their difference in count between describing "GitHub strong" and "npm strong" packages.

4.3 Evolution of Popularity

Having established differences in the meaning of different popularity measures, we now focus on one measure and assess how package popularity evolves with regard to it over time. The npm rank denotes how central a package is to the npm ecosystem. To obtain npm ranks over time, we start with the complete dependency graph as of September 1st 2015. Using the date annotations between all edges in the graph, we then create a filtered version of that graph for every week between September 1st 2010 and September 1st 2015. For every one of the resulting 257 graphs, we calculate the PagerRank value of every package present at that point in time and assign the npm rank thereupon.

4.3.1 Identifying Top Packages

An immediate question to answer is which packages perform the best over the whole existence of npm. One way to answer this question is, as illustrated in Figure 6, to determine the packages with the lowest mean npm rank (i.e., the highest ranked packages). We use the geometric mean for this purpose as it is less prone to outliers as compared to the arithmetic mean. We limit the y axis in Figure 6 from 1 to 100 for readability.

The top packages illustrated in Figure 6 are diverse in nature. `should` and `nodeunit` are tools for testing, `uglify-js` is used to minimize and obfuscate (client) code, `coffee-script` is a language compiling down to JavaScript, and `underscore` provides a set of generic utility functions.

While the packages presented in Figure 6 stand for long running success, a more fine grained analysis is needed to gain insights into momentary package success. Figure 7 breaks down the 5 packages with the lowest mean npm rank per year from 2011 to 2015. We see that, for example, `coffee-script` ranks in the top 5 only in 2011 and 2012. Other packages, like the file system utility `glob` or the evolution of testing tool `tape` (built on top of `tap`) only make it into the top 5 in later years, i.e., 2014 and 2015. Interestingly though, the npm rank of packages in the top 5 remains relatively stable, especially from 2013 on.

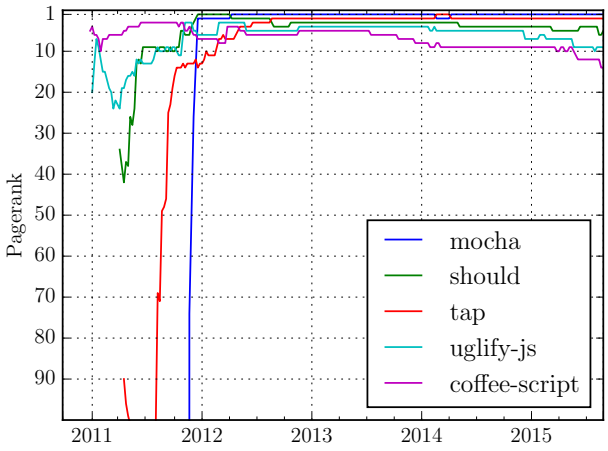

Figure 6: npm ranks (PageRanks) over time of the 5 packages whose npm ranks have the lowest geometric mean.

4.3.2 Top Package Dynamics

As there seems to be little dynamics in the yearly top 5 packages as shown in Figure 7, we aim to determine how many packages manage to enter top npm ranks over time. Table 4 depicts the overall number of packages entering top 10, top 100, and top 250 npm ranks per year. While Table 4 indicates declining numbers of new packages entering top ranks, it also shows that there is still a considerable amount of them, even in 2015.

Year	Top 10	Top 100	Top 250
2011	15	180	445
2012	5	46	142
2013	4	36	116
2014	2	44	99
2015	3	27	60

Table 4: Number of packages per year entering top npm ranks for the first time.

4.3.3 Comparing Popularity of Similar Packages

Another capability arising from being able to determine npm ranks over time is to compare selected packages against each other. Figure 8 shows the npm ranks of selected utility packages over time. These utility packages typically provide a broad set of capabilities like improving convenience in dealing with data types like objects, arrays, or strings. As Figure 8 shows, one package in particular, `underscore`, has been introduced early in npm's history[5]. Since its release, `underscore` held a top npm rank. Nonetheless, multiple competitors have entered npm since then, some of which directly position themselves as alternatives to `underscore`. For example, `lodash` evolved from a fork from the `underscore` project and kept API compatibility[6], and `lazy.js` proclaims to be "[...] similar to `underscore` and `lodash`, [...]"[7]. The selection of presented utility packages is based on a web search for `underscore` competitors. While various `underscore` competitors have entered npm throughout its history, most of their npm ranks are on a declining trajectory since 2014, even if they exhibited growth before. The one exception is `lodash`, which has gradually risen in npm rank since its introduction and was able to surpass `underscore` for good in May 2015 according to our data.

Takeaways. Calculating npm ranks (i.e., PageRanks) over time allows to identify well-performing packages across the life-cycle of a software ecosystem. They can also be used to determine how dynamic or static the ranks of the most popular packages are. For npm, we find that, while decreasing, there is still considerable amount of change in the top ranks. Nonetheless, comparing functionally similar packages indicates that high popularity for some packages may be long lasting. In our example, we find that most utility libraries are declining as compared to the dominant `underscore`, except competitor `lodash` which positioned itself well by providing API compatibility.

5 Version Numbering & Adoption

While the previous sections treat each npm package as a single entity, in this section we consider some questions that arise from studying the diversity and evolution of version numbers of given packages, as well as the usage of package versions by application developers.

As in most software repositories, npm artifacts (packages) are *versioned* to indicate their evolution and to let developers rely on older or newer features as desired. By convention, version numbers follow the *semantic versioning* format [22]: three dot-separated numbers indicating, respectively, the *major*, *minor*, and *patch* versions of an artifact.

Semantic version numbers are lexicographically ordered, i.e. version $m_1.n_1.p_1 > m_2.n_2.p_2$ if and only if $m_1 > m_2 \vee (m_1 = m_2 \wedge n_1 > n_2) \vee (m_1 = m_2 \wedge n_1 = n_2 \wedge p_1 > p_2)$. In order to visually present data where one axis represents a spectrum of version numbers, we often need to convert these triples into a single value. Because none of the version components have upper bounds, we cannot simply consider them to be fractional parts with

[5] It first ranks at January 14th 2010 in our data. However, its earliest commit on GitHub stems from October 25th 2009 and a first `package.json` was added in version 0.2.0, which was released on October 28th 2009, c.f. `https://github.com/jashkenas/underscore/commits/master`

[6] `http://kitcambridge.be/blog/say-hello-to-lo-dash/`

[7] c.f. `http://danieltao.com/lazy.js/`

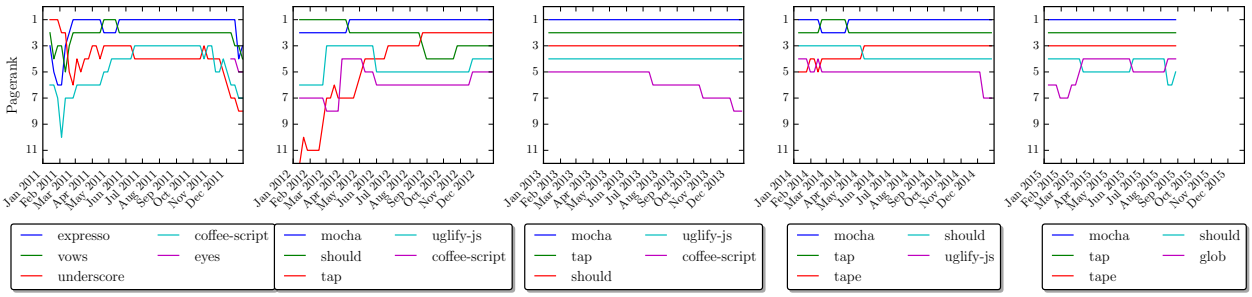

Figure 7: npm rank (PageRank) over time of 5 packages whose npm ranks have the lowest geometric mean *per year*.

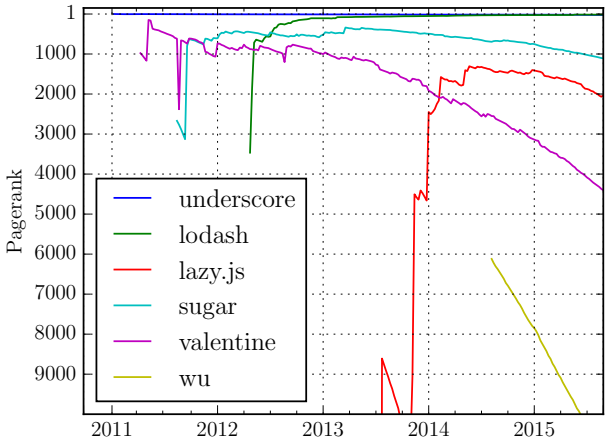

Figure 8: npm rank (PageRank) over time of selected utility packages.

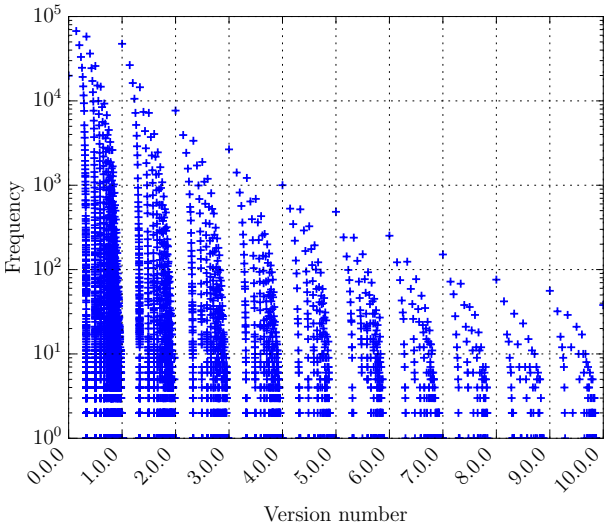

Figure 9: Frequency of version numbers (using (1)) across all npm packages.

a fixed denominator. Instead, to combine a principal and secondary component (e.g. minor and patch), we sum them up, applying to the secondary one a mapping $f : [0, +\infty[\to [0, 1[$. For this paper, we chose the function:

$$f(x) = 1 - \frac{1}{k \cdot x + 1}$$

with $k = \frac{1}{2}$ as a scaling factor.[8] In order to produce a single rational number from a semantic version triple, we apply the function twice:

$$r(m.n.p) = m + f(n + f(p)) \qquad (1)$$

With these preliminary considerations in mind, we first look at how package maintainers work with versions.

5.1 Attribution of Version Numbers

The first question we look at is the distribution of version numbers across all npm packages. Figure 9 displays this for all versions less than 10.0.0 (which adds up to 95% of all packages). As one could expect from a rapidly growing ecosystem, low version numbers dominate not only at the major level, but also within each major number. This is confirmed by looking at the frequencies of each version component in isolation (Figure 10); lower numbers are always more common, with the exception of the minor version number 9 which is more common than 8. A possible explanation is that developers would use a minor version of 9 to indicate that the next major release is "almost there", even when not all minor numbers have been used. Also not pictured in Figure 10 is the major version number 2014, which is overall the 8th most common, indicative of an ad-hoc convention of numbering versions by the year.

Having seen that small values dominate all components of the package version numbers, we can try to establish to which extent this is due to the relative young age of the ecosystem; we do this by comparing package version numbers to their age as they are released. We define as the age of package the time since its first

[8] The choice of any positive k is somewhat arbitrary. We picked this value to appropriately spread out the version numbers we observed.

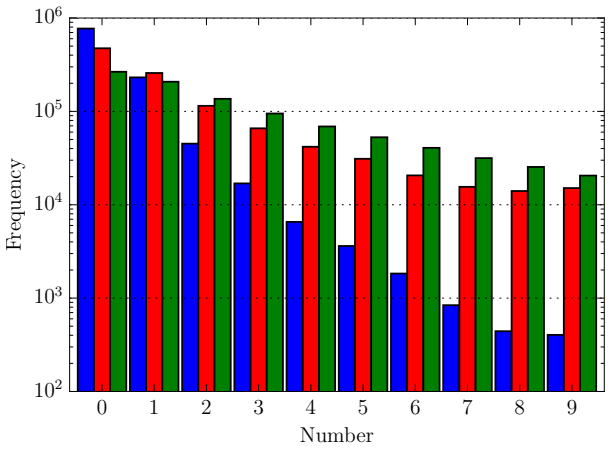

Figure 10: Frequency of individual version components for `npm` packages; blue, red and green denote major, minor, and patch numbers respectively.

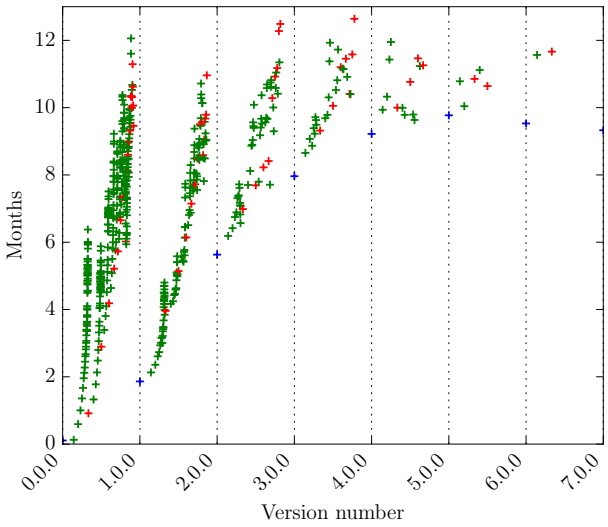

Figure 11: Average time in months after the initial release to reach a given version number (using (1)). Dataset limited to version numbers reached by at least 100 packages. Blue, red, and green denote major, minor, and patch versions respectively.

version was released. Figure 11 shows, for all version numbers for which at least 100 packages exist, the average package age at the time of release. The graph shows a combination of trends: generally, higher version numbers come later in the development of a package. However, we also observe trends *within* major version numbers. In fact, the graph shows that on average, it takes about a year for a package to reach either version 0.9.0 or version 6.1.0. We interpret this as a manifestation that package authors adopt numbering schemes that may not be strictly in accordance to the semantic versioning principle; a large number of package authors are reluctant for instance to ever release a version 1.0.0.

Takeaways. Although `npm` package authors are encouraged to follow the semantic versioning scheme, other numbering conventions have emerged, resulting for instance in a large set of pre-1.0.0 packages, irrespective of their age.

5.2 Adoption by Version Number

We now look at how developers declare dependencies on package versions, based on the data we collected from a large set of open-source applications hosted on `GitHub` (see Section 2.2). Dependencies on versions in `npm` can be declared using *queries* built with a variety of operators; the simplest cases are fixed, explicit, dependencies, where a project author requests a specific version of a package, but the author can also request, for instance, any version with a fixed major component, any version with a fixed minor component, any version within a range, the most recent version in general, etc.[9] Using historical package release and project evolution data, we have

[9] See `http://semver.npmjs.com`.

Query	Rel. freq.	Avg. # vs.
*	1×	7.59
`^n.n.n/n.x.x`	7.28×	2.36
`~n.n.n/n.n.x`	17.53×	1.66
`n.n.n`	10.73×	1.0

Table 5: Relative popularity of version dependency query types, from most permissive to most restrictive, and average number of versions to which the queries resolve within the time between two dependency updates. Data aggregated from approximately 4M version updates.

the ability to retroactively resolve these requirements and know precisely which version of a package would have been returned for which project at any time.

Towards this goal, we processed the 4M+ versions of `package.json` we obtained and recorded for each dependency the points in time at which the version query changed. This gave us 3,955,338 version update points, indicating that on average, when `package.json` is updated (which happens every 90 days on average), 0.93 dependencies are updated. Table 5 shows the relative frequencies for selected types of queries, namely requesting the latest version, allowing patch or minor updates, allowing patch updates, and requesting an exact version. The table also displays the average number of versions to which queries resolve over their life time.

For each dependency query, we then computed the set of all possible versions it can have resolved to. We obtain

this set by intersecting the time intervals of the updates to `package.json` with the release dates of the packages. We find that on average, within the lifetime of a commit, a package query will resolve to 1.88 different versions.

Finally, our data also allows us to answer a question that package developers may find crucial as they issue releases; given that many package consumers use flexible queries, what is the fraction that will obtain a new version when it is released, without changing their `package.json`. We call this measure "implicit adoption ratio", and obtain it by computing, for each package version at its release date, the size of the set of projects resolving to the latest version divided by the size of the set of projects resolving to *any* of the versions. Figure 12 shows the implicit adoption trends for the popular `express` package for building web applications. Note that the second part of the graph indicates that releases are continually issued both for the 3.*x*.*x* and 4.*x*.*x* version families. From it we get several insights: first, patch versions have higher implicit adoptions ratios than minor versions, which have higher ratios than the two major versions visible in the chart. This is explained by the tendency to adopt version queries which minimize incompatible updates. Second, as new releases come out in the 4.*x*.*x*, the implicit adoption ratio *increases*, indicating that the fraction of projects configured to accept these new releases grows over time. Finally and as a complement to the second observation, the fraction of projects implicitly resolving to the latest version in the 3.*x*.*x* family shrinks gradually and decisively over time. The last two points can be explained either by a combination of 1) the continuously growing number of projects using `express`, which tend to use the latest version when they are created (not visible on the graph), and 2) existing `express` projects that migrate to 4.*x*.*x* series when they can afford to.

Takeaways. Through declaring dependencies with queries, application developers can benefit from automated upgrades, at various levels of granularity. This mechanism is used widely in practice, and new releases, particularly patch ones, have high implicit immediate adoption ratios.

6 Related Work

Empirical analysis of software ecosystems is an important aspect of software ecosystem research as a whole [26]. Correspondingly, related work focuses on specific aspects like visualization [13], depicting ecosystem maturity [1], or how to aggregate software quality metrics [17].

Some works empirically analyze software ecosystems that evolve around a specific programming languages, as we did for `npm`. Raemaekers et al. present a crawled dataset containing basic metrics, dependencies, and changes with some aggregate statistics about `Maven`, a popular package manager for `Java` [23]. Another work runs software to identify bugs in source code of libraries shared in the same ecosystem [16]. In contrast to these works, our study of the `npm` ecosystem focuses on the ecosystem evolution, popularity measures, and package versioning. An analysis of the statistical computing project `R` [8] finds a super-linear growth in packages as we report in Section 3. In addition, the study focuses on characterizing contributions to user-contributed versus core packages. We refrain from running a similar analysis as `npm` does not differentiate packages explicitly in such a way, although we did identify different types of packages based on our analysis of popularity measures (see Section 4.2). In [11], the authors present results of a quantitative study of the `Ruby` ecosystem. The paper presents a graph visualization of the whole ecosystem as well as some descriptive statistics and histograms about selected characteristics of packages, including downloads and package size. In contrast to our work, the dataset is much smaller, having only around 10K gem nodes and 13.1K dependencies. Furthermore, the paper does not go into the dynamics of the ecosystem, considering instead a single point in time. We did not find any published empirical analyses of the `npm` ecosystem.

Some works have studied the evolution of versions and corresponding change of software projects. For example, in a recent empirical study [5] regarding two ecosystems (including `npm`) the authors find that developers struggle with changing versions as they might break dependent code. Similar assessments on the effects of changes have been made regarding the `Apache` ecosystem [2] or the `Maven` ecosystem [24]. In contrast to these works, we assess versions in `npm` from a black-box perspective: we do not assess how version changes are reflected in the implementation of individual packages, but focus on the occurrence of version numbers and how they are adopted by application developers.

Finally, `npm` has occasionally been analyzed out of the context of peer-reviewed venues. *`npm` packages pagerank* provides a keyword-based search for packages, and presents the results as recommendations based on their PageRank [21]. While we also consider the PageRank as a possible popularity measure, we have shown that this metric may not be adequate for packages most useful to application developers (Section 4.2). The project *`npm` by numbers* analyses a snapshot of the `npm` ecosystem from September 2015 and presents various statistics on it, including the distribution of version numbers and releases of packages and the dependencies between packages [25]. In contrast to our work, `npm` by numbers con-

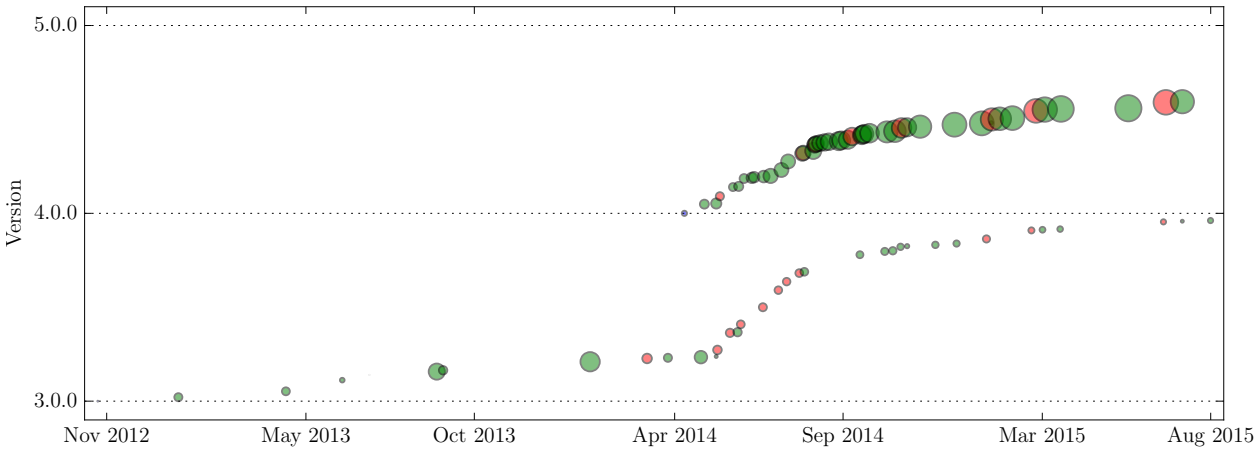

Figure 12: Implicit adoption of new releases of the `express` package. A circle indicates a new release, where blue, red, and green indicates whether it is a major, minor, or patch release. The diameter of the circle denotes the fraction of applications that immediately resolved to the new version as it was released. The circles denote values ranging from 2% to 48%.

siders only a single point in time, whereas we focus on the evolution of the ecosystem, and provides no insight derived from client applications.

7 Conclusion

In this paper, we conducted an analysis of the `npm` ecosystem, one of the largest software ecosystems encompassing application frameworks, libraries, and utilities used in both server-side `Node.js` and browser-side JavaScript applications. We find `npm` to be a striving ecosystem with ongoing and even accelerating growth of packages and increasing dependencies between them. Our findings regarding the differences in popularity measures can be used to improve the search and recommendation systems targeting `npm`, as well as help developers to make informed decisions when choosing packages for use in their applications. Finally, our assessment of version numbers indicates different conventions embraced by developers, despite the prescribed usage of semantic versioning, and our assessment of version adoption shows that flexible version queries can lead to significant immediate adoption ratios.

References

[1] A. M. Alves, M. Pessoa, and C. F. Salviano. Towards a Systemic Maturity Model for Public Software Ecosystems. In *Software Process Improvement and Capability Determination*, pages 145–156. Springer Berlin Heidelberg, Berlin, Heidelberg, May 2011.

[2] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 280–289, Sept 2013.

[3] K. Blincoe, F. Harrison, and D. Damian. Ecosystems in GitHub and a Method for Ecosystem Identification Using Reference Coupling. In *Proc. of the Working Conference on Mining Software Repositories (MSR)*, 2015.

[4] R. Bloemen, C. Amrit, S. Kuhlmann, and G. Ordóñez-Matamoros. Gentoo Package Dependencies over Time. In *Proc. of the Working Conference on Mining Software Repositories (MSR)*, 2014.

[5] C. Bogart, C. Kästner, and J. Herbsleb. When it Breaks, it Breaks. In *Proc. of the Workshop on Software Support for Collaborative and Global Software Engineering(SCGSE)*, 2015.

[6] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1–7):107–117, Apr. 1998.

[7] M. Gasparic and A. Janes. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software*, 113:101–113, 2016.

[8] D. German, B. Adams, and A. E. Hassan. Programming Language Ecosystems: The Evolution of R. In *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2013.

[9] G. Gousios. The GHTorrent Dataset and Tool Suite. In *Proc. of the Working Conference on Mining Software Repositories (MSR)*, May 2013. `http://ghtorrent.org`.

[10] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Software Engineering*, 31(3), 2005.

[11] J. Kabbedijk and S. Jansen. Steering Insight: An Exploration of the Ruby Software Ecosystem. In *Second International Conference on Software Business (ICSOB)*, pages 44–55, 2011.

[12] P. Louridas, D. Spinellis, and V. Vlachos. Power Laws in Software. *ACM Transactions on Software Engineering and Methodology*, 18(1), October 2008.

[13] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing Software Ecosystems. *Science of Computer Programming*, 75(4):264–275, 2010.

[14] K. Manikas and K. M. Hansen. Software Ecosystems A Systematic Literature Review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[15] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 2003.

[16] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis. The Bug Catalog of the Maven Ecosystem. In *Mining Software Repositories*, pages 372–375, New York, New York, USA, 2014. ACM Press.

[17] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse. Software Quality Metrics Aggregation in Industry. *Journal of Software Evolution and Process*, 25:1117–1135, 2013.

[18] C. R. Myers. Software Systems as Complex Networks: Structure, Function, and Evolvability of Software Collaboration Graphs. *Physical Review E*, 68(4), 2003.

[19] I. npm. npm. `http://www.npmjs.org/`. Last visit: March 3rd 2016.

[20] I. npm. npm Discover. `http://www.npmdiscover.com/`. Last visit: March 3rd 2016.

[21] npm packages PageRank. `http://anvaka.github.io/npmrank/online/`. Last visit: January 27th 2016.

[22] T. Preston-Werner. Semantic Versioning 2.0.0. `http://semver.org/`.

[23] S. Raemaekers, A. v. Deursen, and J. Visser. The Maven Repository Dataset of Metrics, Changes, and Dependencies. In *Proc. of the Working Conference on Mining Software Repositories (MSR)*, 2013.

[24] S. Raemaekers, A. van Deursen, and J. Visser. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *In Proc. of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, September 2014.

[25] I. Ros. npm by numbers. `http://npmbynumbers.bocoup.com/`.

[26] A. Serebrenik and T. Mens. Challenges in Software Ecosystems Research. In *Proc. of the ACM European Conference on Software Architecture Workshops*, 2015.