An Analysis of Dependency Network Evolution in PyPI

Md Monir Hossain, Nima Mahmoudi & Changyuan Lin

ABSTRACT

Programming languages have packaging ecosystems in place to make it easier for developers to reuse previous libraries rather than staring from the scratch. While this makes the ecosystem more friendly in terms of rapid development, it also paves the way for a complex ecosystem that is needed to be kept in check. In this work, we take a look into Python's package manager (PyPI) in a quantitative manner to explore different properties of the dependency network and make a comparison of PyPI's evolution against 7 other popular packages to peek into the similarities and dissimilarities. We found that while Python in general followed the laws of software evolution, it has interestingly different trend than other packaging ecosystems in terms of dependency. We hope that these insights can help us to design more platform and developer friendly tools, put in place appropriate policy and choose which platform has a better prospect for a specific project.

KEYWORDS

Software Ecosystem, Package Manager, PyPi, Dependency Network

1 INTRODUCTION

Package managers are integral part of software ecosystems. They allow for convenient reuse of libraries while taking care of dependencies. The exploration of the the ecosystem's attributes and dependency network properties can provide us insight towards their evolution and activity, opening the door towards an improved package recommendation system. All packaging ecosystems provide some tools to help the developers keep track of their packages. Our analysis can also provide these platforms insights to improve these tools, allowing developers to make informed decision based on the understanding of their packages' usage.

Python Package Index (PyPI) is the software repository for Python, contributed by the Python developer community. It is the home of 175,476 packages as of April 10, 2019 and growing with each passing day. We explore different attributes of the packaging ecosystem along with interconnected dependency among the packages. The research questions we are interested to answer for PyPI are:

RQ1: How has Python dependency network grown over time? The motivation behind this RQ is that the growth rates of the dependency network dictates the complexity of the ecosystem and knowing about this enables the ecosystem to put in place tools and policies to manage it properly.

RQ2: How frequently are Python packages updated? The motivation this RQ is to understand the dynamic nature and activeness of the community.

RQ3: To which extent do packages depend on other packages? The motivation behind this RQ is to suggest appropriate change in tools pertinent to managing dependencies. It also helps us to understand whether developers are aware of dependency issues and any change in their behaviour.

RQ4: How prevalent are transitive dependencies? The motivation behind this RQ is to understand to what extent packages are dependent on other packages. Also, like the previous RQ it helps us to identify any change in developer behaviour compared to other packaging ecosystems.

RQ5: How does social network characteristics come into play for dependency network evolution? The motivation behind this RQ is to understand the social structure of the packages. This informs us regarding which packages are influential in the ecosystem and what type of projects should consider python as a prime candidate.

2 RELATED WORK

The research in the arena of software ecosystems has a rich history. For software ecosystems, studies have been done in the context of both individual project and cross project dependencies. Dietrich et al. performed cluster analysis on Java programs' dependency graphs to improve reusability and maintainability [11]. Lungu, Robbes, and Lanza proposed an algorithm to discover dependencies among the software projects of an ecosystem [16]. Constantinou and Mens studied Ruby ecosystem in GitHub over a period of nine years to analyze the socio-technical aspects of the co-evolution among a large number of base projects and their forks in the context of source code and contributing developers [5]. Plakidas, Schall, and Zdun tried to quantify R ecosystem based on documentation medadata. They introduced the notion of ecosystem participants, both in the software marketplace and in the developer community. They used the quantification parameters to discover their interrelationships and contributions [17].

Decan, Mens, and Claes investigated the impact of GitHub in R ecosystem, from the perspective of both package distribution and package dependency management among repositories [8]. In another one of their work, they studied how the inter dependencies cause issues within the system and whether the measures that are in place to tackle them are sufficient or not [7]. Kikas at el. took a look at JavaScript, Ruby, and Rust ecosystems to understand the evolution and dependency network structure [13].

Several works have been carried out to understand how software packages ecosystems evolve, the factors that affect them and the extent of that effect. Decan, Mens, and Claes studied CRAN, PyPI, and npm and compared the structure of their package dependency graphs [6]. Wittern, Suter, and Rajagopalan analyzed JavaScript package ecosystem (npm) based on package descriptions, dependencies, download metrics and their relation to GitHub to understand its growth and activity [20]. Korkmaz et al. tried to model the impact of R packages in the form of downloads and citations, leveraging information from dependency and contributor networks [14]. Decan, Mens and Grosjean did a comprehensive analysis on seven packaging ecosystems including Cargo, CPAN, CRAN, npm, NuGet, Packagist and RubyGems to find out the similarities and dissimilarities among their evolution [10]. Valiev, Vasilescu and Herbsleb

performed both quantitative and qualitative analysis on sustainability of open source Python projects in the context of ecosystem level determinants [19].

One of the critical impact that dependency in a package manager ecosystem can have is in the realm of security. Cadariu et al. investigated security vulnerabilities introduced in software system due to packages being dependent on each other. In order to track these vulnerabilities they also proposed a tool named Vulnerability Alert Service (VAS) [4]. Decan, Mens, Tom and Constantinou studied the propagation of security vulnerabilities through npm dependency network in JavaScript packages [9]. Ruohone studied security vulnerabilities in Python packages for web development. He mentioned that historical or current vulnerability of a package can play a role on conditional probability of another dependent package being vulnerable [18].

A few works have been done considering the social aspects in these ecosystems' evolution. Bogart et al. explored Eclipse, CRAN, and npm to study the changes from software ecosystems perspective. They observed significant difference in practices and expectations toward change due to different community values in different ecosystems. They took a qualitative approach to study developers' reaction to and management of change in their dependencies [2].

3 DATASET AND PREPROCESSING

We have used two datasets for this study. Firstly, We have used data from libraries.io (version: 1.4.0 - December 22, 2018) that gathers data from 36 package managers and 3 source code repositories[1]. RQ1 and RQ2 were studied using this dataset. We extracted the relevant csv files and put them in tables them into a PostgreSQL database. Then we queried the tables and created new ones which contain data pertinent to PyPI only. Associated queries for creating the tables, importing the csv files into tables and creating new tables from there are provided online. At the end, we extracted three datasets, namely pypi_dependencies, pypi_projects and pypi_versions. Relevant codes and scripts are provided online. In addition, the final dataset is provided on Zenodo[2].

As data extracted from libraries.io didn't capture enough dependency information that we required, we used raw_dependencies file from [19] and used it for dependency relevant RQs, dependency graph building and metrics calculation. Thus, we have used the Dataset used in [19] that contains package and dependency names[3]. RQ3, RQ4 and RQ5 have been studied using this dataset.

4 EXPERIMENTAL RESULTS

4.1 RQ1: How has Python dependency network grown over time?

As our first research question, we look at the growth of number of packages and dependencies over time. Being aware of the speed of growth is important for managing the process, infrastructure and guidelines for publishing new packages in the PyPI ecosystem. Without careful consideration, allowing rapid growth of the dependency network might increase the complexity of the PyPI ecosystem beyond the manageable level. To make sure this doesn't happen, the ecosystem requires proper quality standards and tool support. On the other hand, new packages must be introduced continuously in order to offer new functionalities to the developers.

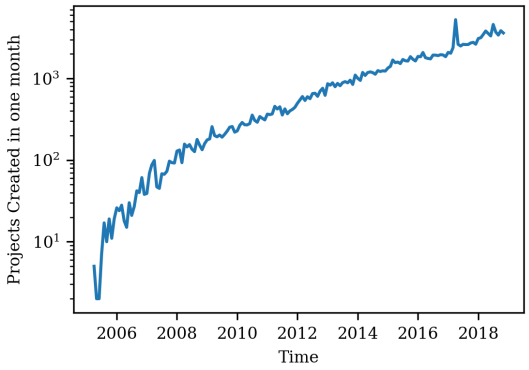

Figure 1: The number of projects published per month.

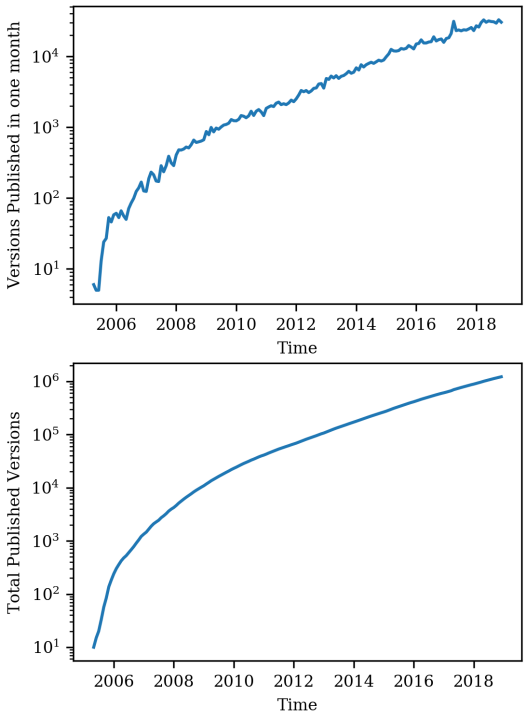

Figure 2: The number of versions published per month and the total number of versions.

To get a better understanding of the growth in this ecosystem, we computed the growth of number of projects, total number of versions published, total number of dependencies in the network and the number of dependencies per package on the monthly snapshots of the dataset.

fig:project-publish-over-time shows the growth in number of packages published per month in PyPI using a logarithmic scale for

[1]https://libraries.io/data last accessed: Apr-10-2019.
[2]https://doi.org/10.5281/zenodo.2620607 last accessed: Apr-10-2019
[3]http://doi.org/10.5281/zenodo.1297925 last accessed: Apr-10-2019.

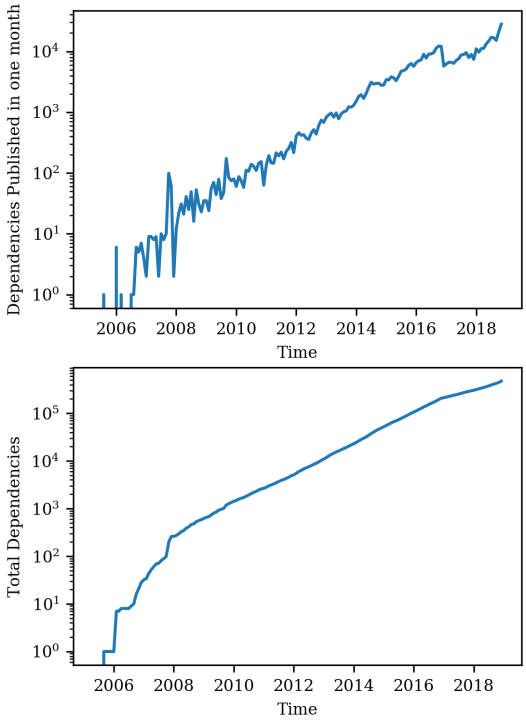

Figure 3: The number of dependencies published per month.

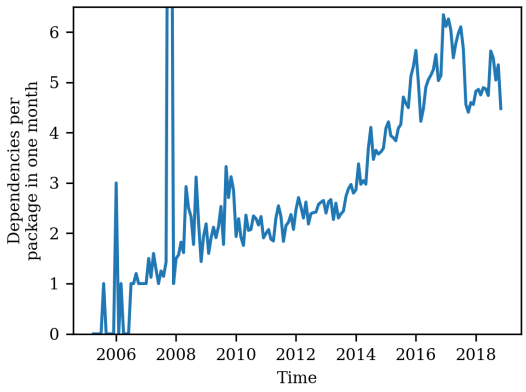

Figure 4: The average number of dependencies per package published in each month.

the y-axis. This figure shows us a very rapid growth in the number of packages in the beginning (due to immaturity in the package manager ecosystem) which becomes exponential after some time as the ecosystem becomes more mature. fig:version-publish-over-time shows the growth in number of versions published over time (which includes updates to older packages). This shows that the growth in the number of versions follows the same trend as number of new packages published.

The number of published dependencies per month along with total number of published dependencies have been shown in fig:dependency-publish-over-time. As is evident from this figure, the total number of dependencies shows an exponential growth in time as well. But to investigate this further and see if the exponential growth is due to increase in number of packages published or number of dependencies per package, we calculated the number of dependencies per package for each monthly snapshot as well. The results is shown in fig:dependency-per-package-over-time. As you can see, the number of dependencies per package does not grow very rapidly, changing from about 2 in 2009 to around 5 in 2018. Thus, the exponential growth in total number of dependencies is mainly due to the exponential growth in total number of packages being published in PyPI.

In order to investigate the speed of growth over time further and find out the model that best describes the data, we performed a regression analysis on the data fitting linear and exponential models to them. The R^2 which shows the goodness of fit is shown in tab:r2-for-dependencies. We can see that for package, version and dependency counts, an exponential model fits the data best. This shows that these measurements follow an exponential pattern. Whereas, dependency per package is following a linear model. This shows that the exponential growth in number of dependencies is mainly due to the exponential growth in the number of packages and not the number of dependencies per package. Decan et. al. [10] observed a growth in number of packages and dependencies over time in the 7 package managers studied, but found out that for the number of dependencies, the Cargo, CPAN, Packagist and RubyGems were experiencing a linear growth while CRAN, npm and NuGet were showing exponential growth. In addition, they found out that Cargo, CPAN and Nuget showed linear growth in number of packages while CRAN, npm, Packagist and RubyGems showed exponential growth. In our analysis, python shows exponential growth in both number of packages and number of dependencies similar to CRAN and npm, but the number of dependencies per packages is not analyzed in [10] for other package managers.

Table 1: R^2 values of regression analysis.

Name	*Linear Model*	*Exponential Model*
packages count	0.78	**0.92**
versions count	0.77	**0.93**
dependency count	0.60	**0.85**
dependency per package count	**0.46**	0.43

Bold cells correspond to highest values.

4.2 RQ2: How frequently are Python packages updated?

Package updating involves functionality improvements, bug fixes, and other changes. It is a common and important phase of the software life-cycle. Package update frequency reflects the activity of software community as software evolves faster in a more dynamic community. Python ecosystem is evolving and changing fast. The motivation of RQ2 is to measure the activity of Python community.

To answer how frequently are Python packages updated, we counted the number of package updates each month from January 2005 to December 2018. fig:number-of-updates-per-month depicts the monthly number of package updates published in PyPI. We recognized that the number of package updates published in PyPI increased significantly over time. From 2005 to 2012, the number of Python package updates grows slowly to about 5,000 per month. Then, it surges upwards to more than 25,000 in 2017. It is also noticeable that the growth is accelerating from 2005 to 2017, and the growth rate tends to remain stable after 2017.

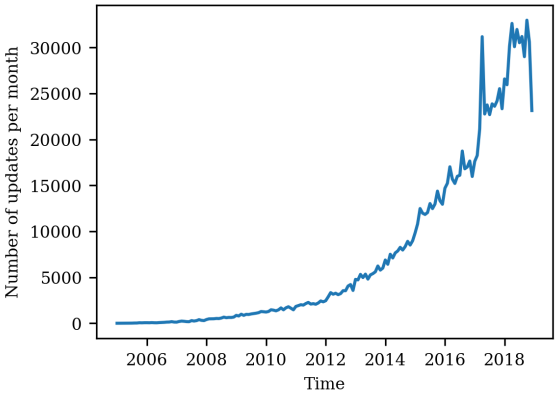

Figure 5: The number of package updates published in each month.

For each package, we calculated the interval between the two updates. We noticed that about 27% of packages in PyPI never updated after release. The obtained metrics regarding the update interval is shown in the following table.

Table 2: Update Interval in Days

Average	Median	25th Percentile	75th Percentile
70.18	18.69	1.59	73.08

While the average update interval is 70 days, the median of update interval is 18.7 days. The 58th percentile is 30.25 days. Therefore, most of the packages in PyPI are updated very frequently. About 58 percent of packages are updated within a month.

We also analyzed the distribution of updates over individual packages by counting the number of package updates for each package after its release. We found that most of the package updates in PyPI came from a small proportion of active packages. We used Lorenz curve [15] to demonstrate the uneven distribution.

The Lorenz curve is usually used for illustrating the inequality of the income distribution. In our case, we used the inverted Lorenz curve, as shown in fig:package-updates-distribution, to demonstrate the uneven distribution of package updates, where the cumulative proportion of packages is on the horizontal axis, and the cumulative proportion of package updates is on the vertical axis. As is evident from the figure, the number of package updates is distributed unevenly. We observed that more than 60 percent package updates came from only 20 percent of active packages. Compared to NuGet and npm ecosystems which are active [10], PyPI also shows the high distribution inequality in package updates.

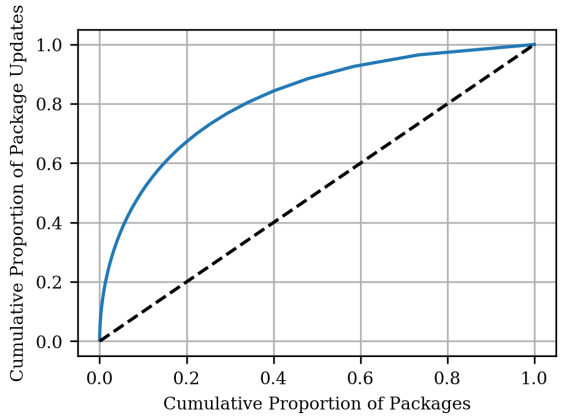

Figure 6: Inverted Lorenz Curve for the Number of Package Updates in PyPI.

The distribution inequality in package updates is highly dependent on the package age. We investigated the impact of package age on the number of package updates. By calculating the date gap between the creation date of the package and the each release date of its updates, we could get the trend of number of updates regarding the package age, as shown in fig:package-updates-versus-age and fig:cumulative-package-updates-versus-age.

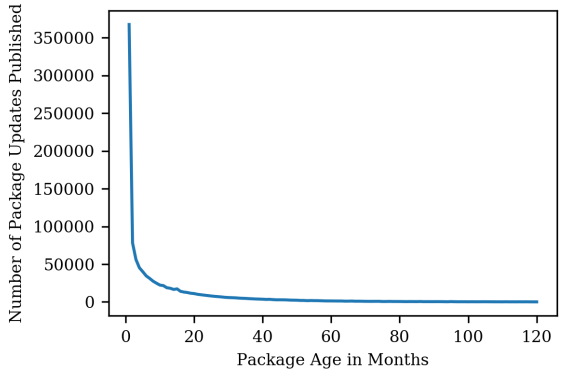

Figure 7: The Number of Package Updates Published versus Package Age.

fig:package-updates-versus-age and fig:cumulative-package-updates-versus-age show that young packages receive package updates more frequently. Over 60% of package updates happen in the first year after the package is released.

For better presenting the uneven distribution of package updates, following the idea of *Changeability Index*, we use it to investigate the change dynamics of Python packages.

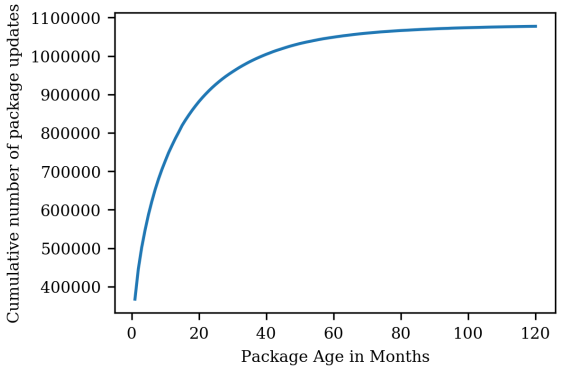

Figure 8: The Cumulative Number of Package Updates Published versus Package Age.

"**Definition 1** The **Changeability Index** of an ecosystem E at time **t** is the maximal value n such that there exist n packages in E at time t having been updated at least *n* times during the last month"[10].

fig:changeability-over-time depicts the changeability index of Python packages in PyPI. The changeability index grows steadily over time from 2005 to 2017 and then reaches a plateau around 2018. By comparing PyPI with NuGet and npm ecosystems which are active, we found that the changeability index of PyPI ecosystem is at the same level with them. There is a high similarity regarding package updates across these three ecosystems.

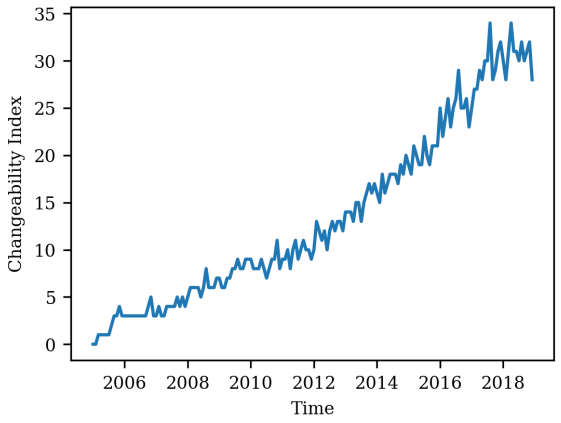

Figure 9: The Changeability Index of Python Packages in PyPI.

4.3 RQ3: To which extent do packages depend on other packages?

Dependencies are strongly prevalent in software ecosystems. Its prevalence in package ecosystems can be attributed to the rapid development opportunity it provides by promoting re-usability for other developers. Instead of reinventing the wheel every time, they get the opportunity to focus on the core development. The ecosystems also acknowledge this fact by providing tools to manage these complex dependencies. However, developers are aware of the complication that can arise from having dependencies in their packages [10]. Sometimes packages are not maintained anymore and sometimes they are removed altogether. Also, there can be version conflict among dependencies of different packages.

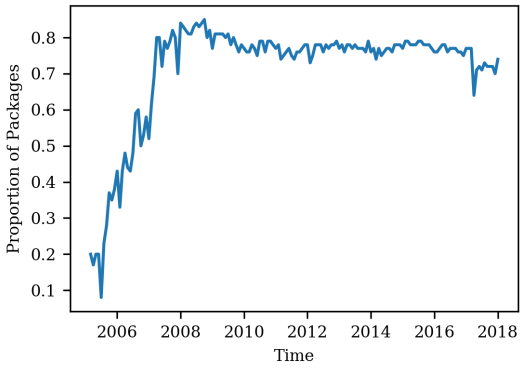

Figure 10: Proportion of Connected Packages

fig:proportion-of-connected-packages shows the evolution of connected packages compared to total number of packages. Connected package are all the packages that are connected to each other through dependency, i.e., either dependent or required package. We found that the proportion of connected packages had a rapid growth between 2006 to 2008. After 2016 there has been a slight drop in dependent package, perhaps due to caution among developers. This is contrary to the findings in [10] that found that for cargo, CPAN, CRAN, npm, NuGet, Packagist and RubyGems, there was a steady increasing trend over time. We hypothesize that Python developer community is actively more conscious regarding dependency issue during recent few years.

However, a connected can be either a dependent package or a required package. fig:proportion-of-dependent-and-required-packages

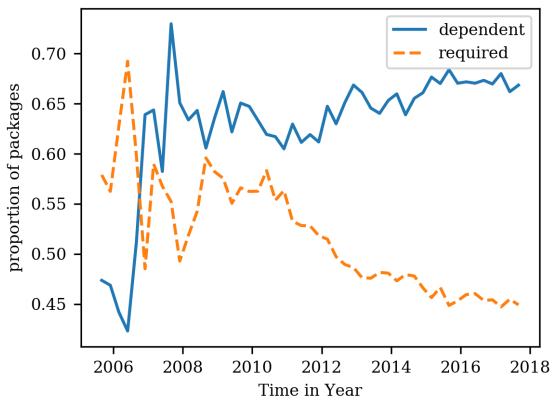

Figure 11: Proportion of Dependent and Required Packages.

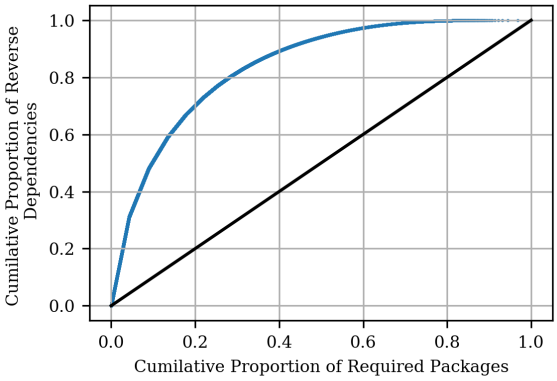

Figure 12: Inverted Lorenz Curve for the Number of Reverse Dependencies for Required Packages Considering the Latest Release.

shows the how dependent and required package have evolved over the years compare to the total number of packages. At the very beginning the proportion of required packages was higher compared to dependent packages. We believe this is due to the fact that at the start of the ecosystem there were not enough core package and as a result some 'seed' packages were released that acted as building block for other packages. This phenomenon is also seen for Cargo in [10]. However, very soon the proportion of dependent packages crossed that of required packages as expected. We observe that while for other ecosystems required package proportion stays same, for PyPI it actually is decreasing. For dependent packages the trend is quite same for all of 8 ecosystems. This shows that a the ratio of influential required packages in python is decreasing. So, developers are limiting the use of dependencies in their packages by keeping only the absolute necessary ones.

Furthermore, as fig:inverted-lorenz-curves-for-reverse-dependencies shows, not all packages contribute equally when it comes to being a required package. This inverted Lorenz curve follows power law

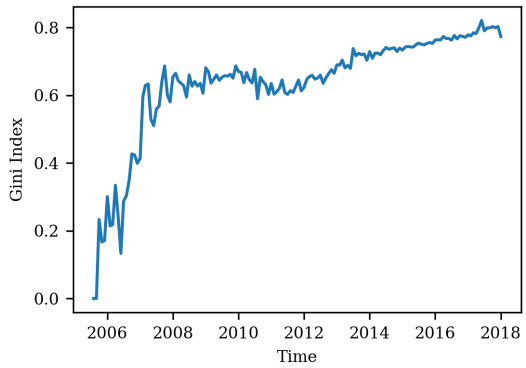

Figure 13: Evolution of the normalized Gini index reflecting the inequality of the number of dependent packages per required package.

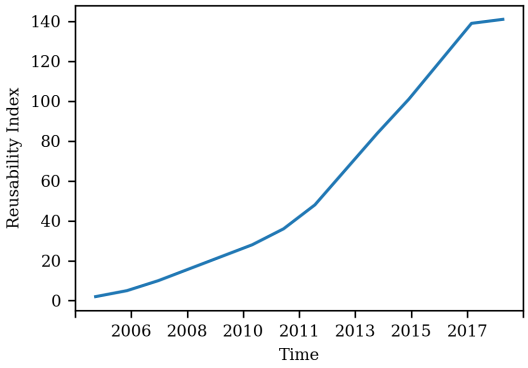

Figure 14: The Evolution of the Reusability Index.

distribution, i.e., minority of packages contributes to majority of requirements. 20 percent of PyPI packages account for more than 95 percent of reverse dependencies. This finding is in agreement with the findings from [10].

We also checked how this inequality has evolved over time. fig:evolution-of-the-gini-index shows the Gini index signifying the inequality in number of dependent packages per required package. An increasing trend in Gini index over the years shows that PyPI packages have been subjected to ever increasing inequality. This also true for other package managers as found in [10].

Next we used the notion of 'Reusability Index' which acts as a combined measure of the amplitude and extent of reuse. Here, amplitude means the number of required packages, while extent refers to their number of dependent packages.

"**Definition 2** The **Reusability Index** of an ecosystem E at time t is the maximal value n such that there exist **n** required packages in **E** at time **t** having at least **n** dependent packages" [10].

fig:evolution-of-the-gini-index shows how reusability has changed over the the years. This follows the similar trend to [10]. The curve for PyPI fits a exponential model slightly better than linear model as shown in tab:r2-for-reusability.

Table 3: R^2 Values of Regression Analysis on the Evolution of Reusability Index.

Name	*Linear Model*	*Exponential Model*
packages count	**0.9313**	0.926

Bold cells correspond to highest values.

4.4 RQ4: How prevalent are transitive dependencies?

Similar to RQ3, the motivation of RQ4 is also to find to which extent do packages depend on other packages, but at a deeper level. While RQ3 mainly focuses on the direct dependency of packages, RQ4 concentrates on the transitive dependency, which captures the deeper layers of the dependency network. The transitive dependency of a package is any dependency introduced by its direct dependencies. Therefore, transitive dependencies have an indirect

impact on the package. If a transitive dependency of a package is broken, one or more direct dependencies of this package may malfunction. Consequently, the package will not work properly either. An example of this is the removal of left-pad, 11 lines of JavaScript code, that caused havoc by breaking thousands of projects after the developer unpublished it along with around 250 other modules. however, it was the widespread use of left-pad that caused a panic in the developer community and forced a unprecedented republish of the unpublished package addressed by CTO and co-founder of npm. But it is really concerning that 11 lines of dependency had such a profound impact and how developers were not aware of the extent of transitive dependencies that their packages and projects depend on.

In an ecosystem where transitive dependencies are more prevalent, a broken or missing package can influence a larger proportion of other packages. Hence, it is necessary to investigate the prevalence of transitive dependencies in the PyPI ecosystem. For each package in PyPI, we counted the number of direct dependencies and the number of transitive dependencies. The fig:distribution-of-dependencies shows the distribution of the number of direct dependencies and the number of transitive dependencies for all packages in PyPI. The average ratio between the number of transitive dependencies and direct dependencies is 2.63 in PyPI ecosystem, which means one dependency of Python packages typically leads to 2.63 transitive dependencies on average. Compared to other ecosystems including npm, CRAN and CPAN [10], PyPI has a significantly lower average ratio between the number of transitive dependencies direct dependencies, reflecting that Python packages tend to have a smaller dependency tree.

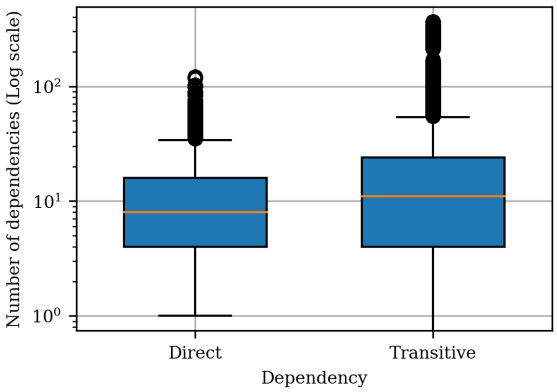

Figure 15: Distribution of the number of direct dependencies and the number of transitive dependencies for packages in PyPI.

We also investigated the evolution of the ratio between the number of direct dependencies and the number of transitive dependencies, as shown in fig:dep-ratio-over-time.

At the beginning, coupled with the development of the Python ecosystem, the ratio rises steadily and reaches the highest point at about 8 around 2011. Then, corresponding to the result of RQ3, the ratio declines to about 3 and remains stable after 2015. We hypothesized that, around 2011, Python package developers were

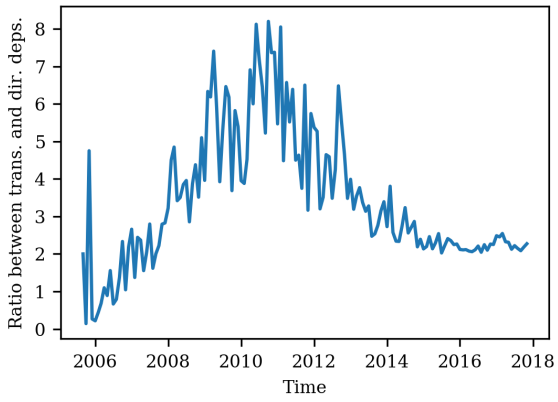

Figure 16: The ratio between the number of direct dependencies and the number of transitive dependencies over time.

aware that packages have a huge dependency tree which imperils the maintainability and availability of packages. So, they tried to slash the number of dependencies of packages after 2011. This trend is unique to PyPI, since we could not see a similar trend in all seven ecosystems including npm, CRAN, and NuGet mentioned by the work published by Decan et al.[10].

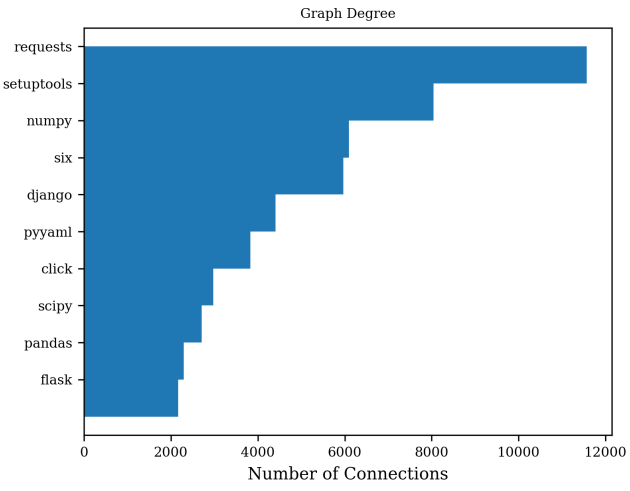

Figure 17: Top 10 packages in the PyPI dependency network according to in-degree.

4.5 RQ5: How does social network characteristics come into play for dependency network evolution?

In this work, Social Network Analysis (SNA) is used to study the social network of dependencies to understand their structure and behaviour. It helps understand which packages are most essential in the PyPI dependency network, which ones are more stable and which packages provide less contribution to the network [12].

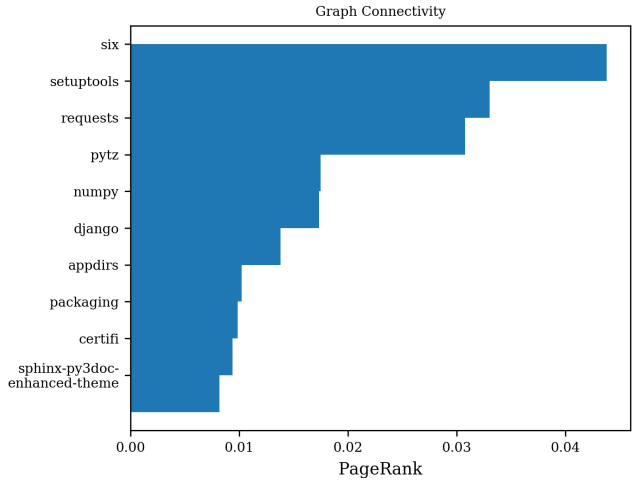

Figure 18: Top 10 packages in the PyPI dependency network according to the page-rank algorithm.

In this section, nodes and directed edges represent packages and dependencies in PyPI. In our definition, package A depending on package B is shown by an edge from the node representing package A to the node representing package B. An interactive version of the directed graph generated which only includes nodes that have a degree larger than 10 is available online[4].

To find out which are the most important packages, we used in-degree and page rank. In-degree is an attribute which shows how many packages depend on a package. The top 10 packages according to in-degree are shown in fig:in-degree-top.

Page-rank algorithm [3] ranks the importance of the packages based on a model of a random browser. Top 10 packages in the PyPI dependency network based on page-rank is shown in fig:page-rank-top.

Both in-degree and page-rank show that the most important packages in the ecosystem are mostly for interacting with the internet, scientific computing, or building websites. This shows that python packages are very stable for these tasks and are considered a viable option.

Another social network analysis that would give us insights regarding the PyPI dependency network is analyzing distinct communities in this network. This analysis is very computationally expensive and is NP-Complete. A fast approximation based on dendrograms to this presented in [1] which works very well for large-scale networks. We used this analysis to get the largest communities in PyPI ecosystem. fig:community-0 shows the largest community in the PyPI network along with its top five packages based on page-rank. As can be seen, these packages are mostly related to scientific computing. This shows that a large potion of python packages are dealing with mathematical and statistical analysis and these packages form the largest community on the PyPI server. The 10 largest communities can be found on the project GitHub repository[5].

[4]http://nimamahmoudi.github.io/cmput663-final-project
[5]https://github.com/nimamahmoudi/cmput663-final-project/blob/master/RQ5/rq5-graph-analysis.ipynb

5 DISCUSSION

We conducted a large-scale empirical study on PyPI ecosystem. In section IV, we proposed and answered the five research questions. We found that PyPI, npm and CRAN share a similar growth trend, which is reasonable since all of them are very popular and active ecosystems today. We discovered that the exponential growth in the number of packages is the underlying reason that led to the exponential growth of the Python dependency network.

6 THREATS TO VALIDITY

We can not verify the accuracy of the data that we used from libraries.io. However, [10] performed a manual checking and cross-validation with metadata from their previous research. We therefore assume the correctness of the data.

In case of the dataset from [19] we found some cases of self dependency and loop dependency. After checking a few of those samples manually from PyPI website, we found that the dataset reported wrong values for those data points. However, given the scale of our analysis and the fact that we are looking at trends for a long period of time, we assume that these impurities will not deviate the findings significantly.

To run our transitive dependency analysis, we considered a dependency depth of 10. This was for resolving dependency loop issue. Due to this loop the program was unable to complete the analysis. Any package that went beyond a depth of 10 was therefore discarded. Although this means that some packages with an actual depth of more than 10 was discarded, the number was low enough considering the scale of our analysis.

7 FUTURE WORK

Our work may be expanded further by the means of socio-technical analysis which takes into account the developers and contributors. One may be able to build a prediction model for packaging ecosystems based on the intrinsic and extrinsic properties. Also, building a verified dataset given enough time can be helpful for these future studies.

8 CONCLUSION

We performed a quantitative and social network analysis of PyPI's ecosystem. We analyzed how the ecosystem has grown over time and what the update pattern tells us about the ecosystem. We explored how vulnerable the system is to direct and transitive dependencies. Along the way, we hypothesize a conscious choice made by developers to reduce number of dependencies in their packages. We confirmed the presence of laws of software ecosystem evolution like increasing growth, change and complexity. By the means of social network analysis we found out which packages are at the forefront of the ecosystem and what type of development tasks Python is better equipped for.

9 ACKNOWLEDGEMENT

This research was enabled in part by support provided by WestGrid (www.westgrid.ca) and Compute Canada (www.computecanada.ca).

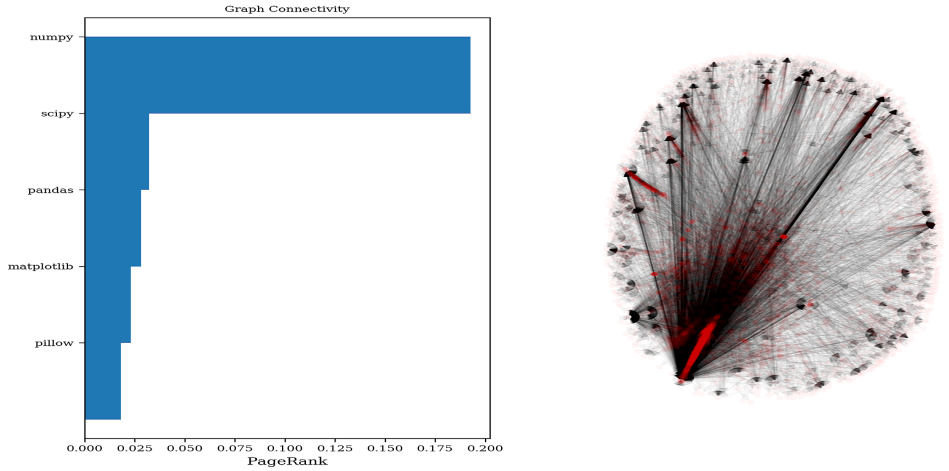

Figure 19: The the largest community of PyPI (right) and five highest ranking packages in it based on page-rank (left). Nodes in this community are shown in red.

REFERENCES

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[2] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 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*. ACM, 109–120.

[3] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1-7 (1998), 107–117.

[4] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 516–519.

[5] Eleni Constantinou and Tom Mens. 2017. Socio-technical evolution of the Ruby ecosystem in GitHub. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 34–44.

[6] Alexandre Decan, Tom Mens, and Maelick Claes. 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*. ACM, 21.

[7] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2–12.

[8] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. 2016. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 493–504.

[9] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 181–191.

[10] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* (2018), 1–36.

[11] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. 2008. Cluster analysis of Java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*. ACM, 91–94.

[12] Kevin Gullikson. 2016. Python Dependency Analysis. http://kgullikson88.github.io/blog/pypi-analysis.html. [Online; accessed 7-April-2019].

[13] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE press, 102–112.

[14] Gizem Korkmaz, Claire Kelling, Carol Robbins, and Sallie A Keller. 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, 511–514.

[15] Max O Lorenz. 1905. Methods of measuring the concentration of wealth. *Publications of the American statistical association* 9, 70 (1905), 209–219.

[16] Mircea Lungu, Romain Robbes, and Michele Lanza. 2010. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 309–312.

[17] Konstantinos Plakidas, Daniel Schall, and Uwe Zdun. 2017. Evolution of the R software ecosystem: Metrics, relationships, and their impact on qualities. *Journal of Systems and Software* 132 (2017), 119–146.

[18] Jukka Ruohonen. 2018. An Empirical Analysis of Vulnerabilities in Python Packages for Web Applications. *arXiv preprint arXiv:1810.13310* (2018).

[19] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 644–655.

[20] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 351–361.