Are Software Dependency Supply Chain Metrics Useful in Predicting Change of Popularity of NPM Packages?

Tapajit Dey
University of Tennessee, Knoxville
Knoxville, Tennessee
tdey2@vols.utk.edu

Audris Mockus
University of Tennessee, Knoxville
Knoxville, Tennessee
audris@utk.edu

ABSTRACT

Background: As software development becomes more interdependent, unique relationships among software packages arise and form complex software ecosystems. Aim: We aim to understand the behavior of these ecosystems better through the lens of software supply chains and model how the effects of software dependency network affect the change in downloads of Javascript packages. Method: We analyzed 12,999 popular packages in NPM, between 01-December-2017 and 15-March-2018, using Linear Regression and Random Forest models and examined the effects of predictors representing different aspects of the software dependency supply chain on changes in numbers of downloads for a package. Result: Preliminary results suggest that the count and downloads of upstream and downstream runtime dependencies have a strong effect on the change in downloads, with packages having fewer, more popular packages as dependencies (upstream or downstream) likely to see an increase in downloads. This suggests that in order to interpret the number of downloads for a package properly, it is necessary to take into account the peculiarities of the supply chain (both upstream and downstream) of that package. Conclusion: Future work is needed to identify the effects of added, deleted, and unchanged dependencies for different types of packages, e.g. build tools, test tools.

CCS CONCEPTS

• **Computing methodologies** → *Classification and regression trees*; *Bayesian network models*; *Learning linear models*; • **Software and its engineering** → **Open source model**;

KEYWORDS

Software Supply Chain, Open Source, Software Popularity, NPM Packages, Software Dependency

ACM Reference Format:
Tapajit Dey and Audris Mockus. 2018. Are Software Dependency Supply Chain Metrics Useful in Predicting Change of Popularity of NPM Packages? . In *The 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'18), October 10, 2018, Oulu, Finland*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3273934.3273942

1 INTRODUCTION

The usefulness and success of a software, as for other products in the market, is reflected by its usage. A true measure of popularity, or in other words usage, of a software would be the number of active users for it. Unless software licenses are sold and carefully tracked, the number of active users is virtually impossible to measure directly and accurately. The number of downloads is, perhaps, the closest approximation for the popularity of a software. NPM (node package manager), which is package manager for JavaScript packages, does track the number of downloads for the packages and makes the data publicly available, unlike most other package managers.

Like most software packages, the JavaScript packages distributed by NPM have a dependency structure, i.e. one package may have other packages as runtime and/or development dependencies (other types of dependencies also exist). This dependency network influences the number of downloads of the individual packages, because when one package is installed by a user, all of its dependencies are also downloaded and installed automatically (unless they are cached). The number of JavaScript packages on NPM exceeds 600K, therefore analyzing the complete dependency network for the whole NPM ecosystem is a challenge. The dependencies listed for a package are direct dependencies, however, those dependencies might have their own dependencies. This complex interconnection forms a dependency network that can be considered a software supply chain, with the entire set of recursive dependencies of package being *upstream* from it, and the packages that are recursively dependent on it being *downstream*.

In a nutshell we'd like to know if the supply chain has effect on downloads:

RQ: Do the number and downloads of upstream and downstream dependencies help predict the downloads for popular JavaScript packages in NPM?

We collected data from npms.io and obtained daily, weekly, monthly, and yearly download numbers for all NPM packages. We focus on predicting monthly downloads, because the daily and weekly downloads for most packages exhibit large variations, which, presumably, are random in nature, and we do not have sufficient history to predict yearly downloads for most packages. We collected snapshots of the statistics on all packages roughly every 2 weeks between 01-December-2017 and 15-March-2018, and used the data from a snapshot to predict the number of downloads for the packages over the next month. To ensure as little overlap as possible, we did not use the data from a snapshot to predict the number of downloads in the next snapshot, which is 2 weeks apart, but looked at the one after that, which is roughly 1 month apart. Also, since the actual number of downloads varies drastically among packages, we use the logarithm of the ratio of downloads in next month and the downloads of the previous month as our response variable. To focus on the effect of the supply chain aspects we consider only packages with at least one upstream and one downstream dependency.

To answer the RQ we used both linear regression and random forest models. Our preliminary findings suggest that both the number and the popularity (measured by the number of downloads) of the dependencies are some of the most important predictors for the change in downloads for NPM packages. These models can help interpret the download counts and reasons that affect the usage of the package. Our findings also suggest that the download number can be affected (or manipulated) through dependencies and frequent updates (probably leading to cache misses).

In Section 2, we present the related work in this field. In Section 3, we describe the data. The data analysis and modeling approaches used for the study, and the results of the analysis are described in Section 4. In Section 5, the implications of the result are presented and the ideas for future work are discussed.

2 RELATED WORK

The term "software supply chain" has been used in 1995 [8]. The concept has since been used for addressing economic and management issues in software engineering [6], for facilitating the Software Factory development environment introduced by Microsoft [7] and elsewhere. The primary use of software supply chain has been for identifying and managing risks related to the software development process [4, 9]. In this study, we are trying to use the dependency network (the supply chain) for understanding and predicting the change of popularity of NPM packages.

The topic of software popularity hasn't seen much attention, possibly due to the lack of reliable popularity measures. Stars for GitHub projects were used to identify the factors impacting popularity of GitHub projects [3]. The relationship between popularity of mobile apps and their code repository was studied in, e.g. [5], where the ratings of the mobile apps were used as a measure of popularity. Studies of code repositories, e.g. [12] considers popularity of Python projects on GitHub, [15] studies the relationship between the folder structure of a GitHub project and its popularity, [2] analyze the popularity, interoperability, and impact of various programming languages.

The NPM ecosystem is one of the most active and dynamic JavaScript ecosystems and [13] presents its dependency structure and package popularity. [14] studies the dependency, specifically the lag in updating dependencies in various NPM packages while [1] looked into the use of trivial packages as part of package dependencies for different NPM packages.

We are not aware of prior work using NPM downloads as a measure of popularity and examining the effect of the popularity and count of immediate and recursive upstream and downstream dependencies for predicting the change in downloads of NPM packages.

3 DATA

The data for our study was collected from the npms.io website, using the API provided [1]. As mentioned earlier, we are only concerned about the packages with at least one upstream and one downstream dependency, and we found that 72,211 packages out of over 600,000 packages in NPM meet our criteria across all the snapshots. We used the API to collect data on all NPM packages twice a month roughly 2 weeks apart, once in the beginning of a month and once in the middle, between 01-December-2017 and 15-March-2018, resulting in 8 different snapshots. The data collection process takes around two days, so the data on all packages are not from the same date, however, we expect the variance to have been evened out across all snapshots. Since we are using the data for predicting the ratio of downloads between the current and the next month, we had 6 usable snapshots (because for the last two we did not have download counts in the next month).

The data collected from npms.io has information on the GitHub repository of the project, e.g. the number of issues, number of weekly, monthly, quarterly, half-yearly, and yearly commits, the list of contributors and the number of commits by each contributor, as well as the number of forks and stars. The metadata information consists of the list of runtime and development dependencies, the monthly, quarterly, half-yearly, yearly, and total number of releases for the project, the name and email of the author and the publisher, the README text, the list of maintainers and contributors (as listed in NPM), and the number of daily,weekly, monthly, quarterly, half-yearly, and yearly downloads. We used the number of these quantities (except the README file) as the variables used for our analysis. The data also has information about some evaluation metrics calculated by the npms.io website, which we did not use for our analysis. For the number of releases and commits, we only used the monthly and yearly numbers, because a quick PCA (Principle Component Analysis) suggested that most of the variance (~80%) in all the commit and release variables are explained by these two components.

As mentioned earlier, the response variable for our study is the logarithm of the ratio of downloads in next month and the downloads of the previous month. Therefore, to maintain consistency, we removed all other download variables from our dataset. There is one more caveat about the NPM download counts, as mentioned in [11]. To summarize, the download counts available in the NPM server is the sum of downloads by automated build servers, mirror servers, robots that download every package for analysis, and downloads by actual human users. Moreover, if a human user had installed a particular package before, it will nearly always be installed from their local NPM cache, so that doesn't get counted to the download numbers available in NPM. Similarly for the mirrors and build servers, the packages are installed from NPM cache unless the version of a package changes. Therefore, the article [11] remarks "Only if [an NPM] package is getting > 50 downloads/day can [the publishers] be sure [they are] seeing signal instead of noise." Keeping this recommendation in mind, we further filtered our list of packages to only include the ones that had > 1500 (50*30) monthly downloads in at least one snapshot. This reduced the number of packages under consideration from 72,211 to 12,999.

Keeping in mind the goal of our study, we designed 12 other variables to examine the influence of upstream and downstream dependencies on our response variable. We constructed the list of one-level (immediate) upstream and downstream dependencies as

[1]https://api.npms.io/v2/package/[package-name]

well as the recursive (all, as mentioned earlier) upstream and downstream dependencies for a package. For each of these four, we calculated the total number of dependencies, the total number of downloads for those dependencies (for the current month), and the average (we used median instead of mean because having a highly popular package in the list would skew the mean substantially) number of downloads for the dependencies. These variables were named using the *convention*: "(one/recursive).(up/down)stream.(dl.avg/dl/count)", where "dl" indicates the total number of downloads, "dl.avg" indicates the average number of downloads, and "count" indicates the number of dependencies.

Due to the skewed nature of all the predictor variables, we took logs of all the variables for our analysis.

4 ANALYSIS AND RESULTS

We performed the analysis of the data in two steps: First, we ran Linear Regression (LR) individually on each of the 6 snapshots, with and without the 12 dependency related variables, and compared the results. In this step we only performed model fitting, and no prediction. In the second step, we combined data from all the 6 snapshots, added a "Date" variable to account for the seasonality component, and fitted a Random Forest (RF) model and performed prediction (70% of the data was used for training and 30% was used for testing). We performed a 10-fold cross-validation using Random Forest by recursively reducing the number of predictors (less important predictors removed first) in each step to obtain the optimal number of predictors by trading model complexity and cross-validation error. Then we verified how many and which of the 12 variables we introduced are in the optimal set of predictors so obtained, and the model performance with these predictors. We used the adjusted R^2 value as our performance metric, since we are trying to predict the ratio of downloads, which is a continuous variable. As a side analysis, we also fitted the Random Forest model on the combined data with a binary response variable that indicates whether the number of downloads will go up in the next month, i.e. if the ratio will be more than 1 or not. For this step, we calculated the AUC under the ROC curve as well as the sensitivity and specificity reported by the model, with the same 70-30 ratio for training and testing sets. The analysis was performed in R.

The linear regression models had adjusted R^2 values ranging between 0.02331 and 0.17300 (median: 0.05301, standard deviation: 0.05373) for the models with the 12 variables we introduced, and between 0.00957 and 0.15640 (median: 0.02008, standard deviation: 0.05646) without those variables. Moreover, the adjusted R^2 value with our extra variables was seen to be better for every snapshot. The Variance Inflation Factor(VIF) for the predictors never exceeded 5, so we do not have a serious multicollinearity problem in this analysis. This leads us to believe the variables we added increase the explanatory power of the model. The set of significant variables were seen to be different between different snapshots. The variables representing the counts, downloads, and average downloads of the recursive downstream dependencies of a package were seen to be significant in five out of the six snapshots, and number of releases and commits in last one year were significant in four. The coefficients for total recursive downstream downloads and number of commits were positive; for number of releases it was positive in two, negative in two, and for the rest the coefficients were always negative. Since we used logs of all the variables, the signs of the coefficients indicate the signs of the powers of the variables in direct relationship with the response variable.

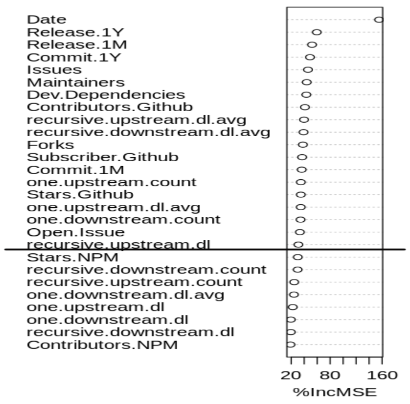

Figure 1: Variable Importance plot

The analysis performed in this step also revealed that the total number of downloads of all NPM packages was very different across the snapshots, indicating the presence of a strong seasonality component, therefore, we added the date of data collection as a seasonality variable when we aggregated the data from different snapshots for further analysis.

We ran a cross-validation exercise using Random Forest by recursively reducing the number of predictors. The exercise indicates the optimal number of predictors to be **19**, from our original set of 27 predictors. The variable importance plot, ordered by percent increase in mean-squared error, of all variables is shown in Figure 1, the variables listed above the straight line are the top 19 predictors. It can be seen that 6 of the 12 variables we introduced are in the list of top 19 predictors. The top predictors include the date of data collection, as a seasonality component, which is the most important predictor by far, along with the number of releases and commits in a year and a month, number of issues (total and open), number of forks, stars, contributors, and subscribers to the GitHub repository of the package, number of development dependencies, with the number of immediate upstream and downstream dependencies, the average popularity (average downloads) of all upstream and downstream dependencies, overall popularity (total downloads) of upstream packages, and average popularity of immediate upstream dependencies.

Finally, we ran a 10 fold cross-validation using the 19 predictors deemed important from the analysis in the last step. The resultant adjusted R^2 varied between 0.380 and 0.496 (median: 0.416, standard deviation: 0.031).

The set of variables that were shown as important from the Random Forest analysis are different from the ones that came out as significant from the LR analysis. One of the reasons for that is likely due to the fact that when we fitted the LR models, we did not perform any prediction, so the variables deemed significant from that analysis are listed based on their explanatory power, but the importance of the variables from the RF analysis are listed according to their predictive power, and these two criteria are not the same [10].

When we used the Random Forest model (with all 27 variables) for predicting if the ratio will be more than one (i.e. if the downloads will increase), the value of AUC under the ROC curve was 0.73, and

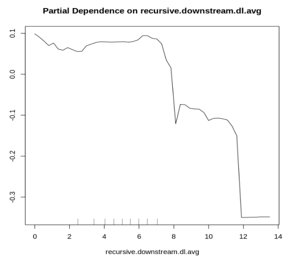

Figure 2: Partial Dependence plot from Random Forest model for "recursive.downstream.dl.avg"

values of sensitivity and specificity were 0.66 and 0.56 respectively. When the model was fitted without the 12 variables we introduced, the values of AUC under the ROC curve, sensitivity, and specificity were 0.65, 0.59, and 0.53 respectively.

5 DISCUSSION

As in many time series, the current number of downloads of a package is a very strong predictor of downloads in the next month (the autoregressive model). We, therefore, focus on the changes in the number of downloads to understand what drives increase (or decrease) in the popularity of a package. Since the number of downloads is a very complicated (and unknown) function of actual usage due to multi-tier caching and usage that is based on regression tests run by automatic builds [11], it is necessary to use modeling procedures to estimate the numbers of downloads. Despite that, it is still possible that at least some of the observed trends are not directly based on increases or decreases in usage but may be an artifact of how the downloads are recorded. From our analysis it is evident that the count and popularity of upstream and downstream dependencies of a package are significant both in terms of explanatory power (as revealed by LR analysis) and predictive power (as revealed by RF analysis). It is also evident that among the dependency variables, the recursive ones are equally or more important, highlighting the value of the supply chain perspective. The seasonality component, activity of a package (number of releases and commits), and overall engagement by users (GitHub stars, forks, subscribers, contributors) were other important predictors of the change in popularity.

We have not performed any causal intervention analysis in this study, so it would be impossible to say if the factors we found to be influencing the change in popularity are the causes behind an increase in popularity or if it is a case of confounding. However, we have a few speculative hypotheses from the results of the analysis we performed.

1: The partial dependence plot of "recursive.downstream.dl.avg" shown in Figure 2 indicates that relationship is essentially non-linear, which explains the low R^2 value reported by linear models. In the plot, negative values in the y-axis mean that the increase in downloads is less likely for that value of the independent variable (x-axis) according to the model, and vice-versa. The flat regions may arise due to no dependence in that range of values for the independent variable, or because of discretization or fewer observations in that range of values for the independent variable. The plot, therefore, likely indicates that up to a value of 8 for "recursive.downstream.dl.avg" the downloads are likely to increase, for more than 8, they tend to decrease, and for 12 or more, a decrease is even more likely.

2: The reason why upstream dependencies are important (RF analysis) might represent a case of reverse causality, whereby a popular package increases the number of downloads of its upstream packages.

3: We found that the seasonality component is very significant (RF analysis). To verify we plotted the total number of downloads by all packages for the different snapshots, and found that the variation was significant. The variation in the median number of downloads was found to be even larger. This is likely an artifact of the NPM ecosystem.

4: The reason why more releases and commits are significant might be due to a combination of effects discussed in [11] and due to how actual humans treat a more active project.

As a continuation of this work, we would like to look more closely into the dependency networks of packages, identifying the dependencies that are deleted, added, and remain unchanged between releases, and identify the effect of each component. We would also like to differentiate between different types of packages (build tools, test tools, general components etc.) and identify the dependency networks and effects of change in the network for them. Overall, we hope our work can help understand the effects of dependencies of a package on its popularity and, more generally, be able to interpret the complicated function that downloads of dependent packages represent.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 385–395.
[2] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 303–312.
[3] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 334–344.
[4] Aparna A Chhajed and Susan H Xu. 2005. Software focused supply chains: Challenges and issues. In *Industrial Informatics, 2005. INDIN'05. 2005 3rd IEEE International Conference on*. IEEE, 172–175.
[5] Diya Datta and Sangaralingam Kajanan. 2013. Do app launch times impact their subsequent commercial success? an analytical approach. In *2013 International Conference on Cloud Computing and Big Data (CloudCom-Asia)*. IEEE, 205–210.
[6] Barbara Farbey and Anthony Finkelstein. 1999. Exploiting software supply chain business architecture: a research agenda. (1999).
[7] Jack Greenfield and Keith Short. 2003. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 16–27.
[8] Jacqueline Holdsworth. 1995. *Software Process Design*. McGraw-Hill, Inc.
[9] Elias Levy. 2003. Poisoning the software supply chain. *Security & Privacy, IEEE* 1, 3 (2003), 70–73.
[10] Galit Shmueli. 2010. To explain or to predict? *Statistical science* (2010), 289–310.
[11] Laurie Voss. 2014. numeric precision matters: how npm download counts work. https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts
[12] Simon Weber and Jiebo Luo. 2014. What makes an open source code popular on git hub?. In *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*. IEEE, 851–855.
[13] 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.
[14] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*. Springer, 95–110.
[15] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2014. Patterns of folder use and project popularity: A case study of GitHub repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 30.