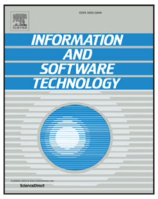
Information and Software Technology 126 (2020) 106365

Contents lists available at ScienceDirect

Information and Software Technology

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



Simplifying the Search of npm Packages

Ahmad Abdellatif[1,*], Yi Zeng[2], Mohamed Elshafei[1], Emad Shihab[1], Weiyi Shang[2]

Department of Computer Science and Software Engineering Concordia University, Montreal, Canada

ARTICLE INFO

Keywords:
JavaScript
NPM
Node.js
NPMS
Code reuse
Empirical studies

ABSTRACT

Context: Code reuse, generally done through software packages, allows developers to reduce time-to-market and improve code quality. The npm ecosystem is a Node.js package management system which contains more than 700 K Node.js packages and to help developers find high-quality packages that meet their needs, npms developed a search engine to rank Node.js packages in terms of quality, popularity, and maintenance. However, the current ranking mechanism for npms tends to be arbitrary and contains many different equations, which increases complexity and computation.

Objective: The goal of this paper is to empirically improve the efficiency of npms by simplifying the used components without impacting the current npms package ranks.

Method: We use feature selection methods with the aim of simplifying npms' equations. We remove the features that do not have a significant effect on the package's rank. Then, we study the impact of the simplified npms on the packages' rank, the amount of resources saved compared to the original npms, and the performance of the simplified npms as npm evolves.

Results: Our findings indicate that (1) 31% of the unique variables of npms' equation can be removed without breaking the original packages' ranks; (2) The simplified npms, on average, preserves the overlapping of the packages by 98% and the ranking of those packages by 97%; (3) Using the simplified npms saves 10% of packages scoring time and more than 1.47 million network requests on each scoring run; (4) As the npm evolve through a period of 12 months, the simplified-npms was able to achieve results similar to the original npms.

Conclusion: Our results show that the simplified npms preserves the original ranks of packages and is more efficient than the original npms. We believe that using our approach, helps the npms community speed up the scoring process by saving computational resources and time.

1. Introduction

According to a recent Stack Overflow survey [1], JavaScript is the most popular programming language today. One of the key reasons for its vast popularity is the fact that JavaScript provides developers with a plethora of online resources [2–4], including an ecosystem where developers can find and reuse packaged code.

The Node Package Manager (npm) is the largest repository of JavaScript packages [5,6]. It has been steadily gaining popularity over the past few years, and currently hosts more than 700 thousand Node.js packages[3]. Although this large number of packages is a major benefit to the developer community, it also brings with it some serious challenges. One of these challenges is *finding the right package* [7]. To help facilitate the effective search of more than 700,000 npm packages, npm uses npms[4] as its official search engine [8]. Developers use npms to search for an npm package to use in their software. Moreover, many SE researchers utilize npms to collect package metrics for their studies [9,10].

Npms ranks packages based on a number of factors that are grouped into three main categories: quality, popularity, and maintenance. Each of the aforementioned categories is composed of multiple metrics, where quality and maintenance are composed of four metrics while popularity is composed of three metrics. Finally, those metrics are aggregated to come up with the final score.

However, recent discussions have shown that sometimes npms scoring may be out of date, hindering its up to date packages ranking [11]. One main reason for this is the fact that npms metrics take significant effort and time (more than 180 h) to calculate, causing npms to only update its scoring every two weeks [12].

* Corresponding author.
E-mail addresses: a_bdella@encs.concordia.ca (A. Abdellatif), ze_yi@encs.concordia.ca (Y. Zeng), m_lshafe@encs.concordia.ca (M. Elshafei), eshihab@encs.concordia.ca (E. Shihab), shang@encs.concordia.ca (W. Shang).
1 Data-driven Analysis of Software (DAS) Lab.
2 Software Engineering and System Engineering (SENSE) Lab.
3 https://www.npmjs.com/

4 https://npms.io/

https://doi.org/10.1016/j.infsof.2020.106365
Received 4 October 2019; Received in revised form 25 May 2020; Accepted 1 June 2020
Available online 7 June 2020

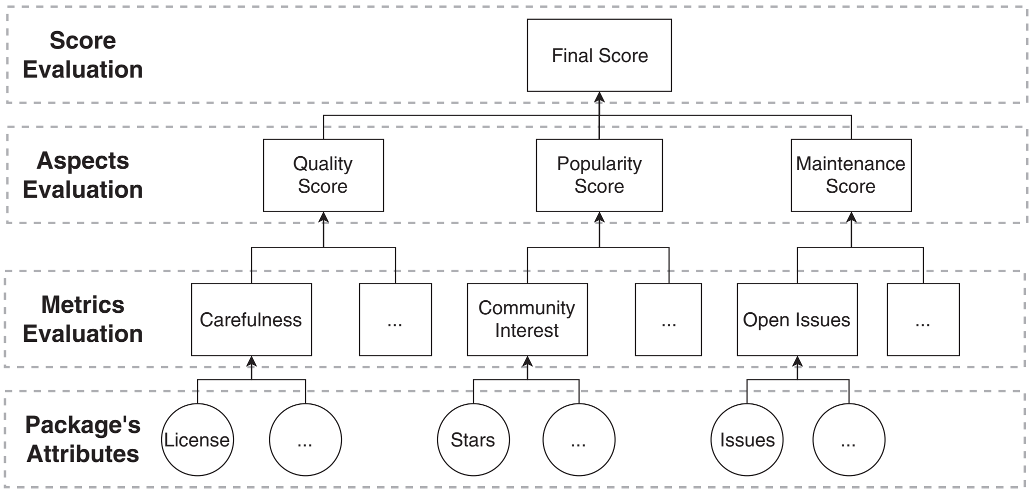


Fig. 1. An overview of npms score evaluation hierarchy.

Armed with the challenge, we set out to have a closer look at npms scoring equation. We carefully reviewed npms documentation and source code to determine exactly why npms scoring is too resource intensive. Our investigation showed that first, npms ranking equation, although comprehensive, is rather complex. Second, there is no clear reasoning behind the composition of the ranking equation or the attributes it includes.

Hence, in this paper, we set out to achieve two main goals. Firstly, we aim to examine the need and importance of each package attribute to the existing npms scoring equations in order to simplify it. Secondly, we perform an empirical study to examine the impact of our simplification on common search results on npms **(RQ1)**. Next, we examine the impact of the change of the final score between the original and simplified npms on the package rank in the search results **(RQ2)**. Then, we aim to measure the impact of simplification on npms resources consumption **(RQ3)**. Finally, we study the impact of our simplification on the package scores as npm evolves **(RQ4)**. Our findings show that (1) our approach reduced the number of used attributes in the package scoring by 31% (2) the difference in the package final score between the original and simplified npms increases/decreases the packages rank by one on average. (3) the simplified equation (which we refer to as npms-simplified) saves 19 h (10%) of the processing time for every npms scoring process run (4) search results of the simplified equation are nearly identical to the original npms results, having an overlapping score of 98.3% and a ranking similarity of 96.7% and (5) the simplified equation does not decay as npm evolves. We believe that improving the efficiency of npms indirectly impacts the npm ecosystem, since npm uses npms to find the packages related to the search query. In addition, our study helps the community to understand the impact of different metrics on the ranking score of the npms-original. Moreover, this paper provides the initial steps for the npms community to further simplify and improve the npms search engine. We make our data publicly available to help advance future research in the community [13].

Paper organization. Section 2 provides the background to the npms architecture and related concepts. Section 3 describes the simplification approach of npms. Section 4 explains the validation of the simplified npms equations and presents our findings. Section 5 discusses our findings. Section 6 presents the related work. Section 7 discusses the threats to validity, and concludes our paper in Section 8.

2. Npms background

Prior to delving into our empirical study, we provide an overview of npms, its components, and ranking mechanism. As mentioned earlier, npms is an open source search engine for npm. Npms analyzes the npm ecosystem, collects different packages attributes from different sources (e.g. GitHub) to compute different metrics (e.g. Community Interest). Then, it uses the computed metrics to calculate the aspect scores (e.g. maintenance). Finally, the packages' final score are calculated based on the computed aspects. The npms scores hierarchy is shown in Fig. 1. This process is done for every single npm package, every 15 days [12]. Fig. 2 shows an overview of the npms components and their modules. There are three main components that are involved in the final score calculation:

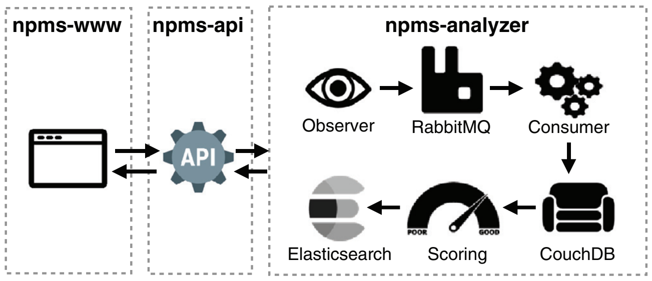


Fig. 2. An overview of npms architecture.

- **npms-www** is the interface of npms (i.e., website). It allows the users to search for relevant packages through keyword search. So, when a user types a search query, it sends the user query to the npms-api component and presents back the search results to the users.
- **npms-api** is the conduit between the npms-www and the npms-analyzer components. It receives the search query from the npms-www and forwards it to the npms-analyzer component to retrieve package information. Finally, it returns the query results to the npms-www component to be presented to the users. The query results contain package information such as name, version, scores in terms of quality, popularity, and maintenance.
- **npms-analyzer** is the npms core component, which is responsible for collecting data and storing the npm packages. The npms-analyzer is composed of five modules namely: **Observer** which continuously pushes packages that need to be analyzed. It contains two types of observers namely: (1) real-time observer, which monitors the npm ecosystem and pushes new or updated packages to the RabbitMQ and (2) stale observer, which pushes packages that have not been analyzed for more than 15 days to the queue in order to be analyzed [14]. The **RabbitMQ** module maintains all the npm packages that are waiting to be analyzed in a queue and supports automatic

Table 1
The 11 equations in the metrics evaluation level and their 22 unique variables at the package attribute level.

Aspect evaluation	Metrics evaluation	Package's attributes	Description	Coefficients before simplification	Coefficients after simplification
Quality Score	Carefulness	License	Evaluation of a package license	0.33	0.25
		Readme	Evaluation of a readme file	0.38	0.33
		Linters	Evaluation of code linters configuration	0.13	0.12
		Gitignore	Evaluation of gitignore file	0.08	**0**
		Changelog	Evaluation of changelog file	0.08	**0**
	Tests	Tests	Size of tests	0.6	0.69
		Status	Build status evaluation	0.25	**0**
		Coverage	Test coverage evaluation	0.15	0.26
	Health	Outdated	Number of outdated dependencies	0.5	0.5
		Vulnerability	Number of vulnerable dependencies	0.5	0.5
	Branding	Homepage	Evaluation of custom homepage	0.4	0.63
		Badges	Package has badges	0.6	**0**
Popularity Score	#Downloads	Downloads30	Mean downloads per month	1	1
	Downloads	Downloads60	Mean downloads rate per 2 months	0.25	0.59
	Acceleration	Downloads180	Mean downloads rate per 3 months	0.25	0.64
		Downloads365	Mean downloads rate per 12 months	0.5	**0**
	Community Interest	Stars	Number of stars	1	1.03
		Forks	Number of forks	1	1.09
		Subscribers	Number of subscribers	1	**0**
		Contributors	Number of contributors	1	**0**
Maintenance Score	Releases Frequency	Releases30	Mean releases per month	0.25	0.25
	Commits Frequency	Commits30	Mean commits per month	0.35	0.31
	Open Issues	#Issues	Number of issues	1	1
		#Closed	Number of closed issues	1	1
	Distribution	#Issues	Number of issues	1	1
		#Closed	Number of closed issues	1	1
		Conditioning	Time to close an issue	1	1

retries of package analysis process when it crashes or fails. The **Consumer** module fetches the package metadata (e.g., package name, version, author, etc.) from RabbitMQ, downloads the package source code, and collects information about the package from GitHub (e.g., star count, fork count, issues count, etc.) and npm (e.g., download count, dependency count, etc.). After the required information is collected, the consumer evaluates the aspect scores of quality, popularity, and maintenance for each package, and saves the collected information and aspect scores in the CouchDB database. The consumer repeats those steps for all the packages in the RabbitMQ. Finally, the **Scoring** module retrieves the aspect scores (i.e., quality, popularity, and maintenance) of all analyzed packages from CouchDB. Then, it calculates an aggregation value by normalizing the aspect scores and uses the computed values to calculate the package final score. Finally, the final score is stored in the **Elasticsearch** module, which provides the search feature responding to requests from the npms-api component.

Fig. 1 presents an overview of the score evaluation hierarchy that **npms-analyzer** uses to evaluate the packages' final score. The hierarchy consists of four levels bottom up: (1) 26 Package's Attributes (2) 11 Metrics Evaluation (3) 3 Aspects Evaluation (4) Score Evaluation (final score). Table 1 shows the aspects, their associated metrics and package attributes that map to the metrics. Initially, the attributes of the packages are collected by npms-analyzer. For example, the **license** attribute examines whether there is a license specified in the package description, the **stars** attribute indicates the number of stars of the package in GitHub, the **issues** attribute indicates the number of issues of the package in GitHub. After all the package's attributes are collected, the **npms-analyzer** calculates the metrics for the package based on the attributes. For example, the **carefulness** score is calculated based on a predefined Eq. (1):

$$\begin{aligned} Carefulness = (&license * 0.33 + linters * 0.13 + readme * 0.38 \\ &+ ignore * 0.08 + changelog * 0.08) * \alpha \end{aligned} \tag{1}$$

The α in Eq. (1) is a special constant that is used to adjust the weights of the equation. It assigns different weights to the equation based on whether the package is stable or deprecated. In particular, the values 0, 0.5, 1 represents the fact that α is deprecated, not stable, or stable, respectively.

Note that the equations and their associated coefficients are decided by npms. We have carefully read the documentation, examined the source code and comments of npms and did not find any rationale for the various coefficients and constants used in its implementation. Hence, we believe that a formal empirical investigation of the package attributes may help in deriving a simpler equation, reducing the performance overhead, while at the same time maintaining the effectiveness of npms.

To understand the npms ranking mechanism in the search results, we examine the npms source code to understand its package ranking mechanism since we did not find any documentation that explicitly explains the packages ranking. Npms utilizes the package's final score to calculate the ranking of the package in the search results page. Since npms is a search engine, it also considers the semantic word similarity between the user's query and the package keywords. In particular, it uses one of the following two equations (extracted from the npms code) to calculate the packages' rank that appears in the query search results

$$package\_rank\_score = 100{,}000 + package\_final\_score \tag{2}$$

$$package\_rank\_score = similarity\_score * package\_final\_score^{15.3} \tag{3}$$

The packages shown in the search results are sorted according to their ranking score (package_rank_score) where the package with the highest score appears at the top of the search result. In case the user search query has an exact match with a package name, npms uses Eq. (2) to calculate the ranking score of a package. The high constant value in the equation (i.e., 100,000) is added to the package's final score to ensure that the package appears at the top of the search results since the user might search for that package.

On the other hand, if there is no npm package name that fully matches the user's search query, the package's ranking score is calculated using Eq. (3). The Elasticsearch component is responsible for calculating the similarity_score between the search query and the package's name, keywords, and description using BM25 Algorithm [15].

3. Simplifying npms

Since our goal is to examine whether the npms equations can be simplified or not, in this section, we first present the collected dataset used in the simplification process. Second, we explain the techniques used to simplify the npms equations, namely feature selection methods. Finally, we illustrate the process of the equations simplification.

3.1. Experimental dataset

To conduct our study, we first needed to know the attributes, metrics, aspects, and final scores provided by npms for all the npm packages. To obtain the dataset for our experiment, we develop a script to query the npms API[5] and save the metadata into our database. In particular, we replicate the npm registry[6] and generate a list of all the npm packages. The list contains the metadata (e.g., author, package name, etc.) of the npm packages. The metadata is used to uniquely identify each npm package when querying the npms API. Then, for each package in the list, we write another script to take the package metadata as input, send a query to the npms API and retrieve the scores of each level (i.e., attributes, metrics, aspects, and final score), and store it in our database. The dataset was collected on July 12th, 2018 and contained data of 735,476 npm packages.

3.2. Feature selection methods

In order to *eliminate any unuseful attributes of a package while preserving its final score*, we employ feature selection methods [16]. Generally speaking, the purpose of feature selection methods is to identify the best subset among many variables to include in an equation. We choose to use feature selection methods since they are a perfect fit for our problem (i.e., removing attributes that are not useful, reducing noise and complexity) [16], well studied in the literature, and have shown to be effective in many domains, including software engineering (e.g., [17–19]).

There are several feature selection methods in the literature, however, we chose to employ the most commonly used ones, namely, forward selection (FS), backward selection (BS), and stepwise selection (SW) [20,21]. To measure the performance of the statistical model, we use the Akaike Information Criterion (AIC) [16] that has been used in similar studies [22–24]. AIC estimates the balance between the statistical properties and parameters of a model. In other words, it measures the trade-off between the goodness of a model fit and its simplicity. Another advantage of using the AIC is that it discourages models from over-fitting as it includes a penalty on the increasing number of estimated parameters [25]. Therefore, it is preferred to select the model with the smallest AIC. That is, the model with the lowest trade-off between its fit and simplicity has the best overall statistical properties and parameter balance.

Forward selection (FS): starts with just an intercept and then sequentially adds the package's attributes which reduces the most of AIC. The process terminates when no reduction can be obtained in the AIC by adding more attributes. For example, to simplify the *Carefulness* (Eq. (1)) and preserve the value of the equation. The forward selection technique starts with no attributes, then, adds one attribute at a time, as long as, the added attribute reduces the model's AIC. In other words, it adds only one attribute which keeps AIC at the lowest value in each round, as shown in Table 2. This table shows that at each round starting from Round #1 to Round #4, each variable is tested to find how it may reduce the model's AIC value if the variable is added. In Round #1, we can see that variable *license* reduces the model's AIC value from 115 to 73; therefore, it is added to the model. Then, each remaining variable is tested to find how it reduces the model's AIC value if an additional variable is added to model besides the existing ones "*license*". In Round #2, we can see that variable *readme* reduces the model's AIC value to 63; therefore, it is added to the model beside "*license*". In Round #3, variable *linters* reduces the model's AIC value to 59; therefore, it is added to the model beside "*license*" and "*readme*". In Round #4, we can see that none of the remaining variables reduce the model's AIC value below 59; thus, the FS terminates.

Table 2
Example of applying four rounds of forward selection method on *Carefulness* equation.

Variable	AIC			
	Round #1	Round #2	Round #3	Round #4
+ *license*	**73**	–	–	–
+ *readme*	76	**63**	–	–
+ *linters*	77	69	**59**	–
+ *ignore*	97	88	68	65
+ *changelog*	103	95	85	68
/ none	115	73	63	**59**

Table 3
Example of applying three rounds of backward selection method on *Carefulness* equation.

Variable	AIC		
	Round #1	Round #2	Round #3
- *license*	183	68	77
- *readme*	167	67	70
- *linters*	157	65	68
- *ignore*	136	**59**	–
- *changelog*	**113**	–	–
/ none	195	113	**59**

Backward selection (BS): starts with a model that contains all package attributes. Then, the attributes are deleted one by one until no reduction of AIC can be obtained by removing more attributes. In each round, the attribute that causes the biggest reduction in the AIC is removed. Table 3 shows an example for the *Carefulness* equation where we start with a full model containing all the attributes of interest. Then, the attribute *Changelog* is removed so that the AIC value drops to 113 which is the lowest possible value in Round #1. We keep repeating this process, as long as, removing attributes reduces the model's AIC. In other words, BS removes the attribute that reduces AIC to the lowest value in each round, as shown in Table 3.

Stepwise selection (SW): is a method that allows to move in either direction, dropping or adding attributes at the various rounds in the process. Stepwise selection involves starting off with a backward approach and then potentially adding back attributes if they later appear to be significant. Stepwise selection reconsiders all the dropped variables (except the recent one) after each round to introduce them again to the model and test their significance. For example, at round#n adding attribute *linters* from the previously removed attributes (*linters, ignore, and changelog*) reduces the AIC to the lowest rather than removing any attributes from the existing ones (*license, readme*), as shown in Table 4.

After explaining the different simplification methods and how they work, we explain the process of simplifying the npms equations in the next subsection.

3.3. Simplifying the npms equations

After describing the collected dataset and feature simplification methods, we present our simplification approach as shown in Fig. 3. We apply the simplification methods for every npms equation at the metrics evaluation level to reduce the number of collected attributes as there is a direct relation between the metric and attribute levels. In

[5] https://api-docs.npms.io
[6] https://docs.npmjs.com/misc/registry

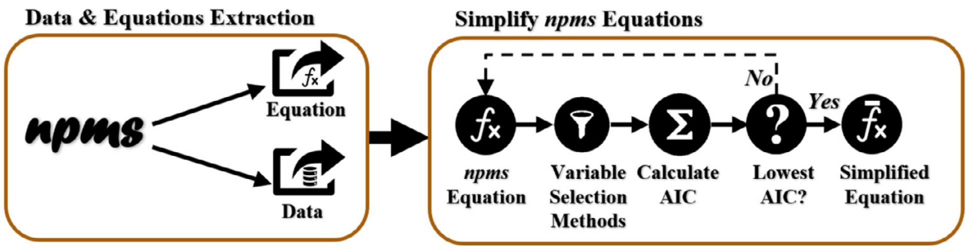


Fig. 3. An overview of our approach to simplify npms.

Table 4
Example of applying stepwise selection method on *Carefulness* equation.

Variable	AIC Round#n
- *license*	76
- *readme*	73
+ *linters*	**59**
+ *ignore*	68
+ *changelog*	85
/ none	63

particular, for each equation, we separately apply the FS, BS, and SW. And for each feature selection method, we select the result that leads to the lowest AIC for that equation. Finally, we choose the equation (simplified) that has the least number of attributes. For example, we apply each simplification method on the 'Tests' equation, then we select the equation results from the SW because it has the lowest AIC compared to other simplification methods (i.e., FS and BS). The SW added 'Tests' and 'Coverage' packages attributes only to the new 'Tests' equation while changing their factors. As the 'Status' attribute is not added to the new equation, we assign zero weight to it as shown in Table 1. It is important to emphasize that we repeat the process on all equations at the metric evaluation level. Table 1 shows the variable coefficients for attributes before and after the equation simplification using the FS, BS, SW methods at the metrics evaluation level. It is important to emphasize that the feature selection methods remove the insignificant variables regardless of their coefficients in an equation, then rebalance the remaining variable's coefficients in order to maintain the same output value as the original equation. For example, the attribute 'coverage' has the smallest coefficient in the 'Tests' equation with coefficient value 0.15. However, after the simplification of the 'Tests' equation, the attribute 'coverage' remains in the equation with a different coefficient (0.26), while the attribute 'status' which had a coefficient higher in the original equation, has been removed in the simplified npms (i.e., 0).

In total, there are 22 unique attributes used to calculate the 11 metric evaluations as shown in Table 1. It is important to note that some attributes are used to calculate more than one metric. For example, 'Downloads' is an attribute that is presented in terms of a period of a month or year and is used to calculate the '#Downloads' and 'Downloads Acceleration' metrics. By using the simplification methods, the new metrics evaluation level equations have 15 unique attributes (i.e., 31% of attributes are removed), as shown in the "After Simplification" column of Table 1 with zero values. During the simplification process, we notice that all of the selection methods give us the same results. This strengthens our conclusion that the removed packages attributes are actually useless for the model.

One might think that simplifying the aspect evaluation level would reduce npms' complexity more by removing the insignificant metrics and their attributes. However, we argue that there is no direct relation between the aspect evaluation level and the package attributes level. Because the package's aspects values are calculated based on the metrics evaluation values of the package itself and the aggregated metric values of all packages [26]. Hence, the complexity of assessing the significance of the attributes at the aspect level is increased because of the aggregation.

4. Empirical case study

In this section, we perform an empirical study to examine the impact of our simplified npms equations. We formalize our case study with the following three research questions:

- RQ1: How well can we preserve the search results?
- RQ2: How is the final package ranking affected by the simplified npms?
- RQ3: How many resources can we save by simplifying npms?
- RQ4: How will our equations perform as npm evolves?

For each research question, we detail the motivation, approach, and then present the results.

4.1. RQ1: How well can we preserve the search results?

Motivation: One of the first questions that comes to mind after simplifying the npms equations is whether our simplification impacts the ranking/output of npms. Consequently, we want to examine the impact of the removed package attributes on the search results. In particular, this research question investigates the differences of the returned search results between the npms-simplified (before simplification) and npms-original (after simplification).

Approach: To address this research question, we run the Consumer and Scoring modules on all packages using both equations: npms-original and npms-simplified. Then, we save the final scores obtained from each equation into two separate database tables (one for each equation). The first database table contains the calculated scores using the npms-original equation while the second table contains the calculated scores using the npms-simplified equation. We searched for the most widely used packages at the time we performed our study in npm [27] which are nine packages in total listed by npm namely, *lodash, async, bluebird, request, express, commander, chalk, react, debug*. We assume that those packages have the most used functionalities. Therefore, users and developers are going to search for those packages to use or modify their functionalities based on their needs. Next, two of the authors read the selected packages' description and keywords in order to extract search keywords. They extract 20 keywords that are related to the main functionality of the packages as shown in Table 5. Then, we use those search words in *npms* and extract the top 100 returned hits, which correspond to 4 pages of results (npms displays 25 npm packages per page). We choose to focus on the top 4 pages since prior work showed that users typically do not consider results that are not in the first 4 pages [28].

We run the *search process* using the original and simplified npms equations and store the results of each into a separate table. Then, we compute two different metrics:

- **Package overlap:** the results overlapping (packages intersection) between the npms-original and npms-simplified search results.

Table 5
The percentage of the preserved overlapping and ranking quality per keyword search results for the top 100 packages.

Search words	Overlapping (%)	Ranking quality (%)
asynchronous	98	96
build user interface	100	99
callback	98	96
container state	100	98
create directory	100	99
date format	100	99
debug	97	96
environment	98	94
future	100	98
http API	99	95
http request	98	96
http	93	93
jQuery parser	100	100
js bundler	100	98
mongodb	97	94
parser	94	94
route	94	93
terminal style	100	99
test framework	100	98
version parser	100	99
Average	98.3	96.7

- **Ranking quality:** using the normalized discounted cumulative gain (nDCG) which is a popular method for measuring the quality of a ranking set of search results and used previously in the software engineering research [29]. We used nDCG to measure the quality of ranking for the search results obtained from npms-simplified in reference to the ranking for the search results obtained from npms-original [29].

Result: Table 5 shows the results overlapping (similarity) between the npms-original and npms-simplified for each keyword. The percentage of results overlapping varies depending on the keyword and ranges between 93% and 100%, and with an average of 98%. The ranking quality also varies between 93% and 100%, with an average of 97%.

We dive into the results to better understand the decrease in the packages overlapping percentages, e.g., in the cases where overlapping reached only 93%. We find that the reason behind that can be explained into two ways (1) the deeper we go through search results pages, the higher the overlapping becomes. The rationale behind that is, as the number of search results pages increases, it is more likely to find similar packages in the results obtained from npms-simplified and npms-original. In fact, the 93% overlapping percentage for “http” was actually increasing as we go from the first page to the fourth page (2) we find that going through more search results pages means having more packages to rank. Therefore, it is more likely to miss rank some packages. This shows the rationale behind decreasing the ranking quality percentage to 93% for some keywords such as “route” where it was actually decreasing as we encounter more packages from page one to four. We consider the achieved overlapping and ranking quality to be good, since the averages are 98% and 97%, respectively.

The simplified *npms* equations preserve the results overlapping and the packages ranks in the search results by an average of 98% and 97%, respectively.

4.2. RQ2: How is the final package ranking affected by the simplified npms?

Motivation: Since npms is a search engine, it might consider the similarity between the user’s search query and a package name (or its meta-

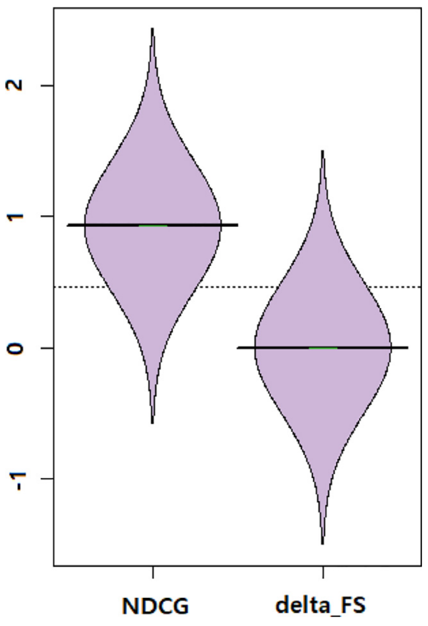


Fig. 4. The affect of differences in the final scores ($delta\_FS$) on the ranking quality (*NDCG*).

data) more than the final score which leads to the high ranking and overlapping results as shown in the RQ1. In other words, npms might be robust enough to maintain the packages’ ranks even if the scores between the original and simplified npms are different. Therefore, we want to study the impact of the final scores of the packages that appear in the search results for the original and simplified equations on the packages rank scores. In particular, we want to examine how the differences in the final scores of the packages computed by the original and simplified npms equations impact the packages’ ranks.

Approach: To accomplish this, we use the original and simplified npms equations along with the 20 keywords listed in Table 5 and their search results from RQ1. Then, we remove the packages that have the same rank in the search results of npms-original and npms-simplified since we want to examine the effect of changing the package final score on the ranking score results. For each package that appears in the top 100 search results of the original and modified npms equations, we calculate the differences in the final score ($delta\_FS$) and ranking quality (*NDCG*). Next, we run ANOVA analysis to examine the effect of the difference in the final score ($delta\_FS$) on the ranking quality (*NDCG*) for the search results. We repeated the previous steps for all the 20 keywords. It is important to emphasize that the final score of a package, and not the semantic word similarity, is the only factor that changes the packages’ ranking scores in our experiment. This is because the semantic word similarity between the user’s query and the packages does not change whether we are using the npms-original or npms-simplified equations to calculate the packages’ final scores.

Result: Fig. 4 shows the impact of differences in the final scores ($delta\_FS$) and ranking quality (*NDCG*) between the original and simplified npms of the 20 keywords search results. We notice from the figure that the difference in the final scores ($delta\_FS \leq 0.1$) changes (increases or decreases) the ranking scores of packages by at least one rank on average. For example, in the case of the package ‘Fluture’, which is returned in the search results of ‘asynchronous’, we find that the difference in the package’s final score is 0.003, which lowers its rank by 2. Furthermore, the results of all ANOVA analyses show that $delta\_FS$ has a statistically significant impact on the ranking quality (i.e., p-value <0.01). Our results clearly show that the difference in the package final score between the original and simplified npms impacts the package’s rank. We believe that our approach helps the community by simplifying the data collection process from npms with almost identical results to the original npms.

> The final score of a package has a statistically significant impact on its ranking where the difference in the final score between the original and simplified npms increases/decreases the package's rank by one on average.

4.3. RQ3: How many resources can we save by simplifying npms?

Motivation: The main purpose of simplifying the npms ranking equation is to reduce complexity and save resources. In our case, we define the performance as the time required to collect packages attributes (e.g. stars count) and evaluate the metrics values (e.g. carefulness) for all npm packages. Npms runs its package scoring process every 15 days, hence, any resource savings will have a lasting impact in terms of time-saving and computation power.

Since we already showed that the equations can be simplified, we do expect some resource savings. However, exactly how much savings is a question that needs to be answered.

Approach: To address this research question, we leveraged 2 Virtual Machines (VMs) on Amazon Web Services [30] to run the npms consumer component for evaluating all npm packages (735,476 packages). Both VMs have the same specifications where each instance is of type *m4.xlarge* [31] with 4-core CPU, 16GB memory, and Ubuntu 16.04 operating system. Next, we setup the same npms environment (e.g. consumer) on both machines using the same configurations to run the package scoring process. We used one of the VMs (VM1) to run the npms-original equations and the other VM (VM2) to run the npms-simplified equations. Two of the authors ran both VMs to calculate the scores of 50 packages and compared the output score against the score calculated manually by writing a script that calculated the packages' score based on the npms-original and npms-simplified equations. This was done as a sanity check to ensure that the environment setup and configuration are correct.

We used the same approach on Section 3 to collect the attributes, metrics, aspects, and final scores for each npm package. Finally, to compare the performance of both equations, we ran the scoring process on each VM and used a script to record the time it took to finish the scoring of the collected packages. It is important to note that we did not include the last step where the final scores are calculated, when recording time because there is no change to the scoring components. So, the time taken to calculate the final score based on the individual package score will be the same in both npms-original and npms-simplified.

Result: We compared the recorded time required for both VMs to finish the scoring process of all packages. The results show that the consumer module in VM1 took 184 h and 9 min, while in VM2 took 165 h and 12 min. So, our simplifications (npms-simplified) saves 19 h (10%) of the time required for each npms run to calculate the packages' score.

It is obvious that one of the reasons for the time reduction in the collection and metrics calculation is the number of attributes involved in computing those equations. In particular, the npms-simplified equations have less attributes (StatusEvaluation and ContributorsCount) than the npms-original equation. Consequently, npms-simplified makes less network requests to GitHub API [7] for getting the values of those attributes. In total, after we reviewed the npms code, we found that there are 10 and 8 network requests to collect the packages attributes for npms-original and npms-simplified, respectively. In other words, the code for npms-simplified saves 2 network requests for each package scoring. As a result, the npms-simplified saves 1.47 million network requests compared to using the npms-original. As we have mentioned earlier, npms runs the scoring algorithm every 15 days, therefore, using npms-simplified equation will save at least 19 h (10%) and 1.47 million requests of collecting and evaluating time compared to the npms-original equation. Finally, the saving increases in time and number of requests as the number of packages in npm grows.

[7] https://developer.github.com/v3/repos/

> The npms-simplified equations save 10% of scoring time and 1.47 million requests compared to npms-original equations for each run of the npms package scoring process.

4.4. RQ4: How will our approach perform as npm evolves?

Motivation: Since the npm ecosystem is rapidly evolving, the features used to rank the packages may also change (e.g., frequency of commits, number of stars, etc.). Hence, this research question aims to test whether the npms-simplified does not decay as the package features changed over time. In other words, we would like to know how the search results from the npms-simplified deviate from the npms-original as the npm ecosystem changes and grows. Answering this research question evaluates our simplification approach and the validity of the simplified equations across the time.

Approach: To fully answer this research question, we performed two types of complementary analyses. First, we obtained an older dataset of npms, from 2017 [32] and performed the simplification process to obtain a npms-simplified equation. Our goal of performing this comparison (between npms-simplified 2017 vs. 2018) is to see if the simplified equations will hold due to the evolution of the npm packages.

It is important to note that npms equations in 2017 and 2018 are almost the same. However, to determine the impact on the ranking of the packages, we ran the consumer and scoring modules for both npms-original and npms-simplified based on the 2017 data. Then, similar to RQ1, we measured the overlapping and the ranking quality across the top 100 packages (first 4 pages) of the same 20 searched words. The rationale behind this, is to simulate a 2017 npms-simplified running on npm packages from the same year and then measure its overlapping and ranking quality with reference to the npms-original in 2017.

Second, we ran the consumer and scoring modules for the npms-simplified in 2017 to calculate the score for all packages in the 2018 dataset. The idea of this analysis is to simulate what would happen if we used a simplification on a new dataset. In essence, we want to observe whether the 2017 npms-simplified will diverge from the npms-original in terms of overlapping and the ranking quality for each keyword as the packages evolved since 2017 to 2018, or not. Again, we measured the overlapping and the ranking quality across the first 100 packages of the same 20 searched words. This helps us to test how the 2017 npms-simplified would decay as npm packages are evolving. Fig. 5 summarizes the used approach to answer this research question.

Result: When we simplified npms based on the 2017 dataset, we ended up removing the same 7 variables reported in Table 1 which indicates the generalizability of the simplifying approach. Also, it proves that the removed variables are not very useful, regardless of the time of removal. On average, as the npm packages evolve, the similarity in term of overlapping and ranking quality between the original and the 2017 npms-simplified results remain almost identical with an average of 99.998% and 99.995%, respectively. This indicates that the npms-simplified equations in 2017 were resilient enough to evaluate the npm packages in 2018 similar to the original npms equation.

To ensure that the high ranking and overlapping is not due to the high similarity in the package attributes between 2017 and 2018 datasets, we investigate those attributes in both datasets. In particular, we perform ANOVA test to measure the difference in all non-boolean at-

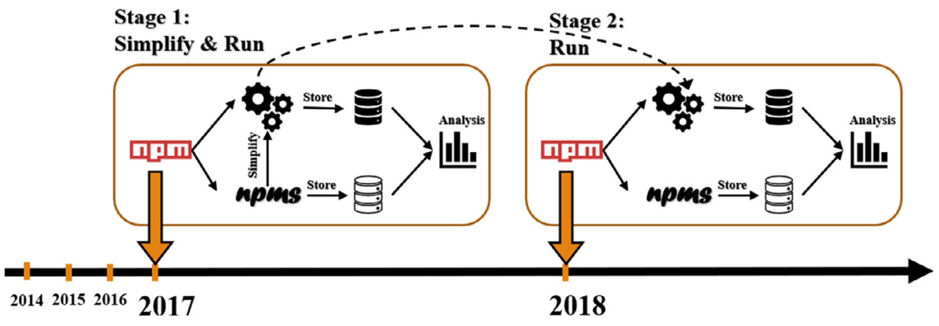


Fig. 5. The workflow for examining the evolution of npms-simplified.

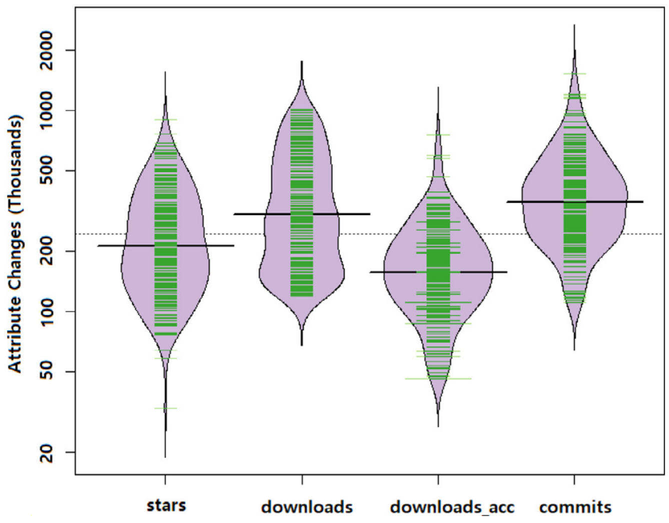


Fig. 6. The change of attributes values as the npm packages evolve.

Table 6
The percentage of the preserved overlapping and ranking quality for the random keywords search results for the top 100 packages.

Search words	Overlapping (%)	Ranking quality (%)
animations	99	98
auth	95	93
connector	100	99
crawl	100	99
editor	96	95
emoji	96	95
font	99	98
gulp	100	99
httpserver	100	97
ink	96	95
log	100	100
observer	94	93
queue	99	98
redis	98	97
render	98	97
scan	99	98
strip	95	94
terminal	95	94
test	96	95
twitter	96	95
Average	97.6	96.5

tributes (e.g., stars) that exist after the simplification process. The results show that the difference is statistically significant (i.e., p-value $\leq$ 0.01). Fig. 6, shows the bean-plot of the mostly changed attributes values from 2017 to 2018. We notice that there is a substantial change in some of the attributes such as commits per month with a median of 410K commits.

Our npms-simplified equations are not impacted by the evolution of npm packages. Moreover, we find that rank-quality and overlap are slightly impacted based by the evolution of npm packages.

5. Discussion

We examined the impact of our simplification on the search results using 20 keywords extracted from the most widely used packages as shown in RQ1 (Section 4.1). This might bias our results since the package usage is one of the inputs to the simplified npms (e.g., Downloads). To confirm our results, we randomly selected 20 packages from the entire list of npm packages, then extracted one keyword per package. The newly derived 20 keywords are shown in Table 6. Next, we followed the same approach discussed in RQ1 to compute the package overlapping and ranking quality between the original and simplified npms using the new keywords.

Table 6 shows that the overlapping and ranking quality results vary between 93% and 100% based on the keywords. The results show, on average, the overlapping and ranking quality between the npms-original and npms-simplified are 97.6% and 96.5%, respectively. These are similar to the results obtained from RQ1. To determine whether there are statistically significant differences in the results from RQ1 and the 20 keywords extracted from the random packages, we performed ANOVA analysis in the overlapping quality results, and then on the ranking quality results. We could not observe a statistically significant difference (*p*-value $>$ 0.01) in the overlapping and ranking quality results. Thus, our findings show that our results are not significantly impacted by the choice of search keywords.

The main goal of this paper is to reduce the complexity and improve the efficiency of the current npms implementation while preserving its results. Our next step is to perform a large-scale study in the future by asking real developers to rank the importance of the packages attributes when selecting a certain package. This would remove the arbitrariness of the used attributes and assign more weights to the important attributes used by npms. Consequently, npms would provide search results that are relevant to the user's search query and contain high quality packages. We believe that our study provides the initial steps to further simplify and improve npms.

6. Related work

In this section, we discuss the work most related to our study. We divide the prior work into three main areas; studies related to package dependencies, work related to ecosystems evolution, and packages searching engine.

6.1. Packages dependencies

Package inter-dependency has been the focus of recent studies [33,34]. For example, Lertwittayatrai et al. [35] leverage topological methods to visualize the high-dimensional datasets of npm. Their results show that the dependencies between packages are shaping the npm topology which indicates their importance. Decan et al. [36] explore the software ecosystem of R packages, focusing on the problem of inter-repository package dependencies in particular. They find that the R package-based software ecosystem would strongly benefit from an automatic package installation and dependency management tool. Kula et al. [37] examine the impact of micro-packages in the *npm* JavaScript ecosystem. In particular, they aim to understand the number of dependencies inherited from micro-packages and the developers' usage setbacks of using a micro-package. Their results show that micro-packages are a significant portion of the *npm* ecosystem. However, micro-packages have a large number of dependencies. Generally, these packages incur just as much usage costs as other *npm* packages. These studies show the importance of dependencies between packages in the ecosystem since the dependency is included in one of the npms equations.

6.2. Ecosystem evolution

Prior research on the evolution of the ecosystems shows that they grow quickly. For example, Bavota et al. [38] investigate the evolution of project interdependencies in the Java subset of the Apache ecosystem. The authors indicate that the interdependencies are increasing and the ecosystem size grows exponentially. Wittern et al. [5] examine and analyze the *npm* packages to understand its evolution. They analyze the metadata (e.g. download count) and dependencies for all packages between Oct 2010 and Sep 2015. The results show that the number of packages and dependencies are growing super-linearly. These studies inspired us to take the rapid evolution of npm ecosystem into our considerations when applying our simplification approach. Also, they motivated RQ4 in our study to ensure that the simplified equations of the original npms will be valid as npm ecosystem grows.

6.3. Packages searching engine

There are researchers who developed search engines that provide developers with the packages they need. Imminni et al. [39] introduce Semantic Python Search Engine (SPYSE), a web-based search engine that lays in the combination of three different aspects meant to provide developers with relevant, and at the same time high quality code through metrics such as code semantics, popularity, and code quality. Other researchers collect different data for their studies using npms [40–42]. For example, Abdalkareem et al. [32] studied the popularity of the trivial packages usage among developers using the npms metrics.

However, to the best of our knowledge, no prior work focused on npms or simplifying its equations, our work complements the previous work in different ways. First, it helps researchers to focus on the metrics that have a significant effect on the package final score. Also, it reduces the effort of the other researchers in collecting the npm data using npms. Finally, we believe that npms has a potential for further improvement, so we encourage the research community to focus on it more.

7. Threats to validity & limitations

In this section, we discuss the threats to internal, construct, and external validity of our study. Also, we discuss the limitation of practical evaluation.

7.1. Internal validity

It concerns confounding factors that could have influenced our study results [43]. First, we measured whether the quality of our equations would decline as npm ecosystem evolves over time by comparing the npms-original and npms-simplified when applied to the 2017 and 2018 datasets. The time period might not be long enough to cause a negative effect on our equations. However, it is difficult to get the data before 2017. Furthermore, we measured the mean progression per month for the search results in term of package's attributes between 2017 and 2018. Table 7 shows the amount of monthly progression for the results of each search words in terms of releases, downloads (100 K), stars (1 K), forks, and commits (1 K).

We selected the top 100 packages returned in our analysis of the packages overlapping and ranking quality in RQ1. Therefore, including more packages might impact our results. However, most users (77%) click-through rate on search engine results go to the first three pages [44,45]. In our study, we selected the first four pages (top 100 packages). To be more confident in our approach and results, we ran the same test using the first 40 results pages (top 1000 packages). We found that the overlapping remained (98%) because more similar packages are found in different ranks. However, the nDCG decreases (91%) because more packages are ranked differently by the end of the search results list. In the future, we plan to include all the packages shown in the search results.

We used 20 keywords extracted from the most depended-upon packages on the npm website. Therefore, using different keywords may lead to different results. Nevertheless, we included the top 100 packages from the search results in our analysis and extract another 20 keywords from 20 random npm packages. The findings showed that there are high similarity in terms of ranking and packages overlapping between the modified and the original npms. Also, we plan to expand our study to include more keywords.

We applied FS, BS, and SW techniques to simplify the npms equation. There are other variable selection techniques which may produce different results. However, the variable selection techniques that we perform are popularly used in software engineering domain [17–19] and have proven their reliability [16]. Also, we are confident in our approach since we applied it on different datasets and gave very high similarity results.

7.2. Construct validity

It concerns the relation between theory and observation [43]. We measured the performance of our modified *npms* in terms of attributes collection and metrics evaluation time, and the number of requests. The performance test is conducted one time due to the time and resources limitations. Other researchers are interested in the usage of CPU, memory and energy savings. However, running the performance test requires much time and effort. Therefore, we intend to conduct performance test several times and use different measures in the future.

7.3. External validity

It considers the generalizability of our results [43]. First, our study is conducted specifically on *npms*, while there are other search engines can be used for npm such as Google, the results might be different on these systems. According to the npm website [6], the official search feature is powered by *npms*. Therefore, our improvement on *npms* could benefit the npm community on a large scale. Moreover, our approach can be applied on different package managers' search engines. The results might be different on packages manager systems of other programming languages such as Python and R. Researchers and practitioners can replicate our study on other non-JavaScript platforms. On the other hand, we applied our approach using 2017 and 2018 npms datasets. Therefore, replicating our study using a new data in the future may lead to different results. However, we tested that generalizability of our approach and results in RQ4. Our findings show that using the original and simplified npms equations return almost identical results.

Table 7
The mean progression per month for the search results in term of package's attributes between 2017 and 2018.

Search words	Mean/month				
	Releases	Downloads (100 K)	Stars (1 K)	Forks	Commits (1 K)
asynchronous	4	38	4	7	0.5
build user interface	2	23	12	9	1.4
callback	11	26	14	1	0.5
container state	2	56	19	4	0.3
create directory	10	40	29	2	0.3
date format	1	26	35	1	0.5
debug	1	39	2	13	0.4
environment	2	26	5	2	6.4
future	1	20	14	1	1.5
http API	1	200	9	8	0.4
http request	6	3	14	4	7.7
http	1	9	23	6	0.8
jQuery parser	5	6	25	4	0.2
js bundler	7	13	4	2	6.8
mongodb	7	11	1	1	0.5
parser	1	37	31	6	0.4
route	15	29	4	1	0.3
terminal style	1	64	7	2	0.5
test framework	0	58	22	1	0.4
version parser	1	32	15	8	0.6

7.4. Limitation of practical evaluation

One of the main challenges in our study is to evaluate the relevance of the results from the simplified npms in a practical setting (i.e., having developers comment on the packages prioritized using the simplified npms). One way to accomplish this is to conduct a user study where we ask developers to search for keywords using the original and simplified npms and use the results of that user study for evaluation. However, we believe that such a study will yield very similar results between the simplified and the original npms in most cases. This is due to the high overlapping (97%) and ranking quality (96%) between the returned packages from the original and simplified npms. Therefore, it is unlikely that the developers would encounter a difference in the outputs of original and simplified npms. It is important to note that the main goal of our approach is to reduce the complexity and improve the efficiency of the current npms implementation while preserving its search results. In other words, we are not proposing an approach that would return more useful and relevant search results to developers.

8. Conclusion

The number of npm packages has seen a steady and sharp increase during recent years. As there are many choices of packages related to a specific requirement, developers use npms to search and compare the quality, maintenance and popularity of different packages, then select the one that suits their needs. However, running npms to collect data and evaluate scores for around 730,000 packages requires more time and resources because the evaluation relies on different variables collected from different data sources. After examining the documentation, source code and comments of npms, we find that there is no rationale behind the selection of variables and their coefficients.

In order to help to save processing time, npms resources and to give an insight into the impact of different metrics in packages' final scores, we performed a statistical analysis to improve the performance of npms equations while preserving their search results. By applying variable selection methods, we reduce 31% of the variables that npms uses, which leads to a decrease 19 h (10%) of scoring time and 1.47 million of network requests that npms needs to collect data and evaluate the final score for the packages. By comparing the packages overlapping and ranking quality when performing search on the original npms and our modified npms, we preserve the overlapping and ranking of packages by 98.3% and 96.7%, respectively. The discrepancy in the packages' final scores between the original and simplified npms increases/decreases the packages' ranks by one on average. We also find that as npm evolves, it has a weak impact in reducing the ranking and inclusion of the npms-simplified equations.

Our work helps to improve the performance of npms and provides a better understanding of software metrics. The npms community can benefit from our findings by saving time and computation resources. In addition, our work also sheds light on potential research areas related to the impact of software metrics on the packages' final scores.

References

[1] Stack overflow developer survey 2018, (https://insights.stackoverflow.com/survey/2017). (Accessed on 01/15/2019).

[2] P. Mohagheghi, R. Conradi, Quality, productivity and economic benefits of software reuse: a review of industrial studies, Empir. Softw. Eng. 12 (5) (2007) 471–516, doi:10.1007/s10664-007-9040-x.

[3] V.R. Basili, L.C. Briand, W.L. Melo, How reuse influences productivity in object-oriented systems, Commun. ACM 39 (10) (1996) 104–116, doi:10.1145/236156.236184.

[4] W.C. Lim, Effects of reuse on quality, productivity, and economics, IEEE Softw. 11 (5) (1994) 23–30, doi:10.1109/52.311048.

[5] E. Wittern, P. Suter, S. Rajagopalan, A look at the dynamics of the JavaScript package ecosystem, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14–22, 2016, 2016, pp. 351–361, doi:10.1145/2901739.2901743.

[6] npm, (https://www.npmjs.com/). (Accessed on 08/07/2019).

[7] npms, (https://npms.io/about). (Accessed on 08/07/2019).

[8] NPM, Searching for and choosing packages to download | npm documentation, (https://docs.npmjs.com/searching-for-and-choosing-packages-to-download). (Accessed on 03/28/2019).

[9] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, E. Shihab, 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, ESEC/FSE 2017, Paderborn, Germany, September 4–8, 2017, 2017, pp. 385–395, doi:10.1145/3106237.3106267.

[10] T. Dey, A. Mockus, Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? in: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'18, ACM, New York, NY, USA, 2018, pp. 66–69, doi:10.1145/3273934.3273942.

[11] Some modules show an older version issue #69 npms-io/npms-analyzer, (https://github.com/npms-io/npms-analyzer/issues/69). (Accessed on 01/15/2019).

[12] npms-analyzer - github, (https://github.com/npms-io/npms-analyzer/tree/103d6209ed62ffb7a2ece26d72e87e5c6be17a86/lib/observers). (Accessed on 04/12/2018).

[13] Npms results, (https://www.dropbox.com/sh/isbebtsc2a2rhkr/AABBez1pXqbOLNqMd76G1cHHa?dl=0). (Accessed on 05/03/2019).

[14] npms io, npms-analyzer, (https://github.com/npms-io/npms-analyzer/blob/103d6209ed62ffb7a2ece26d72e87e5c6be17a86/lib/observers/stale.js). (Accessed on 07/27/2018).
[15] Practical bm25 - part 2: The bm25 algorithm and its variables | elastic blog, (https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables). (Accessed on 03/10/2019).
[16] I. Guyon, A. Elisseeff, An introduction to variable and feature selection, J. Mach. Learn. Res. 3 (2003) 1157–1182.
[17] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496, doi:10.1109/TSE.2008.35.
[18] H. Wang, T.M. Khoshgoftaar, A. Napolitano, A comparative study of ensemble feature selection techniques for software defect prediction, in: 2010 Ninth International Conference on Machine Learning and Applications, 2010, pp. 135–140, doi:10.1109/ICMLA.2010.27.
[19] K. Gao, T.M. Khoshgoftaar, H. Wang, N. Seliya, Choosing software metrics for defect prediction: an investigation on feature selection techniques, Software 41 (5) (2011) 579–606, doi:10.1002/spe.1043.
[20] B. Ghotra, S. McIntosh, A.E. Hassan, A large-scale study of the impact of feature selection techniques on defect classification models, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017, pp. 146–157, doi:10.1109/MSR.2017.18.
[21] X. Chen, Y. Shen, Z. Cui, X. Ju, Applying feature selection to software defect prediction using multi-objective optimization, in: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 2, 2017, pp. 54–59.
[22] M. Kuutila, M.V. Mäntylä, M. Claes, M. Elovainio, B. Adams, Using experience sampling to link software repositories with emotions and work well-being, in: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 18, Association for Computing Machinery, New York, NY, USA, 2018, doi:10.1145/3239235.3239245.
[23] J. Chi, K. Honda, H. Washizaki, Y. Fukazawa, K. Munakata, S. Morita, T. Uehara, R. Yamamoto, Defect analysis and prediction by applying the multistage software reliability growth model, in: 2017 8th International Workshop on Empirical Software Engineering in Practice (IWESEP), 2017, pp. 7–11, doi:10.1109/IWESEP.2017.16.
[24] D. Posnett, V. Filkov, P. Devanbu, Ecological inference in empirical software engineering, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 362–371, doi:10.1109/ASE.2011.6100074.
[25] H. Akaike, A new look at the statistical model identification, IEEE Trans. Autom. Control 19 (6) (1974) 716–723.
[26] npms-analyzer/architecture.md at master npms-io/npms-analyzer, (https://github.com/npms-io/npms-analyzer/blob/master/docs/architecture.md. (Accessed on 01/16/2019).
[27] npm, npm, (https://www.npmjs.com/browse/depended). (Accessed on 04/17/2018).
[28] G.E. Dupret, B. Piwowarski, A user browsing model to predict search engine click data from past observations., in: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '08, ACM, New York, NY, USA, 2008, pp. 331–338.
[29] Y. Wang, L. Wang, Y. Li, D. He, W. Chen, T.-Y. Liu, A theoretical analysis of NDCG ranking measures, in: Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013), 2013.
[30] Amazon, Amazon web services (aws) - cloud computing services, (https://aws.amazon.com/).(Accessed on 08/07/2019).
[31] Amazon, Amazon ec2 instance types, (https://aws.amazon.com/ec2/instance-types/). (Accessed on 08/07/2019).
[32] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, E. Shihab, 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, ESEC/FSE 2017, ACM, New York, NY, USA, 2017, pp. 385–395, doi:10.1145/3106237.3106267.
[33] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empir. Softw. Eng. (2018), doi:10.1007/s10664-017-9589-y.
[34] A. Decan, T. Mens, E. Constantinou, On the impact of security vulnerabilities in the npm package dependency network, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, ACM, New York, NY, USA, 2018, pp. 181–191, doi:10.1145/3196398.3196401.
[35] N. Lertwittayatrai, R.G. Kula, S. Onoue, H. Hata, A. Rungsawang, P. Leelaprute, K. Matsumoto, Extracting insights from the topology of the JavaScript package ecosystem, in: 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4–8, 2017, 2017, pp. 298–307.
[36] A. Decan, T. Mens, M. Claes, P. Grosjean, 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), 1, 2016, pp. 493–504, doi:10.1109/SANER.2016.12.
[37] R.G. Kula, A. Ouni, D.M. Germán, K. Inoue, On the impact of micro-packages: an empirical study of the npm JavaScript ecosystem, CoRR abs/1709.04638 (2017).
[38] G. Bavota, G. Canfora, M.D. Penta, R. Oliveto, S. Panichella, How the apache community upgrades dependencies: an evolutionary study, Empir. Softw. Eng. 20 (5) (2015) 1275–1317, doi:10.1007/s10664-014-9325-9.
[39] S.K. Imminni, M.A. Hasan, M. Duckett, P. Sachdeva, S. Karmakar, P. Kumar, S. Haiduc, SPYSE: a semantic search engine for python packages and modules, in: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, ACM, New York, NY, USA, 2016, pp. 625–628, doi:10.1145/2889160.2889174.
[40] T. Dey, A. Mockus, Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? in: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'18, ACM, New York, NY, USA, 2018, pp. 66–69, doi:10.1145/3273934.3273942.
[41] A. Trockman, S. Zhou, C. Kästner, B. Vasilescu, Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 511–522, doi:10.1145/3180155.3180209.
[42] K.C. Chatzidimitriou, M.D. Papamichail, T. Diamantopoulos, M. Tsapanos, A.L. Symeonidis, Npm-miner: an infrastructure for measuring the quality of the npm registry, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, ACM, New York, NY, USA, 2018, pp. 42–45, doi:10.1145/3196398.3196465.
[43] S.B. Merriam, Qualitative Research and Case Study Applications in Education. Revised and Expanded from" Case Study Research in Education.", ERIC, 1998.
[44] P. Petrescu, M. Ghita, D. Loiz, Google organic ctr study. 2014, 2014.
[45] B. Clifton, Advanced Web Metrics with Google Analytics, John Wiley & Sons, 2012.