The NPM Dilemma: Too Many Packages or Too Few Standards?

Aakash Kotha, Hemachandran Balaji, Nikita B. Emberi, Taher Travadi, Savali Sandip Deshmukh

1. Introduction

In the realm of software development, the ever-expanding npm package landscape presents a double-edged sword. While developers have access to vast resources exceeding 800,000 packages, managing these dependencies presents a significant challenge. Building upon existing research on download trends and dependency size [1], our study delves deeper into the nuanced aspects of package classification. We seek to establish a robust classification framework encompassing code metrics, activity measures, and developer attributes, categorizing packages into distinct categories **(trivial, inactive, dead, deprecated)**. This classification, informed by prior studies [2] [3], will equip developers and the npm community with effective tools for confident navigation through the expansive package landscape.

Our research aims to explore overlooked aspects of package health beyond size or popularity metrics like download trends and dependency size. By understanding correlations between parameters, we develop precise classification systems for npm packages. Misclassification can result in developers missing critical updates, leaving users vulnerable to security threats and hindering software development progress. Identifying vulnerable packages is crucial. These packages can contain security holes that attackers can exploit to gain unauthorized access to user data or systems. Vulnerabilities often arise due to unmaintained code or coding errors and can significantly impact users, potentially leading to data breaches, malware infections, or even complete system compromise.

Our research code and data can be found on GitHub (https://github.com/travaditaher/SE_NPM_packages) for further exploration and validation.

Ultimately, our research seeks to empower developers and the npm community with actionable insights. By revealing the hidden factors influencing package performance and popularity, our **"goals"** are:

- Explore and understand the correlations between various parameters to identify the most significant factors for developing precise classification systems, distinguishing between trivial, inactive, dead, and deprecated npm packages.
- Offer practical suggestions leveraging the insights gained from classification, empowering stakeholders to make informed decisions for better package management, development practices, and community engagement.
- Provide guidance for enhancing the NPM Search Engine by incorporating refined parameters into the ranking algorithm, improving the capacity of detecting and flagging potential abandoned or deprecated packages, which can help developers make more informed decisions, ultimately improving the overall security and reliability of the npm ecosystem.

In accordance with our research findings, this paper **"contributes"** as follows:

1. Developed a methodology to categorize packages into distinct categories based on predefined parameters and thresholds and achieved classification accuracy, with 73.11% classified as dead, 7.29% as inactive, 3.06% as deprecated, and 15.96% as trivial.

2. Conducted an in-depth analysis to identify strong predictors for each package category, utilizing various statistical tests (Mann-Whitney U and Chi-Squared tests) and data preparation techniques and presented correlation values and significance levels in a heatmap, providing a comprehensive understanding of factors influencing package categorization within the npm ecosystem.

3. Investigated relationships between different package categories to ascertain the influence of one category on the likelihood of a package being classified into another and revealed significant correlations between pairs of categories like trivial-dead and deprecated-dead, offering valuable insights into interdependencies among package health categories.

4. Explored the relationship between package categories and vulnerability metrics, focusing on package health scores. Identified significant predictors for categorizing packages as healthy or unhealthy across different health categories, including security recommendations, dependency count, and download activity.

In summary, this research advances the understanding of package health and vulnerability within the npm ecosystem, offering valuable insights for developers, researchers, and stakeholders in software development and package management.

1.1 Research questions

Building upon the foundations laid by prior studies, these are our research questions that delve deeper into the nuanced world of package health:

RQ1. Classification and Prevalence:

1a. Based on code size, download trend, time since last commit, and other relevant metrics, identify the respective measures contributing to categorizing npm packages into the following classes: dead, inactive, trivial, and deprecated.

1b. Using these categories, determine the respective percentage of npm packages falling into each category.

RQ2. Correlations of Package Health:

2a. Do specific code metrics (e.g., lines of code, cyclomatic complexity) or activity measures (e.g., time since last commit, download trends, stars) significantly correlate with the defined package categories?

2b. If these correlations exist, which metrics are the strongest predictors of each defined package category?

RQ3. Exploration of Inter-Category Relationships and Vulnerability Correlation:

3a. Investigate the relationships between different package categories to understand how the presence or absence of one category affects the likelihood of a package being also classified into another category. For example, does the presence of a trivial package increase the likelihood of it also being classified as dead?

3b. Examine how each package health category correlates with package vulnerability, shedding light on the relationship between package health and security risks.

The **"motivation"** behind our research questions is based on the multifaceted nature of package management within the npm ecosystem, aiming to address concerns raised by recent studies regarding the identification of packages [3] [4]. Key studies inspired our inquiries:

1. The prevalence of "trivial" packages sparked our curiosity, inspiring us to investigate their distribution and categorization metrics (RQ1). By examining how they are classified alongside categories like dead, inactive, and deprecated, we aim to gain insights that could enhance software development practices.

2. Drawing inspiration from research on deprecation and maintenance, we are motivated to understand the dynamic nature of package categorization within npm. We aim to explore relationships between different categories and how they influence a package's classification into another (RQ2 & RQ3), potentially reshaping package management strategies.

3. Inspired by the intersection of package categorization and security, we explored the correlation between package health categories and vulnerability (RQ3). We aim to highlight the importance of integrating security considerations into npm package management, contributing to a more secure software ecosystem.

2. Background Work

Our research aimed to align our classification of "dead" npm packages with the authors' approach to identifying "dormant" packages [5]. Drawing inspiration from their methodology, we expanded by incorporating activity features such as the last commit and version dates to categorize packages. Their study provided insights into package abandonment, including package revitalization and ecosystem preservation discussions. We extended their approach by considering factors affecting package abandonment and highlighting how detecting such packages can help avoid vulnerabilities, ensuring the safety and preservation of the npm ecosystem. This would also help to allocate resources more effectively, enabling stakeholders to make informed decisions regarding project stability and sustainability.

Drawing upon the techniques introduced in a previous study, our research extends the approach by incorporating observations of download trends to identify packages exhibiting signs of decline in user engagement within NPM ecosystems [6]. While the aforementioned study focuses on utilizing package centrality trends to predict package decline, we extend this approach by incorporating observations of download trends. We leverage download trend data to classify packages into inactive categories, thus providing a comprehensive understanding of package dynamics. By integrating these techniques, we enhance the effectiveness of identifying packages at risk of decline, contributing to the broader goal of improving software maintenance and evolution practices.

We studied a comprehensive investigation where the authors thoroughly explored the deprecation of packages within the npm ecosystem. Their work provided valuable insights into the reasons for package deprecation, the impact on dependent packages, and the evolution of deprecation practices over time [3]. Inspired by their findings, we aim to extend this research by developing a classification framework encompassing not only deprecation but also other facets of package health. While their study focused primarily on understanding the dynamics of deprecation, our research seeks to broaden the scope by examining additional metrics such as code size, downloads trend, and time since the last commit to classify npm packages into distinct categories. Abdalkareem et al.[7] examined the prevalence and characteristics of trivial packages on npm, revealing that they constitute 16.8% of studied packages and are increasingly popular. Developers favour these packages for their perceived quality despite concerns about maintenance and dependency risks. Inspired by this study, we aim to categorize npm packages based on metrics like SLOC and McCabe's cyclomatic complexity, identifying packages as trivial or non-trivial. While the aforementioned paper focuses solely on trivial packages, we aim to leverage the framework introduced by Abdalkareem's research to systematically categorize npm packages into other classes mentioned in our first research question, providing a comprehensive understanding of package quality and usage.

Extracting data from npm packages is only the first step. A thorough analysis is paramount to glean actionable insights that guide informed decision-making within software engineering projects. This analysis unveils hidden patterns, trends, and correlations that would otherwise remain obscure. Statistical tests like the Mann-Whitney U test and the chi-square test emerge as powerful tools in this process, as demonstrated in the work of Arunkumar et al. [8]. The Mann-Whitney U test, a non-parametric test, shines when we need to assess differences between two independent groups. It doesn't require the data to follow a specific distribution, making it a versatile tool for software engineering projects where data characteristics might be unknown. The chi-square test, on the other hand, tackles categorical variables. It helps us determine if there's a statistically significant association between these categories, allowing us to identify potential relationships within our npm package data. By leveraging these tests, software engineers can validate hypotheses, identify statistically significant differences, and ensure the robustness of their results. Perhaps most importantly, statistical analysis allows us to understand the relationships between variables. This unveils underlying dependencies within the data, which can inform more refined package categorization. We can ultimately enhance project efficiency and reliability by creating a more nuanced understanding of package relationships. Inspired by this approach, we employed similar statistical techniques on our npm package data. This analysis helped us uncover hidden dependencies within the packages, leading to a deeper understanding of the data and ultimately improving the efficacy of our software engineering project.

3. Methodology

3.1 Study design

Our project employed a **quantitative** methodology to gain a comprehensive understanding of the npm ecosystem, allowing us to uncover both overarching patterns and intricate details related to package health, usage trends, vulnerability, and inter-category relations. This approach enabled us to capture broad trends while delving into specific nuances essential for a holistic comprehension of the npm ecosystem. To achieve this, we utilized the following methodologies:

- In analyzing the data, we utilized descriptive statistics, including mean, median, and standard deviation, to examine various metrics such as code size, dependency count, and time since the last commit for each npm package.
- Furthermore, we employed statistical tests such as the chi-square test and Mann-Whitney U test to explore relationships between code metrics (e.g., lines of code, dependency count) or activity measures (e.g., time since the last commit, download trends) and package categories, as well as package vulnerabilities. We applied the same approach to identify inter-category relations. If correlations were identified, we determined the metrics that best predicted package health, elucidating critical factors influencing package characteristics.
- Additionally, we calculated the Variance Inflation Factor (VIF) for each predictor to identify and mitigate collinear predictors.
- We employed bootstrapping techniques to bolster confidence in our results, which is particularly beneficial given the relatively small sample size [9].

3.2 Variables considered

In our study, we considered a comprehensive array of variables to capture various aspects of package health, activity, and code quality within the npm ecosystem. We incorporated a diverse set of metrics, including both code-related and activity-related variables, to ensure a thorough examination of npm packages. Specifically, we selected key code metrics like: Lines of Code (LOC), Source Lines of Code (SLOC), and Cyclomatic Complexity. These metrics offer insights into the size, complexity, and structure of the codebase, providing valuable indicators of code quality and maintainability.

Additionally, we included pivotal activity metrics such as Download Trends, GitHub statistics like Stars, Pull Request Counts, Issues/Bugs Reported and Resolved, and Maintenance & Community Engagement to gauge the strength of community support. Download Trends reflect the popularity and usage trends of a package, GitHub Stars signify its community endorsement and engagement level, Maintenance measures the level of ongoing support and updates provided by the package developers, while Pull Request Counts and Issues/Bugs Reported and Resolved offer insights into the active development and issue resolution processes. By integrating these metrics, we aimed to capture both the technical aspects of package quality and its broader ecosystem dynamics, including the responsiveness and robustness of its community support.

In total, our study incorporated over 40 variables, encompassing various dimensions of package health, activity, and code quality. These variables were meticulously selected to provide a comprehensive understanding of the npm ecosystem. All variables used in our analysis are listed clearly in the Appendix, ensuring transparency and reproducibility in our research methodology.

3.3 Data collection

3.3.1 Data collection methods used:

Embarking on a journey to unravel the intricacies of the npm ecosystem, we delved into multiple data sources to power our analysis. Initially, we gathered all package names utilizing the "all-the-package-names" package [10], revealing a vast repository of over 2.6 million packages as of January 25, 2024. Due to computational constraints and time limitations, we opted to randomly sample 70,000 parent packages for in-depth analysis. Subsequently, we procured extensive package data, comprising various features, to ensure meticulous analysis and facilitate future investigations.

Following the cleaning process and the exclusion of packages with broken or deleted Git repositories, our dataset narrowed down to approximately '28,037' packages for thorough examination. Our data collection methodology involved three primary sources:

- **NPM Repository:** Metadata for all packages, encompassing information such as the latest version information, package age, dependency counts, documentation, download statistics for the last 12 months, and more, were obtained by querying the ***NPMJS registry API*** [11]. Custom Python scripts were employed to interface with the API and retrieve the requisite metadata.

- **Git Repositories of NPM Packages:** We extracted both code and activity metrics, including lines of code, cyclomatic complexity, stars, forks, pull requests, issues, commit history, and contributors, from the Git repositories of npm packages. ***PyDriller*** [12] facilitated this process. Additionally, we scraped GitHub for supplementary information utilizing the ***lxml library with XPath***.

- **Vulnerability Statistics from Synk.io:** [13] Statistics, comprising health scores and vulnerability counts, alongside development activity metrics, were gathered from Synk.io employing web scraping techniques with ***Beautiful Soup***.

By harnessing data from these diverse sources and employing custom scripts and libraries tailored to each source, our objective was to conduct a comprehensive analysis of the npm ecosystem. We aimed to explore package characteristics, activity levels, and vulnerability trends to gain deeper insights into package health and security within the ecosystem. From these three sources, we aggregated approximately 40 features for each package, facilitating a nuanced understanding of all facets and enabling the investigation of feature thresholds to efficiently categorize each package into distinct categories.

3.3.2 Data Cleaning:

Data cleaning involves various tasks to ensure the quality and consistency of gathered data. In our dataset, a substantial portion (40%) of entries lack links to their corresponding GitHub repositories or have common npmjs GitHub links, particularly for retired or deprecated packages not marked as such on the registry. Additionally, some entries required filtering to extract only GitHub repository URLs, as they were found in different formats and subsequently standardized to HTTPS GitHub links. Due to data collection from diverse sources, some entries may lack significant information, leading to their removal. Moreover, certain values were recorded in different units, necessitating conversion into a unified unit system. Furthermore, all the date or timestamp related fields were standardized into months, facilitating numerical data analysis. Following these cleaning procedures, the resulting dataset comprises 40 columns and over 28k entries.

3.4 Outline analysis methods

3.4.1 Analysis method for RQ1(a):

We identified parameters for categorizing npm packages into the classes of dead, inactive, trivial, and deprecated. These parameters were either sourced from existing research papers or manually selected, along with their corresponding threshold values.

To categorize npm packages as **"dead"**, we drew inspiration from a research paper that classified such packages as "dormant" based on their latest activity features [5]. We focused on each package's latest version release date and the last commit date to identify our latest activity features. Following data cleaning, we faced the challenge of determining appropriate threshold values. After initial visualisation of our distributions, we observed the central tendency and variability within the data, as highlighted by the IQR range plot depicted in the box plot attached in the appendix. Notably, the skewed nature of our data prompted us to rely on the IQR method for threshold calculation, given its robustness in handling variability [14]. We opted for a conservative approach by filtering the IQR from the 25th percentile, offering a more robust threshold that accounted for potential variability and outliers [14]. Consequently, we concluded the threshold values of **31.2 months** for the latest versions and **18 months** for the last commit.

In our research, we developed a robust method for classifying npm packages as either **"Inactive"** or active based on their download statistics [6]. An inactive package is characterized by reduced engagement or usage, potentially indicating diminished relevance or ongoing development.

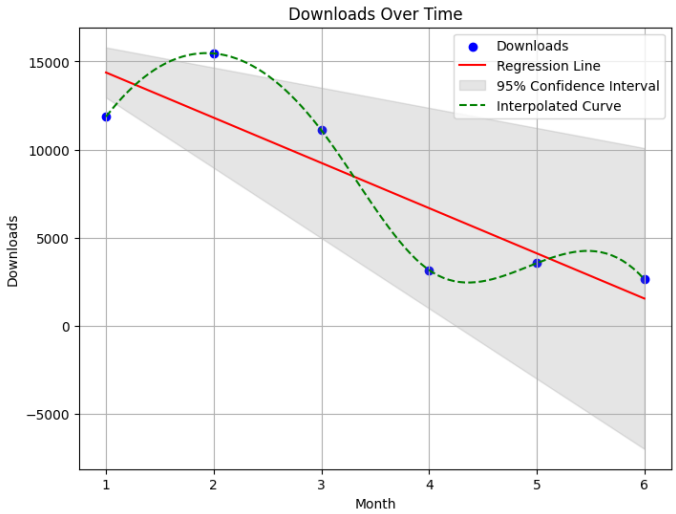

Figure 1: Downloads over past 6 months of package - "react-rrule-generator"

Drawing inspiration from previous studies [6], we analyzed the download trend of each package over the latest six months (Sep 23 to Feb 24). Using least-squares regression, we fit a linear function to the download data and examine the slope (m) of the trend to identify packages in decline. In our study, a package is classified as inactive if its download trend exhibits a

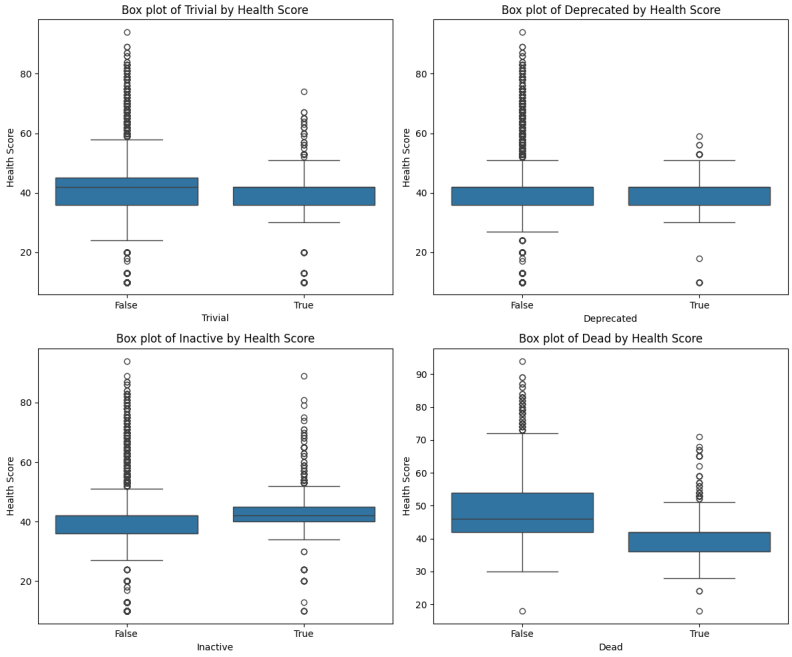

Figure 2: Box plot of each package health category with respect to Health Scores

significant negative slope. Specifically, we establish two criteria:

1. Slope Criterion: The slope of the download trend for the last six months should be $m < v$, with a default threshold value of $v = 0$.

2. P-Value Criterion: To verify the statistical significance of the negative slope, we perform the Wald Test with a conservative p-value (p) threshold, i.e., $p < \alpha$, with a $\alpha = 0.1$ (found after manually analyzing 10% of the population sample).

As depicted in Figure 1, our method effectively identifies packages such as "react-rrule-generator" experiencing a clear decrease in download rankings, thus classifying them as "inactive".

In our study, we devised a method for classifying npm packages into **"Deprecated"** or non-deprecated categories based on the deprecation mechanism employed by developers. To begin our classification process, we fetched the deprecation message from the NPM API for each package. When a package release is deprecated, the package.json file contains a deprecated field storing the associated deprecation message. Drawing inspiration from prior research [3], we observed that a few of all deprecated releases featured the "False" or empty string (as per the npm documentation, developers can remove deprecation by setting the message to an empty string) as their deprecation message. After conducting manual analysis, we concurred with the approach of classifying such releases as non-deprecated.

For the trivial category, we utilized two parameters, SLOC and McCabe's Cyclomatic Complexity. The threshold values were set as $\boldsymbol{SLOC \leq 35}$ ***and*** $\boldsymbol{McCabe's\ Cyclomatic\ Complexity \leq 10}$, as suggested in the paper [7]. These thresholds were determined through empirical analysis detailed in the paper. To gather the necessary data, we retrieved package names and their corresponding GitHub URLs. Using these URLs, we employed the npm] ***'cloc'*** [15] package to calculate SLOC, and the ***'lizard'*** [16] Python package from PyPI to determine the cyclomatic complexity for all npm packages obtained.

3.4.2 Analysis method for RQ1(b):

We adopted a bootstrapping strategy to mitigate the inherent bias introduced when sampling 70K packages randomly from 2.6 million packages. Starting with the 28k cleaned packages, we conducted ten iterations of bootstrapping, each time selecting 3,000 packages with replacements. This process resulted in ten subsamples, from which we calculated the percentage of packages falling into each category. We then computed the mean percentage for each category across these ten subsamples. By employing this approach, we aimed to enhance the reliability of our estimates and mitigate potential biases arising from the limited sample size.

3.4.3 Analysis method for RQ2(a):

Having obtained classification results from RQ1, we possessed labelled data. Therefore, in this RQ, we aimed to investigate the correlation metric to define the predictors contributing to each category. We began by preparing the dataset for analysis, ensuring that the data types of selected parameters were appropriate for statistical analysis. For instance, we transformed certain parameters, such as converting the license type from nominal to boolean and changing the last_modified parameter from date type to months count, to make them suitable for analysis. Additionally, we eliminated unnecessary parameters like dependencies_list to streamline our analysis process.

To analyze the relationship between package health categories and the selected parameters, we approached categorical and numerical variables differently:

Trivial Category	
Parameter	**Threshold**
SLOC	≤ 35
McCabe's Cyclomatic Complexity	≤ 10
Dead Category	
Parameter	**Threshold**
Months since latest version	≥ 31.2 months
Months since last commit	$\geq$ months
Deprecated Category	
Parameter	**Threshold**
Deprecation Message	Not 'False' and Non-empty
Inactive Category	
Parameter	**Threshold**
Monthly downloads for last 6 months	Slope < 0
For the formed regression line	slope should be significantly negative ($p < 0.1$)

Table 1: Parameters and their respective thresholds for categorizing packages into defined categories

Categorical variables (object/boolean type): we utilized the chi-square test. Since traditional correlation coefficients like Spearman's correlation cannot be directly applied to categorical data, we used contingency cross-tables derived from the chi-square test [8] to infer correlation values. If the p-value was less than 0.05, we considered the parameter to be statistically significant for the particular health category.

Numerical variables: we initially calculated the Variance Inflation Factor (VIF) to detect multicollinearity among the variables. If the VIF exceeded 5, indicating multicollinearity, we retained only one of the similar parameters for further analysis. While the t-test was initially considered, it required the data to be normally distributed. However, despite our efforts, we couldn't achieve normal distribution due to the random sampling process. Hence, we shifted to the Mann-Whitney U test [8], which does not require normality. We then used Spearman correlation to identify significant correlations, considering a p-value threshold of 0.1.

We repeatedly performed similar tests on all the bootstrapped subsets to ensure robustness and reliability. We identified common predictors for each package health category from these results and calculated the mean correlation values. This iterative process helped us extract stable and consistent predictors while considering the variability inherent in our dataset.

3.4.4 Analysis method for RQ2(b):

In addressing RQ 2(b), we focused on determining the strength of the correlations between various parameters and each package health category. Utilizing the correlation values obtained for significant parameters identified in the analysis of RQ 2(a), we categorized the parameters based on their correlation strength with package health categories. Parameters exhibiting an absolute correlation value greater than 0.5 were classified as 'Strong' predictors for the respective health category. Similarly, parameters with correlation values exceeding 0.3 but not meeting the threshold for 'Strong' predictors were categorized as 'Moderate,' while those below 0.3 were considered 'Weak' predictors. This classification approach enabled us to discern the degree of influence each parameter exerted on package health.

3.4.5 Analysis method for RQ3(a):

In this RQ, our objective was to explore the relationships between different package categories, focusing on whether the presence of one category increases the likelihood of a package being classified into another category. For instance, to examine whether the presence of a trivial package affects the probability of it being classified as dead, we would create a pair of categories: (Trivial, Dead) and sample the data accordingly. For example, if a package was classified as Trivial, we would sample from the subset where Trivial was labeled as 'Yes' and set the target variable as Dead (True/False). Subsequently, we apply the methods outlined in RQ2 to identify significant predictors and their corresponding correlation values. These values offered insights into whether specific properties, as determined by significant predictors, were linked to an increased likelihood of a package falling into both categories.

This iterative process can similarly be applied to examine the inter-category relationships between all the other category pairs. Notably, each pair, such as (Trivial, Dead), was distinct from its counterpart (Dead, Trivial), indicating unique relationships between package categories.

3.4.6 Analysis method for RQ3(b):

Leveraging the categorization results obtained from RQ1, we analyzed how these categories correlate with vulnerability metrics obtained from Synk.io [13], specifically focusing on health scores for each package.

Upon observing the Boxplot (Figure-2) of each package health category plotted with respect to "Health Score," we identified a threshold of 42. If the health score was less than 42, the package was categorized as unhealthy; otherwise, it was deemed healthy. This threshold was determined based on its ability to effectively separate the dead, trivial, and inactive categories into "yes" and "no" segments as observed in the Boxplots. Although the representation of deprecated packages in our dataset was limited, making it challenging to observe clear distinctions, the ranges of health scores for deprecated packages with deprecation status as true or false (10 to 60 and 10 to 94, respectively) appeared meaningful, thus supporting the selection of 42 as the threshold.

To examine how each package health category correlates with package vulnerability, we employed statistical methods such as correlation analysis, similar to our approach in RQ2. For instance, to determine the relationship between a 'trivial' package and package vulnerability, we sampled from the subset where Trivial was labeled as 'Yes' and set the target variable as Unhealthy (True/False). Subsequently, we identified the significant parameters that correlated with a trivial package being categorized as healthy or unhealthy. Similarly, this analysis can be repeated for all four categories, to discern patterns or associations between package health status and security vulnerabilities.

4. Results

4.1 RQ1

In accordance with the analysis method outlined for RQ1(a), we identified various parameters and their respective thresholds to categorize packages into each of the four defined categories. The specific parameters and thresholds are illustrated in Table 1.

Upon implementing these parameters and thresholds on the dataset, we classified **73.11%** of the total packages as dead, as depicted in Figure 4. Approximately **7.29%** of the sample were identified as inactive. Furthermore, about **3.06%** of the sample fell into the Deprecated category. Additionally, **15.96%** of the packages were classified as trivial. Notably, our findings align with those reported in previous studies [3] [7], serving as a valuable reference for our results concerning the deprecated and trivial categories, respectively. Moreover, we followed the approach outlined in [6] to classify packages as 'inactive'.

4.2 RQ2

In our analysis, we investigated the correlation between various parameters and each package health category, as outlined in sections 3.4.3 and 3.4.4. We meticulously prepared the dataset, identified significant predictors for each package's health category, and found strong predictors for each category.

For the trivial category, parameters such as **Lines of Code (LOC), Total Commits, and Source Lines of Code (SLOC)** emerged as strong predictors. In contrast, for the dead category, parameters like **Lines Deleted in the Past 1.5 years and Package Install Size** exhibited notable correlations. In the case of inactive packages, parameters such as **Popularity and Dependency Count** were identified as significant predictors. Additionally, for deprecated packages, attributes like **Number of Versions and Maintenance** emerged as influential factors.

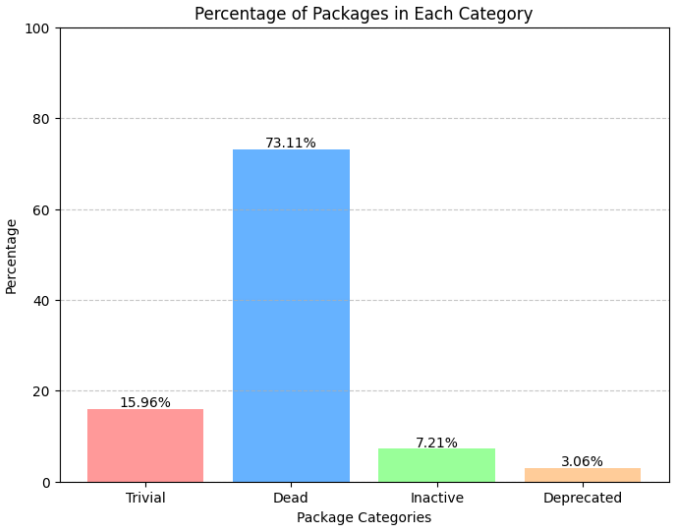

Figure 4: Percentage of Packages in Each Category

The correlation values and significance levels (Strong/Moderate/Weak) for each parameter and package health category are presented in the heatmap Figure 3. For instance, in the heatmap, a correlation value of -0.503 for Lines of Code (LOC) in the trivial category indicates a strong negative correlation, suggesting that higher LOC values are associated with a lower likelihood of a package being trivial. Similar interpretations can be made for all parameters across all four categories, providing valuable insights into the factors influencing package categorization within the npm ecosystem.

4.3 RQ3(a)

In exploring the relationships between different package categories, particularly investigating whether the presence of one category influences the likelihood of a package being classified into another category, we found compelling insights. The most significant pair observed was (Trivial, Dead) with a percentage of 76.83%, indicating that 76.83% of Trivial packages were also classified as Dead. Noteworthy predictors contributing to a Trivial package also being Dead included **package_install_size, #_of_files, and lines_deleted_one_and_half_year**. Additionally, the pair (Deprecated, Dead) exhibited a significant correlation of 69.53%, with notable predictors such as **Community, total_commits,** and **lines_deleted_one_and_half_year**. Further details on significant predictors and their correlation values for various pairs are provided in Table 2. For comprehensive insights into all significant predictors across remaining pairs, please refer to the Appendix.

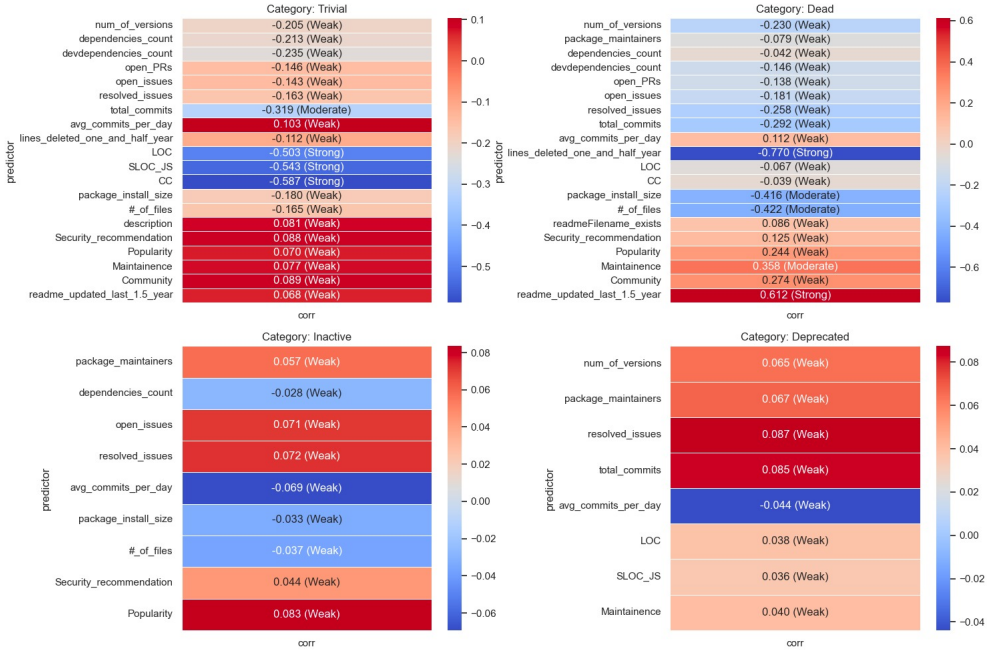

Figure 3: Correlation heatmap illustrating predictors' strength for package health categories

4.4 RQ3(b)

The results of our analysis for RQ3(b) revealed significant relationships between package categories and package vulnerability metrics.Based on the threshold defined on Health Scores, we identified significant predictors for categorizing packages into healthy or unhealthy (vulnerable) states across different health categories:

1. For the trivial category, where trivial was set as 'Yes' and the target variable was classified as unhealthy (True/False), strong parameters were found to be **Security Recommendation** and **Dependencies count**.

2. Similarly, in the dead category, significant predictors included **Security Recommendation** and **Dependencies count**.

3. In the inactive category, some influential factors were **last_3_month_downloads, Security Recommendation**, and **Dependencies count**.

4. For the deprecated category, significant predictors included **Last_commit_before_months, Community, Security Recommendation**, and **Dependencies count**.

The heatmap in Figure 5 presents all significant parameters along with their correlation values and significance levels (Strong/Moderate/Weak) for categorizing each package health category into healthy or unhealthy (vulnerable). In the heatmap, for instance, for the inactive category, last_3_month_downloads is shown as -0.240. The negative sign indicates an inverse correlation, implying that higher values of last_3_month_downloads are associated with a lower likelihood of an inactive package being categorized as unhealthy. Similar interpretations can be made for all parameters across all four categories, providing comprehensive insights into the factors influencing package vulnerability within the npm ecosystem.

5. Discussion

Our project presents some important findings that could help both the developers and users of the NPM ecosystem. Based on the classification that has been carried out, as seen in Figure 4, **73.11%** of the packages were categorized as dead. Majority of the packages which exist in the NPM ecosystem are not recommended to be used. This can be associated with most packages not being maintained properly or better packages being developed for the same functionality. Trivial packages make up **15.96%** of the total packages. Although these make up a significant portion of the packages, trivial packages are not as detrimental to the ecosystem as dead packages. Trivial packages are primarily made to handle any single functionality and could still be maintained properly. Some parts of the trivial packages are also dead. This could be due to the reasons discussed earlier. Although numerous packages are dead, very few have been categorized as deprecated. Only **3.06%** of the packages are marked as deprecated by the developers. This also shows that most packages which are developed are not maintained properly. Inactive packages making up **7.21%** shows that a notable amount of packages are in decline with regard to being active among users.

Other than just identifying the percentage of packages in each category, it is also necessary to check if other predictors could make the classification more robust. Based on the methodology implemented, a number of predictors were found for each package category, as seen in Figure 3. One thing to be noted is that a negative value shows that there is a negative correlation. For example, LOC for trivial has a negative correla-

('Trivial', 'Dead') $---> 76.83\%$		('Deprecated', 'Dead') $---> 69.53\%$	
Predictors	**Correlation**	**Predictors**	**Correlation**
num_of_versions	-0.1547166236	num_of_versions	-0.2309308076
total_commits	-0.1085275624	devdependencies_count	-0.1896936336
avg_commits_per_day	0.1090453408	open_PRs	-0.2791764736
lines_deleted_one_and_half_year	-0.6488728006	open_issues	-0.2570519251
CC	0.09725685798	resolved_issues	-0.2940450189
package_install_size	-0.4052610226	total_commits	-0.3390617505
#_of_files	-0.4141214384	lines_deleted_one_and_half_year	-0.8278261839
readmeFilename_exists	0.07920577805	package_install_size	-0.4731305488
Security_recommendation	0.1267695887	#_of_files	-0.4467435802
Popularity	0.1298140411	Popularity	0.3688438841
Maintainence	0.2387291271	Community	0.3269877773
readme_updated_last_1.5_year	0.5427016377	readme_updated_last_1.5_year	0.7442885888

Table 2: Predictors and their correlation values for two of the most significant category pairs

tion, meaning that as the value of LOC increases, it is less probable to be a trivial package. Some predictors, such as the negative correlation of total commits for Trivial category can be linked to trivial packages usually having lesser number of commits since they are made for a single functionality and might not be updated as often as other larger packages. Although not having a high correlation as total commits, other predictors such as dependencies count, package install size, number of versions etc. also have some correlation. This can also be associated with trivial packages being single purpose and smaller than other packages. But a package being smaller, not to be confused with LOC, doesn't have to necessarily make it a trivial package.

Dead category also has some strong predictors, which is a good point since dead and deprecated packages need to be identified before it is being put into use. One strong predictor is lines deleted in the past year and half having a negative correlation. This is an interesting finding, since having more deleted lines in the past year and half means that a package is being actively maintained and is being optimized well. Another strong predictor along with a moderate predictor would be README updated and maintenance. These are directly related to the meaning of a dead package. A package which is not maintained properly for a long time is considered dead, and this is what these predictors take into account. The predictor community can also be associated with this. Other predictors such as number of files and package install size is also an interesting finding. This could be understood as packages being more likely to be dead if they are smaller, since larger packages might more often be handled by larger groups and hence more likely to be maintained for a longer period of time.

Although not strong predictors, some were identified for both inactive and deprecated. Metrics such as popularity, open issues and resolved issues could be easily understood since a package is considered inactive if it lacks active participation from the community. A package having less number of commits could be associated with it not being active. Deprecated categories not having strong predictors could be understood since it depends on the developer to inform if a package is deprecated or not. But some weak metric associated with the package being maintained regularly could be seen as weak predictors. These predictors are average commits, total commits, issues resolved etc. A deprecated package is officially not supported by the developer, hence making these metrics relevant.

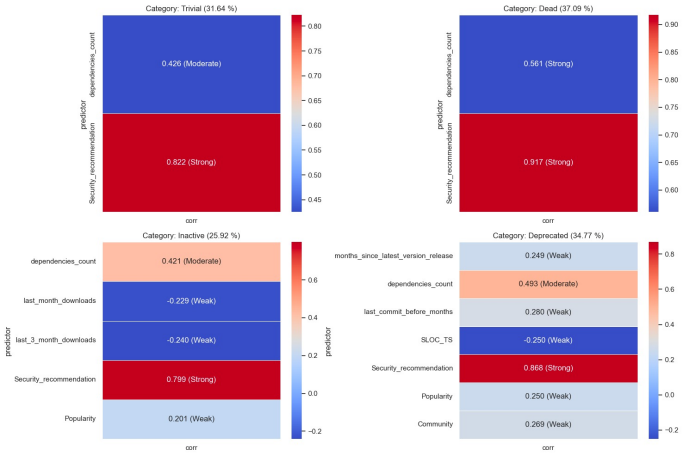

Figure 5: Correlation Values and Significance Levels of Predictors for Package Vulnerability Across Health Categories

Classifying packages based on metrics and activity could greatly improve selection of packages from the ecosystem. Identifying chances of a package being in another category based on predictors could aid this process even more, assuring developers and users of using or investing in time in a package. Doing this produced some strong correlations for some of the categories. One such pair is 'trivial' being 'dead', with a very high correlation of 76.83%. Trivial packages are smaller packages, usually made by individuals and smaller teams. Therefore, it is understandable why trivial packages might not be maintained as long as

others. The predictors are also very similar to the ones identified for dead packages, showing that a trivial package not being maintained for some time has a high chance of being dead.

Another strong pair is 'deprecated' being 'dead', since dead packages are packages which are not maintained properly and deprecated packages are officially unsupported. The predictors are also very similar to dead category's predictors. 'Inactive' packages being 'dead' due to certain predictors is also high. This shows that packages which are not popular among the community and having a decline in activity are more often not to be supported in the longer run. One interesting pair is 'Inactive' being 'Trivial', which shows that it is possible that packages which are declining in user activity could be trivial. This can be associated with trivial packages not being used as much once the necessity is satisfied, or if a better package for the same use case is found. Other pairs have relatively very low correlation. 'Dead' being in other categories is justified, since a package being dead doesn't necessarily mean that it needs to be a trivial or an inactive package. A package which is still actively used could still be dead since it is not maintained by the developers properly. These are the cases which needs to be avoided, since packages not being maintained properly might have security issues.

Package vulnerability can have a major part in some of the package health categories. One common aspect that can be observed is that packages not categorized into any of the categories have a higher upper bound for vulnerability score. Once an important category in which vulnerability plays a major role is dead packages, since they are packages which are not recommended to be used and have not been maintained for a considerable period. A clear difference can be seen for dead categories in this aspect, making it evident that packages which are categorized as dead have a lower vulnerability score, i.e. chances of being more vulnerable. Packages which are not categorized as dead also have a higher upper bound compared to dead packages. This drastic difference cannot be observed for inactive packages, assuring that a package becoming inactive or being used lesser might not be due to the vulnerability of that package. A package becoming inactive might also be due to other factors, including vulnerability. There are very few packages categorized as deprecated, which makes a very small sample size. But a difference in the upper bound for vulnerability score can be seen even if there is not a huge difference in the mean value of the vulnerability score. Trivial packages are developed to satisfy any functionality and hence does not have to be more vulnerable than packages which are not trivial. However, a difference in the upper bound can be observed even for trivial packages. This can be due to the fact that trivial packages are usually made by individual developers and might not be maintained as often as normal packages.

6. Threats to Validity

Construct Validity: The validity of our findings may be affected by several threats to construct validity inherent in our methodology for categorizing npm packages into distinct classes particularly the measures used to classify packages into categories like dead, inactive, trivial, and deprecated may not capture the intended constructs accurately.

External Validity: Possible issues in our study's external validity include: the sampled npm packages might not fully represent the ecosystem due to random sampling limitations, findings may not apply to future times due to changing package characteristics, and our study's applicability to other ecosystems may be limited due to platform dependency and biased data sources.

Internal Validity: In our study, threats to internal validity arise from sampling bias, where certain package types may be over or underrepresented despite repeated random sampling. Additionally, confounding variables that aren't sufficiently controlled pose another threat. Furthermore, measurement errors in categorizing npm packages into different groups could introduce bias, impacting the validity of our findings.

7. Conclusion

Our classification models and predictors identified have resulted in some interesting findings which could greatly aid the NPM ecosystem. These predictors along with the classification model can categorize any package provided to it, making it a robust method to check for a package's category before it is being used. This method can be incorporated into the NPM search engine to make it easier for users and developers to avoid packages which might not be safe to use. There have not been any studies on the NPM ecosystem which have worked on the correlation between categories and identifying the significant predictors for the same. This could significantly contribute to the stability of the ecosystem. However, it might not be feasible to put it into use immediately due to the nature of the dataset. There are very few samples for packages which are deprecated, which makes it difficult to identify significant predictors for the same. The metrics have all been calculated for a smaller subset of the 70,000 random packages in the NPM ecosystem. Therefore, various techniques have been incorporated to ensure accurate results. However, it is still possible that the actual metrics might be different from the results obtained, although the difference being minimal due to the measures taken against this.

To aid the NPM ecosystem, it would greatly help the study if the **"Future Work"** contains some kind of qualitative aspects to the study to understand the perspective of the developers and users regarding the packages which are classified into each category. This could also be complemented with using sentiment analysis. Rather than just categorizing packages as dead and avoiding them, it would help the NPM landscape if some kind of reusability factor for packages is also calculated. Reusing a package would be a better than introducing newer packages which solve the same problem. This could also be improved by introducing package subcategories to highlight the differ-

ence in the levels of activeness, if a dead package is still safe to use etc. To help the overall NPM package selection, it is also essential to perform some kind of dependency analysis. Packages which are using dead or deprecated packages as dependencies also have a risk of being harmful when put to use. This is a vast domain which could be studied in numerous ways.

Team Membership and Attestation of Work

Taher, Nikita, Aakash, Hemachandran, and Savali have significantly contributed to the project's progress.

References

[1] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. What are the characteristics of highly-selected packages? a case study on the npm ecosystem. *Journal of Systems and Software*, 199:111272, 2022.

[2] Xiaowei Chen, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Xin Xia. Helping or not helping? why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering*, 25(3):1494–1528, 2020.

[3] Filipe R. Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. Deprecation of packages and releases in software ecosystems: A case study on npm. *ACM Transactions on Software Engineering and Methodology*, 70(8):1–43, 2021.

[4] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. On the impact of using trivial packages: an empirical case study on npm and pypi. *Empirical Software Engineering*, 23(6):3533–3572, 2018.

[5] J. Khondhu, A. Capiluppi, and K. J. Stol. Is it all lost? a study of inactive open source projects. In E. Petrinja, G. Succi, N. El Ioini, and A. Sillitti, editors, *Open Source Software: Quality Verification*, volume 404 of *IFIP Advances in Information and Communication Technology*. Springer, Berlin, Heidelberg, 2013.

[6] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. Toward using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Management*, 69(6):3618–3632, December 2022.

[7] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad 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*, page 385–395. Association for Computing Machinery, 2017.

[8] S Arunkumar, T Murali, and N Achuthan. Software reliability prediction using neural networks. In *Proceedings of the 1998 International Symposium on Software Engineering (ISSE '98)*, pages 626–631, December 1998.

[9] Ning Deng, Jeroan J Allison, Haijun J Fang, Arlene S Ash, and John E Ware Jr. Using the bootstrap to establish statistical significance for relative validity comparisons among patient-reported outcome measures. *Health Qual Life Outcomes*, 11:89, May 2013.

[10] npmjs all packages. https://www.npmjs.com/package/all-the-package-names/.

[11] npmjs registry. https://www.npmjs.com/.

[12] Pydriller. https://pydriller.readthedocs.io/en/latest/index.html#.

[13] synk.io advisor. https://snyk.io/advisor/.

[14] Kaushal D. Buch. Decision based non-linear filtering using interquartile range estimator for gaussian signals. In *2014 Annual IEEE India Conference (INDICON)*, pages 1–5, 2014.

[15] cloc. https://www.npmjs.com/package/cloc.

[16] Lizard. https://github.com/terryyin/lizard.

[17] LLM's : ChatGPT & Gemini

Appendix

Link to Github Repository: https://github.com/travaditaher/SE_NPM_packages

Each category might hold a distinct meaning based on the diverse end goals one aims to achieve, and interpretations may vary subjectively. In the context of the aforementioned objectives, we define each category as follows:

- **Trivial:** Characterized by a small codebase, limited functionality, concise structure, basic utility, and designed for a singular, uncomplicated purpose.
- **Dead:** A package is considered dead if it lacks updates, commits, or maintenance for a significant period, indicating an abandoned state.
- **Inactive:** Exhibits reduced engagement or usage, possibly due to diminished relevance or ongoing development.
- **Deprecated:** Officially discouraged for use, a package is labelled deprecated if it includes explicit notices, warnings, or announcements from maintainers, guiding users to migrate to newer alternatives.

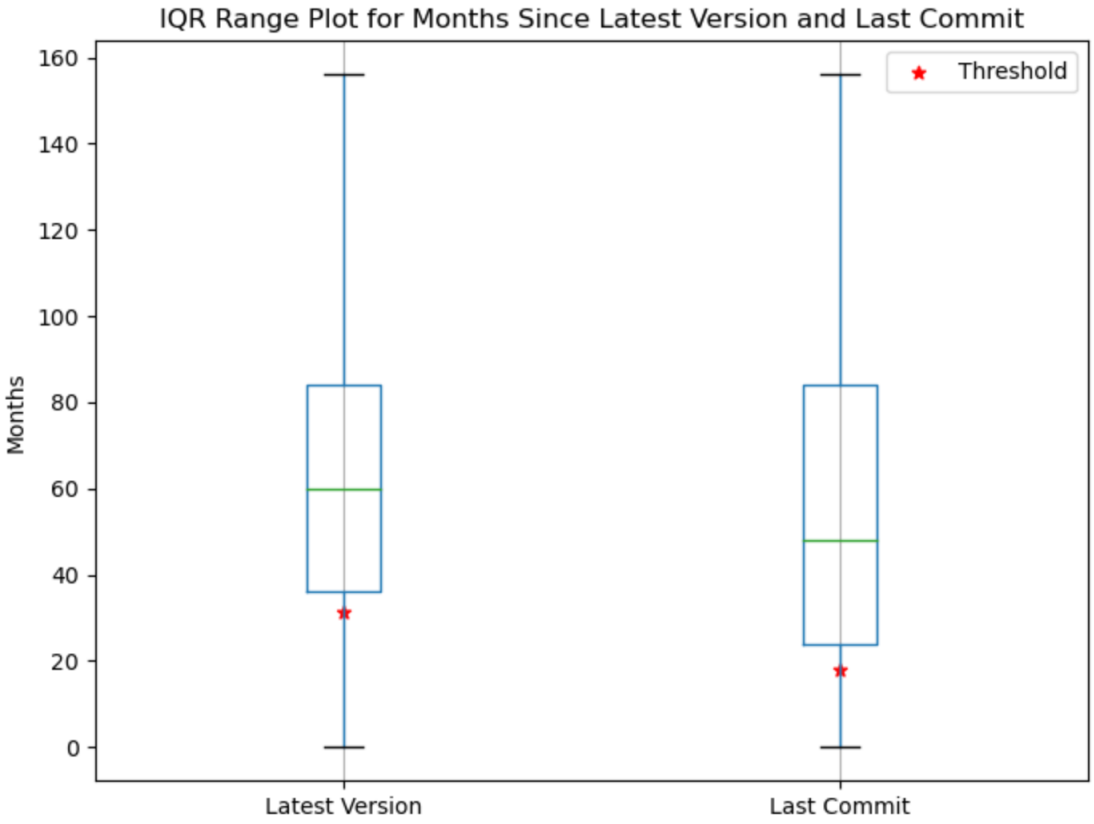

Figure 6: Shows the distribution of data among the two columns using boxplot

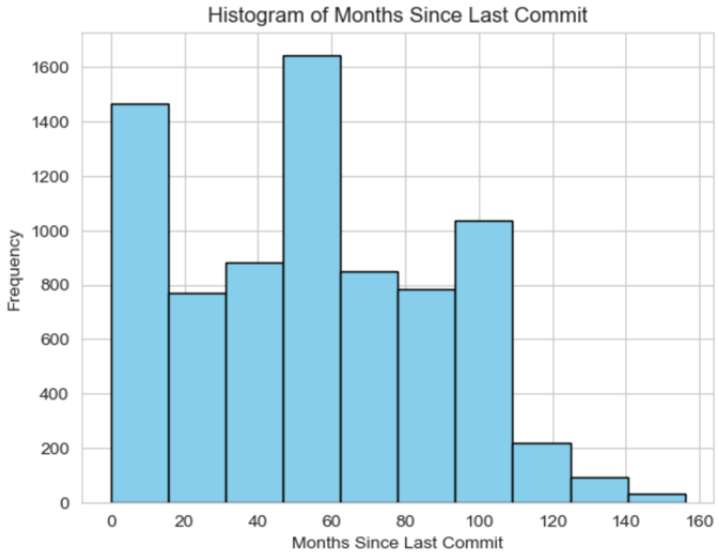

Figure 7: Histogram of Months Since Last Commit

Predictors	**Correlation**
('Deprecated', 'Dead') → 69.53%	
num_of_versions	-0.230930808
devdependencies_count	-0.189693634
open_PRs	-0.279176474
open_issues	-0.257051925
resolved_issues	-0.294045019
total_commits	-0.339061751
lines_deleted_one_and_half_year	-0.827826184
package_install_size	-0.473130549
#_of_files	-0.44674358
Popularity	0.368843884
Community	0.326987777
readme_updated_last_1.5_year	0.744288589
('Deprecated', 'Trivial') → 10.55%	
devdependencies_count	-0.261745098
resolved_issues	-0.212677073
total_commits	-0.258691799
LOC	-0.396281368
SLOC_JS	-0.432436559
CC	-0.477067536
package_install_size	-0.161743316
('Deprecated', 'Inactive') → 7.42%	
package_maintainers	0.238732461
Popularity	0.284243167

Predictors and their correlation values for 'Deprecated'

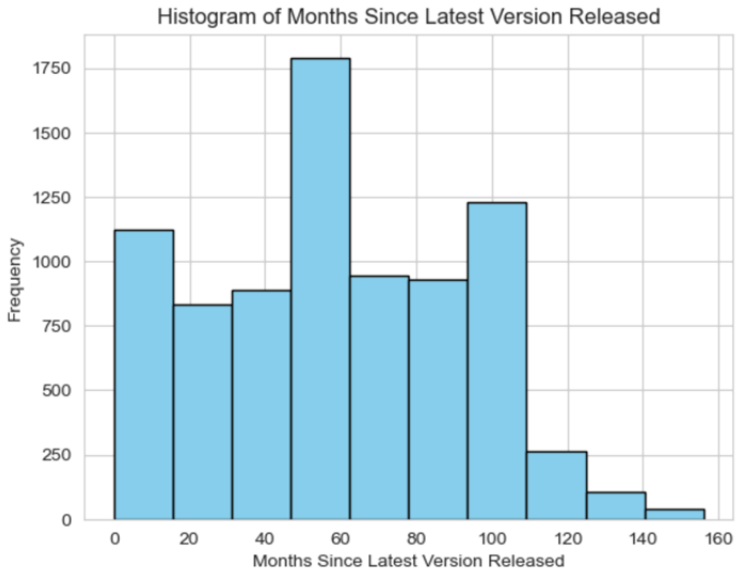

Figure 8: Histogram of Months Since Latest Version Released

Predictors	**Correlation**
('Dead', 'Trivial') → 17.56%	
num_of_versions	-0.21153327909401426
dependencies_count	-0.209432238110000
devdependencies_count	-0.22557142299090366
open_PRs	-0.11995984476139258
open_issues	-0.1097560393211054
resolved_issues	-0.1205491652205136
total_commits	-0.30022441807983935
avg_commits_per_day	0.10144011113616853
LOC	-0.5571713980926116
SLOC_JS	-0.6035282292322821
CC	-0.6096223416902598
package_install_size	-0.1295834849735405
#_of_files	-0.117639120721181736
description	0.05847153826512036
Security_recommendation	0.10834628940588462
('Dead', 'Inactive') → 7.08%	
package_maintainers	0.04561229401473418
open_issues	0.08447143547158456
resolved_issues	0.07157510042660177
avg_commits_per_day	-0.05203386171885279
Popularity	0.09269317520939119
('Dead', 'Deprecated') → 3.08%	
num_of_versions	0.06923872978759336
package_maintainers	0.07218016053635223
resolved_issues	0.0887519373222643
total_commits	0.0887
LOC	0.03869380582329421

Predictors and their correlation values for the 'Dead' category.

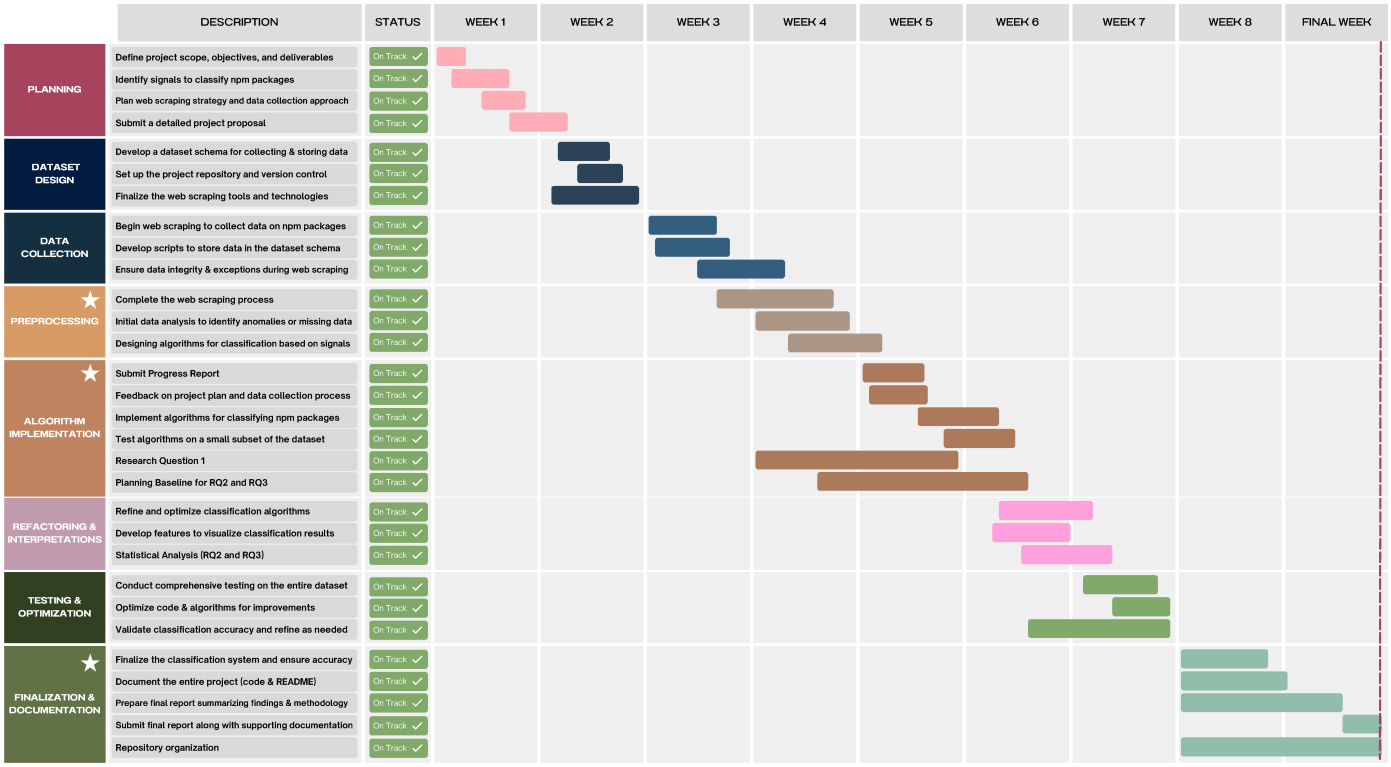

Figure 9: Gantt Chart - Shows the Updated Timeline & Milestones

Predictors	**Correlation**
('Trivial', 'Dead') → 76.83%	
num_of_versions	-0.1547166236
total_commits	-0.1085275624
avg_commits_per_day	0.1090453408
lines_deleted_one_and_half_year	-0.6488728006
CC	0.09725685798
package_install_size	-0.4052610226
#_of_files	-0.4141214384
readmeFilename_exists	0.07920577805
Security_recommendation	0.1267695887
Popularity	0.1298140411
Maintainence	0.2387291271
readme_updated_last_1.5_year	0.5427016377
('Trivial', 'Inactive') → 6.21%	
avg_commits_per_day	-0.065551442
Popularity	0.036677933
('Trivial', 'Deprecated') → 2.04%	
num_of_versions	0.041519671
avg_commits_per_day	-0.068122141

Predictors and their correlation values for 'Trivial'

Predictors	**Correlation**
('Inactive', 'Dead') → 68.56%	
num_of_versions	-0.338327106
devdependencies_count	-0.133275902
resolved_issues	-0.209068396
total_commits	-0.365114132
avg_commits_per_day	0.123741009
lines_deleted_one_and_half_year	-0.802497706
package_install_size	-0.418878795
#_of_files	-0.409774921
Popularity	0.267880164
Maintainence	0.371608246
Community	0.22938423
readme_updated_last_1.5_year	0.590611095
('Inactive', 'Trivial') → 13.71%	
num_of_versions	-0.203737844
package_maintainers	-0.115935593
dependencies_count	-0.162266429
devdependencies_count	-0.222536564
open_PRs	-0.185207263
open_issues	-0.142268535
resolved_issues	-0.200286862
total_commits	-0.281759611
LOC	-0.495184104
SLOC_JS	-0.510059586
CC	-0.55170612
package_install_size	-0.184128097
#_of_files	-0.168063025
('Inactive', 'Deprecated') → 3.18%	
num_of_versions	0.127736511
package_maintainers	0.245529075
lines_deleted_one_and_half_year	0.151576336

Predictors and their correlation values for 'Inactive'

Dataset Schema with Description

Parameter	**Description**
Package Name	String - Name of the package
Description	String - Description provided for the package by the developer in npm
Package Creation	Date - Package Creation date
Package last Modified	Date - Package latest Modification date
Package Age	Float - Number of months since the package Creation
Latest Version	String - Latest/Ongoing version release
Latest Version Released since	Float - Number of months since the last version released according to synk.io
Release Frequency	Float - Version Release frequency in months and years
Total Versions Released	Integer - Number of version released till date
Maintainers	Integer - Number of maintainers
Direct Dependencies	List of Strings - List of dependent packages
Direct Dependencies_count	Integer Count of dependent packages
Development Dependencies	List of Strings List of development dependent packages
Development Dependencies_count	Integer Count of development dependent packages
Readme_existst Documentation	Bool README file existence
Download Trends	Integer Number of downloads recorded from Jan 23 to 22nd Feb 24 (last 14 months)
License	String - Licensing of the project/package
Health Score	Integer - Package health score according to synk.io
Vulnerability count	Integer - Number of vulnerabilities identified as per synk.io
Security Recommendation	String - Security review by synk.io
Popularity	String - Popularity among users
Maintenance	String - Package Maintenance (by developers) metric
Community	String - Package Community activeness measure
GitHub Repo	String - Link to the GitHub Repository
GitHub Stars	Integer - Number of stars in GitHub
GitHub Forks	Integer - Number of forks in GitHub
GitHub Contributors	Integer - Number of unique contributors in GitHub
PR Statistics	Integer - Number of GitHub Open and Closed PR's
Issues Statistics	Integer - Number of GitHub Open and Closed Issues
Total Commits	Int - Total number of Commits on GitHub till date
Months since Last Commit	Float - Number of months since the last commit
Commit Frequency	Float - Github commit frequency in days
Added Lines	Integer - Addition of Lines for all commits and for commits in last 18 months
Deleted Lines	Integer - Deletion of Lines for all commits and for commits in last 18 months
LOC	Integer - Lines of code in GitHub repo
SLOC	Integer - Source lines of code on GitHub without binary files and index/git/readme files
Cyclomatic Complexity	Integer - Cyclomatic Complexity of the code
Package Install Size	String - Installation size of the package
Num of Files	Integer - Number of files associated with Package
TS Typings	Bool - Package follows Typescript Coding
Deprecation Status	Bool - Package marked as deprecated by developer