Hindawi
IET Software
Volume 2024, Article ID 4488412, 23 pages
https://doi.org/10.1049/2024/4488412

Research Article

An Empirical Study on Downstream Dependency Package Groups in Software Packaging Ecosystems

Qing Qi and Jian Cao

Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 201100, China

Correspondence should be addressed to Jian Cao; cao-jian@sjtu.edu.cn

Received 29 May 2023; Revised 4 March 2024; Accepted 30 March 2024; Published 30 April 2024

Academic Editor: Shariq Hussain

The role of focal packages in packaging ecosystems is crucial for the development of the entire ecosystem, as they are the packages on which other packages depend. However, the evolution of dependency groups in packaging ecosystems has not been systematically investigated. In this study, we examine the downstream dependency package groups (DDGs) in three typical packaging ecosystems—Cargo for Rust, Comprehensive Perl Archive Network for Perl, and RubyGems for Ruby—to identify their features and evolution. We also identify and analyze a special type of DDG, the collaborative downstream dependency package group (CDDG), which requires shared contributors. Our findings show that the overall development of DDGs, particularly CDDGs, is consistent with the status of the whole ecosystem, and the size of DDGs and CDDGs follows a power law distribution. Furthermore, the interaction mechanisms between focal packages and downstream packages differ between ecosystems, but focal packages always play a leading role in the development of DDGs and CDDGs. Finally, we investigate predictive models for the development of CDDGs in the next stage based on their features, and our results show that random forest and Gradient Boosting Regression Tree achieve acceptable prediction accuracy. We provide the raw data and scripts used for our analysis at https://github.com/onion616/DDG.

1. Introduction

As software systems become increasingly complex, they often rely on multiple other software systems and evolve in tandem with them. Open collaboration platforms like GitHub have facilitated the development of many open-source software (OSS) projects that require internal dependencies [1]. These projects are now frequently managed by communities comprised of diverse contributors. This phenomenon has led to the emergence of the concept of the OSS ecosystem, which describes how a set of actors interact on a shared technological platform to create various software solutions or services [2]. Given its growing impact, the health of the OSS ecosystem has become an active research topic [3].

Software packages distributed by package managers form large software ecosystems. Packaging ecosystems contain a large number of package releases that are updated regularly, and there are many dependencies between package releases. Researchers have studied the issues relating to dependencies in various packaging ecosystems. For example, Wittern et al. [4] studied the dependency evolution of JavaScript packages in npm, and the dependency network of the R ecosystem has also been studied previously [5]. Mora-Cantallops et al. [6] recently used complex network analysis tools to assess the CRAN software package ecosystem. Valiev et al. [7] studied the likelihood of project development entering a period of dormancy as a function of the project's position in their dependency networks, organizational support, and other factors in the PyPI ecosystem. Decan et al. [8] evaluated the dependency networks in seven packaging systems, namely Cargo for Rust, Comprehensive Perl Archive Network (CPAN) for Perl, CRAN for R, npm for JavaScript, NuGet for the .NET development platform, Packagist for PHP, and RubyGems for Ruby. These studies reveal the overall trends of the dependencies among packages or the whole dependency network in different packaging systems.

When many packages are dependent on one particular package, the latter is called the focal package, and all the dependency packages are called downstream packages [7]. It has been proven that less than 30% of packages are

required by other packages, and less than 17% of all focal packages concentrate more than 80% of all dependencies [8]. This shows that focal packages and their downstream dependency packages play a vital role in the packaging ecosystem. Therefore, in this paper, we investigate the downstream dependency package group (DDG), which consists of the focal package together with its downstream packages, to understand the interaction of dependencies between packages at a micro level (e.g., the behavior of the focal packages will lead to the feedback of its downstream dependency packages) and more importantly, understand its impact on the entire packaging ecosystem at a macro level. Specifically, to verify the influence of developers on the dependency package group, a special DDG, the collaborative downstream dependency package group (CDDG), which only includes the focal package and its downstream packages with common contributors, is also studied and compared with its corresponding DDGs. Section 3 provides further details on the construction of DDGs and CDDGs.

So far, it seems difficult to evaluate the development status of a packaging ecosystem. To address this issue, we propose using DDGs to model package interdependencies and CDDGs to capture collaborative contributions. In this way, the major members of the packaging ecosystem, including packages and contributors, are effectively connected through their interactions (including dependencies among packages and cooperation among contributors). This approach is crucial for gaining a deeper understanding of packaging ecosystems. Additionally, by studying the evolution of DDGs and CDDGs across different packaging ecosystems, we can identify the reasons behind their varying development statuses. Such insights will help guide regulation efforts and maintain the stable growth of packaging ecosystems.

To comprehensively study all packaging ecosystems poses a challenge. Therefore, we have identified several ecosystems with distinct developmental trends to conduct comparative analyses. Through this approach, we aim to establish the impact of DDGs and CDDGs on their development. Following a rigorous screening process, we selected three widely used packaging ecosystems with comparable total package numbers but exhibiting contrasting developmental patterns. These include Cargo for Rust, CPAN for Perl, and RubyGems for Ruby.

To understand the formation and evolution of DDGs, it is important to understand the sustainability of focal packages and their downstream packages. In addition, it also benefits our understanding of the evolution of the whole packaging ecosystem. Unfortunately, most research only investigates the relationships between the total number of dependencies and the status of the packaging system.

Different from previous work, in our study, by treating DDGs as subecosystems in which all components have complex internal interactions, we answer the following three research questions:

(1) RQ1: What kind of roles do DDGs and CDDGs play in the packaging ecosystem? How do the features of DDGs evolve with the development of the ecosystem over time?
(2) RQ2: How does the focal package influence the development of the DDG and CDDG? Are the features of downstream dependency packages relevant to the development of the focal package?
(3) RQ3: Can we predict the development of DDGs and CDDGs?

By answering RQ1, we aim to uncover the positions and influence of DDGs within evolving packaging ecosystems. Answering RQ2 will help us identify the factors that impact the development of both DDGs and CDDGs. Through answering RQ3, we intend not only to provide prediction methods for the development of individual CDDGs but also a way to predict the evolution of the entire ecosystem.

The rest of the paper is organized as follows: Section 2 introduces the related work and compares our research with the existing research. Section 3 explains the definition, selection of packaging ecosystems, extraction of features, and prediction methods. Section 4 presents the research process, results, and summaries for three RQs. A discussion is presented in Section 5, and some threats to validity are detailed in Section 6. Finally, we conclude the paper in Section 7.

2. Related Work

The term ecosystem was first proposed by British ecologist Tansley [9] in 1935. An ecosystem refers to any area in nature where living organisms and nonliving environmental components interact to establish a relatively stable, dynamic equilibrium over a defined period. As a flexible, open system with variable boundaries, an ecosystem represents a fundamental structural and functional unit in ecology.

Similarly, a software ecosystem can also be regarded as a major structural and functional unit in the field of software engineering [2, 10], in which a group of participants interacts with each other to form a large number of software solutions or services on a common technological platform [11]. For a software ecosystem, the environment can be a software company, a research group, or a virtual open-source community [12]. It has been recognized that the software ecosystem is now an effective way to build large software systems on software platforms by combining components developed by internal and external participants [10]. Software ecosystems are active research topics, as shown by the increase in the number of publications in this area since 2007 [13]. Software ecosystems incorporate a wide range of research areas, including software engineering, social networking, and technology management [2].

2.1. Dependency in Software Ecosystems. With the rapid growth in OSS development platforms, the number of available OSS projects has increased dramatically [14]. A key feature of a software ecosystem is that software projects or components have internal dependencies and teamwork [15]. Therefore, some researchers have studied dependency types and dependency identification approaches [16]. Generally, dependencies can be divided into technical and social dependencies [17]. In most work, technical dependency information can be obtained from a project's configuration files.

More accurate technical dependency information can be identified from a project's source code [18], but they require large amounts of memory and computation time [19]. A new approach is proposed in [20], which identifies technical dependency through reference coupling. However, this approach can only be applied to GitHub. Social dependencies are often based on the degree of participation and contribution of the same members between different projects [3]. On the contrary, we want to carefully study the internal development of the project from the micro level of dependency packages in packaging ecosystems.

2.2. Dependency in Packaging Ecosystems. Package managers play a crucial role in managing and distributing software packages, creating distinct packaging ecosystems for different programing languages. The focus of research on dependencies in these ecosystems is primarily on technical dependencies, as social dependency information is often unavailable. Technical dependencies can be extracted from configuration files, allowing for their study in most cases. The proliferation of OSS development has led to a large number of available software packages, which greatly reduces the development cost and time [21].

However, some confusion caused by the need for two or more incompatible package versions in one software artifact is often referred to as "dependency hell" [22]. Many works have tried to escape from "dependency hell." Therefore, researchers have been actively studying the dynamic evolution of the packaging dependency network to solve this problem [23, 24]. Fan et al. [25] proposed a unified dependency graph to find dependency errors. Tanabe et al. [26] considered context-oriented programing as a solution. In previous work [27], it is found that 41% of the errors in CRAN packages are caused by backward incompatible changes in one of its dependencies. In previous work [28], the npm software ecosystem topology is generated to uncover insights and extract patterns of existing libraries by studying its localities.

Packages that are more likely to be used within an ecosystem are located separately from packages meant for application usage outside the ecosystem. Studies on open-source package-based software ecosystems such as Debian and R [29] have highlighted common dependency issues. Moreover, research conducted in [30] revealed that changes in dependencies affect only a small percentage (around 5%) of the source code of client projects. Nevertheless, frameworks or libraries with broadly applicable services may impact client project source code significantly when their dependencies are upgraded.

While examining a single packaging ecosystem may yield some fundamental principles, previous research [31] has indicated that when exploring the dependency network structure and evolution of JavaScript, Ruby, and Rust ecosystems, significant differences emerge across language ecosystems.

To take social dependencies into consideration, social information can be obtained from GitHub if the projects of these packages are hosted in GitHub. For example, in previous research [32], the relationships between the developer coordination activities and the project dependency structure in the Ruby ecosystem have been studied by collecting data from https://RubyGems.org and GitHub. The study shows that the collaboration pattern among developers in the Ruby ecosystem is not necessarily shaped by the communication needs indicated by the dependencies among its ecosystem projects.

It can be seen that although the dependencies of packages have been explored in recent years, most studies are carried out from a global view. However, it is also necessary to pay attention to the local context of a package to understand the factors behind the development trends of packages or the evolution of ecosystems, which has attracted the attention of academia. For example, in previous work [7], the relationships between the dormancy risk of a package and the upstream dependencies or downstream dependency packages are studied. Our paper focuses on the downstream dependency packages. In particular, we treat all downstream dependencies of a package as a group and explore the general characteristics of the group and the interrelationships of group members.

In a natural ecosystem, a subecosystem is often used to divide a large ecosystem into several smaller parts that have a common dividing property. However, this has only been mentioned very briefly in software ecosystem research [33]. The DDGs studied in this paper are a type of subecosystem.

3. Methodology

In this section, we provide definitions for DDGs and CDDGs and proceed to analyze these structures in three representative packaging ecosystems. Our focus is on understanding the roles of DDGs and CDDGs in packaging ecosystems and identifying key features that characterize their development. To address RQ1, we investigate the correlations between focal packages and downstream dependency packages and manually select features to describe their development. We also measure the impact of packages by examining these correlations to answer RQ2. Additionally, we employ 16 features to predict the size of CDDGs to answer RQ3.

3.1. Definition of DDGs and CDDGs. Before introducing DDGs and CDDGs, we start with an introduction to the concepts of upstream and downstream dependencies. Here, we only consider direct dependencies.

Figure 1 is an example. We can observe that packages $\{C_1, C_2, \ldots, C_n\}$ depend on package B, which in turn depends on packages $\{A_1, A_2, \ldots, A_m\}$. Here, package B is regarded as the focal package. Packages $\{A_1, A_2, \ldots, A_m\}$ are the upstream dependencies of B, while packages $\{C_1, C_2, \ldots, C_n\}$ are the downstream dependencies.

In this paper, we mainly focus on downstream dependencies. As previously mentioned, we identify the DDGs, each of which is composed of a focal package and its downstream dependency packages. Moreover, to verify the impact of developers on packaging ecosystems, we further define the CDDG, whose dependencies and focal package should have common contributors. Here, we take Figure 2 as an example. Package B has a list of contributors $\{a, b, c, \ldots\}$, which has three downstream dependency packages C_1, C_2, and C_3, all of

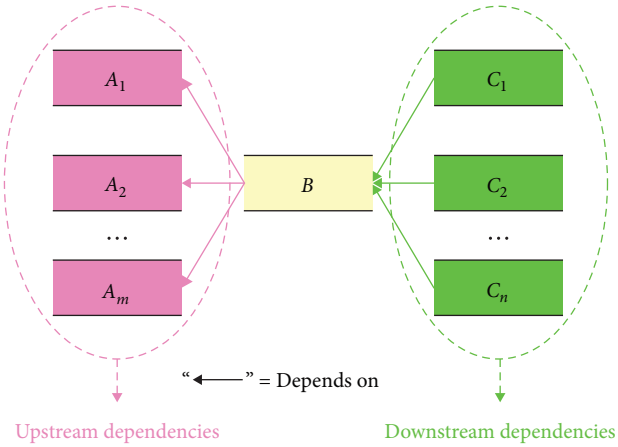

Figure 1: Definition of upstream dependencies and downstream dependencies.

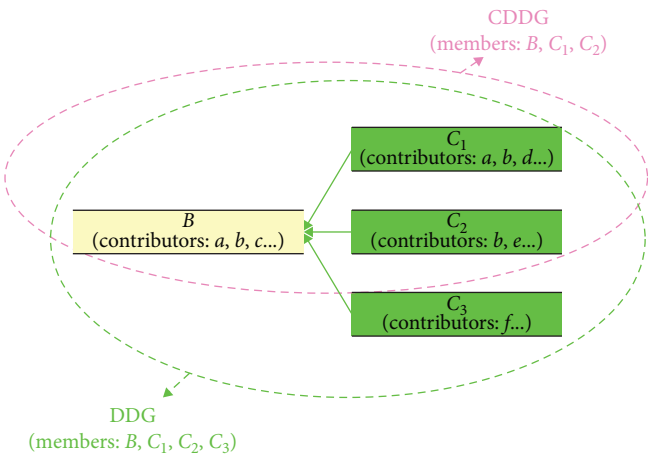

Figure 2: Construction of DDG and CDDG.

which are members of the DDG. Of these, packages C_1 and C_2 have common contributors with package B, so the CDDG can be constructed according to our definition.

3.2. Selection of Packaging Ecosystems. We conducted an empirical study on the DDGs and CDDGs in packaging ecosystems using data from libraries.io (https://libraries.io/), which monitors over 4 million open-source packages across 32 popular package managers for specific programing languages. As of December 1, 2021, Table 1 summarizes the top 12 package managers based on their number of packages, which account for more than 88% of the total. These managers are at different stages of development, reflecting the evolution of packaging ecosystems over time. Some, like Maven, PyPI, and RubyGems, emerged early and are now mature and stable, while others, such as npm and Cargo, are relatively young and still growing. Finally, some older package managers, like CPAN, have become stagnant.

To investigate the development trend of different packaging ecosystems, we calculate the growth rates of 12 popular package managers with time, as shown in Figure 3. The growth rate of the packaging ecosystem p is calculated using the following formula:

$$G_i^p = \frac{P_i^p - P_{i-1}^p}{P_{i-1}^p}, \tag{1}$$

where G_i is the growth rate of the packaging ecosystem p in year i, P_i is the number of the packages in the packaging ecosystem p in year i.

We can observe that the growth rate of packaging ecosystems can be broadly categorized into three patterns: steady growth, fluctuating growth, and almost no growth. The steady growth pattern applies to larger packaging ecosystems whose growth rate remains stable (with a difference of less than 0.2 between maximum and minimum rates). The fluctuating growth pattern is characterized by a relatively high growth rate, which may vary over time (with a difference of more than 0.2 between maximum and minimum rates). The no-growth pattern pertains to cases where the number of packages is small and remains almost constant (with a growth rate consistently below 0.1). Table 2 illustrates the distribution of these growth patterns across different packaging ecosystems.

For our study, we selected three widely used package managers that have comparable numbers of packages but exhibit distinct growth patterns. These package managers are Cargo, CPAN, and RubyGems. We chose Cargo due to its highly variable growth rate, RubyGems due to its minimal difference in growth rate, and CPAN because its growth rate has consistently remained below 0.1.

Below is a brief overview of each of these package managers:

(1) Cargo is the official Rust package manager. Rust, released by Mozilla in 2012, is a safer alternative to existing system programing languages. Cargo is a multi-functional front end for building, packaging, and configuring Rust projects. Crates.io (https://crates.io/), maintained by the Rust team, is the default public registry for the Rust ecosystem. It is the youngest and smallest of the three selected ecosystems.

(2) The CPAN can help programmers find modules and programs that are not included in the Perl standard release. CPAN first appeared in 1993 and was officially introduced to the Perl community in 1995. CPAN (https://www.cpan.org/) has been active online since October 1995. It is the oldest of our selected ecosystems.

(3) RubyGems is a package management framework for the Ruby programing language, providing a standard format for distributing Ruby programs and libraries. It was created during RubyConf 2004 and was released to the public on March 14, 2004. RubyGems.org (https://rubygems.org/) is the Gem hosting service of the Ruby community. It is the largest of the three packaging ecosystems.

Table 1: Information of the top 12 package managers on December 1, 2021.

Package manager	Creation year	Language	Number of packages	Website	Top five keywords
npm	2010	JavaScript	2,121k	https://www.npmjs.com	React, javascript, typescript, nodejs, vue
Go	2007	Go	449k	https://pkg.go.dev	Golang, go, kubernetes, cli, hacktoberfest
Maven	2004	Java	445k	http://maven.org	Java, scala, hacktoberfest, kotlin, android
PyPI	2003	Python	409k	https://pypi.org/	Python, python3, django, testing, api
Packagist	2012	PHP	339k	https://packagist.org	Laravel, php, api, yii2, framework
NuGet	2010	.NET	288k	https://www.nuget.org	Dotnet, csharp, Tag1, Tag2, ios
RubyGems	2004	Ruby	176k	https://rubygems.org	Ruby, rails, gem, ruby-gem, hacktoberfest
CocoaPods	2011	Objective-C	85k	http://cocoapods.org/	Swift, ios, cocoapods, objective-c, carthage
Cargo	2012	Rust	73k	https://crates.io	Rust, cli, blockchain, async, parser
Bower	2012	JavaScript	70k	http://bower.io	Angular, javascript, jquery, css, web-components
CPAN	1995	Perl	39k	https://metacpan.org	Perl, per5, localization, unicode, hacktoberfest
Pub	2011	Dart	27k	https://pub.dartlang.org	Flutter, dart, pub, pubspec, flutter-plugin

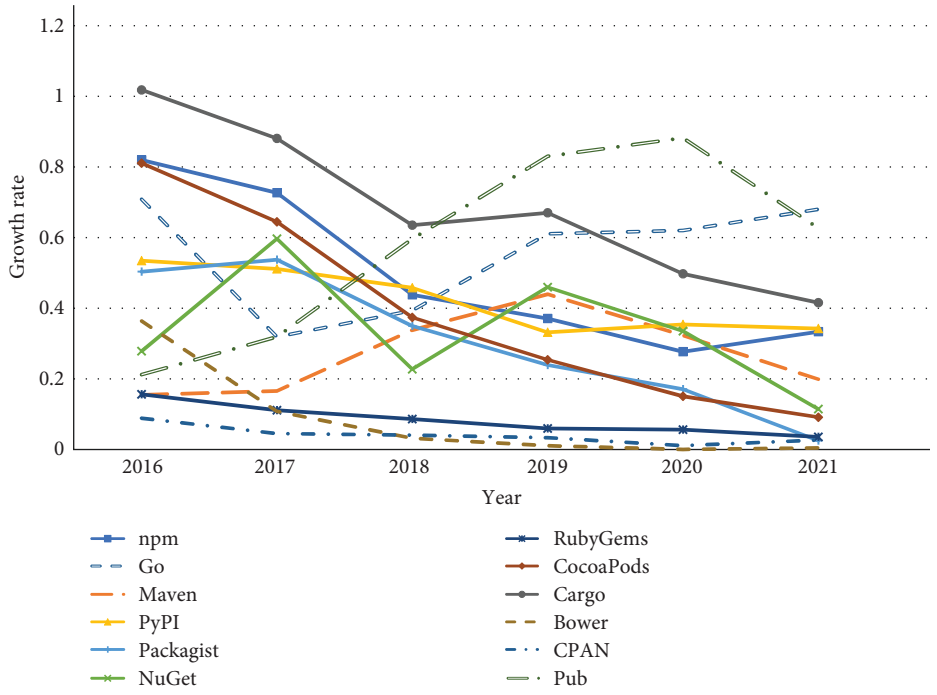

Figure 3: Evolution of the growth rate of the packaging ecosystem.

Table 2: Packaging ecosystems with different growth patterns.

Type	Steady growth	Fluctuating growth	No growth
Package manager	PyPI, RubyGems	npm, Go, Maven, Packagist, NuGet, CocoaPods, Cargo, Bower, Pub	CPAN
Language	Python, ruby	JavaScript, Go, Java, PHP, .NET, Objective-C, Rust, Javascript, Dart	Perl

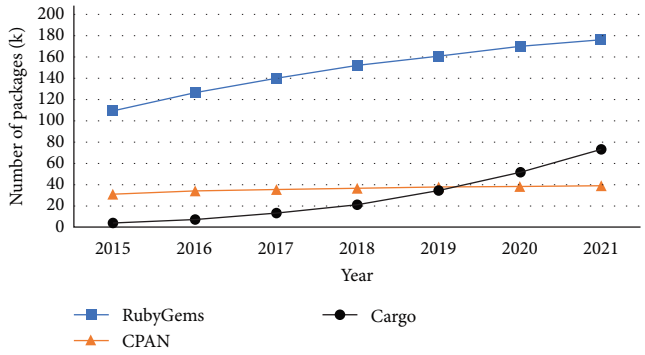

Figure 4: Evolution of the number of packages in the three packaging ecosystems by year.

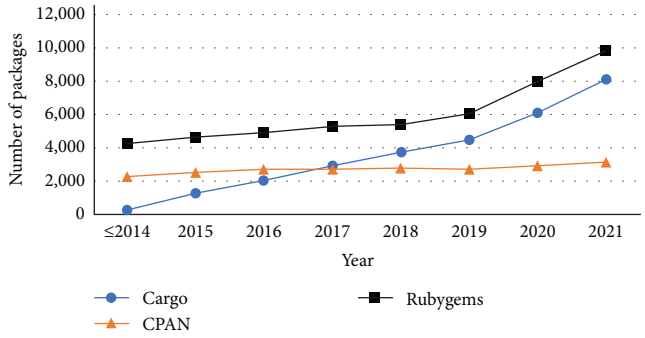

Figure 5: Evolution of the number of packages with downstream dependency packages in three packaging ecosystems by year.

We present the evolution of the number of packages and packages with dependents in the three selected packaging ecosystems through Figures 4 and 5, respectively. At the outset, we observe that the RubyGems ecosystem had the highest number of packages, whereas the Cargo ecosystem had the fewest. Both ecosystems witnessed rapid growth, while the CPAN ecosystem demonstrated negligible growth. Importantly, the Cargo packaging ecosystem has seen a remarkable surge since 2015 and surpassed the CPAN ecosystem in 2020. In contrast, the RubyGems ecosystem has been developing steadily over time.

Figure 5 reveals that the RubyGems ecosystem also boasts the largest number of packages with dependents and continues to grow consistently. Meanwhile, the Cargo ecosystem experiences the fastest growth rate in the number of packages with dependents and exceeded CPAN's figure in 2017. Notably, the growth of the number of packages with dependents correlates with the overall ecosystem's development trend. Therefore, this study compares the DDGs of these three packaging systems, given their different growth trends.

We then obtained the packages of different ecosystems from libraries.io, which covers the data till December 1, 2021.

Our statistical analysis reveals that 20% of packages in three packaging ecosystems have over 20 dependencies. In these ecosystems, keywords are utilized to describe package types, functions, or languages, with a total of 11,715, 653, and 8,235 keywords present in Cargo, CPAN, and RubyGems, respectively. Interestingly, this subset of packages (comprising 20% of the total), along with their dependent packages, covers more than 80% of all the ecosystem keywords, specifically 11,255 (96%), 525 (80%), and 7,892 (95%) in Cargo, CPAN, and RubyGems, respectively.

Therefore, we set the threshold of the number of dependency packages in a DDG to 20 to filter out the DDGs we study. This resulted in obtaining 705, 730, and 1,526 DDGs from Cargo, CPAN, and RubyGems, respectively. Accordingly, the number of CDDGs is 185, 178, and 329, respectively. It is worth noting that the number of CDDGs is notably less than that of DDGs, indicating that collaborations by sharing contributors between packages within the packaging ecosystem are not so extensive.

3.3. Development Trend Prediction for DDGs and CDDGs. The emergence and evolution of ecosystems have given rise to the development of DDGs and CDDGs. Furthermore, a precise trend prediction model for these entities can aid in comprehending the fundamental drivers that steer their growth.

We select features from both the focal package and dependency packages, as shown in Table 3, to construct the prediction models. The details of our feature extraction are listed as follows:

(1) The package age is calculated from its first release year to 2021.
(2) The numbers of stars and forks are the metrics of the corresponding repositories on GitHub of the packages.
(3) The number of keywords of the package is crawled from the libraries.io platform.
(4) Rank refers to the "SourceRank" on the libraries.io platform which is the ratings of the packages. SourceRank is the score for a package based on fourteen metrics; it is used across the site to boost high-quality packages.
(5) Dependency packages and repositories count the number of downstream dependency packages and repositories, respectively.
(6) Number of releases equals the total number of version updates (including major and minor releases).

The features of dependency packages in DDGs and CDDGs are determined by their mean average values.

Multicollinearity is a serious problem in many predictive modeling approaches, which leads to an inaccurate conclusion in relation to the relationship between predictor variables and response variables [34]. We apply collinearity diagnostics to detect multicollinearity. Tolerance and the variance inflation factor (VIF) are two closely related statistics for the diagnosis of collinearity in multiple regression [35]. They are based on the R-squared value obtained by regressing all the other predictors in the analysis. Tolerance can be defined as follows:

Table 3: Features of focal package and downstream dependency packages.

Label	Type	Features
f_1	Focal package	Age
f_2		Number of stars
f_3		Number of keywords
f_4		Number of forks
f_5		Rank
f_6		Dependency packages
f_7		Dependency repositories
f_8		Number of releases
f_9	Downstream dependency packages	Average age
f_{10}		Average number of stars
f_{11}		Average number of keywords
f_{12}		Average number of forks
f_{13}		Average rank
f_{14}		Average dependency packages
f_{15}		Average dependency repositories
f_{16}		Average number of releases

$$\text{Tolerance} = 1 - R^2. \tag{2}$$

According to the previous research, if tolerance is less than 0.2 [36], the problem of multicollinearity between predictor variables should be considered.

The VIF is defined as the reciprocal of tolerance:

$$\text{VIF} = \frac{1}{1 - R^2}. \tag{3}$$

The VIF exceeds its cutoff point, 10 indicates multicollinearity [37]. After analyzing the collinearity statistics for DDGs and CDDGs, we find that all tolerances are higher than 0.2 and VIFs are less than 2.5, indicating that the predictor variables are independent. Therefore, the features we select will not cause the multicollinearity problem.

We apply in this paper five traditional approaches to predict the development trend of DDGs and CDDGs, including linear regression, random forest, K-nearest neighbor (KNN), AdaBoost, and Gradient Boosting Regression Tree (GBRT).

The performance of the prediction model is evaluated by comparing the predicted values with the observed values. In this paper, three evaluation criteria are selected to measure the performance, i.e., mean absolute error (MAE), root mean square error (RMSE), and mean absolute percentage error (MAPE). Their definitions are as follows:

$$\text{MAE}(y, \widehat{y}) = \frac{1}{n}\sum_{i=1}^{n} |(y_i - \widehat{y}_i)|, \tag{4}$$

$$\text{RMSE}(y, \widehat{y}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \widehat{y}_i)^2}, \tag{5}$$

$$\text{MAPE}(y, \widehat{y}) = \frac{100\%}{n}\sum_{i=1}^{n} \left|\frac{(\widehat{y}_i - y_i)}{y_i}\right|, \tag{6}$$

where n is the number of samples, y_i is the observed value of each sample while $\widehat{y}_i$ refers to the predicted value. The smaller the MAE, RMSE, and MAPE, the more accurate the prediction results will be.

4. Research Questions and Results

In this section, we present the research process, results, and summaries for the three research questions.

4.1. RQ1: What Kind of Roles Do DDGs and CDDGs Play in the Packaging Ecosystem? How Do the Features of DDGs Evolve with the Development of the Ecosystem over Time?

4.1.1. What Kind of Roles Do DDGs and CDDGs Play in the Packaging Ecosystem? The DDGs model all the direct interactions between the focal package and dependency packages. Since packages and their interactions are the premises of maintaining the development of the ecosystem, DDGs capture the underlying cooperative relationships among packages in the ecosystem. CDDGs further model the cooperative relationships between package contributors. In this way, the participants in the packaging ecosystem (including packages and contributors) and their interactions (including dependencies among packages and cooperation among developers) are taken into account. Therefore, DDGs and CDDGs play a vital role in the packaging ecosystem.

To explore the roles of DDGs and CDDGs in the packaging ecosystem in a quantitative way, we analyze the features of DDGs and CDDGs.

The average number of dependency packages in DDGs for Cargo, CPAN, and RubyGems is 59.34, 21.82, and 30.68,

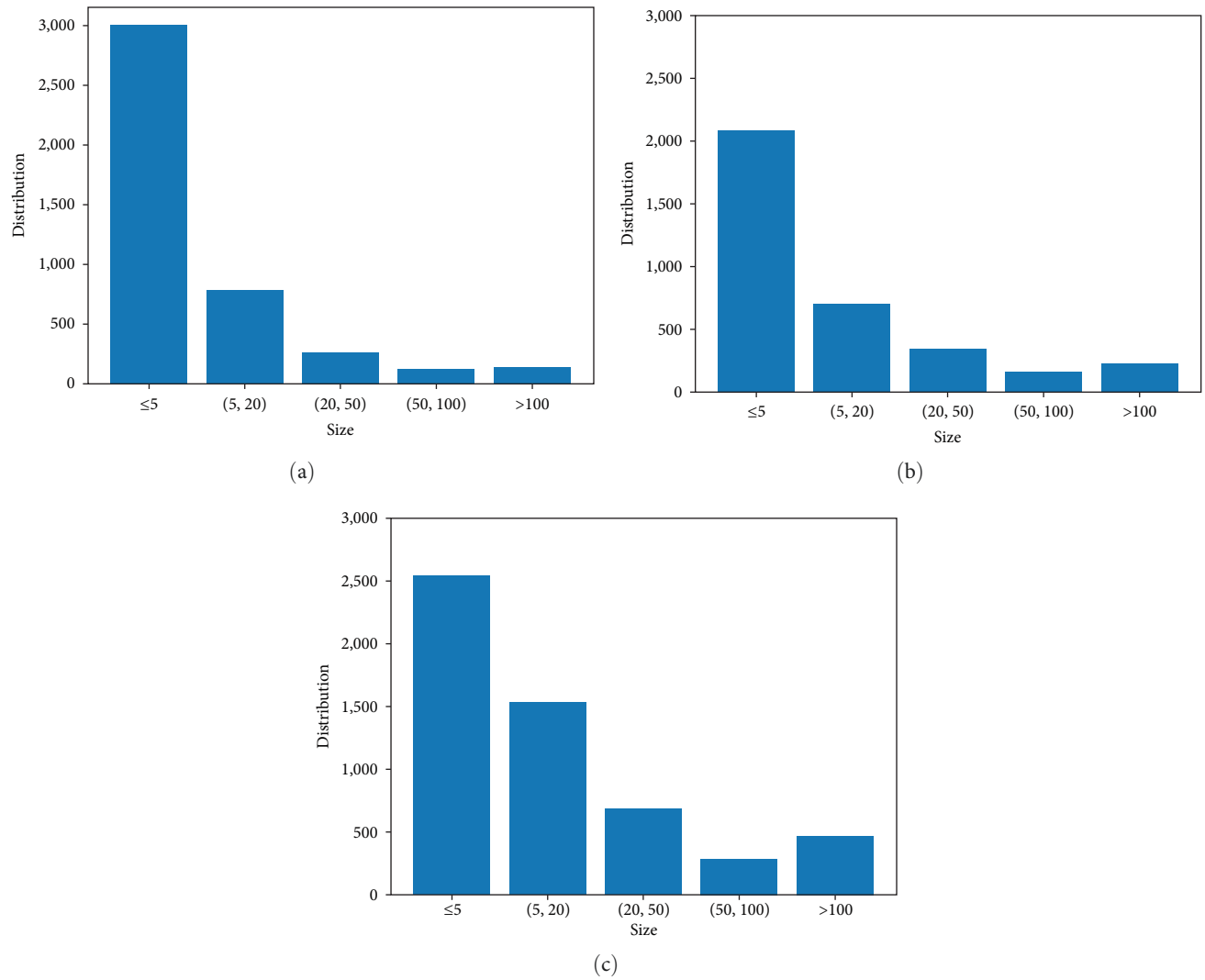

FIGURE 6: Size distribution of downstream dependency packages in (a) Cargo, (b) CPAN, and (c) RubyGems packaging ecosystems.

respectively. Figure 6 shows that the number of dependency packages in DDGs follows a power-law distribution, indicating that most packages are relied upon by only a small number of other packages. Specifically, more than 80% of DDGs have less than 20 dependency packages. Notably, Cargo has the lowest average number of dependency packages in DDGs among the three packaging ecosystems, with over half of its DDGs having fewer than five dependency packages. This can be attributed to Cargo's relative youth, as many of its packages have not yet been widely adopted by others. In contrast, stable packaging ecosystems tend to form large DDGs when some dependencies continue to grow. For instance, in RubyGems, nearly 500 DDGs have over 100 dependency packages.

Tables 4 and 5 show statistics of the dataset and list the five largest DDGs and CDDGs, respectively. Cargo, CPAN, and RubyGems have 705, 730, and 1,526 DDGs, respectively, and the number of CDDGs is 185, 178, and 329, respectively. The number of packages used for research in DDGs is 20,102, 22,983, and 102,655 in Cargo, CPAN, and RubyGems, respectively, and 5,790, 4,335, and 13,940 in CDDGs. We also provide their information, including size, the ratio of focal packages, average age, average version updates, average contributors, and main keywords.

Tables 4 and 5 both exhibit the presence of certain focal packages. Notably, the members of a CDDG form a subset of those belonging to the DDG with the same focal package. In terms of Cargo, four of the top five DDGs have overlapping CDDGs. However, for CPAN, only one DDG has a corresponding CDDG in the list, indicating that a DDG may not necessarily have a proportionate CDDG size. This is understandable since a package often depends on another package for a specific function, and the developers may not necessarily be involved in the development of the focal package. The number of developers contributing to dependency packages varies across ecosystems. As rapidly growing ecosystems like Cargo are more likely to have larger CDDGs corresponding to larger DDGs, the size of the DDGs is not closely related to

Table 4: DDGs in the three packaging ecosystems (the five largest DDGs are listed).

No.	Focal package name	Size	Ratio of focal packages	Average age	Average version updates	Average contributors	Keywords
DDGs in Cargo (total DDGs: 705, total packages: 20,102)							
1	Serde	3,758	0.03	2.63	8.48	9.63	Rust, api, google, cli, web, protocol
2	Serde_derive	2,839	0.03	2.69	8.99	10.16	Rust, api, google, cli, web, protocol
3	Serde_json	2,747	0.02	2.71	8.31	9.48	Rust, api, google, cli, web, protocol
4	Log	2,494	0.03	2.81	9.23	12.09	Rust, cli, http, api, web, log
5	Libc	2,283	0.05	3.46	8.82	7.42	Rust, ffi, bindings, linux, database, gnome
DDGs in CPAN (total DDGs: 730, total packages: 22,983)							
1	Module-build	4,130	0.03	9.60	11.11	0.90	Perl, perl5, localization, unicode, dist-zilla, game
2	Moose	2,457	0.05	8.88	10.23	1.21	Perl, localization, unicode, perl5, dist-zilla, object-oriented
3	Exporter	1,709	0.06	9.67	14.51	1.58	Perl, perl5, perl-module, testing, debugger, cpan
4	Test-exception	1,667	0.04	9.28	12.99	1.71	Perl, perl5, localization, unicode, code4lib, cme
5	Version	1,619	0.07	10.12	14.82	1.89	Perl, localization, perl5, unicode, dist-zilla, compile
DDGs in RubyGems (total DDGs: 1,526, total packages: 102,655)							
1	Rails	8,860	0.01	6.25	8.66	3.85	Rails, ruby, gem, ruby-on-rails, activerecord, rails-engine
2	Activesupport	8,556	0.02	6.55	9.82	6.93	Ruby, rails, activerecord, gem, ruby-gem, rubygems
3	Pry	7,522	0.02	5.10	9.82	4.18	Ruby, sensu-plugins, rails, monitoring, metrics, gem
4	Sqlite3	6,476	0.01	5.89	8.85	3.37	Ruby, rails, activerecord, gem, ruby-on-rails, ruby-gem
5	Json	5,882	0.02	6.88	11.92	6.21	Ruby, rails, gem, api, api-client, monitoring

Table 5: CDDGs in the three packaging ecosystems (the five largest CDDGs are listed).

No.	Focal package name	Size	Ratio of focal packages	Average age	Average version updates	Average contributors	Keywords
CDDGs in Cargo (total CDDGs: 185, total packages: 5,790)							
1	Serde	896	0.07	3.55	16.45	34.94	Google, rust, web, protocol, api, cli
2	Serde_derive	740	0.05	3.56	16.35	33.75	Google, web, rust, cli, protocol, api
3	Libc	683	0.06	3.96	14.84	20.53	Rust, ffi, bindings, gnome, linux, gtk-rs
4	Serde_json	646	0.05	3.62	15.17	33.55	Google, web, api, cli, protocol, rust
5	Hyper	606	0.01	3.67	12.77	26.71	Google, web, api, protocol, cli, rust
CDDGs in CPAN (total CDDGs: 178, total packages: 4,335)							
1	Moose	548	0.09	8.74	15.75	4.18	Perl, localization, perl5, unicode, dist-zilla, aws
2	Test-rinci	545	0.04	5.99	10.55	1.23	Test, metadata
3	Dist-zilla	476	0.13	8.60	14.76	3.71	Dist-zilla, metaprogramming, perl, object-oriented, amazon
4	Perl-osnames	306	0.07	5.92	14	1.26	Unix, bsd, sysv, posix
5	Dist-zilla-Plugin-git-contributors	302	0.15	8.65	15.04	3.72	Perl, dist-zilla, loading, packaging
CDDGs in RubyGems (total CDDGs: 329, total packages: 13,940)							
1	Activesupport	1,531	0.03	7.92	19.81	34.69	Ruby, rails, activerecord, html, activejob, gem
2	Rails	1,315	0.02	7.59	16.75	21.25	Ruby, rails, activerecord, gem, json, ecommerce
3	Activerecord	921	0.02	7.94	15.04	27.93	Ruby, rails, activerecord, ruby-on-rails, testing, sql
4	Rubocop	613	0.06	5.90	30.12	30.59	Ruby, rails, gem, rubocop, ruby-gem, activerecord
5	Railties	455	0.03	7.43	20.20	47.73	Rails, ruby, activerecord, authentication, devise, mongodb

the corresponding CDDGs' size in ecosystems that cease growth, such as CPAN.

There are notable differences in the keywords used by the top five DDGs and CDDGs. The top five DDGs tend to focus on fundamental tools such as building and framework, whereas the top five CDDGs prioritize technology-oriented keywords. In other words, contributors in CDDGs tend to collaborate based on shared interests in tech-related topics. Taken together, DDGs and CDDGs encompass over 80% of all keywords associated with the packaging ecosystem. This underscores the fact that packages found in DDGs and CDDGs account for a majority of the package types available across the entire ecosystem.

The analysis reveals that the packages found in CDDGs have a longer history compared to those in DDGs. This finding suggests that it takes time to establish collaborative relationships among different package developers. Moreover, CDDGs tend to have a higher number of version updates and contributors on average, indicating that they are more active than their DDG counterparts. These observations align with our expectations.

In addition, we observed that Cargo, despite being the youngest packaging ecosystem, has the highest average number of contributors in DDGs. This indicates that Cargo has successfully attracted a sizable community of developers who frequently collaborate within this ecosystem. It is one of the key factors contributing to the rapid growth of Cargo.

4.1.2. How Do the Features of DDGs Evolve with the Development of the Ecosystem over Time? To analyze the evolution of DDGs in the packaging ecosystem, we began by conducting a statistical analysis of their total number each year. Next, we manually selected five key features to assess the changes in DDGs over time. These features include the average number of stars and forks for packages, as well as the average rank, number of dependency packages, and releases for packages in DDGs. By examining these features, we aimed to gain insight into how DDGs have evolved within the larger ecosystem.

Figure 7 presents the evolution of DDG features over time. Figure 7(a) displays the annual growth in the number of DDGs across the three packaging ecosystems. Notably, RubyGems consistently has the highest number of DDGs, followed by CPAN and Cargo, which have the fewest DDGs. Interestingly, there has been exponential growth in the number of DDGs within the Cargo ecosystem. In the case of RubyGems, the number of DDGs grew rapidly before 2018 but has since slowed down. Conversely, the growth rate of DDGs in CPAN has nearly plateaued. These trends align with those observed in the overall number of packages within each ecosystem, underscoring the significant role played by DDGs in shaping these systems. Panels (b) through (f) of Figure 7 exhibit similar patterns, with the average metric values of DDGs in Cargo showing a significant decreasing trend, while those in RubyGems exhibit a slight decrease and those in CPAN remain mostly unchanged. Typically, these metric values increase over time for a given DDG. However, when new DDGs appear within an ecosystem, their metric values tend to be small in the initial stages. As the number of DDGs increases each year, the average metric values for all DDGs gradually decrease. This phenomenon is particularly evident in the case of Cargo, where the total number of DDGs has grown exponentially. For RubyGems, there was a noticeable drop in the metric values of DDGs, particularly before 2018, which aligns with the trends observed in Panel (a) of Figure 7. After 2018, the metric values continued to decline slightly.

Observing Figure 7, it can be noted that among the three package ecosystems, Cargo's DDGs have the highest average number of stars and ranks, which is due to its rapid development and significant attention from users. Such attention and feedback further facilitate its growth. Conversely, RubyGems' DDGs exhibit a larger average number of forks and version updates as compared to Cargo, given RubyGems' longer history in the market.

Moreover, Figure 7(e) illustrates the average number of dependency packages among the three packaging ecosystems. In earlier years, specifically 2015 and 2016, Cargo had the largest average number of dependencies for DDGs. Nevertheless, this figure significantly decreased over time and became the smallest. This pattern can be attributed to the growth trajectory of the number of DDGs in the Cargo ecosystem. It is important to note that attracting downstream dependencies takes time for any DDG; hence, the lower average number of dependencies of DDGs in the later years does not necessarily translate to smaller-sized DDGs in Cargo compared to the other two packaging ecosystems. Rather, it demonstrates that DDGs in Cargo had more dependencies in the earlier years, as evident in the data.

Regarding the metric values of DDGs across the three ecosystems, Cargo, being a new ecosystem with only 75 DDGs in 2015, is currently undergoing fast-paced development with highly active DDGs. In contrast, RubyGems, having been around for a longer period, maintains a certain number of new DDGs and exhibits stable increases in stars, forks, dependencies, and version updates, indicating a stage of stable development. Meanwhile, CPAN is experiencing stagnation with a minimal number of newly formed DDGs each year, and the forks and version updates of its DDGs have stabilized. Regrettably, it receives little attention, resulting in the smallest number of stars and average rankings among the three ecosystems.

Summary of findings: DDGs model the dependency relationships among packages, and CDDGs further identify the collaborations between contributors. The size distribution of DDGs follows the power-law distribution. The average metric values of DDGs are consistent with the development stage of the packaging ecosystem. A fast-developing ecosystem receives considerable user feedback and witnesses numerous new DDGs formations annually. Additionally, the metric values of DDGs in a stable development ecosystem exhibit

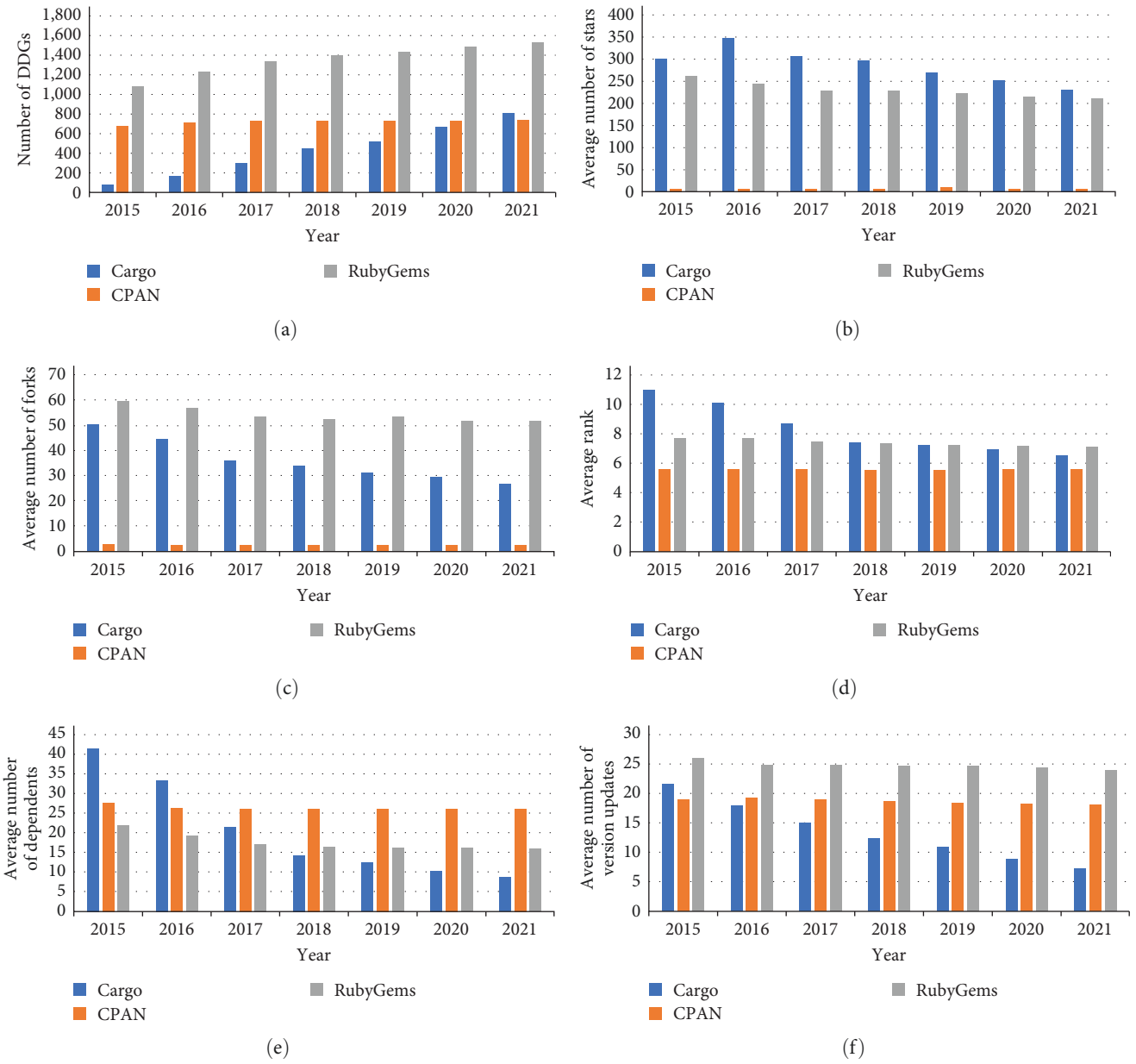

FIGURE 7: Evolution of features in DDGs: (a) evolution of the number of DDGs; (b) evolution of an average number of stars; (c) evolution of an average number of forks; (d) evolution of average rank; (e) evolution of an average number of dependents; (f) evolution of an average number of releases.

steady growth. Conversely, for an ecosystem in the stagnating stage, there is little change in the metric values of DDGs.

4.2. *RQ2: How Does the Focal Package Influence the Development of the DDG and CDDG? Are the Features of Downstream Dependency Packages Relevant to the Development of the Focal Package?*

4.2.1. How Does the Focal Package Influence the Development of the DDG and CDDG? The development of a DDG is influenced by its component packages, including the focal package and downstream dependency packages. Studies indicate that the focal packages play a vital role in shaping the evolution of DDGs [7, 8]. Consequently, we conducted a quantitative analysis to examine the relationship between features of the focal package and the development of a DDG.

To evaluate the development status of a DDG, we measured its size and the average number of releases of its packages. Our study pursued an answer to RQ2.1 by examining how focal packages are linked to DDG development through an analysis of their relationship with DDG size and average number of releases. We focused on eight features of the focal packages, including age, number of stars, number of

forks, rank, number of keywords, number of dependency packages, number of dependent repositories, and number of releases. Using the Spearman correlation coefficient, we investigated the links between these features and DDG size, as well as the relationship between these features and the average number of releases of the DDGs. Notably, the number of stars, number of forks, and rank indicate package popularity, while the number of dependent repositories and number of releases measure the activity status of package development.

The analysis presented in Table 6 highlights that in Cargo and RubyGems, nearly all features of the focal packages display a positive correlation with DDG size. Notably, rank holds a medium (in Cargo) or strong (in RubyGems) positive correlation with DDG size, while the number of dependency repositories has a medium positive correlation. These correlations suggest that package popularity and activity level are key factors driving attraction to downstream dependency packages. Moreover, our findings indicate that age is not a significant factor influencing DDG size; rather, a package's influence holds greater sway over attracting downstream dependencies than its age.

Our analysis did not reveal any significant correlation between the features of focal packages and the average number of releases of DDGs. Notably, we found no link between the number of releases of focal packages and their downstream dependencies. This finding initially appears to contradict the fact that a version update of a focal package can create version compatibility issues, leading to necessary updates to downstream dependency packages. However, prior research indicates that updates to downstream dependency packages often occur with delays [38]. This may explain the absence of correlation in our findings.

Table 7 reveals that in Cargo and CPAN, the features of focal packages display a positive correlation with CDDG size, along with the number of keywords. These correlations prove much stronger than those observed for DDGs, thereby emphasizing the heightened influence of focal packages on CDDG development. Unlike the correlation analysis between focal package features and the number of releases of DDGs in Table 6, Table 7 demonstrates that some focal package features also exhibit a positive correlation with the number of releases of CDDGs. Specifically, rank holds a weak positive correlation with the average number of releases of CDDGs, while the number of dependency repositories shows a medium positive correlation with the average number of releases of CDDGs. Notably, age also displays a positive correlation with the number of releases of CDDGs, suggesting that time plays a role in fostering collaborations among project members and subsequently promoting package activity within the DDG.

Owing to its stagnant state, CPAN exhibits distinct patterns compared to Cargo and RubyGems. Notably, focal packages in CPAN have no discernible influence on DDG development. However, within CDDGs, we found a weak positive correlation between focal packages and CDDG development in CPAN. Furthermore, our analysis indicated that focal packages displayed no attraction to downstream dependency packages, providing evidence of the sluggish pace of ecosystem growth.

4.2.2. Are the Features of Downstream Dependency Packages Relevant to the Development of the Focal Package? We also want to know whether the features of downstream dependency packages are related to the status of DDGs and the focal packages. We study this by analyzing the correlation between the features of downstream dependency packages and the size of the DDGs, and the correlations between the features of the downstream dependency packages and the number of releases of the focal packages.

Since the number of downstream dependency packages in the DDGs and CDDGs follows the long-tail distribution, to remove the impact of a large number of new packages on the overall correlation, we select the top 20% of downstream dependency packages in terms of the number of their downstream dependency packages for the correlation analysis. The results are shown in Tables 8 and 9.

It can be seen that in DDGs and CDDGs, some features of the downstream dependency packages have a weak positive correlation with the size of the DDGs. The correlation becomes stronger in CDDGs. This phenomenon indicates that the size of DDGs is related to the popularity and activity of the downstream dependency packages. In CDDGs, the number of releases of the downstream dependency packages and the number of releases of the focal package have a weak positive correlation, while there is no correlation in DDGs.

After validating the configuration files of different versions in Figure 8, we found that when the focal package was updated, the update frequency of downstream dependency packages that responded to the update in CDDGs was twice as many as for packages without collaboration in DDGs. Due to the emergence of collaboration between contributors, downstream dependency packages in CDDGs are more sensitive to updates, such as bug fixes and new functions in focal packages, and will tend to update accordingly. This shows there is an inherent interaction between downstream dependency packages and focal packages in CDDGs. For example, the requirement of the downstream dependency packages promotes the version update frequency of focal packages in turn, which improves the number of releases of the focal packages.

It is also worth noting that the correlation in the Cargo is the weakest, followed by RubyGems, while the strongest is CPAN, which implies that the role of downstream dependency packages in DDGs differs at different stages of packaging ecosystems. In the early stage, focal packages play a leading role, which directly affects the development of the downstream dependency packages. With the development of the downstream dependency packages, their influence gradually increases, which can then affect the development of focal packages.

Summary of findings: Focal packages, especially those with high rankings, play a leading role in the development of CDDGs. More popular and active packages generally attract more dependency packages. In contrast, the age of a package does not necessarily correspond to attracting more

Table 6: Spearman correlation coefficients between the features of the focal package and the development of the DDG.

	Features of focal package and size of DDG (correlation coefficient (p-value))			Features of focal package and average number of releases of DDG (correlation coefficient (p-value))		
Features	Cargo	CPAN	RubyGems	Cargo	CPAN	RubyGems
Age	0.17 (0.03)	0.11 (0.00)	**0.25 (1.48e-22)**	0.16 (0.00)	0.12 (0.00)	0.10 (0.00)
Number of stars	**0.26 (0.00)**	0.13 (0.00)	**0.42 (1.35e-63)**	−0.09 (0.04)	0.04 (0.27)	0.01 (0.65)
Number of forks	**0.35 (1.17e-06)**	0.16 (2.49e-05)	**0.38 (5.72e-51)**	−0.05 (0.19)	0.05 (0.17)	0.04 (0.11)
Rank	**0.54 (1.29e-15)**	**0.26 (0.00)**	**0.65 (2.24e-40)**	0.09 (0.04)	0.03 (0.44)	0.10 (0.00)
Number of keywords	0.10 (0.18)	0.05 (0.16)	0.03 (0.27)	−0.07 (0.35)	−0.07 (0.05)	−0.02 (0.52)
Number of dependent repositories	**0.47 (1.02e-11)**	0.01 (0.87)	**0.50 (1.79e-22)**	0.18 (5.19e-11)	−0.05 (0.13)	0.09 (0.05)
Number of releases	**0.25 (0.00)**	0.13 (0.00)	**0.29 (2.46e-29)**	−0.0 (0.13)	−0.02 (0.66)	0.04 (0.13)

The correlation coefficient is not less than 0.20, highlighting a higher correlation using bold font.

Table 7: Spearman correlation coefficients between features of the focal package and the development of the CDDG.

	Features of focal package and size of CDDG (correlation coefficient (*p*-value))			Features of focal package and average number of releases of CDDG (correlation coefficient (*p*-value))		
Features	Cargo	CPAN	RubyGems	Cargo	CPAN	RubyGems
Age	**0.26 (9.64e-10)**	0.16 (0.03)	**0.40 (3.59e-14)**	**0.49 (9.34e-13)**	**0.24 (3.42e-10)**	**0.20 (0.00)**
Number of stars	**0.43 (9.02e-5)**	**0.23 (0.00)**	**0.49 (1.37e-21)**	0.16 (0.03)	**0.25 (0.00)**	0.15 (0.01)
Number of forks	**0.43 (2.19e-24)**	**0.26 (0.00)**	**0.50 (2.67e-22)**	0.19 (0.04)	**0.30 (3.66e-05)**	0.18 (0.00)
Rank	**0.72 (1.42e-85)**	**0.47 (3.58e-40)**	**0.71 (1.97e-222)**	**0.38 (1.37e-07)**	**0.34 (2.49e-06)**	**0.39 (1.30e-13)**
Number of keywords	0.01 (0.73)	0.11 (0.16)	**0.23 (2.97e-05)**	0.14 (0.06)	**0.33 (5.61e-6)**	0.02 (0.73)
Number of dependent repositories	**0.56 (2.27e-44)**	0.11 (0.00)	**0.61 (3.91e-148)**	**0.49 (4.94e-13)**	0.04 (0.19)	**0.48 (6.10e-21)**
Number of releases	**0.41 (1.89e-22)**	0.20 (0.01)	**0.44 (1.22e-16)**	0.19 (0.01)	0.17 (0.03)	**0.23 (3.32e-05)**

The correlation coefficient is not less than 0.20, highlighting a higher correlation using bold font.

Table 8: Spearman correlation coefficients for features of downstream dependency packages and the development of the focal package in DDG (top 20% members).

	Features of downstream dependency packages and size of DDG (correlation coefficient (*p*-value))			Features of downstream dependency packages and number of releases of focal package (correlation coefficient (*p*-value))		
Features	Cargo	CPAN	RubyGems	Cargo	CPAN	RubyGems
Average age	0.04 (0.32)	0.09 (0.02)	0.07 (0.00)	0.07 (0.11)	0.18 (8.56e-07)	0.05 (0.04)
Average number of stars	0.10 (0.03)	**0.26 (9.61e-13)**	0.19 (1.02e-13)	0.03 (0.44)	0.10 (0.00)	0.02 (0.52)
Average number of forks	0.08 (0.09)	**0.29 (1.86e-15)**	**0.22 (7.69e-17)**	0.00 (0.92)	0.05 (0.19)	0.07 (0.01)
Average rank	0.11 (0.01)	**0.27 (3.94e-13)**	0.04 (0.12)	0.16 (0.00)	0.08 (0.32)	0.09 (0.00)
Average number of keywords	0.02 (0.65)	**0.36 (4.68e-24)**	0.02 (0.53)	0.09 (0.03)	0.03 (0.47)	0.02 (0.56)
Average number of dependent repositories	**0.21 (2.09e-05)**	**0.43 (2.83e-33)**	**0.31 (9.75e-34)**	0.03 (0.56)	0.13 (0.00)	0.02 (0.53)
Average number of releases	0.05 (0.23)	0.07 (0.07)	0.01 (0.59)	0.05 (0.29)	0.06 (0.10)	0.07 (0.01)
Average number of dependent packages	0.17 (7.11e-05)	**0.36 (5.31e-23)**	**0.24 (1.30e-19)**	0.12 (0.01)	0.01 (0.85)	0.03 (0.21)

The correlation coefficient is not less than 0.20, highlighting a higher correlation using bold font.

Table 9: Spearman correlation coefficients for features of downstream dependency packages and the development of the focal package in CDDGs (top 20% members).

	Features of downstream dependency packages and size of CDDG (correlation coefficient (p-value))			Features of downstream dependency packages and number of releases of focal package (correlation coefficient (p-value))		
Features	Cargo	CPAN	RubyGems	Cargo	CPAN	RubyGems
Average age	0.14 (0.06)	**0.60 (1.01e-18)**	**0.41 (1.26e-14)**	**0.25 (0.00)**	**0.26 (0.00)**	0.01 (0.83)
Average number of stars	0.18 (0.01)	**0.65 (3.61e-23)**	**0.56 (1.45e-28)**	0.09 (0.20)	**0.39 (5.03e-08)**	0.18 (0.00)
Average number of forks	0.13 (0.07)	**0.67 (1.79e-24)**	**0.52 (2.95e-24)**	0.05 (0.50)	**0.38 (1.49e-07)**	**0.21 (0.00)**
Average rank	0.18 (0.01)	**0.65 (4.64e-23)**	**0.58 (1.93e-30)**	0.02 (0.77)	**0.25 (9.55e-12)**	0.10 (0.08)
Average number of keywords	0.14 (0.06)	**0.27 (0.00)**	**0.26 (2.89e-06)**	0.16 (0.03)	0.07 (0.33)	0.16 (0.00)
Average number of dependency repositories	**0.29 (7.23e-05)**	**0.60 (1.08e-18)**	**0.59 (3.45e-32)**	0.05 (0.51)	**0.22 (0.00)**	0.13 (0.02)
Average number of releases	0.13 (0.08)	**0.38 (2.49e-07)**	**0.38 (4.63e-13)**	**0.23 (0.00)**	**0.34 (4.48e-06)**	**0.25 (4.76e-06)**
Average number of dependency packages	**0.33 (4.43e-06)**	**0.56 (2.29e-16)**	**0.59 (1.79e-32)**	0.01 (0.86)	**0.20 (5.50e-08)**	0.12 (0.03)

The correlation coefficient is not less than 0.20, highlighting a higher correlation using bold font.

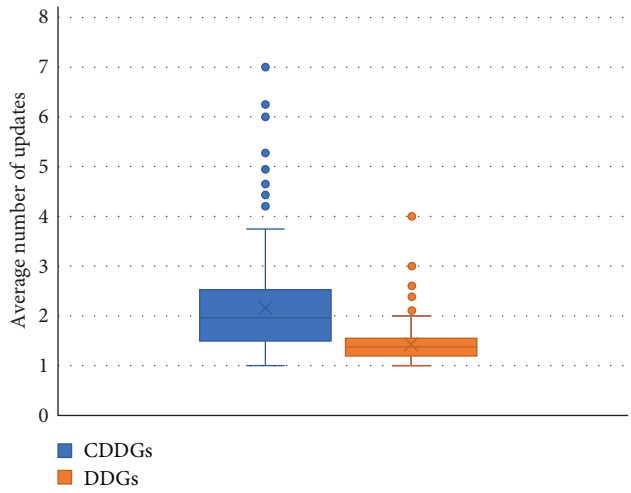

FIGURE 8: Comparison of update frequency of DDGs and CDDGs.

dependency packages. In CDDGs, focal packages have a stronger influence on the development of DDGs and also the dependency packages. In an ecosystem, the ability of focal packages to attract dependency packages in DDGs aligns with its development status. Moreover, the popularity and activity of dependency packages exhibit a positive correlation with the size of DDGs and CDDGs.

4.3. RQ3: Can We Predict the Development of DDGs and CDDGs? When addressing RQ2, our findings indicate that the majority of features within the focal package exhibit a more robust positive correlation with the development of CDDGs as opposed to DDGs. Additionally, CDDGs serve as a reflection of the innate cooperation amongst developers across packages. Consequently, predicting the development of DDGs based on selected features proves challenging, but it is possible to construct models for forecasting the development of CDDGs.

To address RQ3, we leveraged features from both the focal package and its dependency packages from the preceding year to forecast the size and number of releases of CDDGs in the subsequent year. We meticulously selected sixteen features that demonstrated a correlation with CDDG development, which are presented in Table 3. For each year, we took one CDDG's features alongside its corresponding size and number of releases for the following year as a singular sample. Consequently, we obtained 801 samples for Cargo, 746 samples for CPAN, and 1,447 samples for RubyGems over 5 years.

We trained our regression model using five conventional models: linear regression, random forest, KNN, AdaBoost, and GBRT. Since the size of CDDGs conforms to a long-tail distribution and CDDGs of varying sizes may exhibit distinct trends, we created separate models for two categories of CDDGs based on different size ranges. Given that roughly 20% of CDDGs exceed the threshold of 100, we used 100 as the benchmark.

In the model training, 80% of the samples are used as the training set, and 20% are used as the testing set. The CDDG size prediction results on the three packaging ecosystems are shown in Table 10. Correspondingly, the prediction results of a number of releases are shown in Table 11.

Different size prediction models have different prediction accuracies on different ecosystems. Among the five size prediction models, *random forest* and *GBRT* have a relatively stable performance. A comparison between the model prediction performances for CDDGs exceeding 100 in size versus those under 100 reveals that CPAN shows the most significant improvement, followed by Cargo and then RubyGems. This can be attributed to the uneven development of these three packaging ecosystems.

As shown in Table 5, the sizes of the three packaging ecosystems vary significantly, with RubyGems displaying the largest CDDGs, almost twice as large as those in the other two ecosystems. However, some of these larger CDDGs have already reached stability, while others are still in rapid development stages, making them challenging to predict accurately. Conversely, in the emerging Cargo ecosystem, both large and small-sized CDDGs tend to experience continued growth, leading to better prediction results. In the CPAN ecosystem, only large CDDGs display slow growth, while smaller CDDGs no longer develop, leading to a more uniform distribution that facilitates size prediction and improves results significantly.

In terms of predicting the number of releases, the performance of prediction models is significantly better for larger CDDGs. This is due to the notable impact that sudden growth in version updates of dependency packages has calculations on the number of releases for CDDGs. In smaller CDDGs, certain packages are frequently updated due to bugs or sufficient attention, while others may remain inactive due to a lack of attention or superior alternatives. However, because larger CDDGs receive ample attention and continue to update regularly, they yield more accurate predictions for the number of releases.

In terms of prediction performance, both the size and prediction of the number of releases for RubyGems are inferior to that of Cargo and CPAN. This can be attributed to RubyGems having the largest number of packages among the three packaging ecosystems, resulting in relatively larger CDDGs. Larger CDDGs exhibit two developmental trends; some continue to develop steadily while others stagnate, making predictions more challenging and adversely affecting accuracy. Conversely, Cargo continues to grow rapidly, with many new DDGs appearing annually, while CPAN is barely developed, making them easier to predict.

Furthermore, we conducted feature importance analyses to understand which normalized features play a more significant role in CDDG size and release prediction. The feature labels correspond to those shown in Table 3. As depicted in Figure 9, the importance ranks of features differ across the three packaging ecosystems. Nonetheless, the number of stars and dependency repositories of the focal package, average age, and average dependency repositories of dependency packages are generally more important in predicting CDDG size, as outlined in Table 3. In contrast, for predicting the number of releases for CDDG, features of dependency

Table 10: Prediction results of CDDG size.

Prediction models		CDDG size ≥ 100			CDDG size < 100		
		Cargo	CPAN	RubyGems	Cargo	CPAN	RubyGems
Linear regression	MAE	35.14	**35.53**	101.74	12.16	14.43	13.39
	RMSE	45.77	61.84	117.75	14.71	22.44	24.10
	MAPE	13.08%	18.78%	72.77%	32.76%	37.66%	25.99%
Random forest	MAE	25.11	37.07	**61.23**	4.53	1.41	**11.79**
	RMSE	33.05	65.41	**87.47**	**6.35**	5.41	**15.80**
	MAPE	**9.11**%	**17.18**%	36.86%	10.65%	2.4%	**25.18**%
KNN	MAE	75.69	44.81	67.61	13.97	12.65	13.08
	RMSE	95.61	74.11	93.80	18.55	15.77	16.90
	MAPE	32.09%	20.10%	37.98%	41.29%	30.26%	30.96%
AdaBoost	MAE	75.48	40.14	99.42	5.59	3.32	14.56
	RMSE	116.62	68.91	150.93	7.02	5.88	18.03
	MAPE	19.53%	19.80%	39.29%	13.47%	7.2%	30.65%
GBRT	MAE	**23.27**	36.23	90.33	**4.51**	**1.20**	12.30
	RMSE	**28.40**	**61.69**	157.87	6.36	**3.58**	16.86
	MAPE	9.20%	17.62%	**35.66**%	**10.59**%	**2.2**%	26.48%

Highlight the optimal results under each indicator of different models.

Table 11: Prediction results of number of releases of CDDGs.

Prediction models		CDDG size ≥ 100			CDDG size < 100		
		Cargo	CPAN	RubyGems	Cargo	CPAN	RubyGems
Linear regression	MAE	3.57	1.52	6.56	3.72	6.87	11.02
	RMSE	5.08	2.46	7.89	5.06	8.63	17.06
	MAPE	18.07%	11.26%	23.82%	26.95%	36.85%	36.50%
Random forest	MAE	**2.51**	1.43	**5.74**	**3.16**	6.57	9.70
	RMSE	2.90	2.27	**7.36**	**4.45**	**8.45**	**14.07**
	MAPE	16.10%	9.59%	**21.75**%	**20.30**%	30.46%	34.31%
KNN	MAE	2.54	**1.41**	7.24	3.74	8.02	9.72
	RMSE	**2.82**	**2.13**	8.55	5.10	9.84	13.57
	MAPE	**15.70**%	**9.12**%	26.20%	24.15%	36.00%	**33.03**%
AdaBoost	MAE	3.23	1.79	7.18	3.80	7.92	12.94
	RMSE	3.92	2.58	9.30	4.97	9.12	15.49
	MAPE	21.12%	12.14%	24.56%	22.14%	37.72%	37.71%
GBRT	MAE	2.70	1.52	6.18	3.59	**6.42**	**9.62**
	RMSE	3.37	2.43	7.70	5.00	9.03	14.21
	MAPE	18.70%	10.43%	22.92%	22.58%	**28.49**%	36.76%

Highlight the optimal results under each indicator of different models.

packages such as the average number of stars, forks, version updates, and rank are more influential.

Summary of findings: Applying random forest and GBRT yields acceptable prediction performance for CDDG size and number of releases, although the performance varies across packaging ecosystems with different development states. Furthermore, CDDGs of varying sizes exhibit differing stages of development, leading to variances in prediction accuracy. Packaging ecosystems in the initial and stagnant stages demonstrate better predictability, while those in the stable stage have worse prediction results due to package development polarization.

5. Discussion

After analyzing the development stages of the top 12 packaging ecosystems on the libraries.io platform, we have identified three growth patterns: steady growth, fluctuating growth, and no growth. More than 70% of these ecosystems fall into the fluctuating growth pattern, indicating that most are still in the early stage of development. Although Cargo, CPAN, and RubyGems share a similar number of packages, their growth patterns differ significantly. Understanding these growth patterns can help us better comprehend the development trends of various packaging ecosystems.

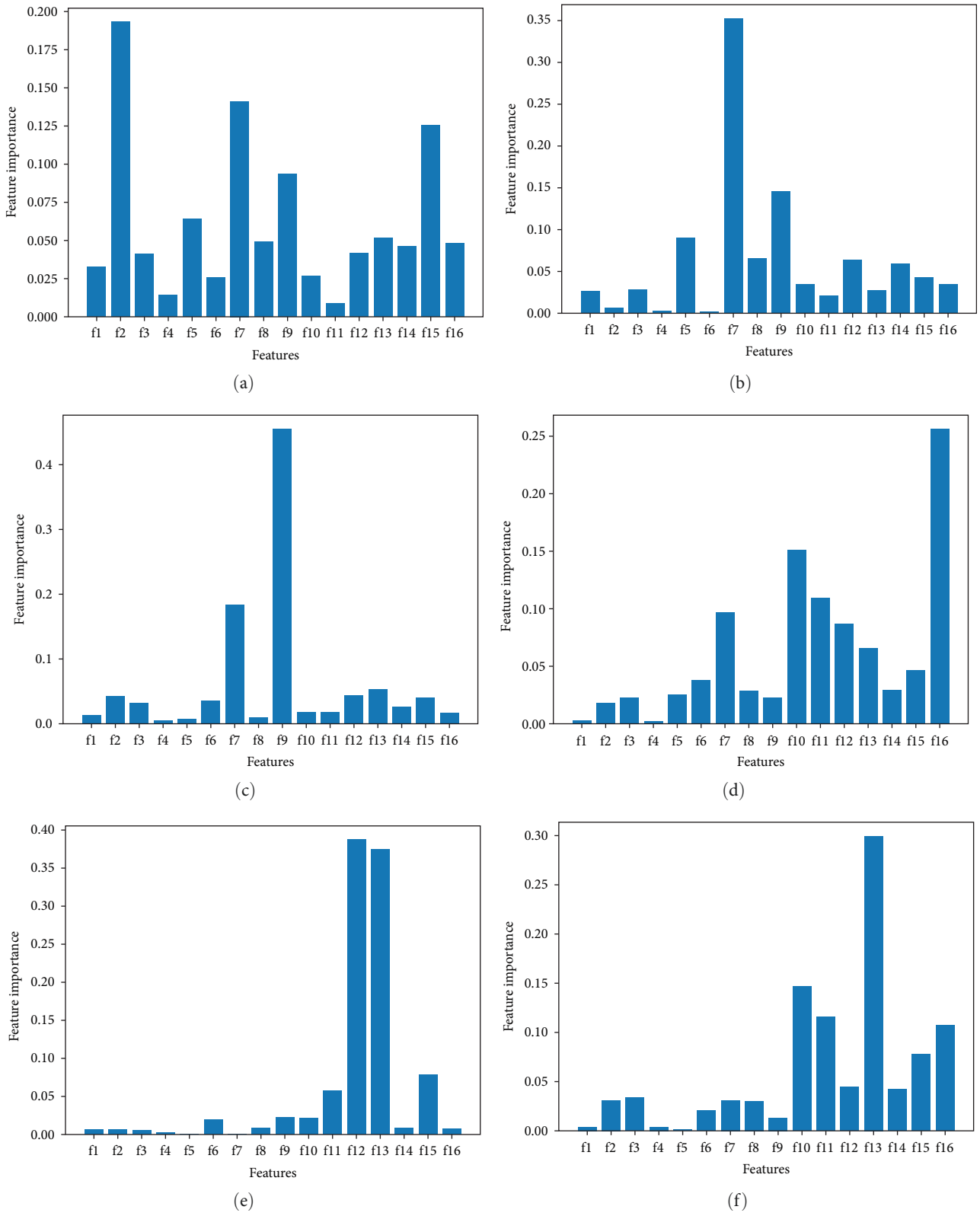

Figure 9: GBRT feature importance of size prediction and number of releases prediction of CDDGs in three packaging ecosystems: (a) GBRT feature importance of CDDG size prediction in Cargo; (b) GBRT feature importance of CDDG size prediction in CPAN; (c) GBRT feature importance of CDDG size prediction in RubyGems; (d) GBRT feature importance of the number of releases prediction of CDDGs in Cargo; (e) GBRT feature importance of the number of releases prediction of CDDGs in CPAN; (f) GBRT feature importance of the number of releases prediction of CDDGs in RubyGems.

The findings reveal a correlation between the overall state of DDGs and CDDGs and the development stages of their packaging ecosystems. When an ecosystem is developing rapidly, many DDGs and CDDGs show increased expansion rates. Conversely, when an ecosystem reaches a stable stage, most DDGs and CDDGs maintain a stable expansion speed. This reflects the fact that DDGs and CDDGs are essential constituents of the entire ecosystem, and their number of releases plays a vital role in driving its prosperity. While it is challenging to evaluate the development of a packaging ecosystem, the overall status of DDGs and CDDGs can serve as a measure of the ecosystem's health status. However, we must keep in mind that every DDG or CDDG within a packaging system has its own status, which is determined by various factors.

Apart from different developmental patterns, distinct languages, and technical domains also give rise to unique dynamics among DDGs and CDDGs in various ecosystems. Therefore, the interactions between the focal package and downstream dependency packages differ across ecosystems, making them highly complex and influenced by multiple factors.

Despite these differences, DDGs and CDDGs share some common laws. Generally, as the size of a DDG expands, the impact of the focal package on the number of releases of downstream dependency packages decreases. Comparing CDDGs with DDGs, we find that the inter-influence between the focal package and downstream dependency packages is more significant in CDDGs. Hence, we can build a reliable size prediction model for CDDGs. In ecosystems with rapid development, such as Cargo, the interinfluence between the focal package and downstream dependency packages is stronger. On the contrary, for stable ecosystems, this interinfluence becomes weaker. Therefore, we can assume that, in healthy ecosystems, the development of the focal package and downstream dependency packages in DDGs should be mutually beneficial.

Furthermore, statistical analysis reveals that CDDGs are more active and have more contributors compared to DDGs. Unlike DDGs, where only features of the focal package and the development of downstream dependency packages are related, some features of downstream dependency packages also affect the development of focal packages in CDDGs. This phenomenon shows that with the establishment of collaboration among contributors, the importance of internal members of CDDGs also increases. The role of downstream dependency packages differs across various stages of packaging ecosystems. In the early stages, focal packages play a leading role, directly influencing the development of downstream dependency packages. As the development of downstream dependency packages progresses, their influence gradually increases, subsequently affecting the development of focal packages. Focal packages and downstream dependency packages mutually influence each other, making the development of CDDGs more stable. Thus, promoting collaborations among contributors can help maintain the development of packaging ecosystems, reducing the risks of dependency hell, which arises when a package you depend on may be abandoned over time due to a lack of maintenance.

In summary, delving into the dependency package ecosystem offers valuable insights to a wide range of stakeholders and holds significant practical implications. The study imparts the following insights to various stakeholders:

Researchers: Investigating package dependencies and ecosystem evolution patterns can provide researchers with valuable quantitative data, analytical models, and deeper insights. Thus, they are encouraged to give greater consideration to DDGs and CDDGs in their software ecosystem research endeavors.

Practitioners: Our findings emphasize the crucial roles played by DDGs and CDDGs within software ecosystems. To gain insight into the current state and future trends, practitioners can benefit from analyzing the distinctive characteristics of DDGs and CDDGs.

Package managers: The healthiness of a CDDG significantly influences the vitality of a focal package. As a package manager, to foster the growth and success of a package, it is essential to strive for broader adoption by downstream packages.

6. Threats to Validity

Although we only analyzed three packaging ecosystems in this research, they are all widely used and represent different stages of development. Thus, our findings have general applicability. However, incorporating more ecosystems may help improve the reliability of our results.

Our research relies on the data from libraries.io, which is the most widely recognized package directory. It provides a large amount of information about each package. However, development platforms, such as GitHub, may also have related information on the repositories of some packages. If we can integrate these data into our study, more comprehensive models can be built to discover more relationships.

We studied the interaction mechanism between the focal packages and downstream dependency packages in DDGs without differentiating the types of focal packages. We believe DDGs of different types may have different interaction mechanisms. Incorporating focal package types into our study can explore the interaction mechanisms more deeply.

Furthermore, the features we listed to describe a DDG and a CDDG are based on our manual selections and are not necessarily complete. For CDDG development prediction, the features we chose are also limited, which may affect prediction precision. If more features can be included in the prediction model, the results would be more precise. However, because we tried to reveal the general interaction mechanisms between the focal package and downstream dependency packages, the features we selected capture these common factors so that the rules we found are more general.

7. Conclusions and Future Work

This study investigates package dependencies in packaging ecosystems. We identified three growth patterns in these ecosystems: steady growth, fluctuating growth, and no growth.

After a screening process, we selected three widely used packaging ecosystems with comparable total package numbers but exhibiting contrasting developmental patterns for analysis: Cargo for Rust, CPAN for Perl, and RubyGems for Ruby. To investigate ecosystem development, we defined two types of dependency groups: DDGs and CDDGs. By treating these groups as subecosystems, we proposed three research questions and provided detailed answers.

First, we conducted a statistical analysis of the features of DDGs and CDDGs in packaging ecosystems. We investigated the factors influencing DDG development and inherent interactions between the focal package and downstream packages. Finally, we combined features from both the focal and dependency packages to compare different prediction models for CDDG development in the three ecosystems. The experiment results showed that prediction performance varied among packaging ecosystems and CDDGs of different sizes. Of all the models, *GBRT* and *random forest* delivered the best performance, indicating that the size and number of releases have nonlinear relationships with the selected features.

Although we analyzed three typical packaging ecosystems in this study, future work could extend our research to include more ecosystems. Currently, we only consider direct dependencies between packages, but we aim to incorporate transitive dependencies into our analysis moving forward. Additionally, we plan to extract more features and examine their dynamic processes to further investigate DDG and CDDG development.

Data Availability

Our data are from libraries.io (https://libraries.io/), which monitors over 4 million open-source packages across 32 popular package managers for specific programing languages.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (grant number: 2018YFB1003800).

References

[1] R. Arora, S. Goel, and R. K. Mittal, "Supporting collaborative software development over GitHub," *Software: Practice and Experience*, vol. 47, no. 10, pp. 1393–1416, 2017.

[2] K. Manikas and K. M. Hansen, "Software ecosystems–a systematic literature review," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1294–1306, 2013.

[3] T. Mens, B. Adams, and J. Marsan, "Towards an interdisciplinary, socio-technical analysis of software ecosystem health," arXiv preprint arXiv: 1711.04532, 2017.

[4] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pp. 351–361, IEEE, Austin, TX, USA, 2016.

[5] K. Plakidas, D. Schall, and U. Zdun, "Evolution of the R software ecosystem: metrics, relationships, and their impact on qualities," *Journal of Systems and Software*, vol. 132, no. 10, pp. 119–146, 2017.

[6] M. Mora-Cantallops, S. Sánchez-Alonso, and E. García-Barriocanal, "A complex network analysis of the comprehensive R archive network (CRAN) package ecosystem," *Journal of Systems and Software*, vol. 170, Article ID 110744, 2020.

[7] M. Valiev, B. Vasilescu, and J. D. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, pp. 644–655, ACM, 2018.

[8] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[9] A. G. Tansley, "The use and abuse of vegetational concepts and terms," *Ecology*, vol. 16, no. 3, pp. 284–307, 1935.

[10] J. te Molder, B. van Lier, and S. Jansen, "Clopenness of systems: the interwoven nature of ecosystems," in *Proceedings of the Third International Workshop on Software Ecosystems*, pp. 52–64, CEUR-WS.org, Brussels, Belgium, 2011.

[11] J. Bosch and P. Bosch-Sijtsema, "From integration to composition: on the impact of software product lines, global development and ecosystems," *Journal of Systems and Software*, vol. 83, no. 1, pp. 67–76, 2010.

[12] M. Lungu, "Towards reverse engineering software ecosystems," in *2008 IEEE International Conference on Software Maintenance*, pp. 428–431, IEEE, Beijing, China, 2008.

[13] K. Manikas, "Revisiting software ecosystems research: a longitudinal literature study," *Journal of Systems and Software*, vol. 117, pp. 84–103, 2016.

[14] J. Choi, B. Ferwerda, H. Jungpil, J. Kim, and J. Y. Moon, "Impact of social features implemented in open collaboration platforms on volunteer self-organization: case study of open source software development," in *Proceedings of the 9th International Symposium on Open Collaboration*, pp. 1-2, ACM, 2013.

[15] A. E. Akgün, "Team wisdom in software development projects and its impact on project performance," *International Journal of Information Management*, vol. 50, pp. 228–243, 2020.

[16] G. Korkmaz, C. Kelling, C. Robbins, and S. Keller, "Modeling the impact of Python and R packages using dependency and contributor networks," *Social Network Analysis and Mining*, vol. 10, no. 1, pp. 1–12, 2020.

[17] E. Trainer, Q. Stephen, C. de Souza, and D. Redmiles, "Bridging the gap between technical and social dependencies with ariadne," in *Proceedings of the 2005 OOPSLA workshop on eclipse technology eXchange*, pp. 26–30, ACM, 2005.

[18] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pp. 101–104, ACM, 2018.

[19] A. Mockus, "Amassing and indexing a large sample of version control systems: towards the census of public source code history," in *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR)*, pp. 11–20, IEEE, 2009.

[20] K. Blincoe, F. Harrison, and D. E. Damian, "Ecosystems in GitHub and a method for ecosystem identification using reference coupling," in *Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR)*, pp. 202–211, IEEE, Florence, Italy, 2015.

[21] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, 2007.
[22] C. Bogart, C. Kästner, D. James, J. Herbsleb, and F. Thung, "How to break an API: cost negotiation and community values in three software ecosystems," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 109–120, ACM, 2016.
[23] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 404–414, IEEE, Madrid, Spain, 2018.
[24] A. Decan, T. Mens, and 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)*, pp. 181–191, ACM, 2018.
[25] G. Fan, C. Wang, R. Wu, X. Xiao, Q. Shi, and C. Zhang, "Escaping dependency hell: finding build dependency errors with the unified dependency graph," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 463–474, ACM, 2020.
[26] Y. Tanabe, T. Aotani, and H. Masuhara, "A context-oriented programming approach to dependency hell," in *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition*, pp. 8–14, ACM, 2018.
[27] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub meets CRAN: an analysis of inter-repository package dependency problems," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 493–504, IEEE, 2016.
[28] N. Lertwittayatrai, R. G. Kula, S. Onoue et al., "Extracting insights from the topology of the JavaScript package ecosystem," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 298–307, IEEE, 2017.
[29] M. Claes, A. Decan, and T. Mens, "Inter-component dependency issues in software ecosystems," in *Software Technology*, pp. 35–56, John Wiley & Sons, Inc., 2018.
[30] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: the case of apache," in *IEEE International Conference on Software Maintenance*, pp. 280–289, IEEE, Eindhoven, Netherlands, 2013.
[31] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pp. 102–112, IEEE, Buenos Aires, Argentina, 2017.
[32] M. M. M. Syeed, K. M. Hansen, H. Imed, and K. Manikas, "Socio-technical congruence in the ruby ecosystem," in *Proceedings of The International Symposium on Open Collaboration*, pp. 1–9, ACM, 2014.
[33] Y. Mijsters, A. Mustafa, I. Mihai, and S. Jansen, "On the nature of software sub-ecosystems and their health," in *Proceedings of the 1st International Workshop on Software Health*, pp. 25–32, IEEE, Gothenburg, Sweden, 2018.
[34] H. Midi, S. K. Sarkar, and S. Rana, "Collinearity diagnostics of binary logistic regression model," *Journal of Interdisciplinary Mathematics*, vol. 13, no. 3, pp. 253–267, 2010.
[35] J. Miles, "Tolerance and variance inflation factor," in *Wiley StatsRef: Statistics Reference Online*, Wiley, 2014.
[36] S. Menard, "Applied logistic regression analysis," 2002.
[37] H. Midi and A. Bagheri, "Robust multicollinearity diagnostic measure in collinear data set," in *Proceedings of the 4th International Conference on Applied Mathematics*, pp. 138–142, ACM, 2010.
[38] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *New Opportunities for Software Reuse*, vol. 10826 of *Lecture Notes in Computer Science*, pp. 95–110, Springer, Cham, 2018.