universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Analysis of the Ruby ecosystem health and taxonomy of gems"

verfasst von / submitted by

David Domonkos, Bc.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2016 / Vienna, 2016

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This paper was not previously presented to another examination board and has not been published.

Vienna, July 2016

_____

David Domonkos

Acknowledgement

I would like to thank my supervisor Uwe Zdun and my advisor Konstantinos Plakidas, who both assisted me during the writing of this thesis and whose detailed feedback brought my critical thinking to another level. I would like to thank the University of Vienna for giving me the opportunity to live and study in the beautiful country of Austria. These years were both wonderful and challenging. I would like to thank my family and my friends who at times I might take for granted, but without whom I would not be where I am. Thank you for believing in me.

Abstract

This thesis studies how the software ecosystem of the Ruby programming language evolved, in what state it is, and what its future might look like. It applies measures researched in a paper by Jansen on the operationalization of measuring ecosystem health and leans on quantitative data collected from RubyGems.org and GitHub. Statistical overview of the data is provided and analyzed.

Secondary objective of the study is the discovery of a basic gem taxonomy inferable from the quantitative data. Classification methods for this taxonomy are proposed and examples of the most prominent representatives are given. Finally, the paper concludes with a demonstration how such ecosystem-wide data can be utilized to meet project-specific decisions.

Zusammenfassung

Diese Masterarbeit erforscht, wie das Software-Ökosystem um die Programmiersprache Ruby sich entwickelt hat, in welchem Zustand es sich befindet und was seine Aussichten für die Zukunft sein könnten. Es werden Metriken angewandt, die im Paper von Jansen zur Operationalisierung des Zustands eines Software-Ökosystems („ecosystem health“) untersucht wurden. Bei den Analysen werden quantitativen Daten aus den Portalen RubyGems.org und GitHub verwendet. Statistische Übersicht der Daten ist ebenfalls beinhaltet.

Nebenbei versucht diese Arbeit eine Taxonomie zu entdecken, welche aus den quantitativen Daten stammen könnte. Methoden zur Klassifizierung werden entworfen und die wichtigsten Beispiele von jener Klassifizierung werden genannt. Am Schluss dieses Papers wird untersucht, wie die verfügbare Daten bei individuellen Projekten ausgenutzt werden können und Entscheidungen steuern können.

Table of contents

1 Introduction

Software ecosystems are an emerging phenomenon in software engineering and product management, aiding companies to outsource product development. [1] Many of the nowadays most familiar and successful software products take advantage of this model, where such externalization of software development brought significant cost reductions and enriched the base product to extents a single company could not hope to achieve alone. [2] As such, it can be considered as a very successful business model and more and more emphasis is put on its study in both, the commercial sector, as well as in the scientific community. [3]

Similar to natural ecosystems, software ecosystems include a community. [4] This community is centered on a software-intensive system and interactions with the system as well as between its members produce a considerable amount of data, especially when interactions are driven through some central place, such as an extension market; in practice, this is more often than not the case. In the information age, data is an increasingly valuable resource and for companies, it does not only provide them with a competitive advantage, but the knowledge drawn from it may not always be flattering to its products or the company itself. As a result, availability of data from proprietary ecosystems is usually very limited and research is difficult. On the other hand, ecosystems around open source products tend not to be owned by a private party and the access to data is much easier, making them especially suitable objects for study.

In this paper we studied how this data can be leveraged to the advantage of the platform owners as well as the individual members of the community and we used the ecosystem surrounding the programming language Ruby as an example. There had been several comparative studies that included this ecosystem as well as studies that focused on it entirely. The most relevant ones are perhaps by Kabbedijk J. and Jansen S., [5] and Syeed et al. [6] However, these papers either provide a broad overview of the ecosystem or concentrate only on a certain aspect. In our work, we focus on areas that were not covered or were covered only slightly. We start with the extraction and description of our dataset in Chapter 2, as well as with some basic definitions laid down in Chapter 3, supporting the remainder of the paper. A broad analysis of the Ruby ecosystem, its actors and artifacts is

provided in Chapter 4. A more in-depth study can be found in Chapter 5, where we analyzed the evolution of the ecosystem and tried to evaluate its health. Finally, we study how quantitative data, which we managed to extract, can be used to infer characteristics about individual extensions, the products that circulate the ecosystem, in Chapter 6 and Chapter 7.

1.1 Software ecosystems

Emergence of software ecosystems (often abbreviated as SECOs) is a relatively new phenomenon and thus a new field of study. As a result, a variety of definitions exist in the literature. Consider as an example the following definition from one of the first papers on software ecosystems [7]:

> *Traditionally, a software ecosystem refers to a collection of software products that have some given degree of symbiotic relationships.*
>
> *- Messerschmitt and Szyperski, 2005*

Alternative, a more detailed definition was provided in 2009 by Jansen et al. [8] They recognize the need to single out an underlying technological platform and a market. The businesses in this case should be understood more broadly (as in actors that have a common interest in being in the ecosystem), since, as we know, members of an open source community do not necessarily have to be companies.

> *We define a software ecosystem as a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently under-pinned by a common technological platform or market and operate through the exchange of information, resources and artifacts.*
>
> *- Jansen et al., 2009*

It seems that there is some natural understanding to what software ecosystems, yet a generally accepted definition was missing. Manikas and Hansen tried to address this problem and they synthesized one from a body of literature, including the definitions above, in 2012. [3] They found that most of them recognize three main elements: common software, businesses, and relationships. The combined definition they laid down reads as follows (shortened):

> *We define a software ecosystem as the interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services.*
>
> *- Manikas and Hansen, 2012*

Building on top of the paper by Jansen and Cusumano from 2013 [4], we will recognize five elements for the purposes of this study. This does not have to cover all existing software ecosystems, but applies to Ruby and many others.

- Platform. The main fundament of the ecosystem, in our case the Ruby programming language and its interpreters.
- Extensions. Third party software, the main object of the ecosystem. In this case reusable Ruby code, libraries, known as ***gems***.
- Community. Actors in the ecosystem, parties creating the extensions. These include individuals as well as teams, commercial parties as well as enthusiasts, developers as well as sponsors or any external influencers. In the case of open source ecosystems, the owners or developers of the platform could be perceived as part of the community as well.
- Consumers. The end-users of the extensions. Consumers differ from an ecosystem to ecosystem, in Android they are the general public. In Ruby, the consumers are a qualified workforce; other developers and parties that use the extensions for some end-product.
- Extension markets. Places and channels through which community distributes its extensions among themselves and to consumers. Jansen et al. recognize that extension markets can come in many forms. There can be a single market or a number of markets. The market can be limited to a simple list of extensions or be a more sophisticated web platform. In Ruby, the primary extension market is known as RubyGems.org.

1.2 The Ruby ecosystem

The history of the Ruby ecosystem begins with the release of the first Ruby interpreter in Dec 1995, known as Ruby MRI[1]. Initially, extensions (reusable Ruby code) were not distributed in the form of packages as it is today, nor was there a single extension market where extensions could be

[1] http://ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html

acquired. This changed when a package manager called RubyGems was released in 2003 (according to Wikipedia[2]) and introduced ***gems***, a special archive type for the packaging of Ruby extensions.

1.2.1 Ruby extension markets

According to Jansen and Cusumano [4], one of the defining characteristics of a software ecosystem is how extensions can be acquired.

> *A software ecosystem can have no extension market, a simple list of extensions, an actual extension market, a commercial extension market, and multiple extension markets.*
>
> *- Jansen and Cusumano*

As of Apr 15, 2016, Ruby had several extension markets. It is difficult to find data with which we could compare their sizes, but with a simple qualitative assessment we can safely proclaim that RubyGems.org has become the most dominant of them by far. Nevertheless, there at least two alternatives with a decent market share. (1) Native operating system package managers and repositories of which the most notable are the RPM repositories used by the operating systems Red Hat Enterprise Linux, Fedora, or CentOS. Here, gems are distinguished from other packages by the prefix ***rubygem***. (2) The preferred market for China is ruby.taobao.org, due to their connectivity problems.

Gems available in RPM repositories are usually taken over from the RubyGems.org platform, i.e. they are first released on RubyGems.org and later adopted by the native repositories. To understand why these native repositories have an importance to their users and why it is unlikely that their size and use will decline in the near future, we should look into the motivations, i.e. shortcomings of the RubyGems.org platform.

- Gems often depend one on another. For example, the gem ***rails*** requires another gem called ***active_record***. Both of these (as well as many others) are installed automatically, when the user asks RubyGems to install ***rails***. However, RubyGems is unable to track dependencies outside of the Ruby ecosystem. Consider the gem ***postgres***, a driver for the PostgreSQL database system. It requires both a compiler for the language C, as well as PostgreSQL

[2] https://en.wikipedia.org/wiki/RubyGems

development libraries. RubyGems cannot install these dependencies automatically and installation of the gem leads to compilation errors if the dependencies are not present, whereas RPM resolves and installs them automatically, substantially simplifying the whole process.

- While Ruby is an interpreted language, it is common for gems to contain parts written in C (or another compiled language) to provide interface adapters to the given gem and these need to be compiled upon installation. In the RPM environment, compilation is done beforehand, meaning the modules that are required only for the purpose of compilation do not have to be present on the machine where the gem is being installed. This is, out of security reasons, often a requirement, especially during the deployment phase of a project.
- Using their own package manager gives the developers of the operating system more control. For example, if there are security concerns associated with a particular gem version, its release can be postponed.

1.2.2 Extension markets and RubyGems.org

As of Feb 29, 2016, RubyGems.org is the most widely used distribution platform for the Ruby ecosystem. It was created in 2009 by Nick Quaranto under the name Gemcutter[3]; since then it has become a collective product of the community and was later integrated by the RubyGems package manager as its default distribution channel. Eventually, it was renamed to RubyGems.org to further emphasize its central role in the Ruby ecosystem.

It must be noted that RubyGems has existed since November 2003 and the language Ruby itself since 1995. Until RubyGems.org was created, there have been other extension markets, such as GitHub, SourceForge and, most importantly, RubyForge[4]. While the data does not exist anymore, according to Wikipedia, RubyForge hosted around 9,600 projects, which is just a fraction compared to the 118,917 gems available on RubyGems.org today.

This evolution shows the difference between a commercial ecosystem owned by a private party and an open source one. Within private ecosystems, the tendency is to have a stable platform and one monopolistic extension

[3] https://rubygems.org/pages/about

[4] http://www.infoq.com/news/2009/10/rubyforge-phased-out-rubygemsorg

market, since the private party has the power to adjust the platform accordingly. With open source ecosystems, the platform owners typically have no preferences over which extension market should be favored as they are rarely motivated financially. Thus, if better alternatives emerge, older and less suitable markets are abandoned, much like the dynamics of an open market in economics. Similar observation was also done by Jansen et al.

> *Market ownership tells the story. Typically, in case a private entity manages the platform the ownership of the market lies in the hands of that entity. As a consequence, when there are multiple extension markets the platform is typically managed by a community...*
>
> *- Jansen et al.*

1.3 Research questions

In the scope of this study, we wanted to clarify if it would be possible to answer selected topics, mostly with the support of quantitative data only. Here we summarize the Research Questions (RQs) we found particularly interesting and that we aimed to answer.

RQ1: *How has the Ruby ecosystem evolved and what is its current state? Is interest in Ruby increasing, decreasing or does it remain unchanged?* This is the most substantial question of the study which we address in Chapter 5 on ecosystem health.

RQ2: *Is the platform alone sufficient for the support of the wider ecosystem? What does the platform provide, what does it not?* The motivation for this question is whether the underpinning platform can continue to provide all fundamental functionality gems need or the functionality comes in the form of (optional) plugins. We try to answer this question in Section 4.5, where we study the complexity of the ecosystem.

RQ3: *Can we categorize gems into one or more taxonomies based on the quantitative data?* As of Jun 14, 2016, the Ruby extension market lacks any form of categorization which could provide a guideline for both end-users as well as researchers. Basic taxonomy could also pinpoint the roles of certain gems and trends present in the ecosystem. We tried to implement automated methods in Chapter 6 and partially in Chapter 7.

RQ4: *Is the ecosystem-wide data relevant for decision-making on a project level?* This is partially answered by the paper as whole, but Section 7.2 provides an insight into how the data can be used for version management.

RQ5*: How much of the popularity of Ruby can be attributed to Ruby on Rails?* Ruby on Rails is one of the best-known web application frameworks. Its apparent popularity has been so high that it would seem that the popularity of Ruby itself was caused by it. Trying to answer a causative relationship like this without an extensive analysis is difficult, nevertheless it was tempting to take advantage of the data we already had at our disposal and get a new perspective. We dedicated Section 4.3 to this question.

2 Data collection

Analyses contained in this paper rely mostly on quantitative data. To be able to work with it conveniently, we created local copies of data collected from the RubyGems.org platform, as well as from GitHub, since there is a prevailing preference among Ruby developers to use it for version control and issue tracking. This chapter briefly describes the specifics associated with the collection and preparation of data from these sources.

The datasets from RubyGems.org were collected at the beginning of March 2016, thus, the last entries are from Feb 29, 2016. For this reason, statements about the ecosystem often refer to the state it was in on this date. Similarly, time series showing certain statistics end on this day. Extraction of data from GitHub was done later, however entries after Feb 29, 2016 were masked from both, the snapshot statistics (like distributions) as well as from time series, in order to provide a unified look at the ecosystem, how it was at one single moment.

2.1 RubyGems.org

Traditionally, researchers would be forced to use web crawlers and public APIs of the platforms they study, especially if they were proprietary. This is different with open source platforms and RubyGems.org publishes all of its data in database dumps, which it updates every week. The data is split between two databases:

- PostgreSQL. A well-known relational database management system, especially popular among open source projects.
- Redis. This database system belongs to the NoSQL, or non-relational, family of systems. The data is stored on a key-value basis, much like the data construct known as ***hash*** or ***associative array***.

The data contained in the PostgreSQL dump has an entry to every gem and to each of its version that was ever present in RubyGems.org. The information includes ***the release date***, ***the description***, ***authors***, ***licenses*** and ***the file size*** of each gem version. On top of that, all dependencies among gems are recorded as well. Lastly, every gem can specify a few relevant links, such as the homepage or link to the source

code, and are also recorded in the dump. This proves particularly useful when extracting and matching with the data from GitHub.

Redis is used by RubyGems.org to store daily download statistics about each gem and each version, as it is much more effective for this particular task. The amount of data is also much higher and the dump we collected on Feb 29, 2016 exceeded 4 GB in size once extracted. The disadvantage of Redis is that the entire database must be loaded in the working memory and, out of practical reasons, we were not able to work with it this way. Instead, we used an external tool for data analysis available under the name ***rdbtools*** to convert the dump into a JSON file. However, a conventional parser would also have problems to work with such a large JSON file, so we had to implement our own solution. We took advantage of the fact that Redis is a simple key-value store. In the resulting file, all key-value pairs are the properties of the root JSON object (thus, there are no nested objects), where each property along with its value is represented by one line. That way, we were able iterate over the file line-by-line when we were looking for a particular property.

2.2 GitHub

GitHub is a repository hosting service based on the Git version control system and is very popular among open source projects. Apart from being a free, cloud-based source code storage, its advantage is that it also has social features, such as an issue tracker. As of Mar 20, 2016, it hosted over 1 million Ruby projects. The majority of these projects are end products, in other words, they do not enter the ecosystem again as a reusable extension. While still being an interesting study object, in our paper we focused on the extensions and the community around them and so we selected those that could be directly linked to a corresponding gem. To connect them, we used the data contained in the table ***linksets*** which holds optional relevant links and it turned out that many gems link to their source code repository. Using regular expressions, we found that 87,577 gems had at least one link containing the string "github.com". With a total of 118,917 gems, this represented almost 74%. Unfortunately, not all of these were a link to a source code repository (e.g. there were occurrences where it linked to the author's profile page instead) and not all of the repositories still existed. In practice, we were able to mine only 68,755 repositories, lowering the percentage to approx. 58%. We should also note that we found repositories

that held the source code of a certain gem, but it was not recorded in the ***linksets*** table. It is unknown how many more such repositories we would be able to find, however we considered 69 thousand repositories to be representative enough. Still, it does not cover the entire ecosystem and certain tendencies would have to be accounted for.

2.2.1 Collection

GitHub is a very popular data source for researches dealing with source code, developer community, or similar. The platform is popular not only in the Ruby community but also in communities around other programming languages. GitHub also keeps track of much more information than RubyGems.org, so the amount of data is enormous, making it virtually impossible to locally download database dumps as it was in the case of RubyGems.org. As such, there have been several projects to get a handle on this amount of data, including the GHTorrent project[5], periodically collecting the data and providing an open online access to it, and even the GitHub Archive[6]. However, we are interested only in a small subset of all of this data and extracting it from these dumps would be very time-consuming. We found that the easiest way how to collect it would be through the official API of GitHub. GitHub is not very restrictive to crawlers and as long as they are using a valid GitHub account; it offers generous traffic limits (5,000 requests per hour).

Collecting data from GitHub is another complex topic that was covered by, among others, Wagstrom et al. [9] and Kalliamvakou et al. [10]. The result of the work of Wagstrom was a crawling program called ***gitminer***, which used the API of the platform. However, it was created in times when the API was not as convenient as it is today and for our purposes and we found that using the Ruby interface ***octokit***[7] provided by GitHub and implementing our own script was the easiest solution. We extracted historical data about the commit history for each of the repositories and aggregated data about the number of contributions. Additional interesting kind of data that could be extracted and analyzed in the future would be historical view on opened issues, pull requests, comments and evolution of code. However, some of these would lead to very high traffic and data requirements.

[5] http://ghtorrent.org
[6] https://www.githubarchive.org
[7] https://github.com/octokit/octokit.rb

2.2.2 Contributions

One kind of statistics we collected from GitHub repositories is the volume of contributions per user. This is aggregate data not only about the number of commits into the repository, but also the number of opened issues and pull requests.[8] It is based on the idea that users can contribute to development in other ways than just new code.

2.3 Post-processing

RubyGems.org uses PostgreSQL and Redis for data representation. However, we anticipated that we would be also interested in characteristics defined by transitive relationships between gems and actors. For such queries relational and key-value databases are ineffective and inappropriate. For this reason, and due to previous experiences, we decided to use Neo4j, a graph database management system, where the data from PostgreSQL was migrated to. All subsequent operations, including data collection from GitHub, were then performed on this instance. Figure 1 shows an overview of the final instance after all post-processing. The graphic does not display some relationships that were created for better performance of certain queries. The relationship `WEEKLY_COMMIT` contains commit statistics on a weekly basis.

You may notice that dependencies are reflected by the relationship `DEPENDS_ON` and that it points from a node of type `Version` to a node of type `Rubygem`. This setting allowed RubyGems.org to store dependencies for each version separately which becomes very useful once we start to study how certain characteristics of the ecosystem evolved in time. However, the edge does not point to a particular gem version, because multiple versions can satisfy one dependency. Take for instance a version restriction `>= 1.0`, which translates to *version 1.0 or above*.

[8] https://github.com/blog/1360-introducing-contributions

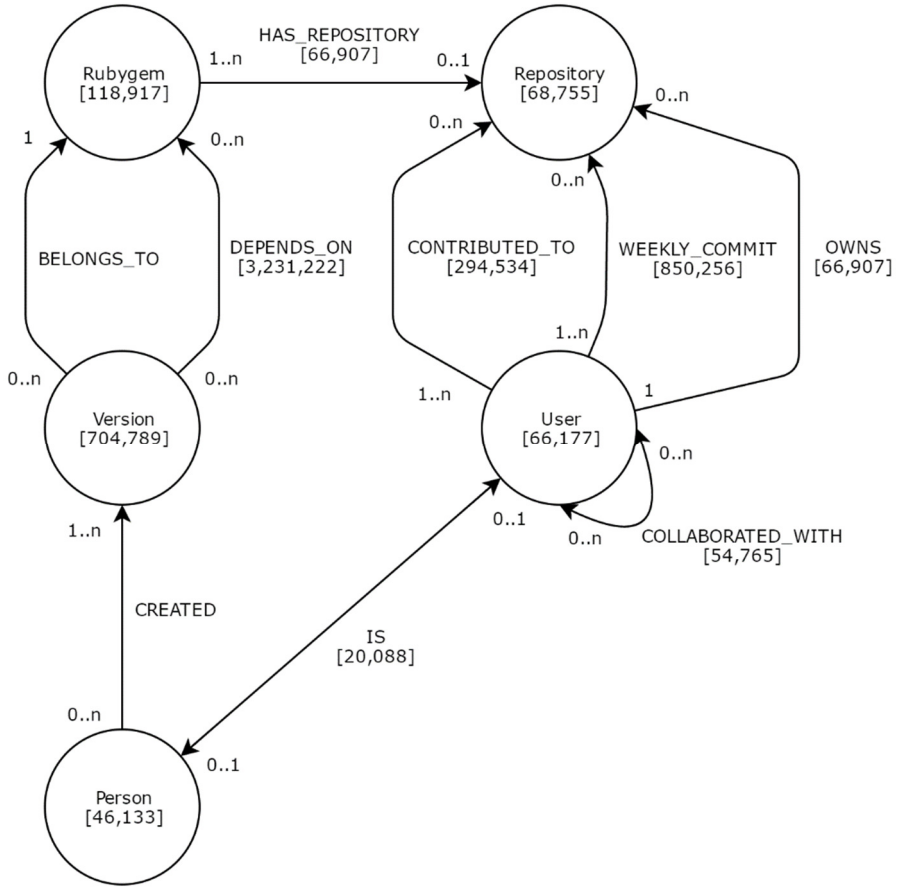

Figure 1. Overview of the final Neo4j instance, with volumes where relevant.

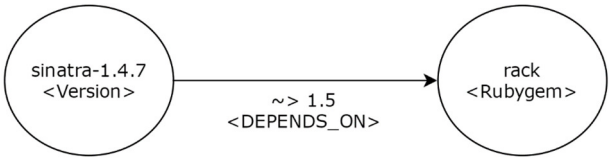

Figure 2. An example of one dependency.

2.3.1 Author extraction

Our Neo4j instance contains nodes of type `Person`. These were derived from the field `authors` in the RubyGems.org database dump. The problem with the `authors` field is that it is free text and is filled in by the person that uploads the gem. That way, authors are separated by a whole range of separators, including commas, quotes and spaces, ampersands and even keywords like ***and*** or ***with***. This representation is very inconvenient for data analysis, so individual names were extracted and `Person` nodes

created. Subsequent listing of people that authored the highest number of gems showed that there were certain exceptions that could not be accounted for, such as ***thoughtbot, inc.***, which does not represent two entities. An effort was made to manually clean up incorrect entries, but some may have slipped through.

2.3.2 Connecting accounts and authors

GitHub is much more sophisticated at tracking authorship than RubyGems.org, since each commit into the repository is recorded. We found it useful to connect the `Person` nodes with their respective GitHub accounts, which could be later used (e.g. for verification purposes), as the two data sources have different traits. This is no doubt a frequently performed operation and similar problem was faced in the work of Goeminne and Mens [11]. They found that the most effective method was the simplest one, that is, they approved only identical matches without case-sensitivity. However, we noticed that the names are sometimes misspelled or use diacritical signs differently, so in our matching method, certain deviation was allowed. To measure it, we used relative Levenshtein distance[9]. For the deviation we selected a threshold of 0.1 (values ranged from 0 to 1). To reduce false matches, we also considered only the contributors to repositories which belonged to gems the particular person authored. This way, we were able to connect 20,088 GitHub accounts to authors from the RubyGems.org database.

[9] https://en.wikipedia.org/wiki/Levenshtein_distance

3 Basic definitions

3.1 Importance

This paper relies heavily on the number of downloads and dependencies to estimate how "important" a gem is to the ecosystem. The very concept of "importance" in the context of a SECO bears some further explanation, however.

First of all, there is a problem of definition. One relatively straight-forward would be that an importance of an extension is directly proportional to how many people rely on it or use it. In our case, we would have to include not only developers that used the gem to build a product (be it an end-user product – such as a web application – or just another gem) but also all transitive users. To give an example, a gem that was used for a single heavily used website would rank higher on the importance scale than a gem used by several products with little to no users, in spite of having a lower number of first-order users.

Another way we could look at importance would be to consider how fundamental are the products that depend on the gem, not only quantitatively in the number of users but also qualitatively. But since we are working with large numbers, determining these qualities is not feasible and we must simplify our method. Even if settled for a simple enumeration of the user base (again, transitively), we still have the problem of collecting the relevant data, as we are unable to track users of products outside the scope of the ecosystem.

Additional complication is that the concept of importance can change with the interested party, i.e. it depends on the perspective. Some gems might be essential for a large number of products to work, which would make them important from the business perspective. On other hand, there could be other gems in the ecosystem that are not so heavily used in the commercial sector, but are used by enthusiast developers and in turn lead to innovations, making it more important from the perspective of ecosystem health and progress.

Due to limitations set by the available data, we restrict ourselves to the following three indicators:

- Dependencies between gems. The more other gems depend on it, the important it ranks. This can be either done simply (number of direct dependencies) or transitively. In our case we can also differentiate between runtime and development dependencies.
- Number of downloads. Download volumes should correlate with the total number of users.
- Activity in a gem's source code repository. High activity indicates that the gem is important to some party (the one that develops it at the very least).

Ideally, we would combine all approaches to estimate a gem's importance. One could argue that the number of downloads reflects the number of dependencies too, as it aggregates the downloads of the gem that depend on it – if a gem B has a dependency on gem A, downloading gem B automatically leads to a download of gem A. However, this does not hold entirely true for two reasons:

- Gem A is not downloaded if it is already installed. This also implies that if another gem C depends on A, downloading B and C leads to just one download of A, in spite of having two dependents.
- There are two kinds of dependencies, runtime and development. Development dependencies are in most use cases not downloaded, but that does not mean that it is of no importance to gem B.

Nevertheless, in most scenarios the number of downloads will actually reflect gem's importance quite well. Looking at the download history, we are also able to detect the evolution of its popularity in rising and declining weekly or monthly downloads. In theory, we should be able to measure a similar property by the number of new dependencies per week or month as well, however this could lack a statistical significance due to much smaller volumes.

3.2 Dependency graph

Complexity of the Ruby ecosystem, and the evolution thereof, is one of the characteristics this paper studies. To be able to examine it on a quantitative level, we consider complexity to be the degree to which projects are interconnected and reused and we find that with our dataset the best instrument for the measurement of such connectedness are the

dependencies between gems. Gems and their dependencies form a ***dependency graph***.

There exist many generally applicable metrics either describing graphs as whole (ecosystem level, in this case) or their individual nodes (project level). For the context of the complexity analysis, we focus on two metrics on the node level. (1) Dependency volume, or the total number of transitive dependencies. (2) Dependency depth, or the length of the longest dependency chain.

3.3 Dependency volume

Given a gem ***x***, outgoing dependency edges (including the transitive ones) form a dependency graph of their own, a subgraph of the complete dependency graph. For the convenience of future references, we will call the number of nodes in such subgraph the ***dependency volume***, or the total number of gems that are required (directly or transitively) by ***x***.

This dependency graph could be falsely seen as a dependency tree. Yet, strictly, it should not be classified as tree, as more paths than one between two nodes may exist. Nevertheless, in tree semantics, analogous metric would be called the ***tree size***.

3.3.1 Dataset post-processing

In order to be able to display the evolution of the dependency volume, the numbers had to be computed not on the level of gems, but on the level of their individual versions. On the first sight, this might seem simple, as dependencies in our database are recorded on the version level. However, remember that these edges point to gems, not to their particular versions (it is a ***Version*** to ***Rubygem*** relationship, see Figure 1). If this fact were to be ignored, the numbers would be biased by the present state of the database, not the state it was back in the time a particular version was released.

As already outlined, our interest will lie in the state of the database (i.e. ecosystem) at the moment a given version was released. To solve this problem, we created ***Version*** to ***Version*** relationships, based on the following criteria. (1) The target node had the highest version number that still satisfied the original dependency edge (***Version*** to ***Rubygem***)

restriction. (2) The release date of the target node was before or equal to the release date of the source node.

Consider the following scenario (dates are ordered numerically): gem $A_{1.0.0}$ was released on date D_4. $A_{1.0.0}$ depends on gem B with restriction `>= 2.0.0`. Gem B has versions $B_{2.0.0}$ released on D_1, $B_{2.1.0}$ released on D_2, $B_{1.5.3}$ on D_3 and $B_{2.2.0}$ on D_5. The correct ***Version*** to ***Version*** dependency edge would be in this case from $A_{1.0.0}$ to $B_{2.1.0}$, because it, both, satisfies the condition and is the highest at the moment of release of $A_{1.0.0}$.

3.4 Dependency depth

Given the dependency graph from Section 3.2, another measurable property would be the length of the longest path, given no cycles. Semantically, this path could be viewed as the longest dependency chain of gems building on top of each other. While such metric is not very conclusive for a single gem or for comparison of gems, the evolution of such number on the ecosystem level could provide an insight into how its complexity changed. If chains tend to grow in time, it could indicate that there is a tendency of gems to provide an increasingly complex functionality. Of course, this is a very rough metric and should only be used in combination with other indicators. We will reference to the metric as ***dependency depth***. In tree semantics, it is analogous to the ***maximum height*** (a.k.a. depth) of the tree.

Formally, we define it as follows:

$$\begin{aligned} Dep(x) &= 0, && iff\ |D| = 0 \\ Dep(x) &= \max\big(Dep(d_1), Dep(d_2), \dots, Dep(d_n)\big) + 1, && else \end{aligned}$$

where $\boldsymbol{D}$ is the set of dependencies of $\boldsymbol{x}$ and $\boldsymbol{d_n}$ are its individual members.

4 Ecosystem overview

This chapter provides a general overview of the ecosystem, which is helpful for understanding the rest of the paper. Shown are extremes, distributions and evolution of certain defining characteristics and statistics. Given the popularity of Ruby on Rails, we also looked into how much of the ecosystem is centered on this framework.

4.1 Platform

Ruby is a dynamic, general-purpose programming language, with its origins dating back to 1995. Until 2012, there was no formal specification of the language and it relied on the interpreter Ruby MRI, originally developed by the creator of the language Yukihiro Matsumoto, to be the reference implementation. There had been an attempt initiated by Brian Shirai to create such specification in 2006 and was called RubySpec.[10] Nevertheless, the Ruby MRI did not follow it[11] and by 2014 the project was cancelled (later to be revived as a test suite under the name The Ruby Spec Suite). The first official specification was laid down in 2012 as an ISO standard ISO/IEC 30170:2012.[12]

The Ruby ecosystem is underpinned by the language specification as well as the interpreters that implement it. As of Jun 14, 2016, apart from Ruby MRI, there had been at least 6 other Ruby interpreters[13], with JRuby[14] and Rubinius[15] belonging to the most mainstream ones. In Chapter 5 we evaluate the health of the ecosystem and, certainly, one of the defining factors is whether the development of the platform continues or has stopped. As all three most popular interpreters use GitHub as their primary source control platform, on Figure 3 we show the commit history in their corresponding repositories. We can see that the number of commits in both

[10] https://github.com/ruby/spec
[11] http://rubinius.com/2014/12/31/matz-s-ruby-developers-don-t-use-rubyspec/
[12] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=59579
[13] http://www.rubymotion.com; http://opalrb.org; http://mruby.org; http://maglev.github.io
[14] https://github.com/jruby/jruby
[15] https://github.com/rubinius/rubinius

Ruby MRI and JRuby repositories had been on rise, showing a stable community and interest from the side of the developers of the platform.

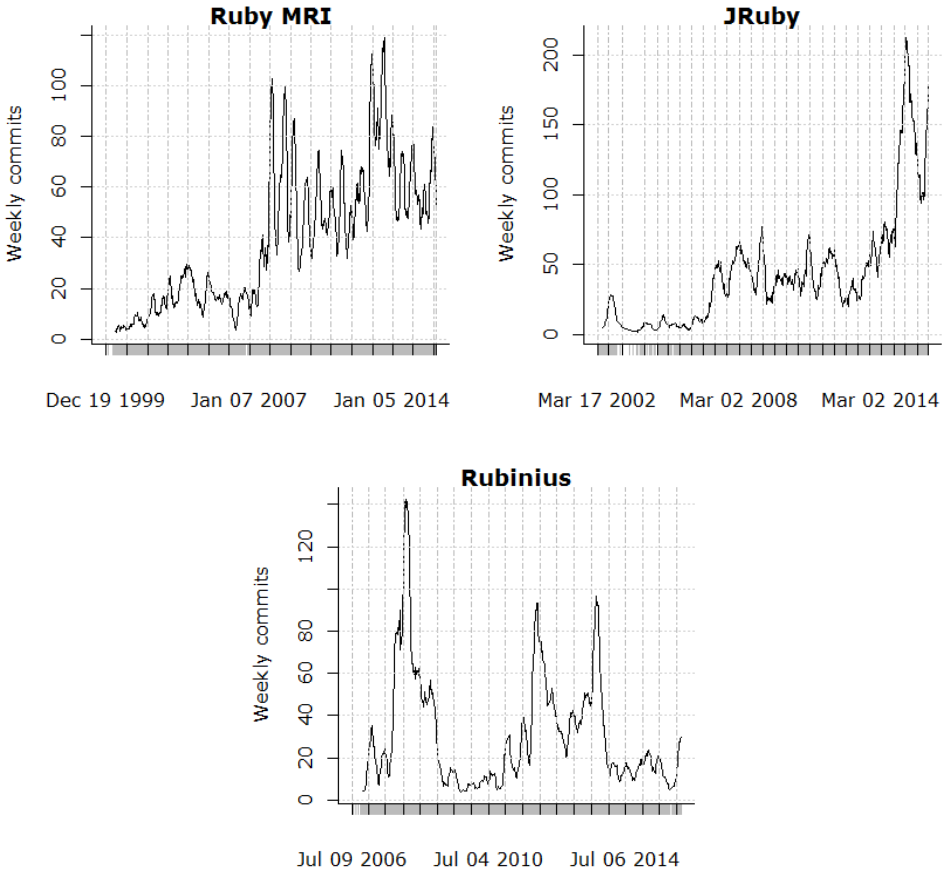

Figure 3. Weekly commit history of three most popular Ruby interpreters. Sliding time window of 10 weeks was applied.

4.2 Gems

The simplest and most straight-forward metric for measuring the size of an ecosystem is the total number of available extensions. As of Feb 29, 2016, there had been 118,917 gems in the Ruby ecosystem. This is significantly more than what was available in the years before the creation of the RubyGems.org platform in 2009. While it has become next to impossible to find out the real number of gems before 2009 (the preceding platforms either do not exist anymore or do not preserve such data), RubyGems.org has tried to keep track of the real creation date of each gem, not just the

date it was added to the database, storing the information in a column called `built_at`, for each gem version separately. On Figure 4 we plot the evolution of the size based on these values.

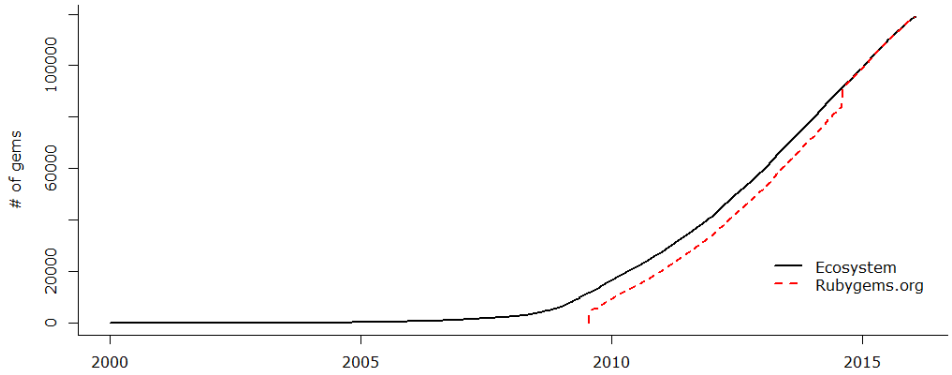

Figure 4. The number of gems available in the ecosystem, as it was approximated by the data from RubyGems.org. Black line represents state inferred from the `built_at` column in the PostgreSQL dump. Red dashed line shows how gems were added to the RubyGems.org database (the `created_at` column in the database dump). Notice the sudden jump at the end of year 2014. Apparently, many gems that existed elsewhere were added to the database at once on this date.

According to this data, the ecosystem seems to have been really small before RubyGems.org, containing only 5,000 gems at first (which were extracted from other platforms that were used for gem distribution at that time, according to past internet articles[16]), as by that time the language was already 14 years old and the RubyGems package manager approx. 6 years old. Simplifying the distribution of gems by a great deal, the platform lowered the entry barrier for new gems and it is likely that this contributed to the amount of gems being released. It is unlikely, however, that it would be the only reason behind this growth and we should be cautious when equating the size of the ecosystem to the number of gems. Nevertheless, it can be said that RubyGems.org has established its position within the Ruby ecosystem for some time now, and that consequently the more recent data drawn from it can be considered reliable indicators of wider trends in the ecosystem.

[16] http://www.rubyinside.com/gemcutter-is-the-new-official-default-rubygem-host-2659.html

As we have already built a database with a considerable amount of data that we also processed for added convenience, here we provide an overview of distributions and extremes based on the number of downloads and the number of dependents (gems having dependencies on a particular gem); these are especially interesting since we consider them as indicators of a gem's importance, as defined per Section 3.1.

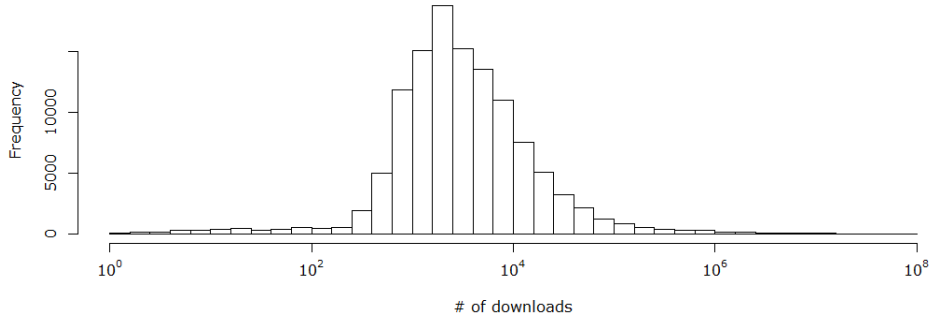

Figure 5. Distribution of downloads. Median is 2,732, maximum 94,993,299.

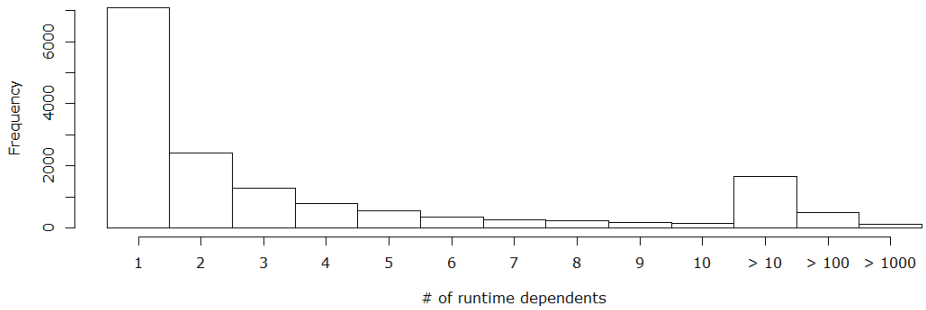

Figure 6. Distribution of the number of runtime dependents. Note that there are 103,354 gems with no dependents at all which were excluded from this graphic for readability reasons. The median is thus 0, the maximum 40,383.

Name	***# of downloads***	***# of runtime dependents***
rake	94,993,299	15,210
rack	90,554,414	20,844
multi_json	87,294,914	9,161
json	83,814,670	40,383
activesupport	82,056,830	24,773
mime-types	81,222,901	14,975
thor	78,684,210	16,048

Table 1. Seven of the highest ranking gems according to the total number of downloads.

Name	*# of runtime dependents*	*# of downloads*
json	40,383	83,814,670
method_source	26,166	21,735,536
i18n	26,081	76,516,680
concurrent-ruby	25,614	1,876,979
tzinfo	25,379	69,161,859
minitest	25,063	41,880,298
activesupport	24,773	82,056,830

Table 2. Seven of the highest ranking gems according to the number of transitive dependents.

4.3 Ruby on Rails centricity

Ruby on Rails (or simply Rails) is a web application framework with an initial release date of Oct 2004. It was developed by David Heinemeier Hansson as part of a project management tool called Basecamp, from which it was later extracted. Since then, the framework has gained so much popularity that the entire Ruby community seemed to have become centered on it. Google Trends shows that the search term ***Ruby on Rails*** is more than half as popular as ***Ruby*** itself (the programming language) and in year 2007 it was more than three quarters (see below).

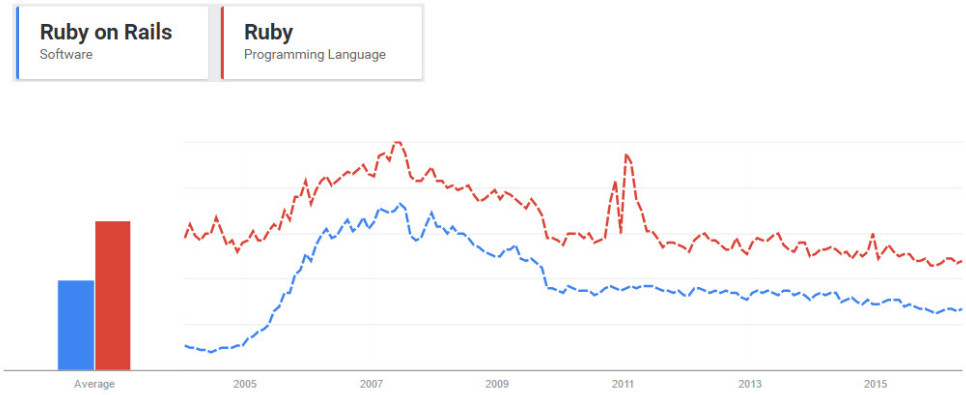

Figure 7. Interest in search terms ***Ruby on Rails*** and ***Ruby***, the programming language. Source: Google Trends

Our interest was in finding out how much of the ecosystem is really centered on the Rails framework, i.e. how many gems directly extend the framework. Since Rails is a complex framework, we assumed that if another gem specified it as a dependency, the gem could be viewed as an extension of the framework, i.e. it is unlikely that a gem would require an entire web

application framework to use it only for some secondary functionality. In that sense, it can be assumed that the existence of the gem depends entirely on the framework.

Nevertheless, we found that taking only the gem ***rails*** into consideration was not enough. While the gem represents the framework, in reality it does not contain any code and serves only as an installation gateway. The framework is split into several modules (such as ***actionmailer*** or ***activerecord***) which are specified as dependencies of gem ***rails***, therefore, installation of ***rails*** automatically installs all of its modules. This is important when we want to count the number of gems extending the framework, because they often do not extend the gem ***rails***, but one of its modules. Therefore, we considered not only dependents of ***rails***, but also dependents of all its modules. Also, we did not account only for gems that depend directly on one of these modules, but also gems that depend ***transitively***. Only runtime dependencies were considered. This method returned 24,774 gems.

Additionally, there are gems that do not specify Rails as one of its dependencies, yet the name or the description would imply that their existence could be directly attributed to Rails; for instance a gem whose name is ***capistrano-rails*** (has no direct or transitive dependency on Rails or any of its modules). As a complement of our previous method, we thus added the set of all gems that contained the string ***rails*** within their name, separated from the rest of the name, or contained the word ***rails*** within their description; as the word ***rails*** in any other meaning is very uncommon in the Ruby ecosystem, a certain relation to the framework was almost a given. This resulted in an additional 3,950 gems, when excluding the gems we have already found before.

With both methods combined, we ended up with 28,724 gems that we consider as extensions of the Rails framework. Considering that the total number of gems is 118,917, this makes up almost a quarter of the ecosystem. We ran this method for a similar, more light-weight web application framework called Sinatra and got 3,067 gems. As such, the ecosystem seems to be very Rails-centered.

4.4 Developer community

In this section we provide similar quantitative overviews as in Section 4.2, with focus on developers instead of gems. Note that data showing the number of commits or committers refers to GitHub repositories.

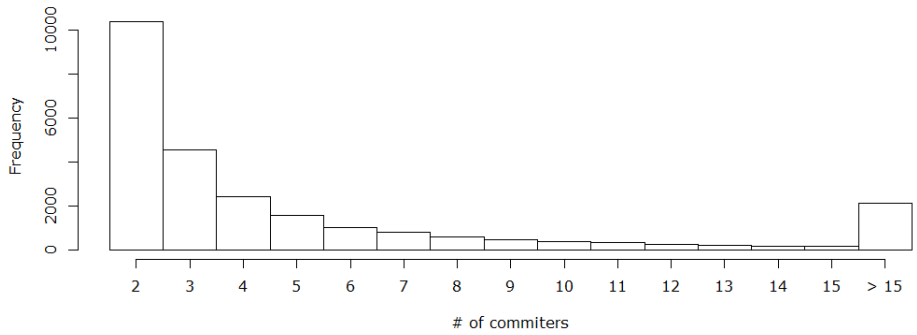

Figure 8. Distribution of the number of committers per GitHub project. Note that there were 39,746 repositories that had only one committer and were excluded from the graphic for better readability. The maximum number of committers per repository was 114.

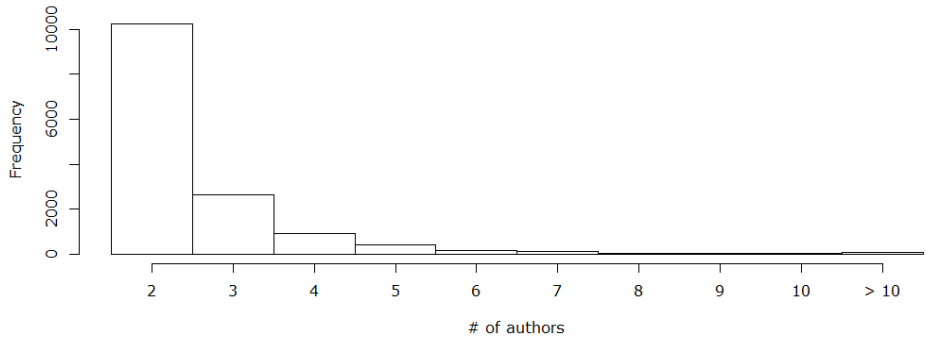

Figure 9. Distribution of the number of authors per gem. There were 102,313 gems that had only one author and were excluded from the graphic for better readability. The maximum was 49.

The number of authors and the number of committers did not correlate well. We ran Pearson's product-moment correlation test[17] and got a coefficient of 0.2523. To certain extent, this lack of correlation can be also recognized on Figure 10. It often happens that the repository has many committers, yet only few authors are specified. A natural explanation would suggest that users must contribute to the code non-trivially before they are

[17] https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient

added into the authors list, i.e. only the core of the team is specified as authors. Also, in some instances only the original authors are specified, as it is in the case of Rails, where the only author that is mentioned is David Heinemeier Hansson, even though there have been many others that contributed and committed to the project significantly. On the other hand, apparently there are also gems that specify more authors than there are committers. This could be caused either by the code base being migrated from another version control system or another platform. It could also happen that code was committed by a different person than the author or that the author committed it anonymously (in which case it is not recorded). In any case, the proportion of gems and repositories with such discrepancies make the validity and accuracy of either metric open to question.

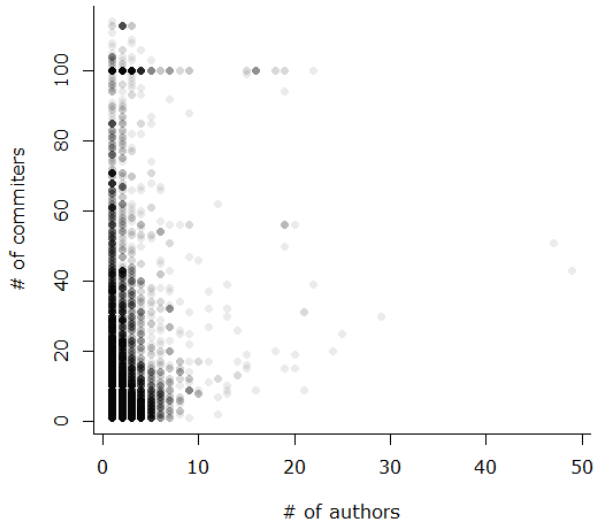

Figure 10. Scatterplot showing the density of gems with regard to the number of authors and the number of committers. Only gems that have a corresponding GitHub repository are displayed (65,128 in total).

Name	*# of co-authored gems*	*Σ of downloads*
james robertson	218	3,116,003
elastic	209	3,497,308
john nunemaker	191	42,548,794
eric hodel	180	183,866,423
michael grosser	165	20,962,901
robert a. heiler	157	1,184,000
brian shirai	146	5,640,261

Table 3. Top seven authors with the highest number of gems where they are specified as one of the authors, with respective cumulative downloads of these gems.

Name	*Σ of downloads*	*Gems*
david heinemeier hansson	545,368,267	rails, activerecord, activesupport
david chelimsky	189,590,282	rspec
jim weirich	186,017,686	rake, builder
eric hodel	183,866,423	rake, rdoc
aaron patterson	174,718,024	nokogiri, arel, sqlite3
erik michaels-ober	170,560,367	multi_json, multi_xml, oauth2
yehuda katz	162,744,855	thor, bundler

Table 4. Top seven authors that created (or collaborated on) gems with the highest number of cumulative downloads. Displayed are also some of their most well-known gems.

4.5 Complexity

As outlined in Section 3.2, we define ecosystem complexity as the degree to which individual projects are interconnected and reused. We also defined two metrics to measure this quality. The higher the total dependency volume of a project, the higher its reuse of functionality, and the higher its dependence on other members of the ecosystem, which can affect such characteristics as updates, error propagation, etc. Similar is the motivation behind considering the maximum dependency depth, whereas longer dependency chains could also be an indication of an increasing specialization or sophistication of a project, loosely translating to higher ecosystem complexity.

Every project contributes to the ecosystem differently; some are simple; others are more complex. On top of that, some projects may depend on other projects, yet in reality need only a part of them for an added side functionality. Such qualities can only be evaluated on a per-project basis and the metrics we used in this study cannot take them into account, since

all gems and all dependencies are deemed equal. Thus, a care should be taken when comparing particular projects on their grounds. Nevertheless, we find that when the number of projects that are evaluated is large enough, special cases are evened out and tendencies are revealed.

We restricted our measurements to runtime dependencies only, as we considered them more relevant for the analysis. For a general overview, Figure 11 shows the distribution of gems according to mentioned metrics as of Feb 29, 2016. For completeness, we also included the outdegree, i.e. the number of direct dependencies a project has.

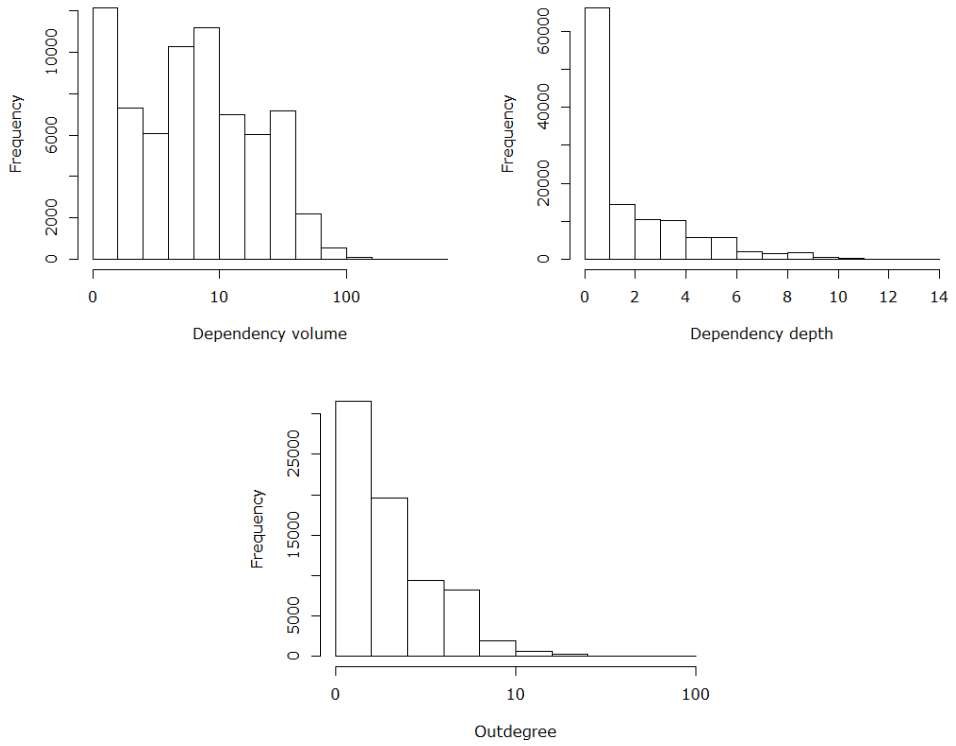

Figure 11. Distribution of gems according to their dependency volume, dependency depth, and outdegree, as of Feb 29, 2016.

On Figure 12 we plotted the means and medians of these numbers for new gem as well as new version releases. All indicators seem to increase in time.

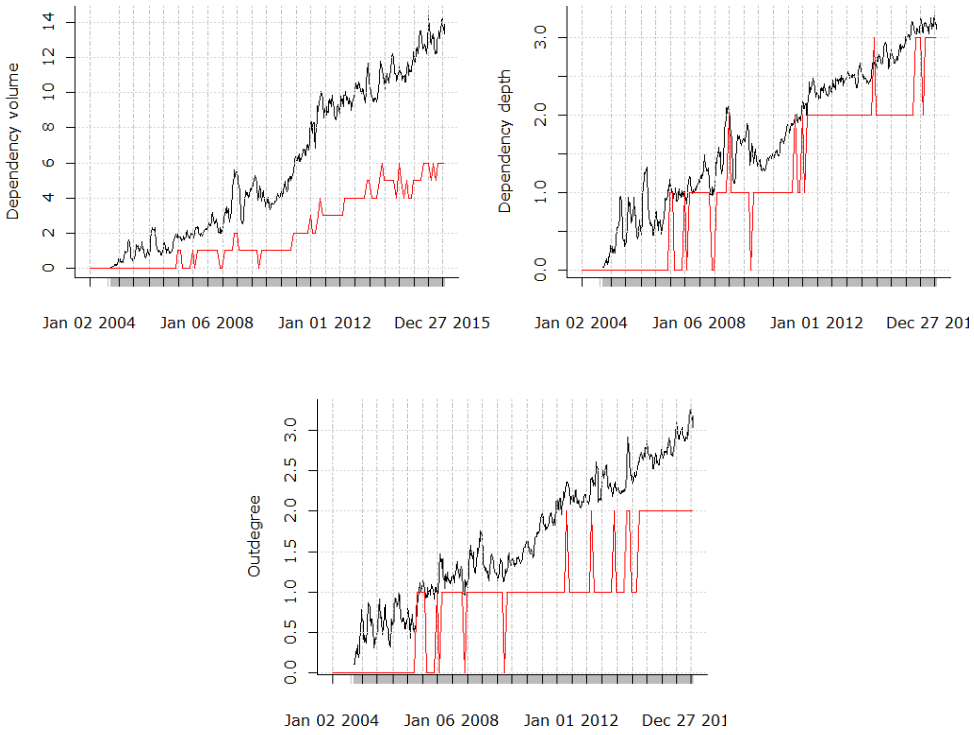

Figure 12. Dependency volumes, dependency depths and outdegrees of new gem or version releases in time. Weekly means are shown, with the application of a 4-week sliding time window for improved smoothness.

The rising medians would indicate that the proportion of projects that have no dependencies is in decline, i.e. more and more projects tend to build on top of other projects. To test this hypothesis, we plotted the proportions of gems with different dependency depths in time on Figure 13. To get a slightly different view to previous two figures, we did not include new version releases this time, only new gems were taken into consideration. Notice that gems with depth 0 (that is, with no explicit dependencies) were, indeed, on a decline, but remain to be significant at approx. 0.4. Proportions of depths 1 and 2 are also in recession, in favor of higher depths.

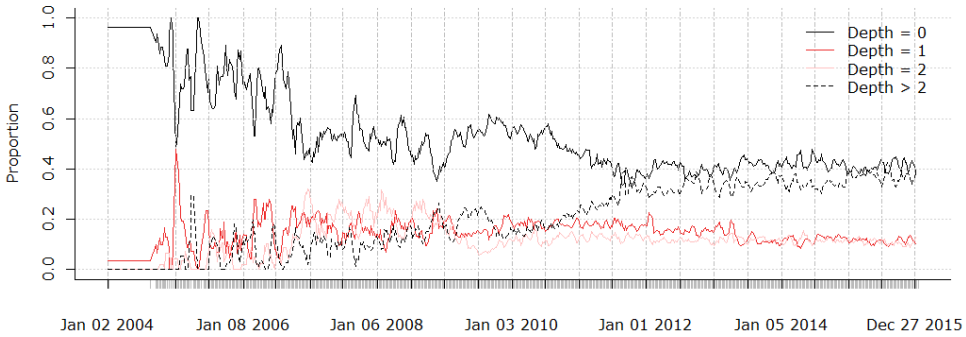

Figure 13. Proportions of platform distances of newly released gems (subsequent versions are excluded from this data). Sliding time window of one month was applied and was smoothed over weekly periods.

5 Ecosystem evolution and state

Let's consider a company trying to decide whether to invest in joining a particular software ecosystem, or an individual developer trying to decide whether it is worthwhile to invest time and effort into familiarizing himself with its technical aspects. Their decision would depend on their perceptions of the ecosystem's "health", that is, its current state and options for future success and growth. If an ecosystem is in evident decline, for instance, it might mean that the general interest in the platform is declining as well, making it more difficult for companies to find developers in the future, or for developers to find jobs in this sector. This makes the measuring of an ecosystem's past and present dynamics an important aspect of our study. Continually assessing "health" would also be relevant for the organization(s) behind the platform, as changes could indicate problems with the platform that need to be addressed.

5.1 Open Source Ecosystem Health Operationalization

In order to study health, the term has to be first defined and operationalized. Similar studies were already conducted on other systems, often on a project level instead of a whole ecosystem. In 2014, Jansen S. provided a sort of meta-study on software ecosystem health. [12] He defined the health of an ecosystem as its *longevity and propensity for growth* and created the framework Open Source Ecosystem Health Operationalization (OSEHO) to provide future guidelines on how to measure it. The framework consists of three pillars and depending on the granularity of data, methods are considered to be either on a network or project level. Network level in our case means using gems and developers as units, thus relying primarily on the data from the relational database of RubyGems.org, whereas the project level translates to finer commit and download statistics extracted from individual projects and then combined to give a general view. The three pillars of the OSEHO framework are as follows:

- Productivity. How productive is the community within the ecosystem? Increasing productivity means that the community becomes increasingly active and the ecosystem blossoms, and vice versa.

- Robustness. How well can the ecosystem cope with disturbances and how well does the community cooperate? Powerful entities in the community that are active in the development can form the backbone for the ecosystem.
- Niche creation. Analogous to this would be the business strategy phrase *get big, get niche or get out*. If there is little opportunity for a newcomer to specialize in one or the other direction, he might be overshadowed by the big players and eventually die out.

It is clear, which Jansen also recognizes, that it is usually not possible to apply all metrics on all ecosystems, either because the necessary data is not available or because the metric is not relevant/applicable. In our study we selected a subset of these metrics and focused on those that can be quantitatively measured. Metrics in the niche creation pillar were not relevant, because due to the lack of data, niche identification in the ecosystem on a large-scale basis was difficult. More information on this can be found in Chapter 6.

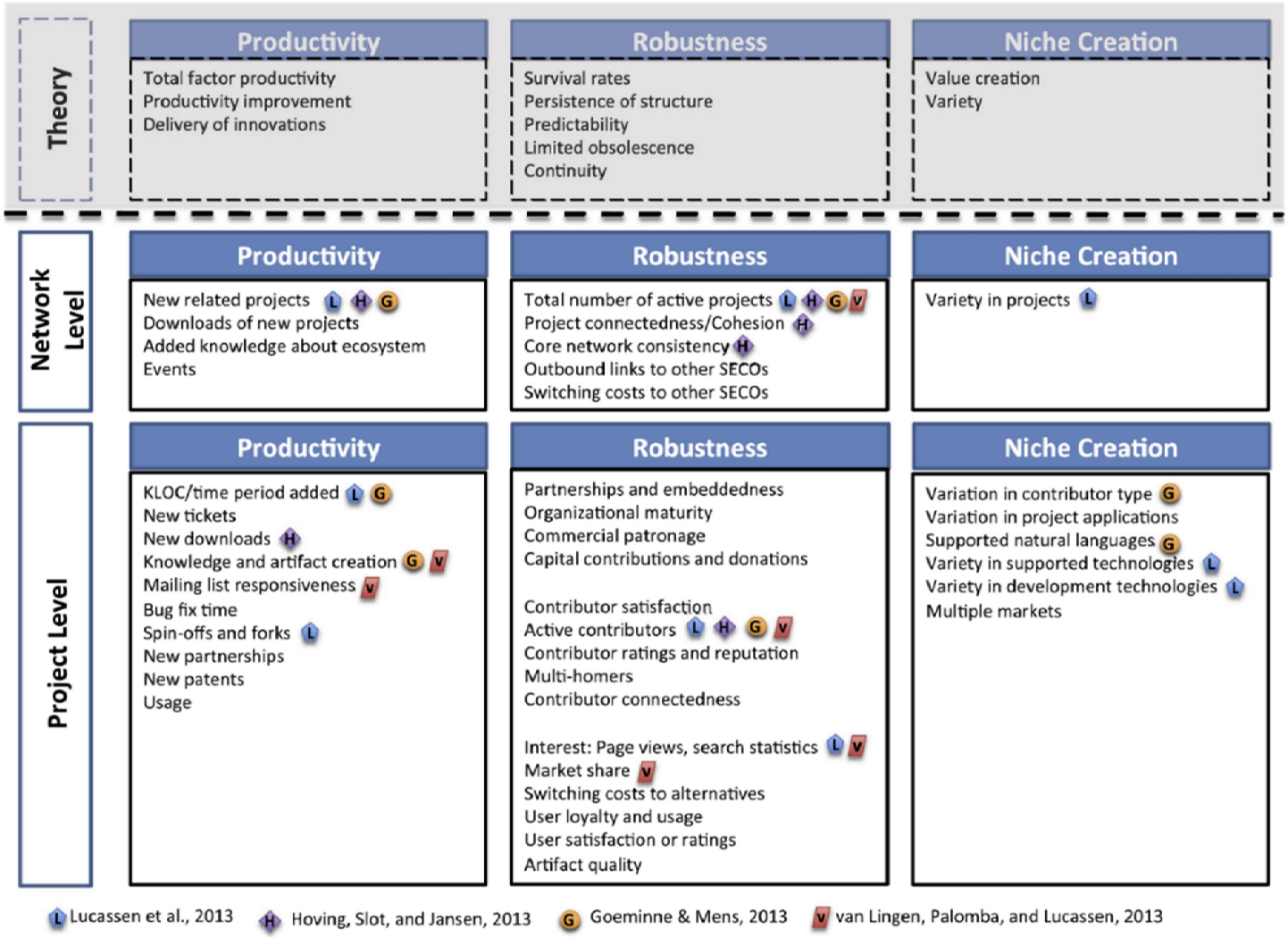

Table 5. An overview of the OSEHO framework and classification of metrics. Source: Jansen et al. [12]

5.2 Productivity

5.2.1 Network level

In Chapter 4 we have seen that the number of gems in the ecosystem keeps rising. However, the total size of an extension market is not a very good indicator of health because, first, we are interested in what the ***propensity for growth*** is, and second, gems are rarely removed from the market once they have been released. Thus, we will not be able to recognize changing trends on a graphic showing only the total numbers. Instead, we looked at how many new gems are added per a given time period and on Figure 14 we chose a three-month time window. Since in this case our data source is RubyGems.org, the accuracy might be lacking for periods that are too far in the past. However, we can clearly recognize a certain plateau being reached mid-2013, by when RubyGems.org was already well-established as the central extension market of the Ruby ecosystem. Since then, the growth of the ecosystem in terms of the number of new gems seems to have been linear, or even slightly slowing down.

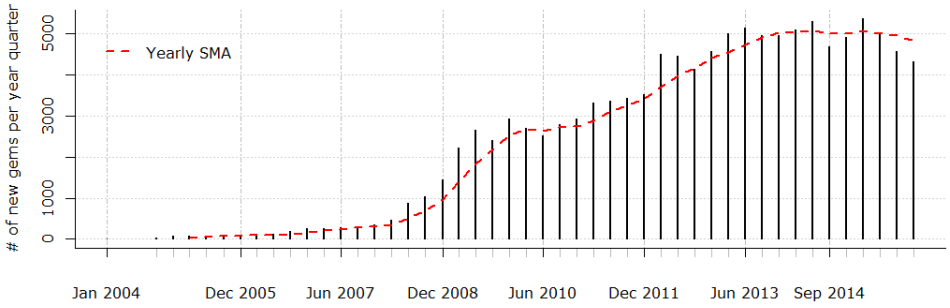

Figure 14. Number of new gems per year quarter. Red line is a simple moving average over a 1-year period.

5.2.2 Project level

A more sophisticated way of measuring productivity is to assess it not only on the level of complete products (gems), but on the level of individual additions to the source code (commits). Figure 15 shows that the activity on GitHub's relevant Ruby projects (the ones that have sooner or later become a gem) rose significantly since 2007, whereas it was negligible before that. There could be many interpretations for this, including some kind of bias introduced by the selection of observed GitHub repositories (described in Chapter 2), sudden gain in popularity of the GitHub platform, or even

the popularity boom caused by the release of Ruby on Rails. The part we are more interested in, however, are the second halves of the charts. Activity seems to have been rising in an almost linear fashion until the end of 2013, where it not only stopped but even started to decline at a rather drastic pace.

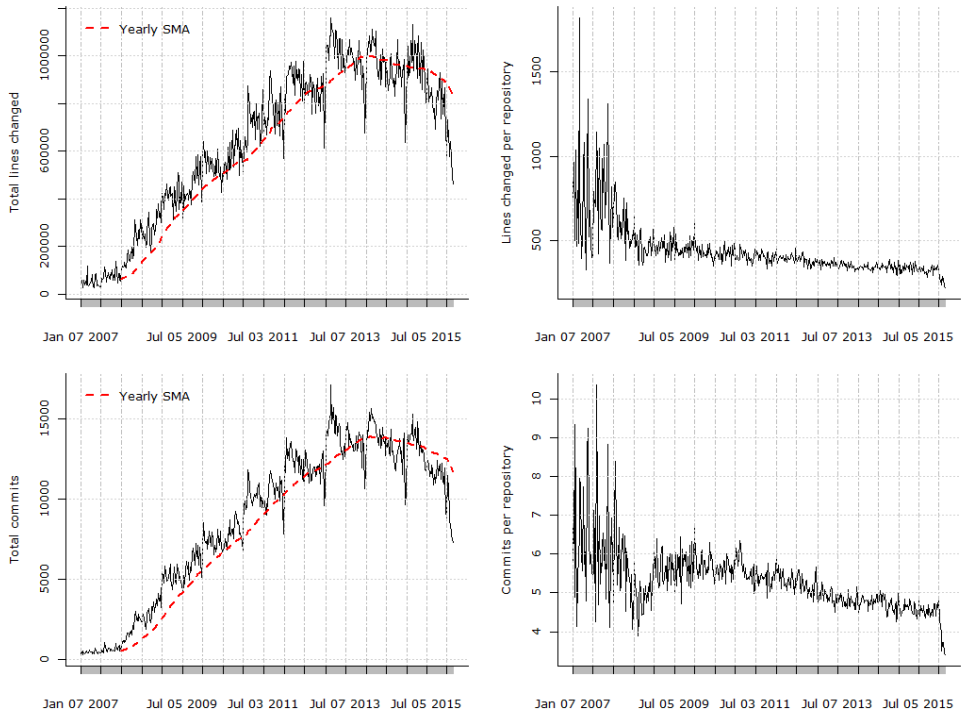

Figure 15. Number of lines changed and commits in a week of relevant GitHub repositories (GitHub repositories that can be directly linked to some corresponding gem). The graphics showing the lines changed were stripped of outliers, some commits contained data files with very large number of lines (e.g. a dictionary), and so any commit with more than 10,000 lines was left out.

This is obviously not a good sign for the ecosystem. Nevertheless, there is one effect caused by our data selection that should be taken into account and that might mitigate this picture. The selected GitHub repositories are only the ones that have a corresponding gem in the RubyGems.org database. However, if GitHub is used for source control in the pre-release stage of a gem's development and the gem was not yet released at the moment of our data extraction, development activity of such (future) gems would not be included in the data. We studied how much this effect might have influenced the declining trend in the most recent periods, and

extracted a table containing the initial release date (defined by the `created_at` column, since we are studying the date of addition to the RubyGems.org database) as well as the date of the first commit into the corresponding GitHub repository for each gem that had such repository. The table below shows the mean and median numbers of commits and days between these two dates. We can see that the numbers are miniscule; the observable decline is over a year long, thus its effect on the data shown in Figure 15 should be negligible.

	# of commits	*# of days*
Mean	16.36	21.08
Median	8.00	5.00

Table 6. Mean and median numbers of commits and days passed between the first commit into a GitHub repository and the first release of the corresponding gem. Only gems whose development started after Jan 1, 2014 were considered (for added relevance) but the numbers were similarly small for the entire history.

We also wanted to test whether the decrease was not caused by a declining preference of using GitHub for source control in the gem-developing community, so we took the number of new gems to which we could identify a corresponding repository and divided them by the number of new gems (with or without repository). To illustrate it on an example, in the week from Dec 13, 2015 to Dec 20, 2015, there were 315 new gem releases, out of which 191 referred to some GitHub repository, resulting in a proportion of 0.61. Such time based analysis is shown on Figure 16 where we can see that, recently, the proportions, in fact, increased. Combined with Figure 14 from the previous section, where the number of new gems shows only a small decrease, we can infer that the decline in activity strongly affects existing projects. Apparently, productivity in the ecosystem is in recession.

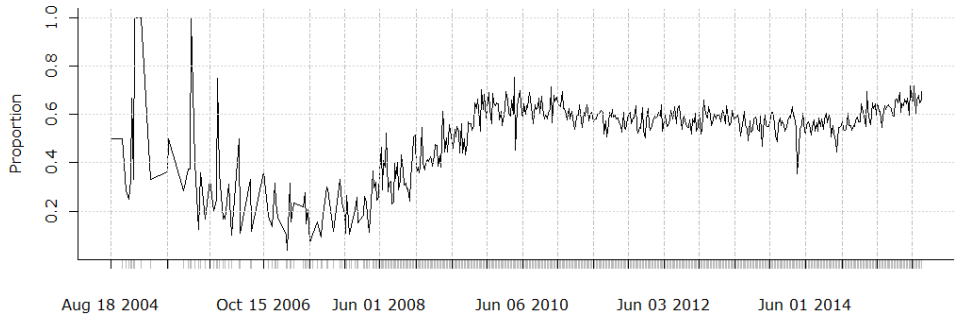

Figure 16. Proportion of new gems in a week with a corresponding GitHub repository to the total number of new gems in that particular week.

Hits of code. This section is a short excursion into an alternative form of effort measurement with an application on randomly selected set of Ruby gem projects. The Lines-of-Code metric (abbreviated as SLoC, or Simple Lines of Code) builds on the assumption that in time, software gets increasingly larger in the number of lines of code, as it is developed and time and effort is invested. Nevertheless, this does not always hold true, as refactoring often leads to code size reductions. In addition, it is a bad instrument for comparing two independent projects, as one problem can have many different solutions (thus different in size), or two developers, as the speed at which they can write code not only depends on their skills but also on their coding style or even the used programming language.

Hits-of-Code (HoC) is a metric proposed by Yegor Bugayenko on his blog[18] in 2014 which tries to address these problems. In particular, it addresses the first problem with refactoring where lines either disappear or are simply modified by recording every time a line is "touched" (modified in a commit), an effort that would not be recorded by the SLoC metric. As Bugayenko mentions, one of the advantages of HoC is that it always increments, just as effort; time that was already invested cannot be taken back. The HoC is not very different from the commit statistics kept by GitHub, only the data source is different and we tried to test if we would find similar trends as in the data we had directly from GitHub.

As part of his blogpost, Bugayenko also created a simple tool[19] that computes the HoC value of a given Git repository. Its only limitation was

[18] http://www.yegor256.com/2014/11/14/hits-of-code.html
[19] https://rubygems.org/gems/hoc

that it could calculate the value only of the present state, but with a small modification, we were able to get the historical values as well. We selected three random samples, each containing 100 gems, for which we extracted the historical HoC values and plotted the results on Figure 17. Notice the S-shaped curvature in our first sample; both graphs have a form very reminiscent of a normal distribution. This would be expected if measurements were taken from a single project and covered its entire lifetime, as the peak is usually experienced in the middle of development. However, we must remember that this shows an aggregation of 100 projects with different beginnings and ends, and so the resemblance is incidental, which was later then confirmed by two other sample sets. In addition, we already know that the rate at which new gems are released was increasing, so we had a higher probability of selecting gems that were younger rather than older and would expect that the monthly HoC would be therefore rising. Unfortunately, the variance was too large for a time-based analysis of the ecosystem with such a small sample and computational costs were too high to create statistics for the entire ecosystem. As a result, this approach was dropped in favor of statistics extracted from GitHub directly.

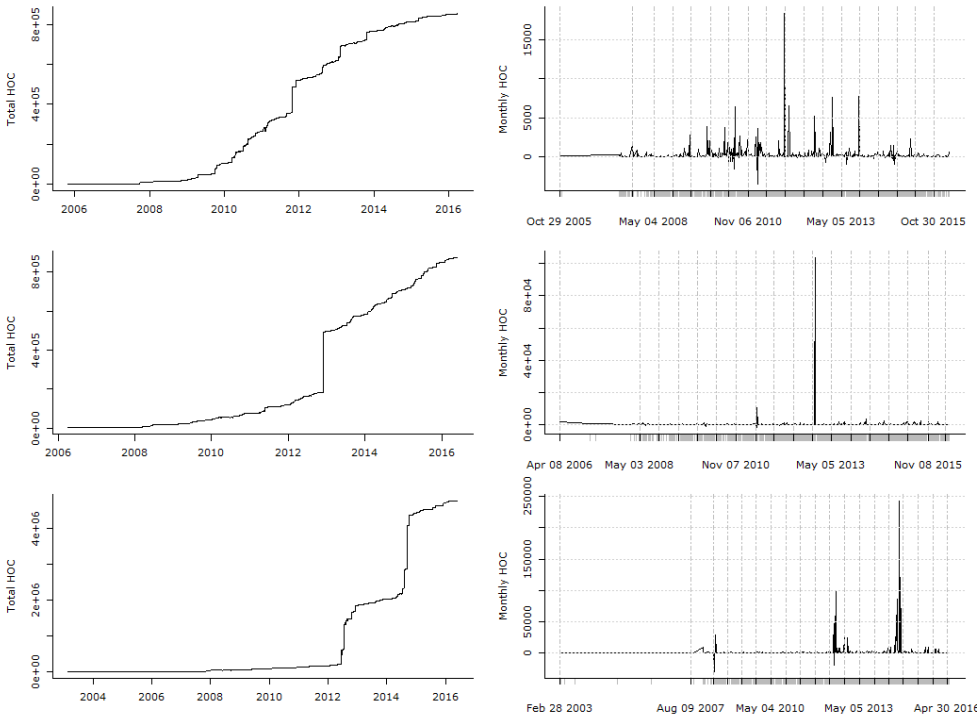

Figure 17. Cumulative HoC (left) and average HoC per month (right) on three random samples of 100 gems.

5.3 Robustness

5.3.1 Active projects

The number of active projects is one of the metrics for "robustness" of an ecosystem, i.e. the volume of activity is an indication of a liveliness of a community and it is understood that the larger the active part of community is, the better it handles changes. A very simple way of measuring activity on the network level is simply looking at the release dates of individual gem versions. Aggregation of these numbers is shown on Figure 18, notice the similarity to Figure 14. While in the productivity section we focused on the newcomer gems, here we address the activity reflected in existing gems as well. Both could be viewed as measures for productivity and there is a certain overlap between the two systems of measurement. But productivity is concerned mostly with absolute growth, whereas new versions do not directly increase the size of the ecosystem. Below we can see that there is no recession visible, as was in the case with new gems, however the rate of new version releases is not increasing either.

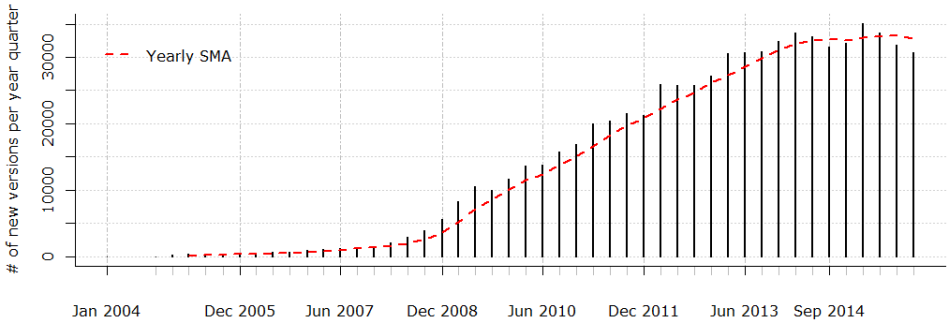

Figure 18. Number of new versions per year quarter. Red line is a simple moving average with a one-year period.

On the project level, the number of active projects can be measured more precisely. Just as in the productivity section, we can use the commit statistics from GitHub. We found that, as of Feb 29, 2016, there were 4,517 gem projects on GitHub which had at least one commit in the past month. We let an algorithm do this evaluation for each week since mid-2007 and created the time series shown on Figure 19. The number of projects that were classified this way as active strongly resembles other plots we have already seen, including a visible decline in the more recent data.

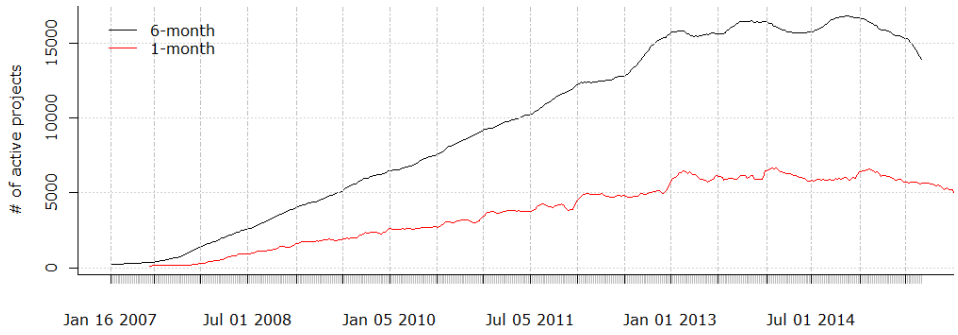

Figure 19. Number of active gem projects on GitHub in time. Active projects are considered projects that had at least one commit in the past 180 days (black) or 30 days (red).

5.3.2 Active contributors

In the paper of Crowston et al. on the success of free and open source software (FLOSS) [13], developer satisfaction and community size are recognized as possible indicators of project success. We did similar data processing as in the previous section and plotted the evolution of the active community size below. Overall, the form is very analogous to how active gem projects recently evolved, with a visible decrease in the past year.

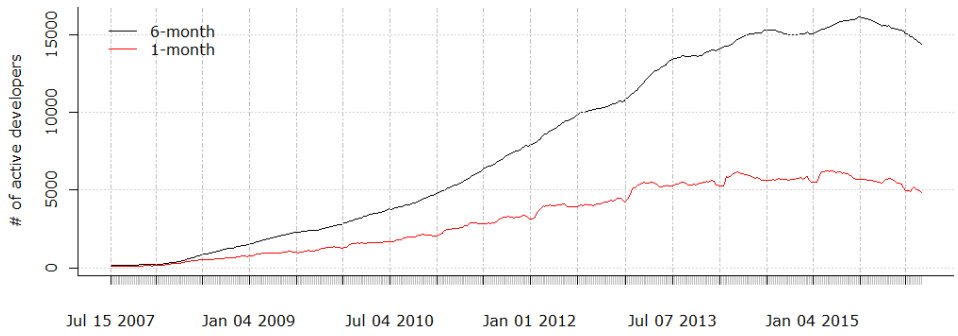

Figure 20. Number of active developers on GitHub in time. Active members are considered those that have committed at least once in the past 30 days (red) or the past 180 days (black).

We also looked at how much time active and inactive members (as of Feb 29, 2016) spent in the Ruby community by comparing their first relevant commit with their last. We observed that both are strongly skewed to the left, being made up mostly by people that stayed in the community less than one year. Especially the distribution of the inactive part is indicative

of developers temporarily joining the community, only to leave it soon again. We applied thresholds of activity/inactivity of 4 weeks and 4 months, and the differences in the resulting distributions are insignificant.

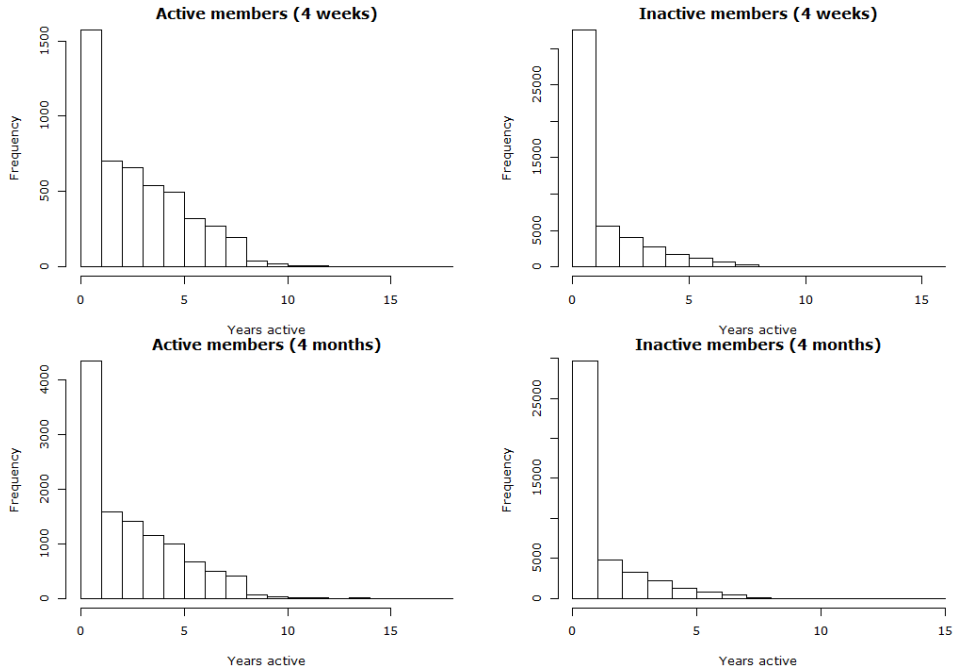

Figure 21. Distribution of active and inactive members based on the number of years they have (had) been active in the community. In the first row, members are classified as active if they have committed in the last 4 weeks. Selection of 4 months in the second row is not incidental, our goal was to skip the Christmas holidays, when the activity temporarily decreases in general. Number of years a developer was active was inferred from his first and last recorded commit.

5.3.3 Total download volumes

An ecosystem's popularity with its end-users is without doubt one of the factors influencing its overall health. When it does not change in time, it means that it does not attract new users and could be an indication of a certain level of maturity. Decreasing trend would be, on the other hand, a bad sign. The question is, how to measure popularity quantitatively? The only relevant kind of data we had at our disposal were the total download volumes, i.e. how large the use of each gem is. In this section, we analyze how it evolved in time to help us determine the overall health of the ecosystem. We must keep in mind, however, that download volumes are prone to invisible external influences. For example, if users tend to

download and install gems more often than before, it does not mean that the user base (and popularity) grows. The same applies if more and more bots crawl the ecosystem and generate these downloads. We are not able to tell how much it affects our observations, therefore, we should take the results with a grain of salt and should not jump to definitive conclusions based on a single metric and data type.

To extract the download history of the Ruby ecosystem, we used the Redis database dump of RubyGems.org. Results are plotted on Figure 22. We can clearly see that the volumes were continually increasing in a linear fashion. Apart from that, there are two interesting features we can observe: (1) the oscillations within the weekly period, where downloads dramatically drop on weekends, which could be viewed as an indication of the ecosystem having a strong commercial deployment, and (2) the short-term drops around Christmas holidays.

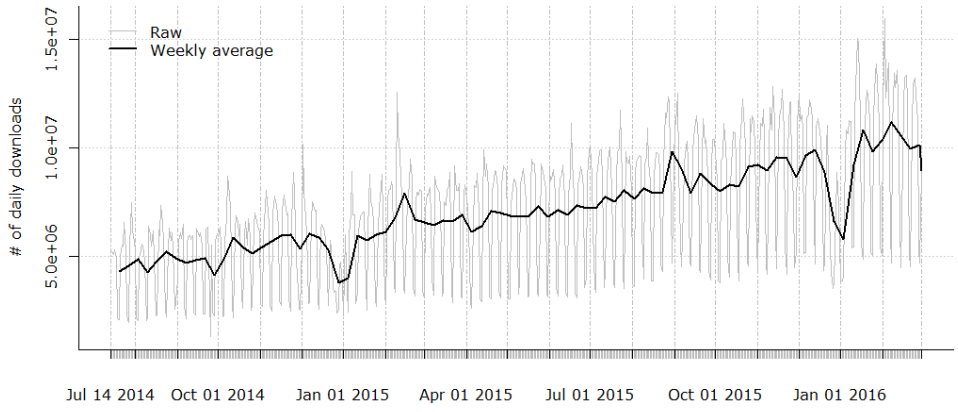

Figure 22. Total daily number of downloads from RubyGems.org.

The OSEHO framework [12] also recognizes download volumes as an indicator of health, however it focuses on new project downloads and sees it as an expression of a welcoming community. Large download volumes for new projects could mean that the community is open to new ideas and is looking for innovative solutions.

Since we have access to the download history, we will not look at the volumes only from the perspective of a current snapshot, but again, we will take an evolutionary view. To accomplish this, we must first define what initial download volumes mean, i.e. how long a project keeps the label "new", as for each duration we might get different results (we will refer to

this duration as ***D***). As we will soon see, daily downloads tend to change as gems age, so we should not include gems that are younger than the chosen ***D***. Since our download history has an approximate length of only 1.5 years, choosing a ***D*** that is too large will limit the amount of gems we can observe (this effect can be seen on Figure 25). Secondly, we must decide whether we are going to take the cumulative downloads of the gem over the entire period ***D*** or use weights (e.g. only consider the final week of the period). Different methods would lead to different results depending on the download history pattern. However, a simple sum makes the results considerably easier to understand and interpret. To illustrate, for ***D*** of one month the interpretation is: the number of downloads within the first month of a gem's existence.

As we can see on Figure 23 this kind of data is very susceptible to outliers. Most observations are located in the range of 200-400 downloads per month, but there are gems that had over 100 thousand downloads in the first month. The maximum is 627,951 downloads belonging to gem called ***mini_portile2*** first released on Nov 17, 2015. The reason why it got so many downloads so quickly was because the gem ***nokogiri*** (having over 67 million downloads in total) specified it as a dependency as of version 1.6.7 released on Nov 30, 2015. It is possible that the gem ***mini_portile2*** was split out of ***nokogiri*** to make the architecture of the project more modular. This dependency and release of new ***nokogiri*** version triggered the huge amount of transitive downloads. Similar explanation would be for ***fog-terremark*** (127,177 downloads in the first month) separated out of ***fog*** on Nov 14, 2014, or ***aws-sdk-v1*** (128,424 downloads) out of ***aws-sdk*** on Aug 26, 2014.

The representation on Figure 23 is not very suitable for the discovery of trends and we would like to see how the initial number of downloads of new projects evolved. Averaging over longer periods would smoothen up the curve but we would end up with artificially high peaks caused by the outliers. On Figure 24 we tried to apply two methods to hide their effects. First, instead of making an arithmetic mean over a given period, we took the median, as it is known for its tolerance against outliers, but we had to choose a longer period (one week) to have enough observations, thus, we were unable to apply the sliding time window method. In the second approach we removed the outliers altogether (the top and bottom 5% quantiles) and then took the mean over a given period. Both methods show a slightly rising trend.

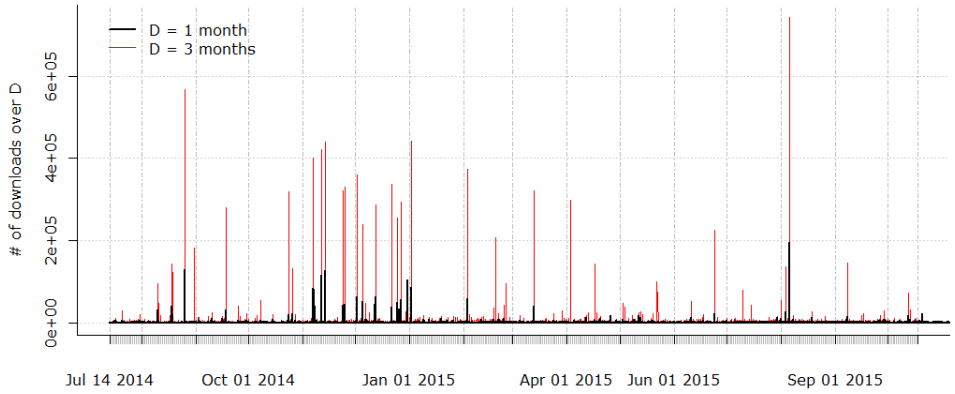

Figure 23. Number of downloads within the first month and first three months of a gem's existence. There are 27,745 observations on this plot. Notice the rare but outstanding outliers.

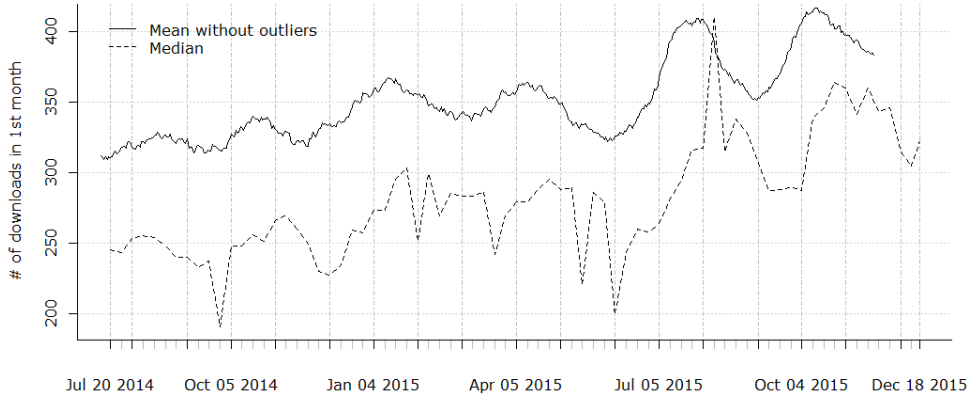

Figure 24. Number of downloads of new gems within their first month of existence. Solid line is a daily mean over 30-day long sliding window, dashed line is a weekly median.

Finally, we tried comparing different durations in Figure 25 to find out whether the rising trend can be recognized in all of them. While the trend is hard to find for the duration of 12 months due to its brevity, we noticed an interesting property. Apparently, the average gem tends to be downloaded mostly in its first month, with download rates dropping quickly thereafter. It is likely that this effect is connected to the fact that the Ruby ecosystem is an open-source ecosystem with no entry barriers, meaning that

anyone can publish their own gem and there is no quality standard. The majority of gems gain a rather small user base.

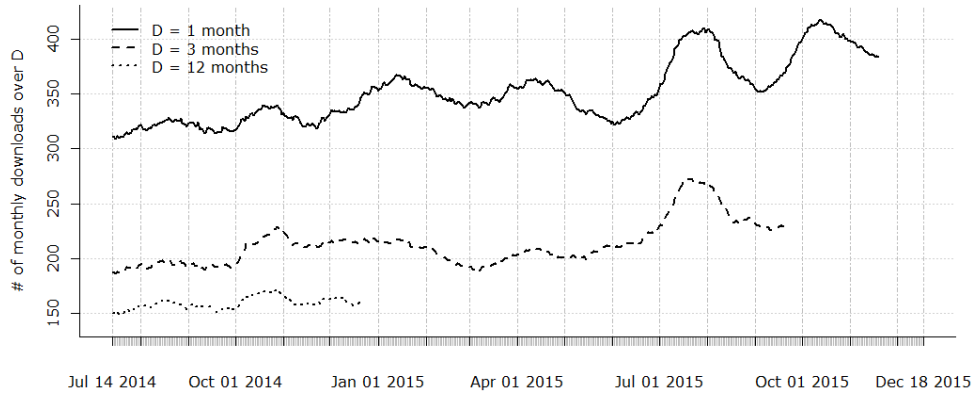

Figure 25. Number of monthly downloads over different durations. Displayed are the daily means without outliers with a 30-day long sliding window.

While the trend seems to be of an increasing nature, we should be careful when interpreting it as the community becoming more welcoming than before. We mentioned that there could be other underlying causes and external factors behind the increase in total number of downloads and the same factors could be active here. In general, 150 monthly downloads is not very much when we compare it with the download volumes of the most visible gems. All in all, the Ruby ecosystem seems to be numerically dominated by gems of low importance and, while the community does not seem to be closed to new ideas, the ecosystem does not provide an efficient way of discovering noteworthy additions.

5.3.4 Community cohesion

Based on the *assumption that a well-connected network is healthier,* [12] the connectivity between community members is an important factor in ensuring robustness. A community that is well-connected will not migrate from an ecosystem easily and will make the ecosystem more resistant to disruptive forces. On a side note, similar observation was done in the paper of Moody J. and White D., [14] defining the cohesion in social groups.

The question that remains is how to measure the connectedness of a community. Moody J. and White D. operationalized cohesion as the minimum number of group members that must be removed in order for the group (graph) to be divided into two subgraphs. However, its application is

somewhat difficult for a group with thousands of members and, secondarily, it does not take the strength of relationships into account. Another obstacle in measuring the connectedness is the actual definition of relationships between the community members. They could be either defined as collaborations, friendships, memberships in the same organization, etc. In such case, strength of relationship should be considered as well, as collaboration of otherwise unrelated developers on a side project should not be put into equality with a team of closely related members (possibly on a personal level) whose primary focus are products within the ecosystem. Finally, inference of relationships between developers based purely on the data from GitHub and RubyGems.org is somewhat arbitrary, if we do not perform any statistical validation of a given inference method (based on polls, for example), which however falls outside the scope of this work. With that in mind, we attempted to provide some quantitative views from the available data that could provide some support to the observations done in the previous sections.

First of all, we tried to get the distribution of GitHub repositories by the number of users that committed to it. According to this method, around 61% of all repositories had only one active developer in their entire history. Such number is very open to interpretation, especially when we lack any comparable figures, so we hoped to gain better informative value by plotting its development in the history of the ecosystem, which we depicted in Figure 26. We also included the numbers for 2, 3 and more than 3 committers in the repository and put them in relation to the total number of active repositories in a given year. Interestingly, the proportions hardly changed at all.

Figure 26. Proportion of GitHub gem repositories that were active in the particular year with regards to the number of active committers.

As mentioned before, it is very hard to infer any relationships between developers from our dataset, as most of the interaction can happen, in fact, outside the measurable ecosystem bounds. An attempt was made in which co-authorship was considered as a proof of collaboration. This could be either determined by the field `authors`, filled by the actual authors of the gem, or the activity in associated GitHub repositories. We found the `authors` field somewhat problematic, since it is in manual control of the developers. In particular, we observed two complications: (1) Some gems mention only few, original authors, however the commit activity in their GitHub repositories indicates that many more developers, in fact, collaborated. (2) There are gems that are copies of other gems, possibly with a small modification which was the reason why they were uploaded in the first place. The number of such gems is unknown, yet for these reasons we found the automatically recorded activity by the version management system to be more reliable, despite its own pitfalls, including projects where only one author (or a subset of authors) was entrusted with committing or projects that did not start on GitHub and were only later migrated there.

On these grounds, we aimed to establish relationships between developers that committed to the same repository. The problem with this approach was that a relationship would be created even if one user had 500 commits and the other one only 1. Such is the case with ***rake***, where the original author has had 590 commits and 94 other users fewer than 10 each. To somehow reflect the strength of the relationship, we filtered out such collaborations, where developers did not reach at least $1/10^{\text{th}}$ of the number of commits of the most active author. Naturally, such threshold could be selected completely arbitrary, so the absolute numbers remain without interpretation; only their evolution promised some value. In addition, in our particular application we included only collaborations where developers shared at least 30 commits into the same repository. Out of 66,177 extracted GitHub users, we ended up with 4,149 that had at least one relationship with another user which satisfied these conditions. To illustrate the sensitivity of our approach to selected thresholds, we recognized 19,732 relationships when we removed the 30-commit restriction.

Restricting to those 4,149 users with at least one collaboration edge, we found that the distribution of the number of relationships per user (the node's degrees) changed in time only insignificantly, as seen on Figure 27. Slightly different it is with "lone-wolf" users – that is, users who shared with any other user less than 30 commits or committed to a repository

insignificantly in comparison to the contribution of other users. According to Figure 28, their proportion seems have been rising, even though very slightly. Numbers from years 2007 and possibly 2008 are hardly representative enough, since the size of the community might have been too small.

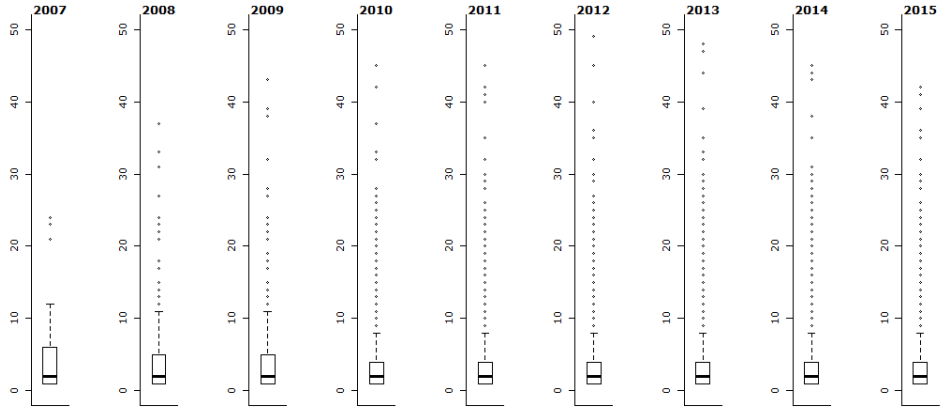

Figure 27. Distribution of users by the number of other users they collaborated with. Those that did not collaborate with anyone were excluded. Only users that were active (that is, made at least one commit) in the specific year were considered.

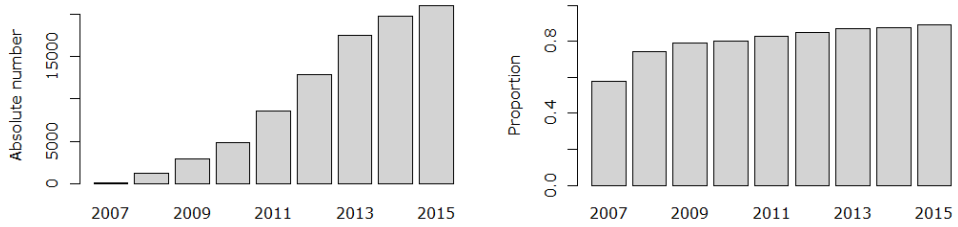

Figure 28. On left, number of users active in the particular year that did not collaborate with anyone. On right, their proportion to all users.

From Figure 27 we can see that the number of users a "median GitHub user" collaborates with is 2 (when we consider only the collaborating part of the community). In certain sense, this group could be considered as the core of the community. To complement these graphs, we plotted the distribution of users from this core by the number of years they were active on Figure 29. We compare years 2016 and 2014, as according to Figure 20 from Section 5.3.2, the size of the active part of the Ruby community should

have been similar. It seems that most users that were part of the core remained active.

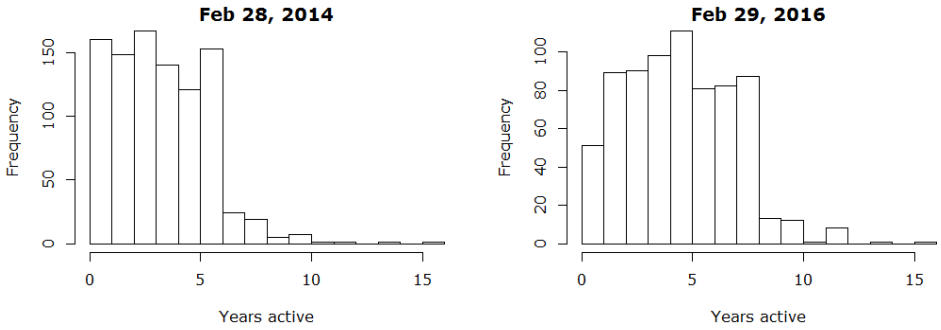

Figure 29. Distribution of developers that have collaborated with at least 2 other developers in the past 6 months, based on the number of years they had (have) been active in the Ruby community (the first recorded commit).

6 Classification methods

With over 118 thousand available gems, browsing the market can become easily overwhelming. Since RubyGems.org provides no categorization of its own, finding the proverbial "right tool for the job" may often depend on word-of-mouth recommendations in platforms like ***Stack Overflow***. Even though Ruby's target group is presumably a qualified work force rather than the general public, meaningful classification could still help new projects reach their potential users. In addition, such classification could be useful for structuring and/or studying an ecosystem's offerings.

The purpose of this chapter is to elaborate on methods by which quantitative, ecosystem-wide data could be used to create categories of gems. The ultimate aim would be to exploit this knowledge and provide categorization for the Ruby ecosystem, and allow comparisons with similar classifications in other software ecosystems. Relying mostly on the data from RubyGems.org and the dependency graph, we identified four categories that could help in selecting gems interesting for practical application and for further study.

Before we delved into an automatic classification, we attempted to find an existing classification already available on the internet. Since the gem specification[20] does not make it mandatory nor possible to specify the category to which a gem should belong, authors are left without an option of how to classify their gems.

That said, there is one web portal that has attempted to do this, called The Ruby Toolbox[21], which classified gems manually; such an approach for all of 118 thousand gems is unachievable, especially when gems can change their category during their lifetime. As a result, there were only 2,036 gems that were classified this way, as of Jun 13, 2016. In addition, the last announcement on the platform dates back to Jun 4, 2013, so whether the portal is still up-to-date remains questionable.

[20] http://guides.rubygems.org/specification-reference
[21] https://www.ruby-toolbox.com

6.1 Fundamental gems

We looked at which gems have the highest number of downloads and noticed that the top is dominated mostly by extensions like ***rake*** (spin-off of the ***make*** program used for Ruby applications), ***rack*** (unifies the API for HTTP calls) or ***i18n*** (an internationalization library). These provide very fundamental functionality that is repeatedly required by other gems, either directly through a direct dependency, or indirectly, through transitive dependencies.

Identification of such gems is worthwhile, because gems that are required by large amounts of other gems are worth studying, especially for the platform owners. They have the potential to show in what direction the platform should be further developed and whether such gems should be incorporated into the platform or not.

It is often the case, however, that the end-users are not interested in this group, but rather in gems that offer specific functionality solutions that use the "fundamental" gems. Hence when sorting gems by the number of downloads or the number of dependencies, it is often useful to filter them out in order to discover what the users are actually interested in.

If we could classify gems by function, one way we could define fundamental gems would be by selecting a set of categories created by such classification. Yet, as we mentioned previously, with our data such classification is not possible. A possible definition for a fundamental gem could be a gem that combines a high number of transitive reverse dependencies with a low number of transitive dependencies of its own, because such a gem is more likely to be "close to the platform" while at the same time sustaining a significant part of the ecosystem.

Using this definition, we applied the dependency volume and the number of transitive dependents to paint the distribution of gems between these two dimensions in Figure 30. The heat map shows that the majority of gems have low dependency volumes but also none or very few dependents. Noteworthy is the concentration on the right side of the graphic, where gems have over 90,000 transitive dependents. With 118,917 gems in total, this means that almost the entire ecosystem ultimately depends on them. Finally, there are also gems that do not rank on the right extremity but still manage to have large amounts of dependents, so while they do not belong to the most elementary group, they apparently either create their

own clusters, where they are the elementary gem, or define specialized branches of the ecosystem.

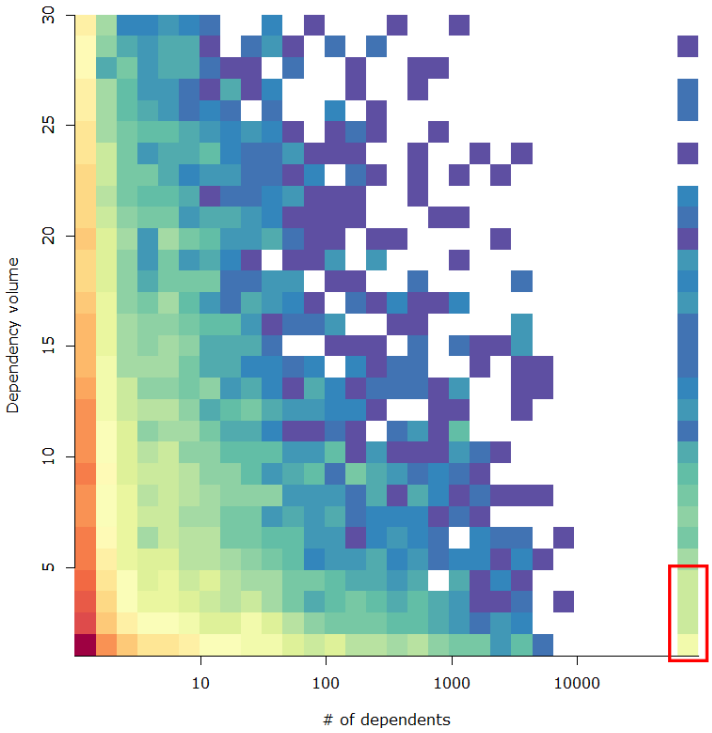

Figure 30. Distribution of gems in regards to their dependency volumes and the number of dependents. Heat colors are logarithm-transformed to get a better view on the cooling effect and reduce the outlyingness of the area around [0,0], which contains significantly more gems than other areas. Gems with dependency volume higher than 30 were omitted. Marked area shows the ones with the highest relevance.

Rephrasing the previous definition, we could characterize fundamental gems as gems that *require little or no other external software and a significant part of the ecosystem transitively depends on them.* This definition gives us room for adjustment of the levels of what we accept as a fundamental gem, depending on what we are interested in. We created a list of gems classified as fundamental, where we selected only the ones on which the majority of the ecosystem depends (margin of 60,000 dependents) and have at most 5 transitive dependencies (dependency volume), corresponding to the area on Figure 30 marked with red. The list contained 288 entries.

6.2 Closely related gems

As it is frequently mentioned throughout this paper, the Rails web framework is one of the best known and most important extensions in the Ruby ecosystem. At the same time, it belongs to one of the most complex elements and, thus, studying it is very convenient for the discovery of various possible characteristics and extremities of the ecosystem. And one such feature we notice is that the gem ***rails*** alone does not, in fact, contain any code, only documentation for the framework. The code is contained in other gems that were split out of the framework, such as ***activerecord***, ***activesupport*** or ***actionpack*** (nine in total, as of version 4.2.6). Modularity is an important software engineering pattern that contributes to, among others, reusability of code, and it seems that we can also find it on the level of entire gems, where they create clusters of modules. The goal of this section is to suggest a method for discovering gems that are closely related to each other. With the implemented algorithm we then performed a brief qualitative analysis of the results.

Before constructing an algorithm, it is important to recognize how such groups of related gems could be represented, as the Rails model does not necessarily present the only option. Different representations require different discovery techniques. In our paper we tried to account for three; naturally, other models are possible as well.

- Direct, centralized. There is one central node that users usually access (download). Other gems represent modules of the project and are direct or transitive dependencies of the core. This is the case with Rails.
- Direct, decentralized. There are more nodes that users access (download), with no clear distinction which one is the more important one. There is also a link (a dependency) between the nodes. Such could be a server and client gems of one project.
- Indirect, centralized. Nodes are accessed separately (similar to the previous case), however, one is recognized as the central one and there is no dependency link between them. The server/client example also applies here.

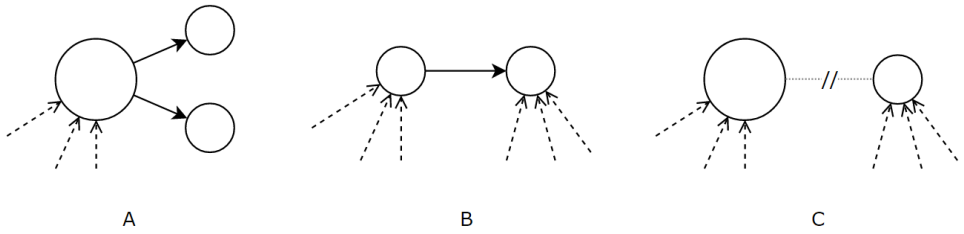

Figure 31. Different gem modularity models: direct, centralized (A); direct, decentralized (B); indirect, centralized (C). Dashed arrows represent user accesses, filled arrows are dependencies.

The models we selected above were subject to the data and techniques we are able to apply for their discovery. In essence, the data we used includes ***the dependency graph***, ***the number of downloads***, ***the author***, ***the name*** and ***the description*** of a gem. We could also include the data extracted from GitHub, but it is unlikely that we would gain any significant improvement without making the method excessively complicated. Using algorithms explained in the forthcoming sections, we were able to find 14,680 gems that could be associated with at least one other gem.

6.2.1 Direct, centralized

If we were interested in constructing an optimal algorithm, we would propose to consider every subset of available gems and identify clusters by maximizing a given similarity function. Considering the amount of gems present in the ecosystem, such algorithm would have an unreasonable time complexity. In addition, the point of this section is not to find an algorithm with the best hit rate/time cost ratio, but to explore possible identification techniques, see how well they perform, as well as assist in possible future qualitative analyses. For these reasons we designed a simple heuristic. We also introduced two assumptions about the direct, centralized modularity clusters. (1) Central nodes are located at the root of the dependency tree, i.e. the central node depends on modules, but no module depends on the central node. This is not always true, see for example the ***doctest-rspec*** module which depends on ***doctest-core***, the actual central node of its cluster. (2) There is a direct dependency from the central node to each of its modules. This should apply in the majority of cases.

While these assumptions decrease the precision, they greatly simplify the algorithm. Assuming there are no dependency cycles in the cluster, we can use the number of transitive dependents to sort all gems in an ascending order. By iterating over this list, we always have the certainty that if a gem does not belong to any cluster yet, it is either a central node or does not belong to any cluster at all.

Next step was the actual comparison of nodes. Here we took the following assumptions: (1) Gems that were built as part of the same project usually share authors. It is possible that the gems have different authors (e.g. when responsibilities in the project were assigned accordingly), but when there is no match in the authors section, it is hard to tell whether the gems have anything in common. (2) Modules either carry the name of the central node in their name, along with an additional string carrying information about the purpose of the module (e.g. ***rom*** and ***rom-mapper***), or the module mentions the name of the central node in the description (e.g. "Databases on Rails..." of ***activerecord***). The latter, however, is a rather weak indicator, as descriptions can be potentially longer texts and the name of the presumed central node might be a commonly used word.

In addition to this, we also considered the use of the number of downloads to help us avoid false positives. Due to dependencies, the modules should be with each user access downloaded as well, i.e. when a gem A depends on B, during the installation of A, both A and B are downloaded, provided they were not installed before. Unfortunately, the matter is not that simple. As already mentioned, it is possible that the module B is already installed on the user's system, which decreases the number of downloads. An opposite of this complication is that it is not exceptional for users to use only module B or for another, independent gem to require module B, increasing the download volumes. The matter is complicated even further when version releases of modules are not synchronized. Nevertheless, it is not our goal to come up with a flawless classifier, but to find an explorative method. Therefore, we will expect that if users directly access the central node and rarely the rest of the modules, the download volumes should be at least similar.

Using techniques mentioned above, we were able to implement an algorithm that yielded 394 groups of 1,233 gems in total. We then performed a short qualitative analysis of a sample of 10 groups and found that 2 were missing some of the nodes that should have been included and

the positivity of one group would be arguable. However, the method seems to have produced good results; if it was meant to be used an assistant for human-driven analyses, it would be advisable to make it less restrictive and increase both its recall and precision.

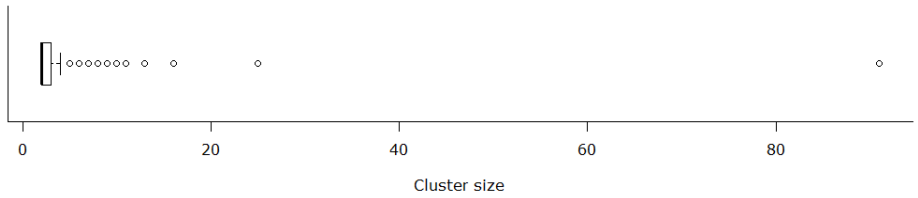

Figure 32. Distribution of sizes of identified clusters.

Gems within cluster	*Max. # of downloads*	*Cluster size*
rails, activerecord, actionmailer, ...	82,056,830	7
rspec, rspec-core, rspec-expectations, ...	39,449,750	4
coffee-script, coffee-script-source	35,313,946	2
aws-sdk, aws-sdk-resources	21,540,237	2
unicorn, raindrops, kgio	15,747,161	3
simplecov, simplecov-html	15,125,231	2
compass, compass-core, compass-import-once	14,412,160	3

Table 7. Highest ranking clusters with regard to the number of downloads. The number of downloads is that of the gem with the highest number within the cluster. The central nodes are always listed first.

6.2.2 Direct, decentralized

In contrast to the previous approach, there is no central node and gems can be regarded as equivalent in importance. We classify according to three criteria. (1) Authors must be similar, otherwise relevance cannot be established, since gems have no control over incoming dependencies. (2) Names must share a common substring of length at least k. We used the length of 3. Without this restriction we would end up with too many clusters that were developed by the same authors but serve a different purpose – in spite of one depending on the other. (3) The number of downloads of a dependent must not be much lower than that of on which it depends. This is to prevent formation of clusters that have seemingly nothing in common, i.e. gems have no power over their dependents and we found that some high-profile gems have dependents that share a portion of the authors, even though they have little to do with the original gem. Similar problem is described in the section that follows.

This method found 7,223 gems, putting them into 2,387 groups. We were also able to find groups that would have belonged to the first type, because modules shared only a substring of the central node's name. However, the results were less reliable, as we also traveled the dependency graph against the direction of the edges (remember, we have no assumption of the presence of any central node) and as mentioned, gems have no control over what other gems depend on them.

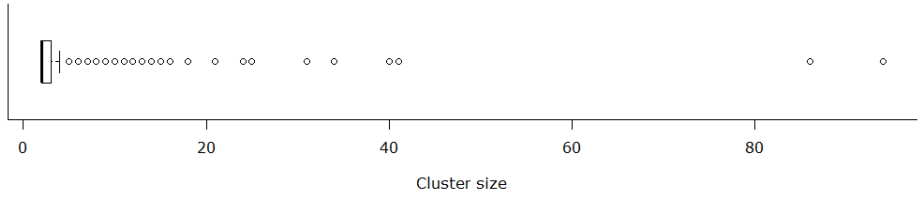

Figure 33. Distribution of sizes of identified clusters.

Gems within cluster	*Max. # of downloads*	*Cluster size*
sprockets, sprockets-rails	47,368,765	2
rspec-core, rspec, rspec-mocks, ...	39,449,750	5
net-ssh-gateway, net-ssh, net-ssh-multi	36,405,750	3
coffee-script, coffee-script-source	35,313,946	2
aws-sdk-resources, aws-sdk-core, aws-sdk	21,540,237	3
sqlite3, sqlite3-ruby	18,410,130	2
rubygems-bundler, bundler-unload	15,685,771	2

Table 8. Highest ranking clusters with regard to the number of downloads. The number of downloads is that of the gem with the highest number within the cluster.

6.2.3 Indirect, centralized

In this case the necessity of shared authors and parts of name was even stronger, since we lacked the information from the dependency graph and potential match could have been any other gem in the ecosystem. Accordingly, the match was also much greater, yielding 5,278 groups with 14,862 gems. Here we noticed a weak spot of having to rely on the user-provided author list. For many popular gems there are gems that could be considered their *reuploads*; they are virtually the same gem, perhaps containing a small modification, and were uploaded by someone else. The `authors` field was not changed and the gem was named similarly. Take for instance gems ***rack*** (a popular HTTP wrapper) with over 90 million downloads, first released on Mar 2, 2007 and still maintained, and ***qoobaa-***

rack, with its first and final version released on Jul 21, 2009 and 1,302 downloads in total. To exclude these cases, we decided that gems belonging to the same relation group must have at least similar download numbers (not less than $1/10^{th}$ and not more than 10 times). The adjusted method resulted only in 3,465 groups of 7,992 gems. As seen below, the majority of clusters were of size 2.

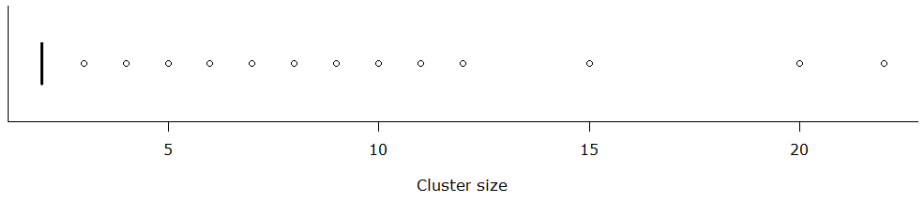

Figure 34. Boxplot. Distribution of sizes of identified clusters.

Gems within cluster	***Max. # of downloads***	***Cluster size***
json, json_pure	83,814,670	2
rspec, rspec-rails	34,228,525	2
aws-sdk, aws-sdk-core, aws-sdk-v1	21,540,237	3
html-sanitizer, rails-html-sanitizer	9,980,893	2
linecache, debugger-linecache	6,299,245	2
ruby_core_source, debase-ruby_core_source, debugger-ruby_core_source	4,696,472	3
http_connection, right_http_connection	4,661,383	2

Table 9. Highest ranking clusters with regard to the number of downloads. The number of downloads is that of the gem with the highest number within the cluster. The central nodes are always listed first.

6.2.4 Optimizations

These algorithms use five independent variables to create the clustering. It is likely that it would perform similarly well with only a subset of them (probably, the name has the largest contribution) so a possible optimization would be to select a minimal subset of these indicators. The three groups that we identified also have a considerable overlap. Another optimization could thus be a reduction into a single algorithm that would cover all three of them.

6.3 Dependency leaves

On the other side of fundamental gems stand gems that represent the leaves of the dependency tree, with no incoming dependency edges. These can be located easily and as of Feb. 29, 2016, there was 101,696 of such gems, over 85% of the entire ecosystem. With a total of 7.6 billion downloads, this subset has a collective number of downloads of only 651 million, with a median of 2,303 downloads. It is no surprise then that the majority of these gems would not rank very high on the importance scale, as we defined it in Chapter 3. Or in other words, the majority of these gems have not gained enough visibility to make it interesting for other developers to extend them, which would turn them from leaves to internal nodes of the dependency tree.

There are, however, outliers in this group. Sorting the gems by the number of downloads, we find that there are nodes that have over 1 million downloads, without having any other gem depending on them. This configuration is unusual, because when gems gain a certain user base, users tend to extend them by added functionality in the form of other gems. Gems in this configuration could be understood as ***functionality endpoints***, products that require no further extensions. Consider the gem ***turbo-sprockets-rails3*** extending only a small part of the Rails framework by speeding up the compilation of static assets (such as images or stylesheets).

Name	***# of downloads***
gem-wrappers	2,742,494
busser-serverspec	1,698,570
bundle	1,604,818
turbo-sprockets-rails3	1,465,235
dpl	1,377,704

Table 10. The most downloaded gems with no dependents.

Since RubyGems.org is an open market with no entry barriers, this method is very vulnerable, as anyone can add a dependency on a leaf gem by releasing a new gem, turning the leaf gem into an internal node. However, not every addition like this can be regarded as a meaningful contribution to the ecosystem; consider the so-called *reuploads*, as we labeled them in Section 6.2.3, as well as the median number of downloads of 2,732, median daily downloads over a lifetime of a gem being 3.34 and 87% of all gems

having no dependents at all. When the ecosystem is dominated by gems with low significance (most of the downloads are caused by robots), it is also questionable, how meaningful are the dependencies caused by them. In other words, it seems to be very likely that some gems that could be considered "functionality endpoints" were hidden by some insignificant gems that specified them as a dependency.

To account for such scenario, we made an attempt in which we also included gems with dependents, however the sum number of downloads of all these dependents was only a fraction of the number of downloads of the particular gem. When considering the downloads as a measure of importance, this would translate to the gem being more important than all its dependents combined. In Table 11 we list a number of gems where the ratio between these two numbers was the smallest. Notice the gem ***rack-ssl*** on the first place, which introduces a very simple extension (redirects all requests to use the HTTPS protocol) to a very frequently used gem *rack*. It has several other gems that build on top of it, but none of them gained a comparable popularity. In summary, we found 6,995 gems that had more downloads than all their dependents combined, as well as having at most 50 of such dependents.

Name	***# of downloads***	***Dep. # of downloads***	***# of dependents***
rack-ssl	21,070,465	307,805	24
journey	18,105,768	1,282,263	33
rubygems-bundler	15,685,771	173,082	11
rubygems-update	14,706,994	74,798	9
daemon-controller	4,763,254	94,495	15
httpauth	4,302,393	114,207	18
quiet_assets	4,513,741	368,259	45

Table 11. Gems, where the difference between their number of downloads and combined number of downloads of their dependents was the greatest.

6.4 Popular inactive gems

Two readily available metrics are the popularity and activity of a gem. We operationalize popularity as the number of downloads per time period, whereas active gems are considered those that are maintained. For both metrics there is a threshold to be defined to fit a gem into one or the other category. Additionally, activity of a gem can be either determined on a repository level (GitHub) or a version release level (RubyGems.org). Because we can observe each individual commit, the metric is more

sensitive on the repository level, but can be applied only to a subset of gems.

We found of particular interest gems that we could classify as popular yet inactive. Previous experience would suggest that when a project is popular, it automatically draws developer attention as well. This should apply even more so in an open source community, where any of the users could potentially become the maintainer of the project, if the original authors leave. Thus, it is rather uncommon for a project that is popular not to be maintained. Yet, in the Ruby ecosystem we discovered that this is actually relatively often the case. We selected gems that had at least 100 downloads a day for the past three months, which delivered 3,228 results. Out of these, 1,233 gems did not receive any new version for the past year, which is significant considering the high average and median version release frequency for the ecosystem overall (on average a new gem version is released every 11 days), as discussed more fully in Section 7.2. Similarly, we found 2,448 gems with over 100 daily downloads that also had an associated GitHub repository, out of which 557 had no commit recorded in the past year. For additional parameter combinations, see Table 12 and Table 13. Notice that the percentage is still high even for gems that have over 5,000 downloads a day and have not been updated for over a year.

Daily downloads	***Maintained***	***Not maintained***	***% (not maintained)***
> 100	1,995	1,233	38.20%
> 5,000	244	94	27.81%

Table 12. Number of gems with the specified number of downloads a day (average over the last 3 months) that ***have*** and ***have not*** received a new version release in the last year.

Daily downloads	***Maintained***	***Not maintained***	***% (not maintained)***
> 100	1,415	1,813	56.16%
> 5,000	176	162	47.93%

Table 13. Same as Table 12, only this time gems are considered as not maintained if they have not received any new version release in the last 6 months.

The most plausible explanation seems to be that these gems provide so basic a functionality that there is little room for improvement left and a stable version has been achieved, e.g. the gem ***i18n***, that dominates the list ranked by the number of downloads, and provides localization and

translation services. Many such gems are very fundamental in function and as such significantly overlap with gems defined in Section 6.1.

Name	***Last update***	***Recent daily downloads***
i18n	Dec 19, 2014	80,466
thor	Mar 24, 2014	74,019
builder	Jun 1, 2013	73,613
erubis	Apr 1, 2011	63,429
rack-test	Jan 9, 2015	57,579
diff-lcs	Nov 8, 2013	44,019
multipart-post	Dec 21, 2013	42,222

Table 14. Seven gems that have not received any update for the past year, ranked by the number of average daily downloads in the past three months.

7 Gem lifecycle

7.1 Expiration

Like in the natural world, extensions in a software ecosystem also have their lifetimes and at the latest expire together with the ecosystem itself. Often an extension could be perceived as expired before that, however. Consider a solution that enjoys high popularity at first, but is later superseded by another product, dropping its user base to practically zero, or a gem that has never gained any user base at all and the author has not touched it in years. We would naturally perceive such gems as deprecated, outdated, or expired. We have seen that there are large numbers of gems that (1) other gems do not depend on and (2) have very few downloads (whereas we must ask ourselves whether the median of two thousand downloads is few – however, downloads are by no means the number of users and we should concentrate more on the distribution rather than on the absolute number). The question is then, how many of them are maintained and used; how many still contribute to the ecosystem at all? Can we identify them programmatically?

Whether such method can be implemented boils down to the definition of an expired gem. Theoretically, we have two possibilities. (1) A gem is expired if it is not used anymore. The question here is, what does it mean not to be used – no user at all or is there a certain user base allowed if it is low enough/shrinking? This also has a practical difficulty in determining the size of the user base, since as we know, number of downloads is largely inadequate. (2) A gem should be classified as expired, if the authors do not maintain it anymore. In this case the question is, how do we choose the threshold over which a gem may not be maintained to be classified as expired? Also, what about the popular inactive gems which are not developed anymore but are still downloaded frequently, as shown per Section 6.4?

Obviously, this task is not as trivial as it would appear at first; ideally we should consider the evolution of a gem's download history as well. Certainly, gems that experienced a peak and then a significant decline would be of interest. However, we do not only lack the data for the most part of the history, but extraction and utilization of such data would be programmatically difficult.

Analysis of how many projects, or gems, are still active was already done in Section 5.3.1. As of Feb 29, 2016, we counted approx. 4,500 gems with GitHub repositories that are active (out of 68,755 GitHub gems in total). If turned around, this could be viewed as an analysis of how many gems are expired, even though we would restrict ourselves only to a maintenance-based approach. To complement it, we attempted to infer a similar methodology based on the user base size.

In order to classify a gem as expired or expiring, we considered two dimensions, both relying on the download statistics. First, we could consider a gem to be expired if users do not download it anymore at all. As illustrated previously, determining the user base purely on downloads is challenging due to some complications. First of all, if a gem is not downloaded anymore, it does not mean it is not used, as the users might have had it installed for a long time without the need to repeat the download, or they install it from their private repositories. Nevertheless, if a gem has only a handful of such users, it can be hardly seen to be a significant part of the ecosystem and could be considered expired. Second, non-negligible traffic is caused by robots, potentially downloading the data for similar purposes as this thesis does. Consider the graphic on Figure 35, where virtually all gems have had at least 1.5 downloads a day for the last three months, yet it is unlikely that every single one of them would be used on a regular basis. Furthermore, it still remains unknown how much of the traffic such robots are responsible for, and even a small difference in thresholds would lead to great differences in numbers (1.5 would exclude less than 10,000 gems, 2.0 more than 30,000). There is also a possibility that some gems are favored more by these robots than others. For further enumerations we selected a threshold of 1.9, the approximate point where the steepness of the trend suddenly changes. This resulted in 33,368 gems being excluded, or classified as having no real traffic at all.

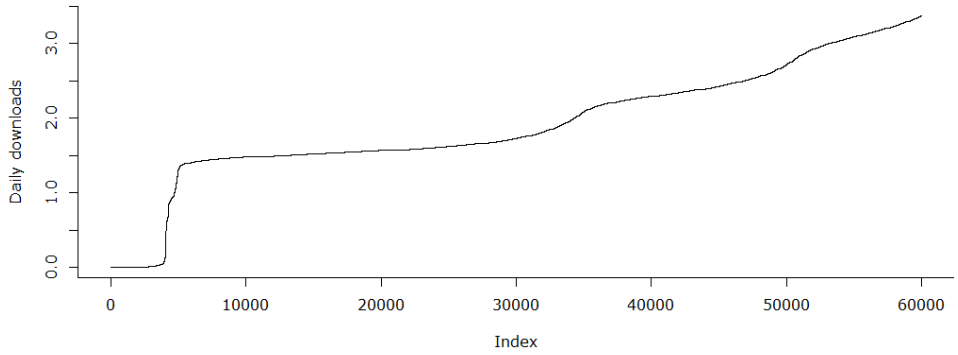

Figure 35. Gems sorted by the number of average daily downloads over the last 3 months (only the lowest 60,000).

The second dimension which we can consider is the actual evolution of downloads. While it is difficult on a massive scale, and data from earlier than the last two years was already erased, it is possible to compute the average daily downloads over the entire lifetime of a gem by taking the total number of downloads and dividing it by the number of days a gem exists in the database. This way, we are able to tell what are the recent daily downloads in comparison to daily downloads over the lifetime of a gem. On Figure 36 we show the ratio between these numbers for gems that are at least one-year-old and have recent downloads above 1.9 (see previous paragraph); ratio below 1.0 means that gems recently have fewer daily downloads than was the average over their lifetime. For more than half of the gems there seems to have been hardly any recent development. From Section 5.3.3 we are also aware of the fact that the total download volumes increase in time, so we tried to adjust the number by the ratio of the entire ecosystem (which was 2.13). As a result, for almost 70,000 the ratio was lower than 1.0 (approx. 30,000 without the adjustment).

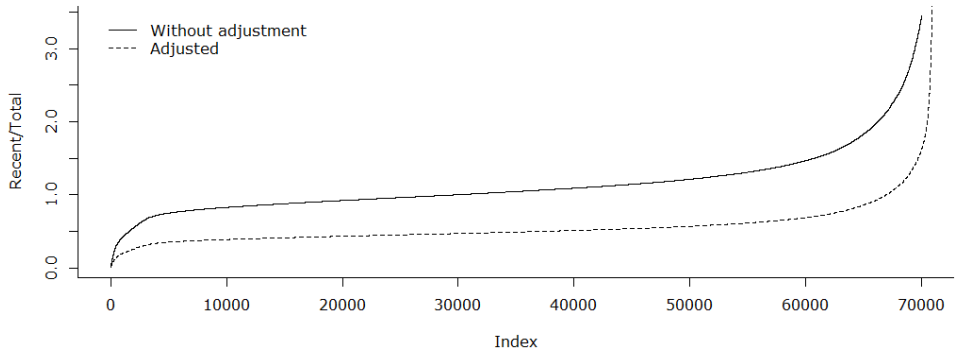

Figure 36. Gems sorted by their ratio of recent and average downloads a day. For readability reasons only the lowest 70,000 are displayed. Adjustment means dividing the ratio by the ratio of the entire ecosystem.

Both dimensions could be combined to create some classification method, whether a gem is already expired or is likely to expire based on this recent development. As always, it strongly depends on the thresholds we choose. If we applied the 1.9 threshold mentioned previously, we would have over 33,368 gems that are already expired. With the second method, apart from the ratio, we could also consider using a threshold in the absolute number of recent downloads in order not to classify gems as soon-to-expire if their daily downloads are still significant. Selecting a threshold for downloads of at most 10 and a ratio of below 1.0, this method returned another 23,844 gems (without adjustment, with adjustment the number was 51,516). While the interpretation remains arguable, the number of gems with very few actual downloads is seemingly considerably high.

7.1.1 Yanked gems

While RubyGems.org does not offer any explicit way how to remove a gem from the database entirely, it is possible for authors to remove them from the listings and make them unavailable for download. The command is called ***yank***. This state is reflected in the database as a gem having no versions that would be marked as `latest`. The number of yanked gems is 7,307, or alternatively, there are 111,610 gems that are really available.

7.2 Version lifetimes

An interesting study object for any developer, be it an individual or a company, is how long old software versions continue to be used and how

quickly new versions are adopted. In Ruby and in numerous other programming language ecosystems, there is a convention for version numbering, which is called the Semantic Versioning standard[22]. This standard defines that a version should be composed of three numbers, the major, the minor and the patch version, each separated by a dot. An increase in the patch number should have no effect on the API of the software. The minor number should be incremented if a backward-compatible change is introduced into the API and the major number should only be changed if the new API is not backward-compatible. Nevertheless, this versioning scheme is not mandatory and gems are free to use their own. On top of that, even if something appears to follow this scheme, there is no guarantee that it is followed correctly, as we will witness later. Nevertheless, for gems following the scheme, observing the adoption speed of new versions can show the creators (1) how fast new APIs are adopted (in case of a new major release) and (2) how long old versions will have to be maintained. Such information is certainly useful for planning and budgeting when the gem is maintained by a commercial entity.

This is also a place where we are challenged by the lack of a good methodology for measuring the user base. Even here we will have to rely on the download history, even though we already know that it does not have to reflect the actual usage accurately. Nevertheless, it still has informative value when the volumes are large enough, so in the following sections we will show the download evolutions of various versions for three different gems: Rails, Celluloid and Chef. The point is to give a basic insight into how quickly versions are adopted, how this information can be utilized and how different gems with different release cycles and version numbers can vary. For an introductory overview, we list the average and median number of days between new version releases of relevant gems or group of gems in Table 15. Note that these statistics do not differentiate between major, minor, or patch versions. We got these numbers by constructing a single list, where we listed number of days between each next version of each gem. Thus, the more versions a gem had, the more emphasis was given to it. Partially, this could explain the median of 11 days which is very low.

[22] http://semver.org; http://guides.rubygems.org/patterns/#semantic-versioning

	Mean	*Median*
All gems	60.16	11
Over 50k downloads	47.36	10
Rails	21.54	10
Celluloid	30.46	22
Chef	12.30	6

Table 15. The mean and median number of days between version releases. Second row represents gems that have at least 50,000 downloads in total (a collection of 5,335 gems), as we would assume that the more popular they are, the more attention they tend to get from their maintainers.

7.2.1 Rails

The gem ***rails*** has around 65,000 downloads daily and over 64 million downloads in total. These volumes make the data less sensitive to actions of single individuals and better display large-scale (external or internal) factors. The framework has had 280 versions so far, the last one being 5.0.0.beta3. Since the major version 5 was first released on Dec 18, 2015 and, as of Feb 29, 2016, is still in the beta stage, it's proportion of downloads is insignificant and the primary version remains to be 4.2. For this reason, we do not include it in the graphs that follow.

Since our download history data reaches only as far as Jul 20, 2014, we can only look at the recent versions. The first beta version of Rails 4.2 was released on Aug 20, 2014 and as we can see on Figure 37, the number of downloads for the next 4 months was, unsurprisingly, rather low. Then on Dec 20, 2014, Rails 4.2.0 was released, leading to a steady growth, as more and more users migrated to the new version.

Absolute numbers might be slightly misleading, since the number of daily downloads is vulnerable to effects of some external factors; in particular, on a scale like this the most evident is the decrease around Christmas holidays. Instead, we can look at the download volumes of individual versions in relation to the combined number of downloads, as depicted on Figure 38. Here we can see more clearly that releasing the first production version 4.2.0 lead to an immediate boom, it had the largest impact on downloads of 4.0 and 4.1 and basically no effect at all on earlier major versions. This is understandable, since minor versions should introduce only backward-compatible changes to the API, so switching should be without problems.

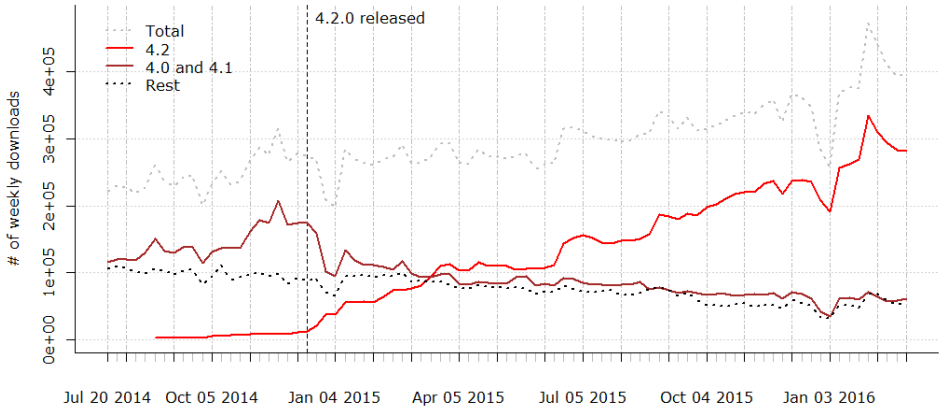

Figure 37. Evolution of weekly downloads for selected versions in absolute numbers.

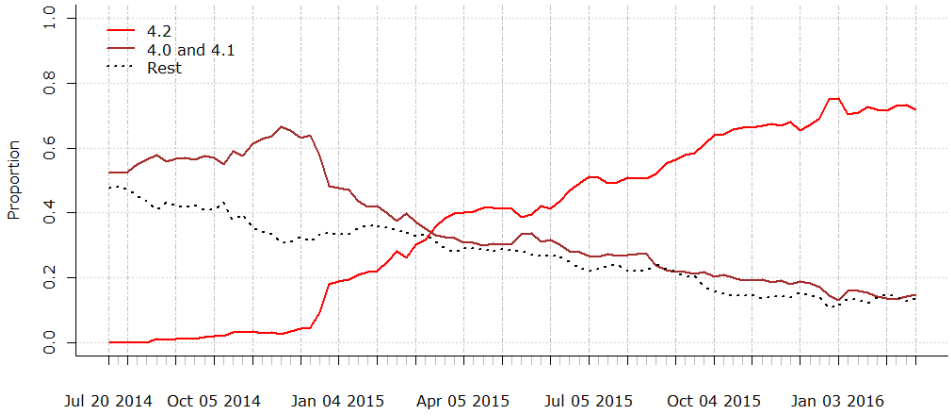

Figure 38. Proportions of downloads for selected versions to total downloads. The data is averaged on weekly basis to smoothen up noise.

The rate at which new minor and major versions are adopted is an interesting study object and can be used to infer some assumptions about the user base. If old versions have long lifetimes, it could mean that the user base is more conservative and represented by larger companies, where stability is a key factor. On Figure 39 we put relevant release dates into perspective and show that in spite of version 4.0.0 being released on Jun 25, 2013, a significant portion of the user base remained with the previous

major version and only reached the 10% mark by the beginning of 2016, 2.5 years after the release of Rails 4.0.0, with a total lifetime of 4.33 years. Regarding the minor versions lifetimes, we see that 4.0 had a significant user base for around 1.75 years, 1 year after the release of 4.1, which in turn similarly had lifetimes of 1.67 and 1 years (in respect to 4.2). New minor versions are, indeed, adopted much faster, even though the users seem to be rather conservative in this case.

These numbers are important for the maintainers of the framework to give them an idea of how long old versions need to be maintained and when it is safe to drop support. By the time the download history starts, Rails 2 seems to be long surpassed, with downloads oscillating below the 5% mark.

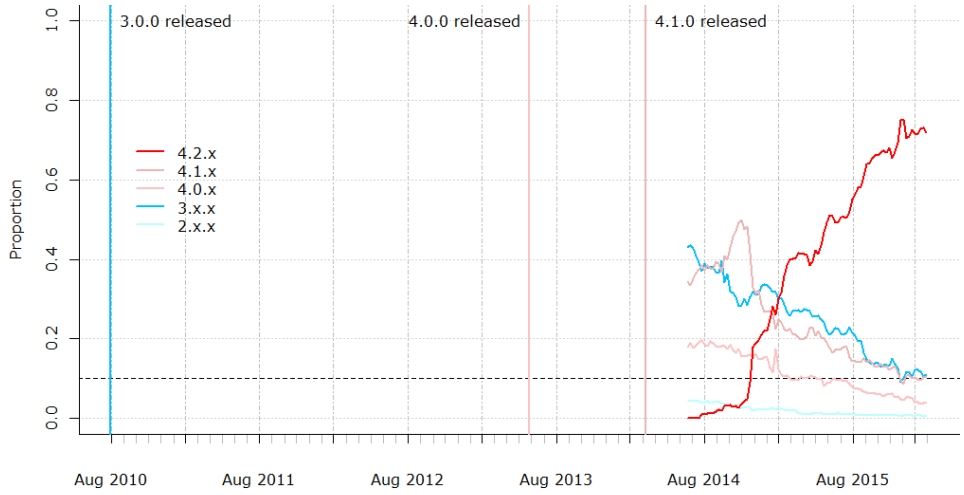

Figure 39. Shows how the proportions of downloaded versions changed in time. While we lack the data before Aug. 2014, we show the release dates of relevant versions to bring them into perspective.

Finally, we can look at how long the community around Rails kept fixing bugs in these old versions by looking at the release dates of the respective major and minor versions, as seen on Table 16. As we have already seen, Rails 2 is long outdated with its latest version released in 2013. On the other hand, all version lines 3, 4.0 and 4.1 seem to be still alive, in spite of seemingly declining user base. We must keep in mind, however, that Rails has a huge amount of users and the developer community around it is similarly large, so, clearly, proportional usage (or downloads, for that matter) is not very conclusive, as even a small percentage can mean a large number of potentially influential users.

Version line	*Latest version*	*Release date*
2.x.x	2.3.18	Mar. 18, 2013
3.x.x	3.2.22.2	Feb. 29, 2016
4.0.x	4.0.13	Jan. 6, 2016
4.1.x	4.1.14.2	Feb. 29, 2016

Table 16. Release dates of the latest versions of relevant major and minor version lines.

7.2.2 Celluloid

Celluloid is a library that simplifies concurrent programming in Ruby, exists since May 11, 2011, and, with over 12 million downloads, belongs to one of the most popular gems. In its history, it has only had one major version (number 0) and 16 minor versions. Therefore, we will not be able to observe the adoption speed of major versions within its user base and we will focus on the minor ones. On Figure 40 and Figure 41 we can notice that the version line 0.16 was taken over much faster than it was in the case of 0.17. Looking at the changelog[23] of 0.17.0 we found out that some classes were renamed and moved in this version, introducing a backward-incompatible change which made it harder for users to switch. On this example we can see how the semantic versioning scheme is not always perfectly followed. According to a guide[24] available on the official website of Ruby on Rails, no such changes were made in Rails 4.2, yet the jump was smaller than it was in the case of Celluloid 0.16.

As seen on Figure 41, version line 0.15 has still not reached the 10% mark as of Feb 29, 2016. If we tried to extrapolate the data, we could expect it to reach it around mid-2016 at this pace. That would give this version a lifetime of around 2.75 years and would happen 1.75 years after the release of the next minor version 0.16. This is almost a year longer (for both numbers) than it was in the case of recent minor versions of Rails. However, as depicted above, 0.15 seems to have been the only version line with a significant user base during the initial release of 0.16, whereas in Rails we had a simultaneous coexistence of multiple minor and major versions. It is hard to come to some definitive conclusions from the small number of releases we are able to observe, but it would appear that the users are ready to switch Celluloid versions faster than of Rails. One

[23] https://github.com/celluloid/celluloid/blob/master/CHANGES.md
[24] http://edgeguides.rubyonrails.org/upgrading_ruby_on_rails.html

possible reason for this would be the fact that the functionality provided by Rails is more complex and there is more room for things going wrong.

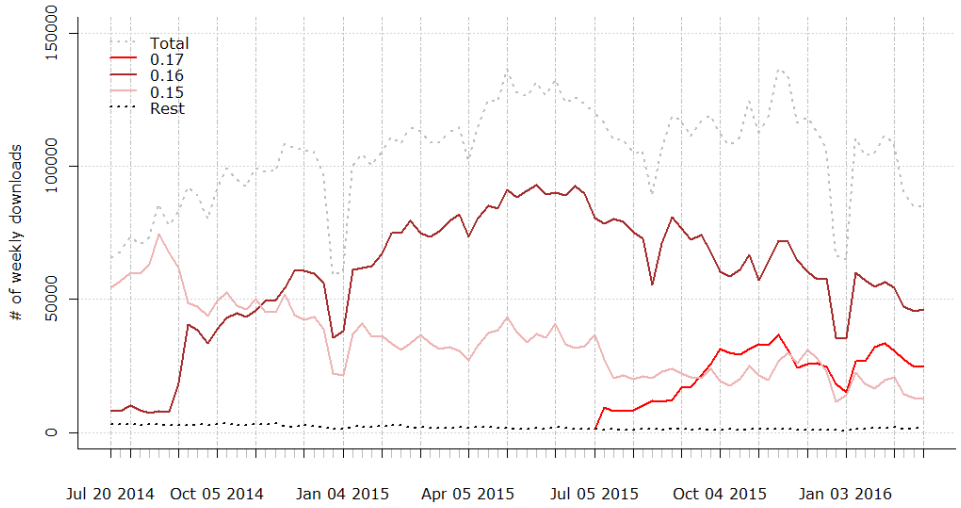

Figure 40. Evolution of weekly downloads for selected versions in absolute numbers.

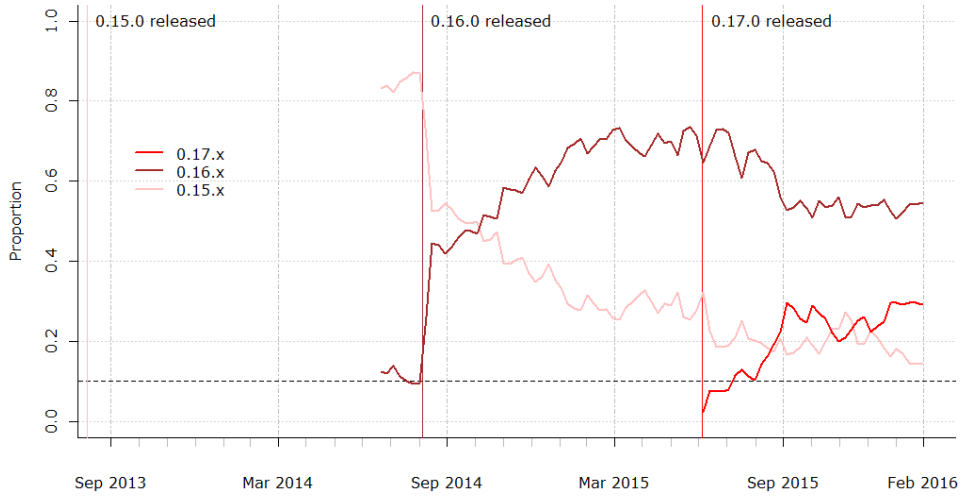

Figure 41. Proportions of version downloads in time. Relevant version releases are put into perspective.

7.2.3 Chef

Chef[25] is a configuration management tool streamlining server configuration. It implements a paradigm called ***infrastructure as code***, where configuration (e.g. what software and services must be installed and running and how these should be configured) is stored in machine-readable files, making them easier to apply on new infrastructure nodes, easier to maintain and enables us to automatically monitor their correct state. The gem ***chef*** represents the server side of the tool, has almost 7 million downloads and has had 4 major versions (0, 10, 11 and 12) by Feb 29, 2016.

On figures below we can see a coexistence of the last three major versions. While a steady growth and decline of versions 12 and 11 respectively can be recognized, the version 10 seems to retain a significant user base even three years after the release of its successor. Compare this to Rails 3 that reached the 10% level after 2.75 years and where a clear declining trend was recognizable. On the other hand, Rails 3 continued to receive patches by the time this study was done, whereas the last patch of Chef 10 was released on Nov 10, 2014, almost 1.5 years ago. There are obviously many factors that influence whether an old version will be maintained, including the complexity of the software, priorities and experience of the developer team or company, and the absolute size of the user base of that version. While we cannot reliably tell the size, the download volumes of Rails were larger by a factor of 10, so as we already mentioned before, even small proportions could mean a large amount of users.

[25] https://www.chef.io/chef

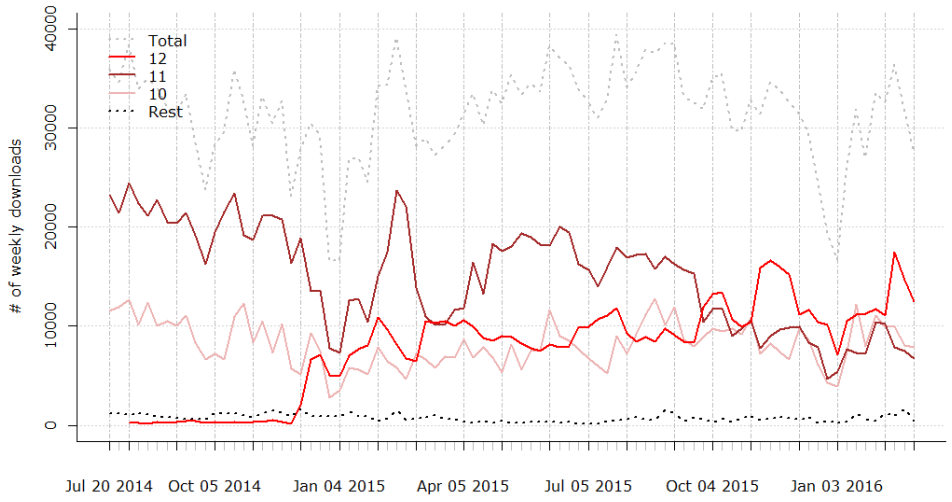

Figure 42. Evolution of weekly downloads for selected versions in absolute numbers.

Interesting is the download history of minor versions. Figure 44 shows that each minor version almost immediately takes over the majority of downloads, then to be replaced by the next version. While it is true that downloads show only new installations, not the actual proportions of use, this chart shows significant differences from those of Rails and Celluloid. One explanation could be that Chef receives minor version upgrades more regularly than Rails and are less distinguished. On the other hand, new minor Celluloid versions are treated almost as major releases. Another question we could ask ourselves is how well users understand the release cycles of gems they use. Answering it we could find out that the differences between the download histories could be attributed to something as trivial as the intuitive "outlook" of the version number, i.e. the lower the version number is, the bigger the anticipation of a breaking change will be. For example, minor versions of Celluloid could be treated more like major versions because the major version has number 0. The subjective importance is then lesser in the case of Chef, where the latest version number starts with number 12. While it is not the subject of our study, it could be one of the factors deciding the lifetimes of old versions.

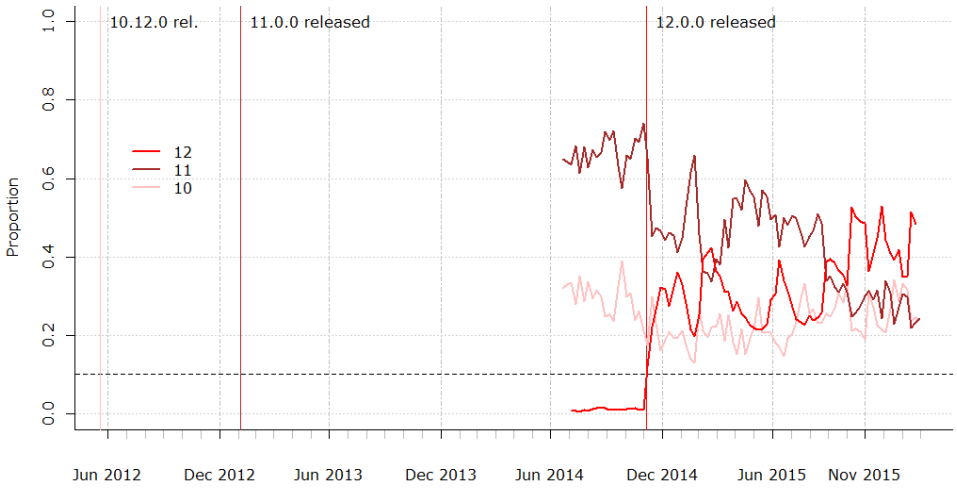

Figure 43. Proportions of major version downloads in time. Relevant version releases are put into perspective. Note that 10.12.0 was the first production version of the major version line 10.

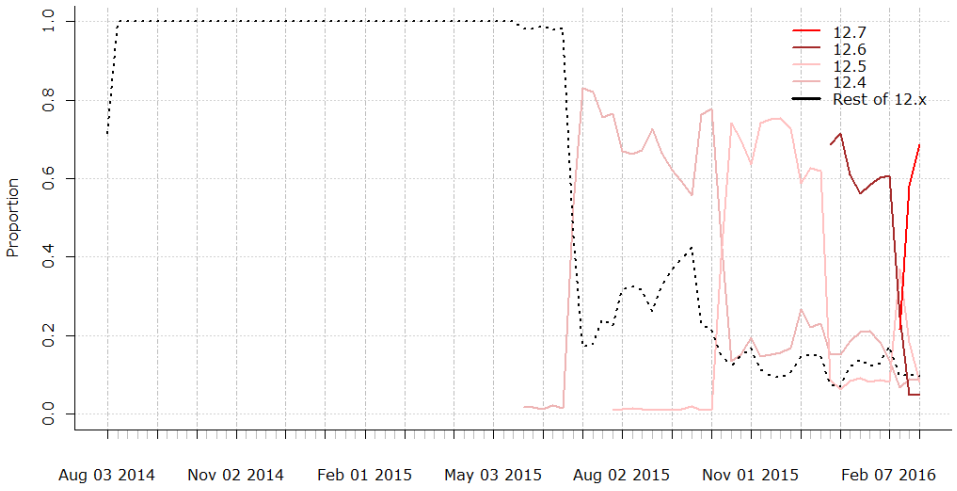

Figure 44. Proportions of recent minor versions.

8 Discussion

> *RQ1: How has the Ruby ecosystem evolved and what is its current state? Is interest in Ruby increasing, decreasing or does it remain unchanged?*

In Chapter 5 we have seen that the number of new gems in the ecosystem is decreasing. Since there are no market entry barriers on RubyGems.org, the ecosystem is now largely populated by gems that have little to none users and, thus, could be deemed rather unimportant to sustaining the wider ecosystem. These gems have both low download numbers (total and daily) as well as no dependents at all.

The decreasing trend was reaffirmed by other indicators; of particular interest were those that used commits to GitHub as the source of data. They showed that the commit volumes plummeted, leading to a significant decline. So did the number of active contributors as well as projects. Based on this data, it also seems that the influx of new members into the collaborating part of the community (which we consider to be the core) has been stagnating as well. Finally, similar development can be observed in the statistics provided by Google in Figure 7, where the interest in Ruby, the programming language, as a search term has been steadily decreasing.

A certain decline in growth and interest in the ecosystem is apparent. Based on the scheme for ecosystem health evaluation proposed by Jansen [12], we would evaluate the health of the Ruby ecosystem as not very positive, since he defines it as the *propensity for growth*. On the other hand, we think that a growing trend that is slowing down could also be an indication of an ecosystem that has become mature and stable. An indication of that could be the download numbers (which should loosely reflect the number of end-users) that continue to grow, although the ever-increasing number of gems and the presence of crawlers must be accounted for.

> *RQ2: Is the platform alone sufficient for the support of the wider ecosystem? What does the platform provide, what does it not?*

In Section 6.1 we have seen that there are gems in the ecosystem, which provide such fundamental functionality that almost every other gem depends on it – either directly or at least transitively. One such example

would be ***rake***, which has become a substantial part of the build process for virtually every gem. We have found 288 gems on which more than half of the remaining gems depend in this fashion.

Similarly, in Section 4.5 we have observed a trend showing an increasing number of dependencies and the degree to which gems are intertwined, meaning that, in time, gems tend to require more and more other gems for them to function. One interpretation would be an ever-increasing need to require more sophisticated solutions than what the platform provides by itself. Alternatively, it could be also an indication of gems enhancing each other rather than creating a completely new solution (i.e. plugins) or that gems become more modular, smaller and more specialized (single responsibility principle). Likely, all three options play a role and, combined with the first paragraph, it seems plausible that the ecosystem complements and improves some basic aspects of the platform.

> *RQ3: Can we categorize gems into one or more taxonomies based on the quantitative data?*

We have seen in Chapter 6, that based on the dependency graph, number of downloads and, if need be, other indicators, we can find gems where similarities in the indicators translate to shared characteristics. These characteristics can be interesting for various interest groups; for instance, gems that are popular, yet inactive, could be used to research universally required solutions that were simple enough so that their further maintenance was unnecessary. Fundamental gems, on the other hand, can show projects which are needed for a healthy ecosystem and what would be sensible to provide right from the start if someone were to create a new ecosystem. To summarize, we found that our data can be used to derive taxonomies, even though it is a coarse-grained tool that can hardly be used to make any inferences about the purpose of every individual gem.

> *RQ4: Is the ecosystem-wide data relevant for decision-making on a project level?*

Sophisticated commercial ecosystems such as Google Play (formerly known as the Android Market) collect large volumes of a variety of data and provide the extension administrators with a large number of views on that data. This is frequently used to drive the development of the extension. The situation is different in an ecosystem like that of Ruby, however, where the developers do not have such detailed statistics at their disposal. The

question was, whether the data still can be used to learn something about the user base.

Tracking the number of downloads on a per-version basis is a very basic service that is very easy to implement and should be a part of every extension market. In Section 7.2 we have demonstrated that the very least we can do with this data is to compare their proportions and, if tracked for a given amount of time, their evolutions. These numbers can be used to make assumptions about the user base, record the adoption rates of new versions and guess the size of the user base, which helps when prioritizing maintenance tasks. Its use still remains limited, however, and plays more of a supportive role.

> *RQ5: How much of the popularity of Ruby can be attributed to Ruby on Rails?*

Section 4.3 was dedicated to elaborating on this question. On the chart provided by Google Trends, we have seen that as soon as Rails was released, the popularity of Ruby in terms of Google searches raised significantly, in accordance to the Ruby on Rails search term. Even though the interest in 2010 seems to have returned back to its before-Rails numbers, the number of Ruby on Rails searches remained high. Naturally, there is no guarantee that the data from Google is reliable; in fact, "ruby" is a relatively common word and in spite of Google showing results for Ruby the programming language, the ability of the search engine to differentiate between semantic meanings of the word has likely changed and influenced the numbers over the years.

Nevertheless, a new view was provided by the data we already had at our disposal. We found that almost a quarter of all extensions in the ecosystem could be linked to Rails, which is significant for a single framework. Therefore, it would seem that the ecosystem is, indeed, very Rails-centered.

9 Conclusion

We have seen various ways quantitative data can be used to our advantage when evaluating a software ecosystem and were able to analyze the ecosystem as whole, as well as its individual projects. An emphasis was put on the analysis of health indicators, out of which we have come to the conclusion that the Ruby ecosystem is slowly declining, as the community shrinks and the productivity of its members decreases. We also witnessed that gems of low quality flood the ecosystem, which creates a distorted view on the ecosystem's size. All these effects are closely tied to the nature of the Ruby ecosystem, which is open-source and offers no barriers for entry in its primary market.

The large number of gems served as a motivation for automated classification methods, as the discovery of interesting gems has become challenging, be it for a practical applications or for research. We tried to derive algorithms and approaches on how to find gems that would be of particular interest based on certain characteristics. Our classification scheme is not exhaustive, but we feel it covers the most obvious avenues of approach, and can serve as a basis for determining other, more refined classification schemes.

Future work could take advantage of the insights gained in this study, complement the gem taxonomy with new categories and the classification methods could be then put into practical use on a standalone portal. It would be bold to state that offering a categorization scheme would improve the fortunes of the Ruby ecosystem; likely, the decline is to be attributed to other factors rather than the discoverability of gems. Nevertheless, better discoverability of gems could contribute to the quality of the infrastructure and could give better chance for innovative ideas to be found. Future research could focus on pinpointing the actual causes behind the deterioration of the ecosystem.

Our methods could of course also be extended to other ecosystems. In a more tightly controlled ecosystem, for example, our insights into structure and behavior of an ecosystem's elements could be applied to improve its governance.

References

[1] G. K. Hanssen and T. Dybå, "Theoretical foundations of software ecosystems," in *Proceedings of the Forth International Workshop on Software Ecosystems*, Cambridge, MA, USA, 2012.

[2] J. Bosch, "From Software Product Lines to Software Ecosystems," in *Proceedings of the 13th International Software Product Line Conference*, San Francisco, CA, USA, 2009.

[3] 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.

[4] S. Jansen and M. Cusumano, "Defining software ecosystems: A survey of software platforms and business network governance," in *Proceedings of the Forth International Workshop on Software Ecosystems*, Cambridge, MA, USA, 2012.

[5] J. Kabbedijk and S. Jansen, "Unraveling Ruby ecosystem dynamics: a quantitative network analysis," in *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, Cheltenham, UK, Edward Elgar, 2013, pp. 322-332.

[6] M. M. M. Syeed, K. M. Hansen, I. Hammouda and K. Manikas, "Socio-technical congruence in the Ruby ecosystem," in *Proceedings of The International Symposium on Open Collaboration*, Berlin, Germany, 2014.

[7] D. G. Messerschmitt and C. Szyperski, Software Ecosystem: Understanding an Indispensable Technology and Industry, Cambridge, MA, USA: The MIT Press, 2003.

[8] S. Jansen, S. Brinkkemper, J. Souer and L. Luinenburg, "Shades of gray: Opening up a software producing organization with the open software enterprise model," *Journal of Systems and Software,* vol. 85, no. 7, pp. 1495-1510, 2012.

[9] P. Wagstrom, C. Jergensen and A. Sarma, "A network of Rails: A graph dataset of Ruby on Rails and associated projects," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, San Francisco, CA, USA, 2013.

[10] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German and D. Damian, "An in-depth study of the promises and perils of mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, Hyderabad, India, 2014.

[11] M. Goeminne and T. Mens, "Analyzing ecosystems for open source software developer communities," in *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, Cheltenham, UK, Edward Elgar, 2013, p. 247–275.

[12] S. Jansen, "Measuring the health of open source software ecosystems: Beyond the scope of project health," *Information and Software Technology,* vol. 56, no. 11, p. 1508–1519, 2014.

[13] K. Crowston, J. Howison and H. Annabi, "Information systems success in Free and Open Source Software development: Theory and measures," *Software Process: Improvement and Practice,* vol. 11, p. 123–148, 2006.

[14] J. Moody and D. R. White, "Structural Cohesion and Embeddedness: A hierarchical concept of social groups," *American Sociological Review,* vol. 68, pp. 103-127, 2000.