Università degli Studi di Pavia

Dipartimento di
Ingegneria Industriale e dell'informazione

Scuola di Dottorato in
Ingegneria Elettronica, Informatica ed Elettrica

XXX Ciclo

Mining Git based Software Repositories

Tesi di dottorato di:
Gianluca Roveda

Relatore:
Dott. Tullio Facchinetti

A.A. 2016/2017

Questa tesi è dedicata a Michela,

che mi supporta e sopporta ogni giorno,

alla mia famiglia e ai miei amici.

Contents

Contents **v**

List of Figures **vii**

List of Tables **xi**

1 Introduction **3**

1.1 Impact of VCS mining in research and industrial fields 6

1.2 Objectives of the work 8

1.3 Structure of the thesis 10

2 Git and GitHub **13**

2.1 Basic concepts of Git 13

2.2 Basic concepts of GitHub 16

2.2.1 Issue tracker and code review 17

2.2.2 Continuous Integration and project metrics 18

2.2.3 Documentation and social features 19

2.3 Data mining of Git and GitHub features 20

3 Related works **25**

3.1 Git repository and their mining 27

4 Tools used **35**

4.1 Python 35

4.2 MongoDB 35

4.2.1 Map-reduce 36

4.3 KNIME 38

4.4 Gephi 38

4.4.1 ForceAtlas2 . . . 39
4.5 HTTP and REST API . . . 41
4.6 GHTorrent . . . 42
4.7 System setup . . . 43

5 Data analysis **45**
5.1 Datasets . . . 45
5.1.1 Datasets format . . . 47
5.2 Data exploration . . . 50
5.2.1 Size of the repositories . . . 51
5.2.2 Popularity of the repositories . . . 60
5.3 Note of commit authors . . . 70
5.4 Sparsity of contributions . . . 74

6 The network graph **81**
6.1 Implementation . . . 82
6.1.1 Data retrieval . . . 82
6.1.2 Data processing . . . 84
6.1.3 Graph visualization . . . 87
6.2 Interpretation of results . . . 91
6.2.1 MSR14: jkbr/httpie . . . 91
6.2.2 MSR14: mangos/MaNGOS and TrinityCore/TrinityCore . . . 93
6.2.3 MSR14: mxcl/homebrew . . . 97
6.2.4 MSR14: django/django . . . 101
6.2.5 MSR14: PHP frameworks . . . 104
6.2.6 Android REST API client libraries . . . 106
6.2.7 ELECTRON: electron/electron . . . 110
6.2.8 LANGUAGES: collaboration between programming languages 113

7 Conclusion and future works **117**

Bibliography **123**

List of Figures

1.1 Comparison between possible CVCS and DVCS scenarios. 5
1.2 Number of questions posted on Stack Overflow by users for a specific VCS until 2016, with Git exhibiting an increasing trend. 8

2.1 Sample Git project representation; from the top two branches (add_sqlite_db and add_css) are developed in parallel by different users and later merged on master. 15
2.2 GitLab graph representation for the Inkscape project shows a more complex usage of the Git VCS. Unlike the fictional example of Figure 2.1, this is read from bottom to top. 16

3.1 Number of MSR papers published per year. 26
3.2 Number of cited MSR papers published per year. 26

4.1 System setup, which illustrates how data is collected from GitHub REST API and is stored in a database, which in turn is accessed by KNIME for data exploration and Python scripts. Results produced are then displayed in Gephi. 44

5.1 Commits, authors and forks counts boxplots for the four datasets. Most of the displayed values are outliers, and the actual boxes appear as flat lines. 54
5.2 Commits, authors and forks counts for the four datasets, with forks excluded. 55
5.3 Graphical representation of projects sizes for MSR dataset. 57
5.4 Graphical representation of projects sizes for ANDROID dataset. . . 57
5.5 Graphical representation of projects sizes for ELECTRON dataset. . 58

5.6 Graphical representation of projects sizes for LANGUAGES dataset. 58

5.7 How many objects for each parameter of the non forked repositories in MSR14. 62

5.8 How many objects for each parameter of the non forked repositories in ANDROID. 63

5.9 How many objects for each parameter of the non forked repositories in LANGUAGES. 63

5.10 How many objects for each parameter of the non forked repositories in ELECTRON. 64

5.11 Popularity map for non forked repositories and non empty issue tracker repositories for MSR14. 66

5.12 Popularity map for non forked repositories and non empty issue tracker repositories for ANDROID. 66

5.13 Popularity map for non forked repositories and non empty issue tracker repositories for LANGUAGES. 67

5.14 Popularity map for non forked repositories and non empty issue tracker repositories for ELECTRON. 67

5.15 Popularity map for non forked repositories, without pull requests and active forks, for ANDROID. 68

5.16 Popularity map for non forked repositories, without pull requests and active forks, for LANGUAGES. 69

5.17 Popularity map for non forked repositories, without pull requests and active forks, for ELECTRON. 69

5.18 Committers for jquery using emails (Top 10 + others). 71

5.19 The number of commits produced by the top contributors for each month. 71

5.20 Committers for jquery using logins (Top 10 + missing + others). . . 72

5.21 Percentage of authors with exactly 1 commit for each repository; 30% of repositories have 50% or more authors with exactly 1 commit. . . 75

5.22 Percentage of authors with 5 or less commits for each repository; 80% of repositories have 50% or more authors with 5 or less commits. . . 76

5.23 Percentage of authors with 10 or less commits for each repository; 86% of repositories have 50% or more authors with 10 or less commits, and 51% of authors have 80% or more authors who contributed with 10 or less commits. 77
5.24 Number of repositories that was contributed from each author; 30.7% contributed only to a fork (0 repositories), 61.6% contributed to exactly one non forked repositories, and only 7.7% of authors contributed to two or more non forked repositories. 78

6.1 Pseudocode of the algorithm for data retrieval. 83
6.2 Gephi interface when data streaming is complete. 88
6.3 Gephi graph for project *HTTPie* after color, size and positions have been set. 90
6.4 The contributions in the first months of *HTTPie*. 94
6.5 Static graph for *MaNGOS* and *TrinityCore*. 95
6.6 Four moments in the history of *MaNGOS* and *TrinityCore*. 98
6.7 Homebrew static graph. 99
6.8 *Django* static graph, with *django-cms* and *django-debug-toolbar* highlighted in blue and yellow color. 101
6.9 Static graph for *PHP* frameworks in MSR14. 104
6.10 Android static graph. 107
6.11 Animated graph at the month in which *JakeWharton* contributed to *Volley*. 109
6.12 Static graph for the *Electron* project (whole view). 111
6.13 Static graph for the *Electron* project (zoomed on center). 111
6.14 Static graph for LANGUAGES. 115

List of Tables

2.1 List of features and their usage in this thesis. 20

5.1 Projects included in the ANDROID dataset. 46
5.2 Projects included in the LANGUAGES dataset. 46
5.3 Projects included in the ELECTRON dataset. 48
5.4 Collections included in a snapshot. 50
5.5 Size of *repos*, *commits* and *users* on the datasets. 51
5.6 Number of commits, forks and authors for 3 repositories of MSR14. . 52
5.7 Missing issue tracker comments. 61
5.8 Counts of different logins that created commits with the same email address. 73
5.9 Average number of commits per author who committed to exactly 1, exactly 2 and 3 or more repositories. 78

6.1 Settings for the network graph algorithm. 85
6.2 Number of commits for repositories and users in *HTTPie*. 92
6.3 *MaNGOS* and *TrinityCore* contributors, and their total contributions on either projects. 96
6.4 Top 10 contributors on *Homebrew*, and the corresponding number of commits on *Homebrew*. 100
6.5 Top 25 contributors to projects other than *Homebrew*, the number of contributed commits and the corresponding repository. 100
6.6 Top 10 committers on *Django*. 102
6.7 Committers for the project *django-debug-toolbar*. In *italic* the ones that also appears in Table 6.6. 103
6.8 Committers for the project *django-cms*. 103

6.9 Top 10 contributors who contributed to more than 1 projects in Section 6.2.5. 106
6.10 Top 10 contributors who contributed to only 1 project in Section 6.2.5. 108
6.11 Repositories of Section 6.2.5, and their commits count. 108
6.12 Commit message for ten randomly selected users who only made few contributions on *Electron*. 112
6.13 Top 10 contributors on *Electron* project 113
6.14 Top 10 contributors on *Atom* project. 113
6.15 Top 10 contributions on projects different than both *Electron* and *Atom* in Section 6.2.7. 114
6.16 Common contributors and their commits for top 5 projects in LANGUAGES. 114

Acronyms

MSR Mining Software Repositories

VCS Version Control Systems

DVCS Distributed Version Control Systems

CVCS Centralized Version Control Systems

CI Continuous Integration

OO Object Oriented

GH GitHub

GL GitLab

OSS Open Source Software

MSR14 Mining Software Repositories 2014 Mining Challenge Dataset

PCA Principal Component Analysis

HTTP HyperText Transfer Protocol

REST REpresentational State Transfer

URI Uniform Resource Identifier

API Application Programming Interface

CLI Command Line Interface

MMORPG Massively Multiplayer Online Role-Playing Game

WoW World of Warcraft

MVC Model View Controller

CMS Content Management System

Chapter 1

Introduction

Version Control Systems (VCS) are software and/or systems used by developers to keep track of changes in their applications' source code. In general, VCS allow to preserve the history of a set of files through the creation of checkpoints, each containing the files content, the creation timestamp and possibly a human written description [67]. Users can review the history of the modifications and eventually revert their code to investigate the history of bugs or undo harmful modifications.

To conceptualize, a trivial VCS form often used by young practitioners is to copy their working directory at regular intervals and before making difficult changes which may break the code; these backups are placed in folders with easily understandable names, such as "graphic interface working", "added feature X", "project final version" or "snapshot before major rework". This allows for partial history review and disaster recovery, with manual restoration of the files in the working directory. While this primitive form of VCS is indeed a working solution for simple projects, it is limited in functionalities, does not scale well with the growing of the project and is difficult to use when multiple programmers are cooperating.

Source Code Control System (SCCS) [64], one of the first VCS softwares, was invented in 1972 and features many functionalities of modern VCS. Still, it lacked collaboration along different machines.

Allowing for seamless cooperation in large teams, in particular, is a hard challenge, which poses problems that even modern VCS are still facing.

A possible solution to add collaboration in a VCS is the introduction of a central server, which is responsible to store the main version of the project. Clients can connect to the central server, obtain a copy of the project, modify it locally

and push the modifications. This paradigm, called Centralized Version Control Systems (CVCS), is implemented by many VCS, such as Concurrent Versions System (CVS)[1] [52], which was released in the middle of the 80s[2], and Subversion[3]. Subversion, in particular, is still one of the most widely used VCS[4].

While CVCS paradigm works for many real-life scenarios, and was/is widely used, its requirement of having a unique central server poses technical problems when said server is unreachable, and ethical questions for open source softwares, such as who should host the repository when the owner is not a company or an individual, but a community. In the early 90s, Sun released a VCS which based its collaboration abilities on different premises: each copy of the repository acts as a fully fledged, self contained repository, and the VCS provides tools to merge the modifications between the different copies of the repository. With this approach, there may not even be a "central repository", as intended in CVCS, or there may be as many as the users so desire. This paradigm is called Distributed VCS (DVCS), since the repository is distributed over all of the machines. Moreover, in CVCS the act of creating a checkpoint is usually contextual to its propagation on the central server, while in DVCS the two operations are independent. Having the two operations divided allows for offline working, and postpone the eventual conflicts between different versions to the act of synchronization, favoring the creation of smaller checkpoints and richer checkpoint histories.

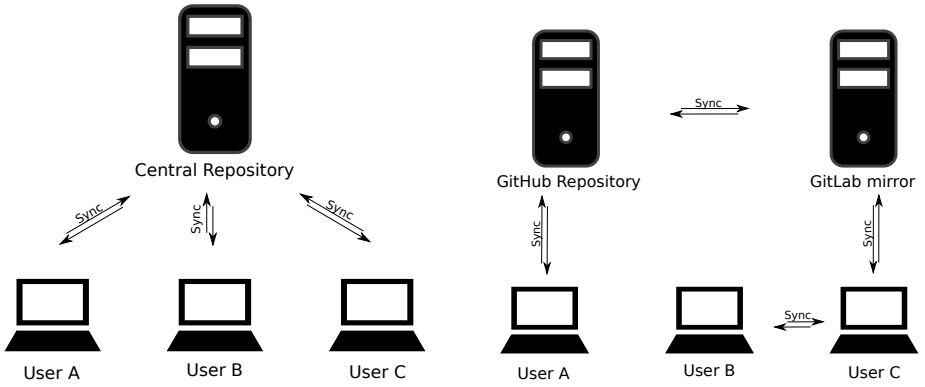
A sample figure, comparing a CVCS and a DVCS scenario, can be seen in Figure 1.1. The CVCS example is pretty straightforward: there is a single central server and three different users, connecting to the central server to synchronize their contents. In the DVCS scenario, instead, a more complex situation can be observed. As previously stated, in DVCS each machine hosts a full copy of the repository and can be used for synchronization by other clients. In this example, GitHub is used as a central server to expose the project to external contributors, here represented

[1] CVS page, last visited on January, 12th 2018: `http://www.nongnu.org/cvs/`.

[2] A copy on Google groups of the original usenet message with CVS, last visited on January, 12th 2018: `https://groups.google.com/forum/?hl=en#!msg/mod.sources/eqze_AHbIK0/uE90wCq3ui4J`.

[3] Official Subversion homepage, last visited on January, 12th 2018: `https://subversion.apache.org/`.

[4] VCS trends for 2016, last visited on January, 12th 2018: `https://rhodecode.com/insights/version-control-systems-2016`.



(a) CVCS: 3 clients with a central server; the clients require the synchronization against the server to properly share the updates

(b) DVCS: 5 nodes with mixed behaviour; client A synchronizes to a server, client C to the other server, client B uses client C as a server and the two server synchronize each other

Figure 1.1: Comparison between possible CVCS and DVCS scenarios.

as User A. User C, instead, synchronizes with a copy of the repository hosted on the company's GitLab server, to achieve faster synchronization. The GitLab copy acts both as a client, since it synchronizes automatically with the GitHub server, and as a central server for the company workers. Finally, User B needs access to the code from the company, but he his forbidden from using the company's network. Therefore, User B establish a private connection with User C and synchronizes with his repository. When User B gets back home, he will find his work on the public GitHub repository, even though he used User C as a central server. This example shows the flexibility of DVCS: out of five machines, two are acting as pure clients, one as a pure server and two as both clients and servers.

Over the years, several CVCS and DVCS alternatives were proposed, such as *Subversion* (CVCS), and *Bitkeeper* (DVCS), however in recent days many developers are choosing *Git* as the DVCS for their open and closed source developments. Git is an open source DVCS written by Linux Torvalds specifically for maintaining the Linux kernel, after using Bitkeeper for many years. Since the Linux project receives thousands of contributions from a huge number of contributors, Git was specifically designed to work with big projects and large number of users and activities. Most of the open source communities switched from CVS to Git for its ability to handle large repositories and many contributors, and created web platforms such as BitBucket

and GitHub to share projects and allow users to freely propose their contributions.

These platforms are nowadays used for both open and closed source projects, and feature more than just the source code history: tools such as issue trackers, code compilation and automated tests, chat and messaging, code reviews, product deployment and milestones tracking are examples of what is commonly included. The recorded data contains many different aspects of software development, such as programmers activities, bugs introduction, discovery and fixing, users reactions, and is therefore interesting for researchers. Extracting useful information from this data is, however, difficult: due to its raw size and the variance between the data format in different repositories.

1.1 Impact of VCS mining in research and industrial fields

At the best of the author's knowledge, an in-depth analysis of the impact of mining VCS in the current scientific literature is not available. From the analysis of the state of the art, which is detailed in Chapter 3, it emerges that the research in the VCS area is focusing on answering practical questions that are related to the efficiency and the effectiveness in software development. For example, [50] addresses a problem that can be summarized by the question "Can I foresee a commit introducing a bug in my codebase?". Other works deal with different aspects: "Will my team be successful?" [48] or "How does the programmers handle multitasking?" [76]. In general, the effort of the research aims at providing valuable information to programmers and their team leaders. For these reasons, in this section we discuss an assessment of the impact of the proposed work by considering the size of the domain that could be interested by new discoveries coming from the mining of VCS, namely, the software industry.

The first aspect that can be useful to consider is the number of programmers world-wide. There are different sources of information on this subject, which lead to different views of the aspect. In [23] the world-wide population of programmers is estimated in 22 million persons in 2017, considering 40 different countries and using information from the International Monetary Fund and the World Bank. On the

other hand, the DataUSA website[5], powered by Deloitte, Datawheel estimates the total workforce of software developers, applications and systems software in 1.17 million at the date of this writing[6]. Finally, the Business Software Alliance (BSA) provides data on the total direct and indirect employment created by the software industry. Their analysis report indicate a total of 2.5 million direct and 9.8 million indirect workers in the USA in the year 2016 [3] and a total of 3.1 million direct and 11.6 million indirect workers in EU in the year 2014 [2].

While it is hard to provide an exact number of the users of the Git VCS, companies such as Rhodecode[7] provide charts that supports the idea of Git being widely used. In particular, Figure 1.2 shows the number of questions for several VCS per year on Stack Overflow, a popular community of programmers. The figure suggests that the interest in Git is steadily increasing, while less questions per year are posted for other VCS. On the other hand, GitHub yearly posts statistics of the users activities on a website called Octoverse[8]. While historic data can not be accessed, the latest information regarding the year 2017 indicate a total of 24 million of user accounts and 1.5 million of organizations, providing a rough estimation on the number of programmers using Git.

Tackling the assessment from a different perspective, the impact of the software development industry can be evaluated from some indicators on the size of the market. One of these indicators is the "Gross Domestic Product (GDP)" of the software industry, which measures the monetary market value for all goods produced by the software industry in a given country in an yearly period. In particular, BSA, in the previously mentioned USA report [3] estimates that the software industry did a total of 1.07 trillion dollars of total value added GDP in the USA for the year 2016, with a direct total value added GDP estimated in 475.3 billion dollars for the same year. Given that the same company, in the year 2007, estimated the total value added GDP in the USA by the software industry to be 261 billion dollars [1], the software industry total value added had a growth of more than 400% over 9 years. The EU report for the year 2014 [2] provides similar data, with a direct total

[5]The DataUSA website, last visited on January, 12th 2018: `https://datausa.io`.

[6]The statistics at DataUSA, last visited on January, 12th 2018: `https://datausa.io/profile/soc/15113X/`

[7]Rhodecode VCS surveys for 2016, last visited on January, 12th 2018: `https://rhodecode.com/insights/version-control-systems-2016`.

[8]Octoverse website, last visited on January, 12th 2018: `https://octoverse.github.com/`

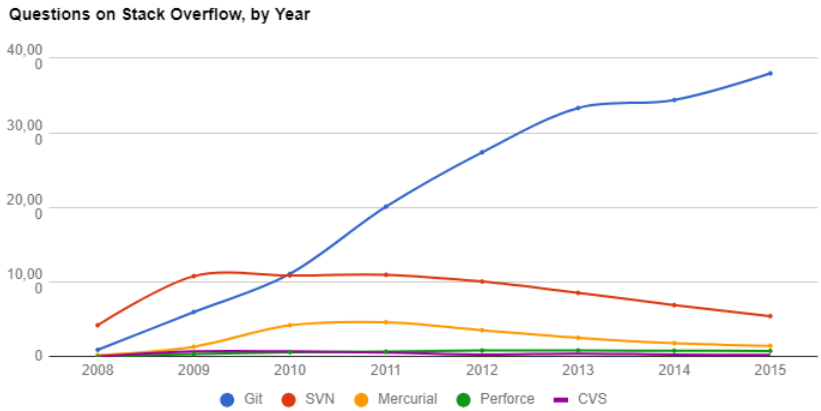


Figure 1.2: Number of questions posted on Stack Overflow by users for a specific VCS until 2016, with Git exhibiting an increasing trend.

value added GDP of 910 million euros, while the indirect total value added GDP is estimated to have been 249 billion euros.

Finally, the existence of several commercial tools to extract insights from private repositories indicate the demand of the market for this kind of tools. For example, the Codacy Automated Code Review tool, from Codacy[9]; Code Climate[10], which is claimed to be used by over 100.000 projects, and analyzing over 2 billion lines of code daily; and Gitential[11].

1.2 Objectives of the work

This thesis focuses on the extraction of valuable information from software repositories managed by the Git VCS. The Git VCS was selected since it is currently the most widely used versioning system, and its adoption in the open source community results in a huge amount of public data available for researches, as shown in Section 1.1. In particular, this thesis applied data mining methods to information from GitHub, which is the most popular platform for hosting and sharing Git projects.

The first objective of this thesis consists in the collection, validation and exploration of different datasets collected through the GHTorrent [33] tool. The experience described may be of use for future research, as most researchers do not describe

[9]Codacy homepage, last visited on January, 12th 2018: `https://www.codacy.com/`.
[10]Code Climate homepage, last visited on January, 12th 2018: `https://codeclimate.com/`.
[11]Gitential homepage, last visited on January, 12th 2018: `https://gitential.com/`.

the collection of their dataset, and may either be using an already available dataset (such as the MSR14) or a complete GHTorrent dump. Other online tools, such as GitHub Archive[12] and Google BigQuery, can be used to access public datasets as well. The author, however, thinks that being able to create a dataset, rather than using a pre-built one, helps in obtaining transparent results, as it ensures that data was not altered by third parties. Further reasons that make the usage of custom datasets valuable are that the complete GHTorrent dumps may be missing recent updates to repositories, that they require powerful machines to be run on, and that they do not contain any private repository. The latter, in particular, may pose a limit for companies that do not develop open source code, and value the secrecy of their source codes.

The existence of problems in the collected datasets, and the discovery of a large number of users with a wrongly set-up Git account, may therefore represent valuable examples for future researches addressing the collection and the analysis of non pre-collected datasets for their studies.

The second objective of this thesis is to propose an approach to achieve a representation of complex attributes, such as "large" and "popular", which people often use when referring to repositories, but cannot be measured by any numeric metrics. This approach includes the aggregation of simple, numeric parameters through principal component analysis (PCA). Considerations can be made through proximity in the resulting maps, i.e. closer repositories are similar sized or have similar popularity.

The third objective of this thesis consists in the analysis, design, implementation and results interpretation of a novel visualization technique, which makes use of graphs to represent users contribution to one or more repositories. Unlike other available tools, such as Gource [18], which are focused on the representation of the activities over a single repository, this methodology aims at studying the "link" between different repositories created by common users' activities. The underlying idea is that the concepts, the methodologies and the techniques are transported among different repositories by common developers, who acquire knowledge on a project codebase in writing their contributions. Said technique is applied on the collected datasets, with results shown and commented in Section 6.2. Results are interpreted and compared to expert knowledge on the projects' history, and

[12]GitHub Archive website, last visited on January, 12th 2018: `https://www.githubarchive.org/`

interesting aspects of the repositories life can be read.

The presented approach, albeit currently using only a limited amount of features per project, scales to large datasets and can be extended to combine different features, in the form of different forms of contribution other than commits). The resulting tool, in its current state, can provide useful information to a team of developers. For example, in Section 6.2.6, it is presented a case where it may help in deciding which library should be included in a project, and Section 6.2.2 presents the scenario of a project ceasing to be actively supported, with developers steadily migrating on an alternative project.

The results described in Section 6.2.1, where a library for web development is used in many web related projects, and Section 6.2.4, where a debug tool was developed for the main project, are examples of how the presented approach is able to provide complex informations, that would otherwise require the expert knowledge on the history of the projects to be acquired. Albeit these results, in the current state of the presented work, requires a human interpretation to be of use, the tool is still useful to researchers as a preliminary step to identify the repositories that match certain conditions.

1.3 Structure of the thesis

To help the reader in having a better understanding of this thesis contents, the basic concepts of both Git and GitHub are presented in Chapter 2. The chapter also describes how repositories attributes can be used as features for data mining. In Chapter 3, a review of the related works is presented. The work is focused on the method applied by researchers to collect useful data from several repositories. Chapter 4 describes the tools used in the context of this thesis, and is concluded, in Section 4.7, with a graph displaying the interactions between each component.

In Chapter 5, the analysis of the 4 datasets used in the thesis is done. The chapter includes a detailed description of the datasets. It illustrates the included repositories and the criteria of their inclusion. Moreover, it describes the datasets format and a statistical analysis of the repositories features. The chapter also includes the description and the results of the visualization technique proposed to compare popularity and size between several repositories.

This thesis proposes a new method to represent information from a repository:

the usage of graphs, both static and time-dynamic, to represent the flow of knowledge existing between different repositories, generated by the users who make contributions. Chapter 6 discusses this approach, from the idea to a working implementation. Several example results are discussed and validated, with comparisons to their actual history. Each specific result presents a peculiar pattern in the interactions between users and repositories. These interactions have been discovered and commented in the text.

Finally, conclusions, critical considerations and possible future works are presented in Chapter 7.

Chapter 2

Git and GitHub

2.1 Basic concepts of Git

The basic operation that a developer performs on Git is the *commit*, a snapshot of the changed files with respect to the previous recorded version of the files, which is recorded with the name and email of the author and the timestamp of the commit.

Simpler projects may be composed by a set of subsequent commits, which can be thought as changes aligned in a sequence on a straight line. More complex projects, however, may have different structures, and may feature several independent "lines" of commits, written in parallel, isolated spaces that are joined later. These "lines", in the Git jargon, are called *branches*, and represent different lines of development. There is always at least one branch, called by default *master*. However, developers are free to create as many branches as they want, and later use a *merge* operation to merge the commits of a branch on another.

Branches can be created starting from any commit on any given branch, and can be later merged on one or more branches. When a branch is merged, the commits are automatically propagated from the branch that will be merged (the *source* branch) to the *target* branch. Sometimes, when the same file has modifications on both the source and the target branch, Git is not able to produce a reliable automatic merge, and presents the user with a *merge conflict* error, requiring him/her to manually fix the ambiguities and produce a merge conflict fixing commit. Fixing a merge conflict may require a huge effort in case the source and the target branch diverged significantly. Therefore, it is advisable to have short lived branches. On the other hand, having to solve conflicts on every commit may distract the developer. For this

reason, it is advisable to work on atomic branches, that will be merged when their specific purpose has been fulfilled.

Most repositories adopt rules to determine how branches should be created and merged. Such set of rules are called *Git flow*. There are several different commonly used Git flows, and online platforms such as GitHub propose their own featured flow that works best with the tools provided by the platform itself. As a commonly followed convention, the master branch represents a stable version of the software and should not be tainted by non compiling/non working commits. Pushing commits directly on the master branch is therefore considered a bad practice, but researchers have to account that a considerable amount of developers is still only using the master branch on their repositories, especially for projects with very few contributors. Features and bug-fixes may either be done on separate branches (feature branches) or on a single development branch; in the first scenario the branch is merged when the atomic purpose it serves is completed, in the latter when it reaches a stable state and the release goals are fulfilled. In bigger projects, the two approaches are often mixed: there are several *long life* branches, such as *release*, *release candidate*, *beta*, *alpha*, *development* and feature branches are used to push new content to *development* or hot-fix the bugs. Branches may be merged on the *release* branch after some criteria are met, such as *all tests pass*, code reviews, etc. Ultimately, it is the project manager who chooses which flow is appropriate, and since the flow is a set of rule he/she must also enforce them on other developers.

Git also allows to *tag* important commits, i.e. attaching a label (the tag) to a commit to easily identify it when reviewing the project history. Git tagging feature is often used for long term support to released versions. In many scenarios, new features are introduced in future releases of the software. However, security and bug fixes must be provided for previous versions as well. Given that the fixes were developed in self contained branches, Git tags simplify this process, since it is sufficient to merge these over the tagged commits of the previous versions to propagate the fixes.

An illustration of a fictional Git repository created to illustrate this concept can be seen in Figure 2.1. In this example, commits are represented as colored dots, in a top-to-bottom timeline. Each vertical line corresponds to a branch.

The project is created with a first initial commit having a commit message equal to *Initial commit*, with the commit unique identifier *f157645*. The unique identifier is a hash calculated using the *SHA* algorithm, which identifies uniquely the snapshot of

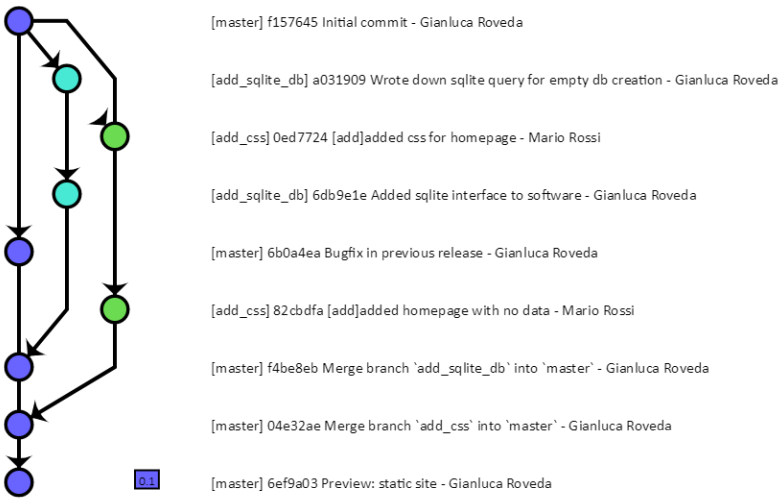


Figure 2.1: Sample Git project representation; from the top two branches (add_sqlite_db and add_css) are developed in parallel by different users and later merged on master.

the content. Two feature branches are created from the first commit: *add_sqlite_db* and *add_css*, represented with the teal and green dots respectively. The work is made in parallel on the two feature branches by the two different developers. Meanwhile, one of the two developers also pushed a hot fix commit on the master branch. Notice that merging the branches created new commits (namely *f4be8eb* and *04e32ae*) on the master branch. The final commit (*6ef9a03*) contains an important version of the code, which may be of interest in the future. Therefore it has been tagged as *0.1* version of the project.

Non fictional Git repositories are often much more complex. As an example, Figure 2.2 depicts a graphical representation of the Inkscape project between December, 30th 2017 and January, 1st 2018. A total of 18 commits are divided in a total of 9 branches. The red line on the right represents the master branch of the project. The remaining branches are a mix of long-lived branches and feature branches, some of which are merged into the master branch. For example, the green branch contains a fix and is merged into master. Notice how the purple branch is also merged in the green branch before being merged into master, and that master itself is being merged over other branches. The latter usually happens when fixes from the master branch are needed in a feature branch, or when the feature branch has diverged too much

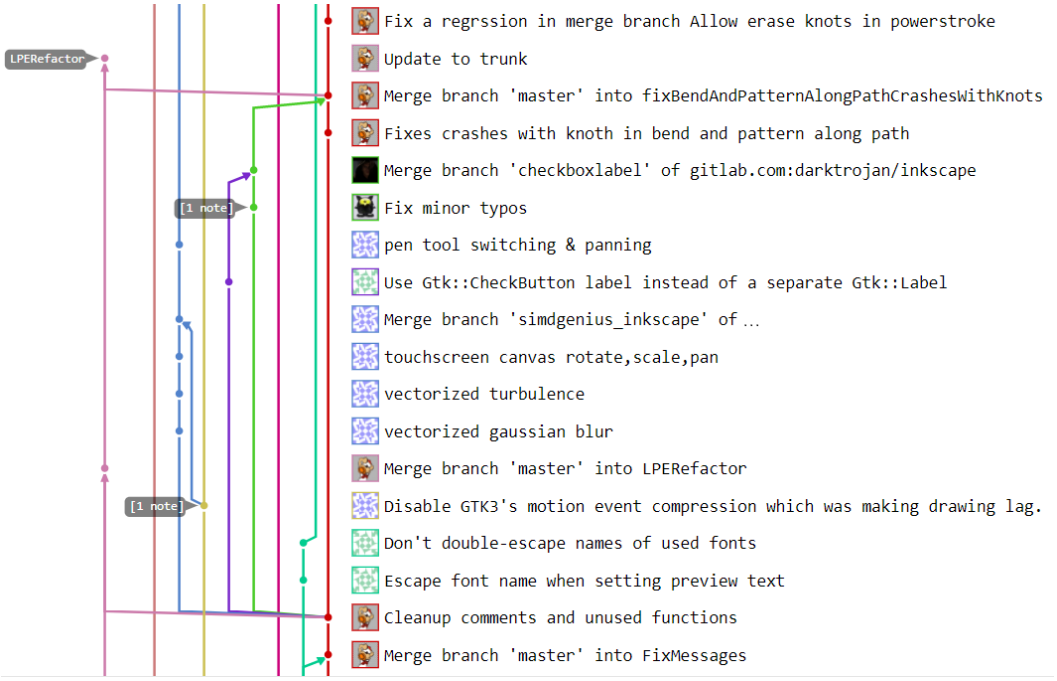


Figure 2.2: GitLab graph representation for the Inkscape project shows a more complex usage of the Git VCS. Unlike the fictional example of Figure 2.1, this is read from bottom to top.

from the master branch and the developer wants to incorporate missing commits to avoid integration hell. It can also be noticed that, mostly, a single developer is working on a single branch, that two fixes and a cleanup commit were made directly on the master branch by a single user, which is a project maintainer.

In its internal representation of the content, Git uses a format that is quite different than what has been presented here; this summary presents Git from the developer's standpoint, and it was written with the purpose of guiding the reader to a better understanding of the following chapters.

2.2 Basic concepts of GitHub

GitHub is a web platform that hosts private and public Git repositories. A key aspect, which is fundamental to understand several analyses performed in this work, is that GitHub promotes the so-called *fork and pull* model. Under the *fork and pull* model, every user is free to create a copy of a public project, called *fork*. The developer can freely modify the forked repository. Moreover, he can eventually propose modifications

to the original (forked) project through the so-called *pull requests* (hence the name "fork and pull"). When a pull request is opened, a notification is sent to the project maintainers. The maintainers review the code and eventually start a discussion on its content. The opened discussion is public: any other GitHub user with access to the project is free to participate, albeit the final decision of integrating the pull request contents (*accept*) or discard them (*close*) can only be taken from one of the project maintainers. A pull request discussion may also result in further modifications of the contributions, until it is either accepted or closed.

When a repository is hosted on GitHub, it is integrated with several other tools such as:

- an issue tracker;
- code review and discussions for pull requests and commits;
- statistics on continuous integration;
- project metrics;
- documentation (in a dedicated Wiki);
- social context (star projects and users, notifications etc.).

2.2.1 Issue tracker and code review

The project issue tracker is where developers, contributors and external testers can report bugs or request for new features. The issue tracker is organized into messages in a forum-like style, where further details may be requested, such as details of the bug, possible ways to fix it, details of the proposed new features, their effective value and pertinence to project and such.

Since the issue tracker may contain thousands of open issues, the project owner can define custom labels to help people navigate through the tracked issues. Example of such labels may be *bug* and *feature*, but also *newcome friendly* to identify easier issues that may be addressed by less expert contributors, or *question* for user questions on the project.

When feature branches are used, a branch can be linked to an issue to follow the progresses made by the developers.

Many projects do not use the GitHub issue tracker, either since they do not use any issue trackers at all, or because the project is configured to use an external issue tracker.

Code review is used in pull request discussions to contextualize comments. The preview of the resulting branch is shown, and the differences with respect to the current target branch are highlighted in green (additions) or in red (deletions). Users can write a comment on a portion of *code diff*[1], and the comment is shown in the pull request discussions. The code review tool can also be used outside of a pull requests, for example to show the differences between a previous version and the current version of the code.

Both in the issue tracker and in the code review, a user can be marked as the assignee. It is expected that an issue assignee is who will work in the next step on the issue completion, usually by writing code, and that a pull request assignee is who should review the code and decide whether or not to accept the pull request. To notify a user that his/her interaction is required on an issue/pull request discussion, it is possible to write his username in the form of *@username* in a comment. A notification is then sent to that user to notify he has been mentioned, along with the discussion link.

2.2.2 Continuous Integration and project metrics

Continuous Integration (CI) is a development methodology that belong to the class of the extreme programming. CI is used to avoid the perils of stale branches and integration hell. Stale branches refer to branches that either contain too much code or diverged too much from master, and are therefore difficult to merge. The integration hell, on the other hand, refers to branches that require more work to be merged than to have their content re-write from scratch, due to the master branch being changed too much during the branch lifetime. Continuous Integration as a programming technique requires developers to work to small, atomic feature branches that are tested and merged frequently into the master branch.

Long term pull requests discussions are not suitable for CI, since they would require too much time and slow the developers' work. On the other hand, it is necessary to ensure that any code that enters the master branch is correctly working

[1] A code *diff* is the portion of different code between the stored snapshot of the project and the current version of the files in the working directory.

and do not introduce bugs. As a results, tools such as Jenkins and Travis were invented to automatically compile the code and run developers written tests.

GitHub can be configured to interact with such tools, and automatically run tests when a pull request is opened. The tools results (compiling or not, list of passed tests) are reported in the pull request, and can be used to speedup the code review process significantly. Even when CI methodologies are not used, the CI tools integration provide value to the project maintainers, and ease the job of evaluating a pull request. Still, since GitHub itself does not run any CI tool, many projects are not configured to use CI tools.

GitHub also features a collection of project metrics called *GitHub insights*. The metrics offered are on a project level, and include contributors' activities, commits distribution over time and a pulse check overview, with the summary of recent activities made on the project.

2.2.3 Documentation and social features

GitHub offers an in-built wiki platform to host the project documentation in markup language (Markdown, RST, Textile etc.). The wiki is basically a Git repository itself, and can be accessed through Git for offline working. It lacks, however, the features offered by GitHub such as issue tracker and pull requests.

As the issue trackers and CI, many projects do not use GitHub wiki for documentation. The majority of the projects, however, do host some form of documentation in the form of one or more Markdown files. The most commonly used one, *README.md*, is by default shown by GitHub as the project homepage. The README.md usually includes the project description, setup and contributions guide, in an unstructured format. The README.md may include links to other Markdown files, so it is possible that it acts as the homepage for the whole project documentation, and that this is contained in the project files and not in the wiki.

GitHub allows some forms of social interaction, such as following a user to receive updates on his/her activity and star a project to receive information on the project. These features generate data that is interesting in analyzing social behaviors.

Feature	Usage
Commits	Commit content (author, time, sha) for both statistics and graphs, Commit count: metric for amount of user work
Branches	Not used
Tags	Not used
Issue tracker/pull requests	Statistical considerations
Forks	Identify active/passive users
CI	Not used
Documentation/Wiki	Not used
User rating	Statistical considerations

Table 2.1: List of features and their usage in this thesis.

2.3 Data mining of Git and GitHub features

Sections 2.1 and 2.2 present an overview on how Git and GitHub are used by developers. Regardless on the tool used by researchers to gather the data, the concepts that have been explained are translated into collections of objects or database tables. These objects, and their properties, are the features used in the works based on MSR. Chapter 3 presents an overview of such related works, and each of them is mostly focused on analyzing a specific set of features.

Basically, in current scientific literature, most works focus on analyzing a specific aspect of repositories, e.g., social interactions using issue trackers and pull requests, or distribution of commits over the day using only the commits.

This section will provide an overview of how the concepts presented in this chapter can be used to obtain useful features, and, for the features that was used in this work, what is their role. For convenience, a summary of the features and their usage in this thesis is shown in Table 2.1.

Commits

A commit contains a number of precious information. The commit author, the message that was written by hand by the developer writing the commit, the date of the commit and the differences of the project files with respect to previous commits

are included and can be analyzed to derive useful data. The work carried out in this thesis, and in particular the algorithm of Chapter 6, is heavily based on commit objects, and in particular to the commit authors and commit timestamps.

By analyzing the commit message, the occurrences of merge conflicts can be detected. Merge conflicts are interesting features that motivate several studies. Due to their negative impact on the developer's productivity, it is worth to study their occurrence and the patterns that lead to conflicts. The study of conflicts, however, is outside the goals of this thesis.

The commit unique identifier, the so-called *sha* field, can be used as a check for a collected dataset: no duplicated commit *sha*s should be found in a valid dataset. A commit also includes its parent(s) *sha*, allowing for the reconstruction of the commit tree. These observations, however, were not relevant in this study, which is assuming that the content of the dataset is reliable.

The count of commits performed by a user, or that was made for a repository, is a feature as well, as it reflects the amount of development that a user made or that was made on a repository. This feature was widely used in the present work for a statistical analysis of the dataset (Chapter 5) and in Chapter 6.

The main advantage of conducting the analysis on the basis of commit-related information is that commit objects are included in every repository, and that regardless of the tool used for gathering the datasets, commits data will always be available.

Branches

Whether branches are used or not in a project is an important indicator by itself. Interesting evaluations can be done on the patterns used when working with branches (*Git flow*), the number of commits in a branch, and the lifetime of the branch. This work does not consider any feature related to branches, albeit Chapter 3 includes many studies focused on branches.

Tags

Git tags are a useful feature to study the evolution of software projects. One of the problems that need to be faced when building an analysis based on tags is that they are not always used by project maintainers. Moreover, their usage is

not standard. This means that, even when they are used, they may be adopted for different purposes. For this reason, this work does not make use use of tags information. This is also motivated by the fact that the tool used to gather the datasets, i.e., GHTorrent (see Chapter 5), does not support them in the collected data.

Issue tracker and pull requests

The number of open, close and total issues contained in an issue tracker are indicators that reflect the activity of users on that repository. For each issue, the discussion activities that are generated from the users contain features, such as the discussion timestamps and the discussion messages, that were studied in literature. For instance, sentiment analysis can be conducted on such data. This work considers the number of issues in a repository as a feature to compare repositories when studying the datasets.

Using data from the issue tracker as features, however, suffers from the problem that not every repository use an issue tracker. Some repositories do use external issue trackers rather than the GitHub integrated one.

Pull requests offer similar features and problems, with the bonus of having the proposed code differences as a feature (code review). In this thesis some considerations are made on the authors who were active in discussion within the issue tracker vs. the authors who prefer to discuss in a pull requests. These aspects are leveraged in Chapter 5.

Forks

The number of forks of a repository is a feature that reflects how many users are interested in the project. Moreover, the commits made on a fork can be compared to those made on the corresponding original repository to evaluate how the developments are influenced by external project contributors.

The results derived in this thesis do consider the number of forks of a repository. Moreover, the concept of *active fork* is introduced, which is defined as a fork where at least one contribution is propagated from the forked repository to the original one. In Chapter 5, active forks are used to differentiate between *active users*, users who actively contributed to the original repository with code contributions, and

passive users, users who forked a project but never pushed code contributions back to the original repository.

Continuous integration

Continuous Integration (CI) data include interesting results on the build process and tests passed, which can be used as starting point for interesting researches. However, many projects are lacking CI data, since CI requires a dedicated machine to be configured and connected to the repository. This forces CI studies to be built around ad-hoc datasets. For this reason, the work presented in this thesis does not consider CI data.

Documentation and user evaluation

The existence of any form of documentation, and the percentage of the codebase covered by documentation, are features that are studied by researchers interested in software engineering aspects. Documentation, however, was not relevant in this thesis.

On the other hand, the total number of stars a project has received is an indicator reflecting the popularity of a repository. The whole history of "who star what" is recorded and can be used as a useful information to study, for example, the influence of popular users on his/her followers. The assessment presented in this thesis uses the number of stars as a measure of popularity when comparing different projects.

Chapter 3

Related works

Software repositories are a source of precious information for project managers and researchers, and have been studied since the beginning to extract useful indications regarding the evolution of a project. The increase in usage of VCS and the growth in Open Source Software (OSS) communities encouraged more researches to address the issues related to this field. As can be seen in Figure 3.1, the number of papers published on the topics has a strongly growing trend.

This chapter covers the most meaningful papers focused on GitHub and Git repository mining in general. The papers published between 2005 and the first half of 2016 will be considered. Figure 3.2 shows that these papers also have a growing trend, similarly shaped to the more generic VCS mining papers.

The first objective of this work, as stated in Section 1.2, is to mine data from a set of GitHub-based datasets, and provide a graphical approach to suitably compare different repositories. The availability of a graphical method for comparison purposes is valuable as a tool for researchers and developers to conduct preliminary analysises on a dataset, as it may be used to understand which repositories are the best fit to match the given criteria. As an example, a developer who requires to select the best library for his project may need to compare tens of alternatives. By providing him a tool to quickly identify the bigger/most popular projects, he may choose to focus his attentions only on a limited amount of such projects, and will save him the time he spent on less important projects. It is worth to note that a graphical approach is not the only possible way to achieve this result. For example, one could think of building a recommendation system that outputs the most meaningful repositories. The graphical approach, however, carries more information, while it is not allowed

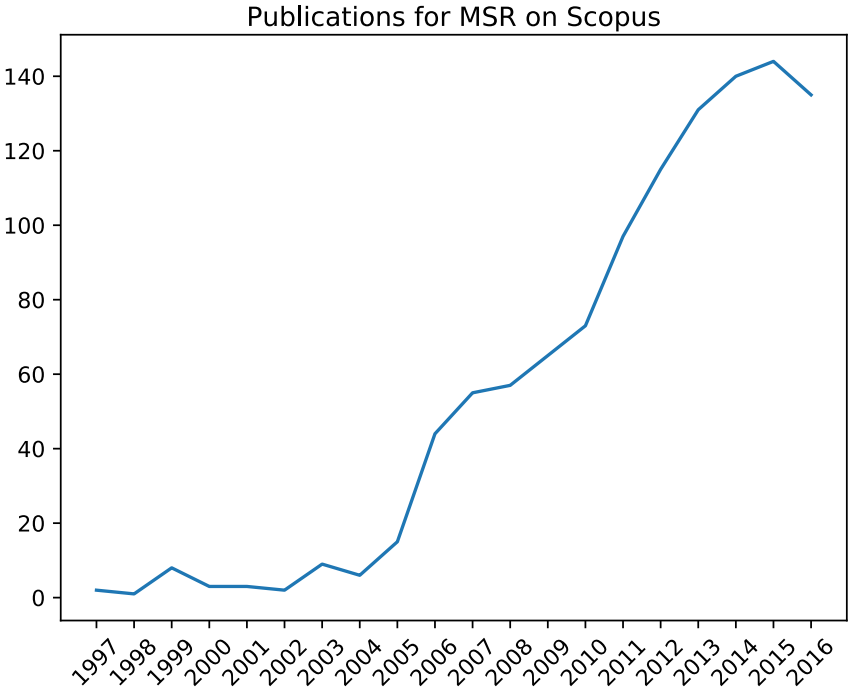


Figure 3.1: Number of MSR papers published per year.

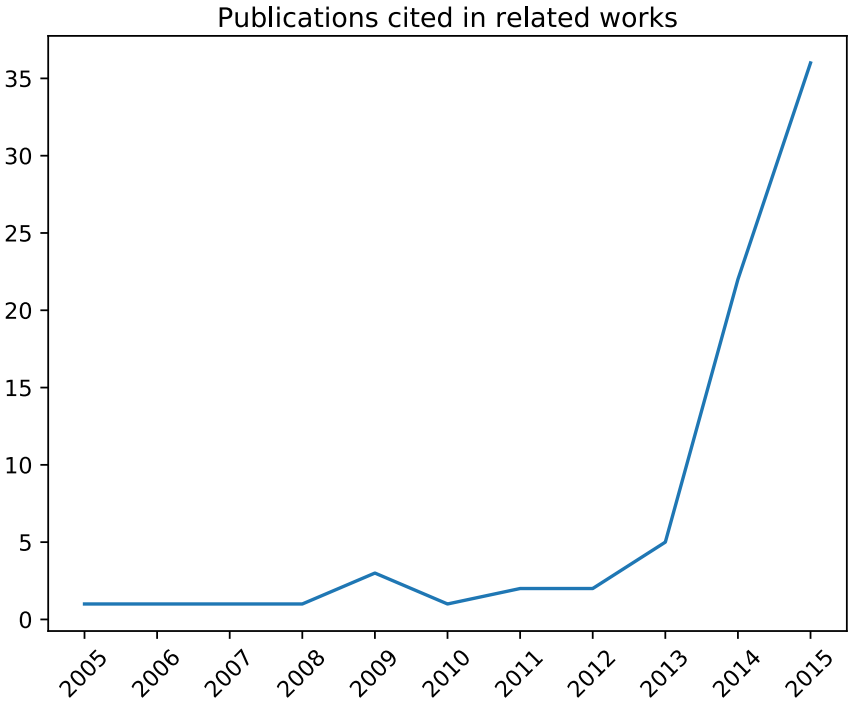


Figure 3.2: Number of cited MSR papers published per year.

to autonomously select which projects should be discarded.

Several previous works have studied GitHub datasets, albeit at the best of the author's knowledge no existing work was done with the goal of providing a graphical representation for comparison purposes. The tool used to collect the dataset, GHTorrent, is presented in [33].

The second objective of this thesis is to provide an approach to analyze the flow of the commits and the contributors in common between repositories, as a way to analyze the relationship between them and their history. Again, no papers were found in the literature that analyze this aspect on different repositories. The closest work is presented in [18], which describes a tool to achieve a rich visualization of a repository history. The aforementioned work, however, does not take into account the interactions among different repositories.

Out of the four datasets used (see Section 5.1), only the MSR14 was widely investigated in scientific literature, and several important aspects and issues were derived in previous researches. The MSR14 dataset is also addressed to apply the analysis developed in this thesis.

3.1 Git repository and their mining

This section discusses several previous works on Git repository analysis and data extraction. The considered existing works are grouped by topic to ease the presentation and to categorize the different investigated aspects.

VCS usage by feature

Researchers have studied how people use the features that Git-based VCS offer. Some works consider how branching and forking are used by developers [9], where the authors compare how branches are used by users that migrate to DVCS. The paper concludes that DVCS usage results in less "interference" and more cohesive branch w.r.t. CVCS. Interference is used in this context to define conflicts in code changes when merging different branches; conflicts are time-costly to fix and are therefore undesirable. A branch is called *cohesive* by the authors when its changes are focused to a single aspect of the project, e.g., fix a specific bug or implement a new feature.

In [13] the sparsity of data over forks is considered and measured using a

proposed set of metrics, while in [12] topology techniques are applied to branches in the repository and recurrent patterns are discovered and analyzed. The research in [43] considers why and how developers fork from other GitHub projects. The authors conclude that repositories are forked to propose code contributions through pull requests, and mostly the fork is done on the original repository.

In [4], the authors present an analysis on the size of the contributions for 9000 code contributions from OSS projects. The article proposes an approach to differentiate over three types of commits: *single focused* (a single passage in the developments of a single contributor), *aggregate team contributions* (aggregated chunk of code with different goals) and *repository refactorings* (merging of different revision systems, initial check-in of projects, etc.).

VCS usage by user category

A research trend focuses on how VCS are used by a particular category of users. The authors of [21] consider how computer engineering students use VCS, their usage of branching and the amount of garbage[1] that they include in their working directory, while [68] further strengthen this study by correlating it with code quality and peer review considerations. In [51] studies are presented on users who exploit the features of VCS for text editing, and the strength and weakness of VCS for collaborative text editing are analyzed in comparison to alternative solutions[2]. The authors in [45] make considerations on how GitHub, a platform born for hosting open source software, can be used effectively for commercial softwares, and use surveys and interviews to inquire developers on its usage in commercial software. The answers show that typical practices of OSS community, such as self organization and reduced communication, are adopted in commercial projects as well. In [17] there are considerations on how digital music maker use GitHub as a platform for collaboration. The main result shows that less commits are produced during the week, and more commits are produced during the weekends w.r.t. software repositories. Moreover, these results show that digital music repositories have similar bug reports and contributions to software repositories. Finally, [35] analyzes the contributor's

[1] Garbage content in a repository consists of generated files that should not be committed

[2] A private instance of GitLab was used to coordinate the development and the review of this thesis, while keeping track of versions and changes. This thesis is thus an example of the result obtained by the use of VCS in collaborative text editing.

perspective in collaborating on a GitHub project, with results focused on problems in communications and social aspects, and results in a set of useful recommendations for practitioners.

Impact of pull requests

Several aspects of pull requests have been studied as well. The authors of [62] provide insights on around 78.000 accepted and rejected pull requests, and the ~20.000 developers who made them. The insights include interesting observations. For example, it is pointed out that "the more experienced developers are not necessarily the ones who produce the most commits/pull requests merged". On the other hand, the authors state that "many pull requests do not succeed due to a few unresolved technical problems". In [79], the authors consider factors that affect the closing time of a pull request. They provide several statistics, and conclude that it is difficult to estimate in advance the time required. The single most impacting factor is the number of comments to the pull requests, but this information is – of course – not available in advance at the opening of a pull request. Therefore, it becomes hard to reliably predict the time required to address a pull request. In [38] it is shown that, if the coding style of the contribution is similar to the codebase, then the pull request is more easily accepted than rejected. To achieve this result, language models were used. The authors also noticed that most of the code review effort is spent on the more dissimilar portion of the code reviewed. Some considerations on the aspects of conflicts that arise during the code review phase of the pull requests are studied in [40]. In particular, the authors compare three possible strategies, namely *rational explanation*, *constructive suggestion* and *social encouragement*, and discover that only constructive suggestions are helpful for not losing contributors when pull requests are not accepted.

Considerations on social aspects

Social aspects related to Git-based projects have been also investigated. In [24] the authors present a tool that automatically evaluates the bus factor of a project. The bus factor is defined as the number of developers that must be struck by a bus in order for the project to stop due to lack of knowledge. This is a useful metric for project manager to prevent missing deadlines due to accidents/illness and to better

manage programmers turnover.

A revisit of the concept of Pareto principle applied to software development is presented in [78], where the authors suggest that the Pareto principle does not seem to apply for software project. They do notice, however, that many projects have a very low bus factor, and that a core developer leaving the project may have a strong negative impact on the project "survival". The authors of [48] evaluate the reasons that make (or prevent a group to be) a successful team, studying projects on GitHub and concluding that small teams, possibly even composed by a single person, tend to achieve success more often; therefore the authors conclude that it is advisable to break a large team in smaller and more focused teams. Popular users, i.e. users with a high number of followers, are considered in [15]. Digging through what makes them popular, the authors conclude that understanding popular users is important since, as stated in [65], they tend to attract contributors to projects where they collaborate (social contagion). In [8] the authors study the dynamics of users influence, by exploring 3 metrics that measure popularity: number of followers, number of forks to one's projects and number of project watchers. They find that these metrics do capture both popularity and – rather unexpectedly – code reusability. The usage of CI services is analyzed in [75]. Among the findings, the authors discover that direct code modifications (commits) have higher chances to not compile or not passing tests w.r.t. indirect ones (pull requests). The research in [58] aims at providing a comparison method between developers by means of specific metrics, such as code survivability[3]. The authors claim that the proposed method matches the project manager's expectations in a real world case study for the most productive developers, but does not necessarily produce accurate results for the least productive ones, for which human investigations are needed.

In [16] the authors analyze the differences in the usage of CVCS and DVCS by the users, and discover that DVCS highly correlates with smaller, atomic commits and in DVCS the commit messages has higher chances of containing references to issue tracking tool entries. An analysis of the usage of Gists, which are the GitHub equivalent of code snippets, is shown in [77]. The authors investigate both how they are perceived and what they are used for. They conclude that Gists are not widely used. Moreover, users using Gists mostly have a few snippets published, and often

[3]Code survivability is defined as the amount of code that survived during the life of the project vs code that was deleted or changed

made by a single file.

In [59] the authors consider the usage of the *goto* function in C programs, which is often considered a bad programming practice. Their findings show that *goto* is actually rather popular, but in most cases its usage is appropriate. In particular, it is used in contexts – such as error handling – where it does not jeopardize the code maintainability. The authors of [28] find a correlation between the number of files per commit and the dependencies among them. In [76] the authors study how users collaborate on multiple projects at the same time (multitasking), identifying several behavioral patterns and trying to understand the impact of multitasking on code quantity and quality. One of the findings is that users who do multitask do not seem to perceive their own limits, and end up lowering their productivity. The authors of [66] focus on analyzing the sentiments from commit logs. Most of the commits have a neutral sentiment, with a strong correlation between the number of files changed and the sentiment expressed in the commit log.

Bug handling

Some studies focus on software bugs. In [50], an approach is presented to automatically determine commits that caused bugs and the corresponding fixes (named *fix-inducing changes*). The authors correlate commit messages with issue tracker (Bugzilla) to perform a syntactic and semantic analysis on the commit message. The authors of [5] consider several problems that may invalidate the results when mining software repositories for bugs. The main correlation is with social, organizational and technical factors that are external to the issue trackers. Therefore, the paper presents strategies that may mitigate the errors accountable to these factors. The paper also features a bug history discussed with professionals to present patterns that may be used by future researchers.

In [54] the authors present a tool that generates a report on bugs introducing commits and fixing commits, as well as computing several Object Oriented (OO) programming language metrics. A correlation between the main programming language and the number of bugs, for several GitHub projects, is investigated in [63]. The results show that the type of language used in the project has a modest impact on the quality of the resulting software. The authors of [27] find a correlation with the time associated to commits and the probability that the commit introduces a

bug. Similarly, a correlation is found between the day of the week and the introduced bugs. In particular, commits made at late night have higher chances of introducing bugs, while smaller commits are generally safer than bigger commits. Moreover, different projects have different daily distributions of bug-inducing commits.

The work in [30] is focused on development turnover in OSS world, and its impact on overall code quality. External and internal turnover are used, respectively, to indicate contributors leaving/joining the project and contributors focusing on different aspects of the same project. In [37] a database collected from 13 Java projects is presented, with an authors handmade characterization of bugs to be used by future researchers.

Tools and frameworks

Some studies present tools that can be used by MSR researchers. Gousios et al. present both Alitheia core [34], a platform for software analytics, and GHTorrent [33], a tool to create a snapshot of GitHub to overcome GitHub API requests limitations. Gousious also created an online, public, collectivity-gathered dataset obtained by GHTorrent [36]. Alternative platforms to analyze Git repositories are presented in [19] and [25]. In [31] a 3D visualization tool for Git repositories, designed for educational usage, is presented. The author of [10] present a framework for applying big data technique to MSR. Torch [72] is a tool for code reviewing, particularly focused on object-oriented programming languages.

Surveys

In the literature there are several surveys on the topic, from pre-2006 surveys [44], to more Git-specific ones [14]. Follow ups of [14] are reported in [47] and [46].

The work in [39] reports a set of suggestions for young MSR researchers, based on papers from 2004 to 2012. In [20], the authors present a comparisons of tool used by researchers in MSR, along with their application. A comparison on the limitations of 93 different papers in the topic of MSR, in terms of data size, replicability and other factors is studied in [26].

Other studies

Further studies involve working on fixing logical problems with the obtained data, such as [69] where an approach is presented to detect file renaming and [80] where incorrect tasks completion times are detected. In [57], an approach is presented to track software performance along different revisions, together with a test-case study. The authors of [42] present metrics to evaluate software quality, and [56] presents a big data approach to discover architectural software designs. In [71], a manual emotion analysis on over 800 issue comments is performed, to analyze the possibility of automatic emotion analysis, and concluding that it may be feasible for some kind of emotions (such as love, joy and sadness) and impossible for others. The authors of [32] conduct a survival analysis of 5 database frameworks used in 3707 Java projects from GitHub. The work in [70] represents activities in a repository as a function, and combines it with events that occurred to the programmers in order to classify the effects of technical debt. A social diversity analysis on the GHTorrent dataset from 2014 is conducted in [74], while the authors of [49] present a model to estimate the growth stage of OSS projects based on project activity and project size.

In [7] a possible approach to evaluate a project bus factor is presented and validated on several GH repositories. A survival analysis on GH projects based on both project features and project members is conducted in [53]. In [6] the authors analyze how exception handling is used in Java projects, and discover that in it is often used improperly.

Chapter 4

Tools used

This chapter provides a reference for the tools used to obtain, analyze, process and visualize data in this thesis.

4.1 Python

Python is a high level, object oriented, interpreted programming language. The first version of Python was released in 1991 by Guido Van Rossum, and nowadays it has become one of the most popular programming languages, according to trend analyzing websites[1] [2] .

What makes Python a widely used programming language is the high number of available libraries, covering most of the high level applications, making it suitable for many tasks.

In the academic world, Python is used for data analysis [60], machine learning [61], molecular biology [22] and more [55].

4.2 MongoDB

MongoDB is a non relational, non SQL, document oriented database. Unlike SQL based databases, data is not represented as a set of entry in fixed structure tables, but is gathered in *collections* of *documents*. Documents inside a collection are not

[1] Code popularity of programming languages in 2016, last visited on January, 12th 2018: `http://blog.codeeval.com/codeevalblog/2016/2/2/most-popular-coding-languages-of-2016`.

[2] Programming languages trends, last visited on January, 12th 2018: `http://pypl.github.io/PYPL.html`.

constrained in sharing the same structure, but may have arbitrary fields and subfields. Although internally data is represented in BSON for performance improvements, documents are presented to user as JSON objects, and JSON can be used for querying data for fields values and regular expressions. In general, MongoDB allows to handle easily flexible/complex structures, with good performances on the insertions [73]. Performances on retrieving data, with respect to SQL databases, greatly depends on how the data is accessed and indexed, as well as the hardware setup of the machine hosting the database (the datasets indexes must fit in ram for best performances). MongoDB offers two paradigms for high performance data retrieval: aggregation and map-reduce.

Aggregation is a multi-step pipelined framework of simple commands, such as filtering, grouping and sorting, and offers the best performances. On the other hand, aggregation is limited in its capabilities to the commands offered by the framework, and the user cannot run a custom coded command step in the pipeline. For its flexibility, in this work the map-reduce paradigm was used instead for querying aggregated data.

4.2.1 Map-reduce

From the official MongoDB documentation[3]: *Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results.* Map-reduce can be seen as a three step pipelined aggregation (map-reduce-finalize), with the first step used to associate (map) values to a key, the second step reducing the multiple values associated with that key to a single value and the third, optional step to apply a transformation to the results. Unlike the aggregation framework, however, the three steps are Javascript function which can contain arbitrary code.

As an example, suppose that one wants to compute the number of commits per repository. The map function should take a commit object, find its repository full name, that will be used as the key, and map the value 1 (this commit count) to the repository key.

```
var map = function () {
        var url = this.url;
```

[3]MongoDB documentation on map-reduce, last visited on January, 12th 2018: `https://docs.mongodb.com/manual/core/map-reduce/`.

```
        // We want to extract the full repository name from URL
        // First we remove the commit sha trough regexp
        var regexp = /(.*)\/[^\/]+/i;
        var result = regexp.exec(url)[1];

        // Then we remove the rest of the URL stuff from left and right
        result = result.replace('https://api.github.com/repos/', '');
        result = result.replace('/commits', '');

        // Finally, we change the dot from . to utf8 encoding
        result = result.replace(/\./g, '\uff0E');

        // This repository now has +1 commit count
        emit(result, 1);
}
```

The map function is called for each commit in the *commits* collection, and for each commit extracts the full repository name through substitutions and emits the repository name (the key) and the commits count - which is always 1 for a single commit. The reduce function also remove any dot (.) character and replaces them with the UTF8 equivalent ('\uff0E'), since the dot is an allowed character in GitHub repository names, but creates many problems in MongoDB, where it is used to indicate subfields in keys.

This example shows why map-reduce is more flexible than aggregation: several non standard operations must be conducted in the map operation to extract the repository name from the commit URL.

The reduce function will receive a couple of key (repository) and values (an array of values, each being either 1 or a partial result produced by a previous reduce call). Therefore, the reduce function must simply sum the values received from the map:

```
var reduce = function (key, values) {
        return Array.sum(values);
}
```

The result can either be consumed immediately or saved in a new collection for future usages.

4.3 KNIME

KNIME [11] is an open source platform for data analytics released in 2004, with machine learning and data mining components. It allows for the creation of data pipelines through a graphic interface which represent each pipeline step as a node. Many nodes come pre-built in KNIME: nodes for creating data table from files, nodes from managing the data, filters, statistics and data representation. Moreover, KNIME have a repository of community made extensions to integrate new nodes.

KNIME can also include WEKA and R nodes, and allows for the inclusion of custom Python, R and Java code. Custom code can be written inside special nodes which handles the interfaces required to interact with incoming data and propagate the results to following nodes.

Among the many available nodes, KNIME also features a set of nodes to interact with MongoDB, making it a good choice for mining GHTorrent data.

4.4 Gephi

Gephi[4] is an open source tool to visualize and analyze graphs and networks. The authors define Gephi as *fast*, *simple* and *modular*. These three concepts illustrate what makes Gephi different than alternatives, and the reasons why it may be the best alternative in several contexts:

- it is *fast* since it uses OpenGL for rendering networks, using graphics card for the rendering. This allows to use Gephi for rendering networks with a large number of nodes. Renderings are made online, to have a live preview while running node positioning algorithms;
- it is *simple* because the live preview allows to tune the algorithms during the run, thus having a better understanding of the influences of parameters on the final results through trial and errors;
- it is *modular* because it can be extended through plugins, therefore serving for many purposes.

Unlike many alternatives, Gephi is not powered by a graph database, but is focused on the visualization and positioning of the nodes. The positioning, in

[4]Gephi on GitHub, last visited on January, 12th 2018: `https://github.com/gephi/gephi`.

particular, can be done by many algorithms, but the default one, ForceAtlas (versions 1 and 2) [41], is oriented at online node positioning, with nodes repelling each other and edge between nodes acting as a spring, attracting the connected nodes with force proportional to the edge weight. The algorithm is not guaranteed to converge to a stable situation, and in general it runs unless the user manually stop it after achieving the desired results.

One of the available plugins, called GephiStreamer, can be used to stream data from Python, thus making it a good combination with Python/KNIME.

Finally, Gephi supports the representation of dynamic graphs, through a moving window over a time interval. Both a node existence and the node's properties can either be static (a single value) or dynamic (an array of timestamp/value pairs). If the node property is dynamic, Gephi will use values inside the current time window to represent the node.

Dynamic graphs are still an unstable feature, and their use often results in crash and software instabilities. Nevertheless, they are still able to provide interesting and unique results for displaying the evolution of the data over a certain period of time.

4.4.1 ForceAtlas2

To make the work self-contained, this section reports the details of the behavior of the ForceAtlas2 algorithm that is used to place a node in the 2D graph, originally proposed in [41].

As aforementioned, ForceAtlas2 is a node layout positioning based on forces and repulsions, specifically designed to be used in Gephi. When executed, the algorithm does not stop automatically, but rather runs infinitely until the user manually end the positioning. While the algorithm is executed, forces and repulsions are computed and applied continuously. Depending on the nodes and the edges, it is possible that the resulting graph does not converge to a stable situation. The main advantage of ForceAtlas2 over many alternative positioning layouts is that the user may see the node positioning during the execution of the algorithm, and even interact with the nodes or change the settings without stopping the execution.

The positioning of each node depends only on the other nodes positions and on its connections with other nodes. Node properties do not impact the node positions. For this reason, as stated by the authors in [41], "the result cannot be read as a

Cartesian projection": a node position makes sense only when compared to other nodes, and the resulting graph may be different over executions. This makes the result produced by ForceAtlas2 not deterministic. Moreover, the algorithm may be stuck when the resulting graph is in a state of local minimum, i.e. when the sum of all of the forces is 0, but the current reached equilibrium is only neutral or unstable.

For the works of this thesis, it is important for the graph to converge to a stable equilibrium. This is because, since information is provide through nodes proximities, a graph in an unstable or neutral equilibrium would carry wrong information to the user. The real time interaction capabilities of ForceAtlas2 and Gephi can be exploited to check the state of the current equilibrium: when the user manually moves nodes, if the graph is in an unstable equilibrium status, the node positioning would change drastically; on the opposite, if the graph is in a neutral state the nodes would not change their positioning. When a stable equilibrium is reached, on the other hand, moving nodes out of their equilibrium position would result in them returning to their original positions, relatively to the other nodes. A stable equilibrium was reached for all of the graphs presented in this thesis, and in particular in those reported in Chapter 6.

In ForceAtlas2, nodes attraction is linearly dependent on their distances:

$$F_a\left(n_1, n_2\right) = d\left(n_1, n_2\right) \tag{4.1}$$

with n_1 and n_2 being two nodes.

On the other hand, repelling force considers a node degree (deg), which is the number of edges connected to a node:

$$F_r\left(n_1, n_2\right) = k_r \frac{\left(deg\left(n_1\right)+1\right)\left(deg\left(n_2\right)+1\right)}{d\left(n1, n2\right)} \tag{4.2}$$

The coefficient k_r is defined by the user settings. The value $deg(n)+1$ is used instead of $deg(n)$ to ensure that even nodes having null degree are subject to some non-null repulsion force.

The two equations of attraction (Equation (4.1)) and repulsion (Equation (4.2)) are the core of how ForceAtlas2 works. However, to understand the algorithm settings that are used in Chapter 6, few more concepts are needed.

The `scaling` setting affects the spacing of the nodes in the graph, and is the value k_r in Equation (4.2). Increasing the scaling results in having nodes more spaced amongst them, while reducing it results in a more compact graph.

Another parameter that affects the positioning of the nodes is called `gravity`: it adds a third force, gravity, to the graph, that attracts nodes to the center and prevent non connected nodes to drift away. The force F_g that gravity applies on each node only depends on that node degree and on a user specified constant, k_g:

$$F_g(n) = k_g(deg(n) + 1) \tag{4.3}$$

If edges are weighted, weight values will be used when computing attraction forces, as in Equation (4.4):

$$F_a = w(e)^{\delta} d(n_1, n_2) \tag{4.4}$$

with $w(e)$ being the weight of the edge e, and δ being a user selectable parameter to change the influence of edges. The value $\delta = 0$ results in edges having no weights, while $\delta > 1$ increases the effects of edge weights.

Finally, the user may choose to use the LinLog mode for the attraction force:

$$F_a(n_1, n_2) = log(1 + d(n_1, n_2)) \tag{4.5}$$

According to the authors of ForceAtlas2, the use of the LinLog mode has "a strong impact on the shape of the graph, making the clusters tighter".

In this thesis, the LinLog mode is used for some graphs in Chapter 6 with many clusters, that would otherwise result in either a too large scale or an unreadable representation due to the graph being too crowded.

4.5 HTTP and REST API

HyperText Transfer Protocol (HTTP) is an application level protocol, widely used for communications between machines in the World Wide Web (WWW). It is a request-response based protocol, where a client machine asks the server for a resource through the resource Uniform Resource Identifier (URI) and the server provides the requested resource (content). HTTP describes several methods to interact with the requested resource, such as *GET*, to fetch the requested resource, *POST*, to update it with new data, *PUT*, to create a new one and *DELETE* to delete a resource. These methods and their behavior, however, are only expected in an HTTP based protocol, since their actual details are strongly dependent on the application which implements

the protocol. An example of usage of HTTP is the communication between web browsers and web servers.

The concept of REpresentational State Transfer (REST) was invented by Roy Fielding in his PhD thesis [29], to describe a stateless implementation based on the HTTP protocol. More in details, a REST application response must not depend on the history of previous requests and responses.

The set of resources provided by a REST service is called REST Application Programming Interface (API). In more general terms, API denotes the set of resources available to a developer using a tool or a service. A service offering a REST API exposes a set of resource URIs (the API*endpoints*), and provides a documentation on how an application should interact with the API. Since the interaction is usually based on standard HTTP methods, several libraries are available for most development programming languages and environments, both for client and server side of implementations.

REST is nowadays a widely used choice to provide resources by many services and they provide a consistent interface. Many websites, such as weather, video streaming (Youtube, Twitch among the many) and Google are also offering a REST interface to provide access with third party applications, such as mobile apps.

4.6 GHTorrent

GitHub offers an API access (Section 4.5) to obtain repository data from scripts and programs. The provided API calls represent a simple way to access repositories data, and therefore to conduct mining experiments. To prevent services abuse, however, GitHub limits the number of API calls a single user can make to a certain limit. To date, the limit imposed by the online platform is approximately 3.000 requests per hour. Unfortunately, this limit is usually not enough to access the amount of data contained in a large single repository.

To overcome this limitation, Georgios Gousios created a software, called *GHTorrent* [33], which automatically navigates through all of the different API endpoints, to collect and store GitHub responses. The calls to the API endpoints performed by GHTorrent are still limited by GitHub, but when GHTorrent dump is completed the collected dataset can be used without any more interruptions. An interested user can freely download the program to create a custom copy of a subset of GitHub, by

collecting repositories of interest. On the other hand, the GHTorrent project also features pre-collected snapshots of GitHub [36]. These snapshots include the history of a huge portion of GitHub hosted repositories, up to a certain date in time. Users who are not interested in running GHTorrent themselves can donate their GitHub API key to help the project maintainers to keep their publicly available snapshot updated.

Data, whether it was collected by the user or downloaded from GHTorrent public datasets, is stored by GHTorrent in two different databases. The first database is SQL-based. The official snapshots are dumped from MySQL. The SQL database has the disadvantage of lacking some data, since several repository data is only partially structured. This organization does not cope with SQL based databases, which must have a strongly structured format. Queries executed on the SQL database are however the fastest option, since they are executed on structured data. The second database is MongoDB, which is a non relational database (so-called NoSQL database), based on the JSON format for the internal data representation. Given that GitHub APIs use JSON for responses, GitHub data fits perfectly in MongoDB. MongoDB is, however, generally slower than MySQL in executing queries.

4.7 System setup

The system setup, shown in Figure 4.1, displays the interactions of the described components. To create a dataset, a file containing the list of wanted repositories must be provided to GHTorrent. GHTorrent queries the GitHub REST APIs for each repository and stores the resulting data in a MongoDB database. Moreover, for each repository in the provided list, all of the forks are identified and their data is recursively added to the database. MongoDB is then accessed by KNIME for data analysis (Chapter 5), and by Python scripts to generate the graphs described in Chapter 6. Python scripts are also using the database as a cache to store the results of map-reduce operations. Finally, Gephi is used to display the resulting graphs, allowing the user for interactions and further exploration of the results.

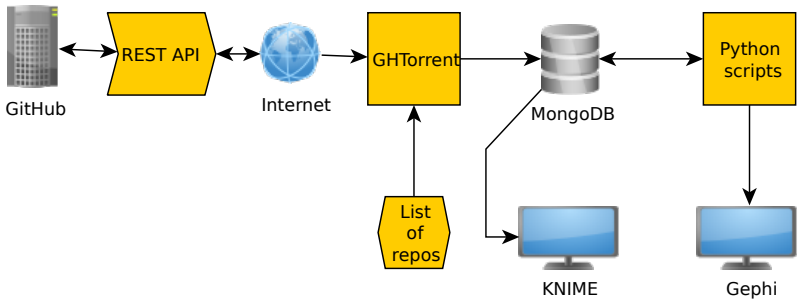


Figure 4.1: System setup, which illustrates how data is collected from GitHub REST API and is stored in a database, which in turn is accessed by KNIME for data exploration and Python scripts. Results produced are then displayed in Gephi.

Chapter 5

Data analysis

Understanding the data is the first step in mining tasks. In order to conduct data analysis, however, it is required to collect the data that will be studied.

Since the analysis requires many access to the data, the proposed work uses GHTorrent (presented in Section 4.6) to construct snapshots of datasets. Only the MongoDB collected data will be used, since in this research it is preferable to have access to more complete data than to perform faster operations on the data base.

5.1 Datasets

Four different datasets have been used for this thesis. The first dataset is the *Mining Software Repositories 2014 Mining Challenge Dataset* (MSR14) [33]. The MSR14 is a subset of a GHTorrent snapshot, containing data from 90 repositories, along with their forks. These 90 repositories are, according to the GHTorrent description page[1], *the top-10 starred software projects for the top programming languages on Github.* The purpose for which the MSR14 was created is a mining challenge, held in 2014, for the 11th edition of the *Mining Software Repositories* conference. For this reason, many researchers have been using the MSR14 for their studies, and therefore many results and considerations are available for this dataset.

However, the use of the MSR14 dataset alone presents two problems: 1) it is a relatively old dataset and 2) the included project may not be optimal for analyzing repositories interaction, i.e. based on the inclusion criteria, the project may have

[1]GHTorrent description page, last visited on January, 12th 2018: `http://ghtorrent.org/msr14.html`.

google/volley	stanfy/helium	nisrulz/OptimusHTTP
SpartanJ/restafari	apptik/jus	greengrowapps/ggarest
square/retrofit	alirezaafkar/JsonRequester	magnetsystems/rest2mobile
studioidan/HttpAgent	orhanobut/wasp	foxykeep/DataDroid
jianastrero/Sweet-Mother-of-Json	jaksab/EasyNetwork	kevinsawicki/http-request
reisub/HttPizza		

Table 5.1: Projects included in the ANDROID dataset.

eclipse/golo-lang	elixir-lang/elixir	chapel-lang/chapel
ruby/ruby	dlang/dmd	micropython/micropython
scala/scala	typelead/eta	apache/groovy
lua/lua	factor/factor	racket/racket
jashkenas/coffeescript	rust-lang/rust	nulang/nu

Table 5.2: Projects included in the LANGUAGES dataset.

limited interactions. Therefore, three more datasets were populated with repository information specifically obtained for the purposes of this research. GHTorrent was used to collect the data. Each dataset focuses on a specific topic. The datasets are labelled as ANDROID, LANGUAGES and ELECTRON. Their content and characteristics are described below.

ANDROID was built with the idea of having a comparison of different REST APIs client libraries (see Section 4.5) for the *Android* platform. The supposed scenario consists of a user who needs to include a REST API library to his *Android* application, performs a search on the web and obtains a list of possible alternatives. It is a relatively small dataset, and includes 15 projects and their forks. The full list of the included projects can be seen in Table 5.1. Among all the alternatives, the most widely known libraries on this topic are *google/volley*, which is mentioned in the *Android* developers official website[2], and *square/retrofit*.

The LANGUAGES dataset includes 15 open-source programming languages hosted on GitHub. The projects were arbitrarily chosen from the list of 49 programming languages showcased by GitHub[3], with the goal of including programming languages only in the results. The list of the included projects are shown in Table 5.2.

Finally, the fourth used dataset, ELECTRON, includes 122 projects that are built

[2]Official Android developers website, last visited on January, 12th 2018: `https://developer.android.com/index.html`

[3]Programming languages showcased by GitHub, last visited on January, 12th 2018: `https://github.com/showcases/programming-languages`.

on top of *Electron*, a framework to build cross platform applications using *Javascript*, *HTML* and *CSS*. Some popular open source examples of *Electron* based projects are *Atom*, a text editor, *Beaker*, a peer-to-peer web browser and *Visual Studio Code*, a cross platform IDE. *Electron* is also used by many closed source applications. Some known examples are the official *Slack* client, a professional communication platform, *GitKraken*, a Git graphics frontend, *Discord*, a chat for gamer and *Whatsapp*, a smartphone communication application. The collection of projects were gathered from *sindresorhus/awesome-electron*, a GitHub repository that is referenced from the official *Electron* repository as a collection of useful resources. In particular, only the open source projects were included. The projects included in the resulting dataset are listed in Table 5.3.

5.1.1 Datasets format

The MongoDB (see Section 4.2) snapshots feature several collections corresponding to different API endpoints from GitHub. The information regarding the dataset format can be retrieved from the GitHub developers documentation[4]. Nevertheless, the most significant aspects on GHTorrent snapshots will be reported for completeness, and as a reference in case of future significant API changes.

The snapshots feature 14 collections, shown in Table 5.4. Collections such as *repos*, *users* and *commits* are a good starting point to investigate the repositories.

The *repos* collection contains general information on the repositories. Among the many fields contained in a *repo* document, the following will always be found:

- *full_name*: the full name of the repository;
- *owner*: the identifier and information on the repository owner;
- *description*: a short description of the repository;
- *fork*: a boolean flag indicating whether the repository is a fork;
- *url*, *forks_url*, *keys_url* etc.: the API endpoints for the repository on GitHub API;
- *created_at, updated_at*: the creation/update date for the repository;

[4]GitHub developers documentation, last visited on January, 12th 2018: `https://developer.github.com/`.

electron/electron	whoisandie/yoda	nurtext/active-collab-desktop
beakerbrowser/beaker	mmckegg/loop-drop-app	appetizermonster/hain
zeit/hyper	frankhale/toby	MeoBeoI/Catify
getinsomnia/insomnia	EragonJ/Kaku	jenslind/minira
wulkano/kap	yeoman/yeoman-app	m0g/ansel
minbrowser/min	minodisk/markn	willmendesneto/build-checker-app
princejwesley/Mancy	rhysd/Shiba	teesloane/moonview
zz85/space-radar	Bahlaouane-Hamza/Yays	tryghost/ghost-desktop
k0kubun/Nocturn	sapjax/TimoFM	mattermost/desktop
muan/mojibar	maxogden/monu	xwartz/PupaFM
mafintosh/playback	jenslind/piglet	sivragav/mediumdesk
railsware/upterm	twolfson/google-music-electron	sachinchoolur/lightgallery-desktop
atom/atom	pomodoro	HR/Crypter
Microsoft/vscode	dvcrn/markright	yakyak/yakyak
oguzhaninan/Buka	officert/mongotron	KeitIG/museeks
brave/browser-laptop	colonizers/colonizers-desktop	fresk-nc/VOX
jlord/git-it-electron	mazehall/eintopf	vesparny/marky
sindresorhus/caprine	mawie81/whatsdesktop	decosoftware/deco-ide
Automattic/simplenote-electron	sqlectron/sqlectron-gui	tofuness/Toshocat
LeeChSien/nuTorrent	khornberg/docker-indicator	ningt/iStats
brrd/Abricotine	LightTable/LightTable	wireapp/wire-desktop
luin/medis	makotot/Tubehead	terkelg/ramme
muffinista/before-dawn	MarshallOfSound/Google-Play-Music-Desktop-Player-UNOFFICIAL-	
evancohen/smart-mirror	auchenberg/chrome-devtools-app	temps
hachibasu/koko	yeobara/yeobara-desktop	web-pal/DBGlass
keeweb/keeweb	steventhanna/proton	dermike/slide-beacon-app
ekonstantinidis/gitify	uxebu/james	saenzramiro/rambox
moose-team/friends	alchen/DTCP	sedwards2009/extraterm
gillesdemey/Cumulus	fgnass/inbox-app	mike-schultz/materialette
pwambach/fat-file-finder	alienbox	vutran/dext
maxogden/screencat	midnightSuyama/tweet-rec	sidneys/pb-for-desktop
sindresorhus/gulp-app	teesloane/snippet-bar	dcrousso/GroupMe
webuildsg/osx	geeeeeeeeek/electronic-wechat	720kb/ndm
hij1nx/levelui	xwartz/dida	dcrousso/GIFBar
maddox/kart	kilian/fromscratch	oguzhaninan/Stacer
yoshuawuyts/vmd	kalpetros/hawkpass-desktop	tinytacoteam/zazu
cheeaun/kyoku	Zhangdroid/Gokotta	sarah-seo/Inpad
Nekle/greader	lumios/shake	KELiON/cerebro
leanote/desktop-app	vitorgalvao/fog	sidneys/desktop-dimmer
pt2121/Snapper	Thomas101/wmail	mifi/lossless-cut
imagemin/imagemin-app	BoostIO/Boostnote	sential/wexond

Table 5.3: Projects included in the ELECTRON dataset.

- *watchers_count*: how many users are watching this repository;
- *language*: the language mainly used by the files in the repository;
- *has_issues, has_wiki*: boolean flags indicating whether the issue tracker/the wiki is used;
- *forks_count* and/or *forks*: how many forks were created for this project;
- *open_issue_counts*: how many open issues are present.

An important aspect that is worth to mention is that GitHub APIs are designed to be explored with a sequence of related calls. For instance, let us consider a user who wants to obtain a list of the forks for a given repository. In this case, the user has to ask GitHub for that repository, obtaining the URL for *forks_url*. Afterwards, he can obtain the fork list through a series of subsequent calls. The same pattern, however, can not be adopted for the GHTorrent snapshots: only the API responses are saved during the snapshot creation, and a layer that translates an API URI to the resulting data is missing. The solution is to reverse the problem: from the collection *forks*, all forks that have their main repository equals to the repo of interest (*parent.full_name == repo_of_interest*) should be found. This operation, however, may be costly as it must navigate through all of the documents in the collection.

The *users* collection features information on the users. The most important information that can be found here are the *login*, the *name* and the *email*. The *login* field is the unique identifier of the user on GitHub, and can be used to safely match documents from other collections. The fields *name* and *email* can be used to match documents when *login* is not present, however this poses threats as the user has many degrees of flexibility on these attributes. More details on this, and in general on the problem of matching documents to the corresponding user, are presented in Section 5.3.

Other potentially interesting fields are *type* (which can be Organization or User), *company*, *location*, *hireable*, *followers* and *following*. The *created_at* field can be used to evaluate for how long the user has been on GitHub.

The *commits* collection contains information on the commits pushed on a repository. In the *commit* field, the information on the Git commit can be found. Examples of important attributes in *commit* field are the *message* (the commit message), the *tree*, which can be used to position the commit in a history of commits, and the

Collection	*Description*
commit_comments	user written messages on commit content (usually in pull requests)
commits	data on commits, such as authors, date and contents
events	history data on starting and ending of events, such as forking and watching
forks	same data as repos collection, but only for forked repositories
issues	data on issues in issue tracker at the moment of the snapshot creation
issue_events	history of issues, such as creation, renaming and closing
issue_comments	replies to issues
org_members	members of GitHub organizations
pull_request	data on pull requests at the moment of snapshot creation
pull_requests_comments	comments posted on the pull requests
repo_labels	the set of labels used in a repository
repos	data about repositories themselves, such as names, dates and programming languages
users	users data, including name, login and email
watchers	information on users watching repositories

Table 5.4: Collections included in a snapshot.

author and *committer* raw data, as logged by Git. When they are present, the *author* and *committer* fields contain information on who created the code in the commit (the author) and who wrote the commit (the committer). For both the *author* and the *committer* fields the *login* field can be used to quickly match a document from the *users* collection, when available.

Another important field in a commit document is *url*, which stores the original URL of the commit. Since the base of the URL contains the repository full name, it can be used to identify the commit repository, as shown in the example of Section 4.2.1.

5.2 Data exploration

The initial questions that have to be made regarding the features of a dataset are related to the size of the involved objects. Size matters, since it provides a rough estimation of the complexity of the problems involved when studying a dataset. The number of documents for the 3 collections *repos*, *users* and *commits*, contained in the 4 datasets, is reported in Table 5.5.

A possible way to carry out the analysis on this data is to use data analytics tools. Among the many options, KNIME (presented in Section 4.3) is a good choice since - among other features - it also offers a MongoDB node to collect data directly from the MongoDB server.

	n_repos	*non_forked_repos*	*n_users*	*n_commits*
MSR14	108,710	90	496,519	601,080
ANDROID	5930	16	13,480	6,941
LANGUAGES	11,785	14	71,319	341,303
ELECTRON	19,926	110	121,603	82,115

Table 5.5: Size of *repos*, *commits* and *users* on the datasets.

When using raw data from the dataset, it must be kept in mind that queries involving a large number of documents are too expensive to be executed, either because the query requires a long time for the execution, or because the results are so large that they can not fit in the maximum space available for MongoDB responses. In these scenarios, the *Map-reduce* method, which was illustrated in Section 4.2.1, could be used to compute pre-processed, aggregated data beforehand. The processed data can feed KNIME to compute the results. For instance, a query for all of the non-forked repositories requires a few seconds to be executed on all of the 4 considered datasets, returning less then 200 documents per dataset. A query to return all of the commits, on the other hand, can not complete successfully on the MSR14 dataset (with $601,080$ commits). It is therefore impossible to obtain results, such as finding all of the commits for a given repository (the example provided in Section 4.2.1) when the matching is done client side on the query resultset.

In the remainder of this section, along with the computed statistics and the related considerations, the corresponding map-reduce applied to the data will be reported to facilitate the understanding and the reproducibility of the results.

5.2.1 Size of the repositories

The basic numbers reported in Table 5.5 are useful to provide a rough estimation of how many elements are included in the datasets. However, several interesting numbers can be further derived for a more accurate characterization of the datasets.

A metric that is really important when considering a repository is how "big" it is. Most users involved in the OSS community will not have any doubt in saying that projects such as Linux and *Firefox* are "big", and for a reason: these two are known projects, with rich history, many famous contributors, a huge codebase. On the other hand, the same users may find difficult to provide a definition of what

repository	commits	forks	authors
mavam/stat-cookbook	56	41	2
django/django	15,981	2,692	476
jquery/jquery	5,983	4,920	320

Table 5.6: Number of commits, forks and authors for 3 repositories of MSR14.

would be considered a big project, or to classify an unknown repository as "big" or "small". Referring to the MSR14 dataset, the repository *mavam/stat-cookbook* contains activities of a single user who writes a book on statistics. The value of every single attribute related to this repository is smaller than *django/django* (the *Django Python* web framework), as can be seen from Table 5.6, therefore it is easy to conclude that *mavam/stat-cookbook* is smaller than *django/django*. On the other hand, when comparing *django/django* with *jquery/jquery* (a popular javascript library), the situation is less clear-cut, since some numbers are bigger and others are smaller.

Defining the size of a repository is not a simple task, as there is not any single attribute that can be chosen to compare the size of two repositories without introducing a certain amount of subjectivity. Yet, it is important for a user to compare different repositories, especially when he must decide which project best serves his purposes.

One of the statistics provided by GitHub in the project page, and that many users are currently using to evaluate the size of a repository, is the number of commits that were pushed to the repository itself. The number of commits is certainly related to the amount of activity that was done on the repository. On the other hand, this number may be misleading. For example, it may happen that a skilled developer, with a limited amount of time, does not want to spend the time needed to write a rich commit history, and therefore produces a small amount of big commits with bug-free, tested code. Comparisons of this repository to another one with many less skilled developers, who are writing buggy code and are producing many bug fixing commits, would be deceiving if they are limited on the commits count.

A better way to assess the size of a repository is to consider a combination of several attributes, which can be related to the size of a project, and either let the user make considerations on the raw values or aggregate them to provide a graphic

comparison. The idea behind this approach is that the error introduced by using a single parameter to estimate the size of a project is reduced and redistributed over the number of parameters used.

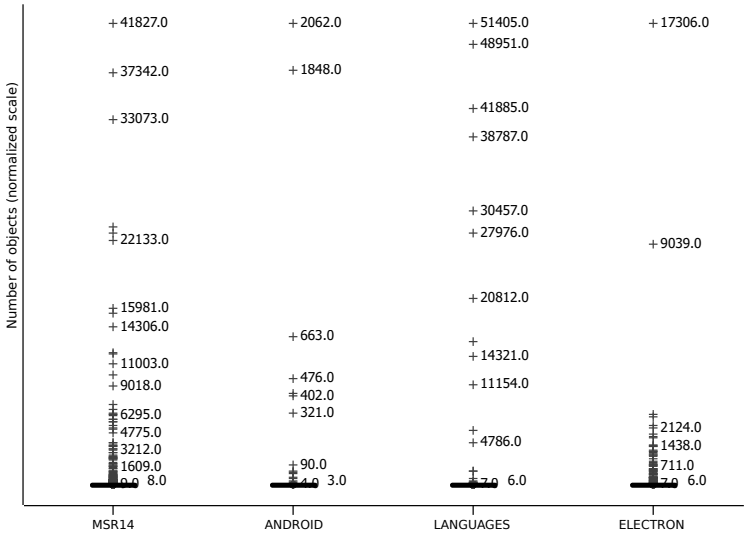
Some parameters that relates to the size of a project are:

- *number of commits*: while this alone is not correct, it is related to the size of a project;
- *number of forks*: bigger projects tend to attract contributors, and in turn to have a large number of forks.
- *number of users*: similar to the number of forks, but different since not all users who create a fork contribute back to the original projects, and also not all users who contribute do so in a GitHub fork.

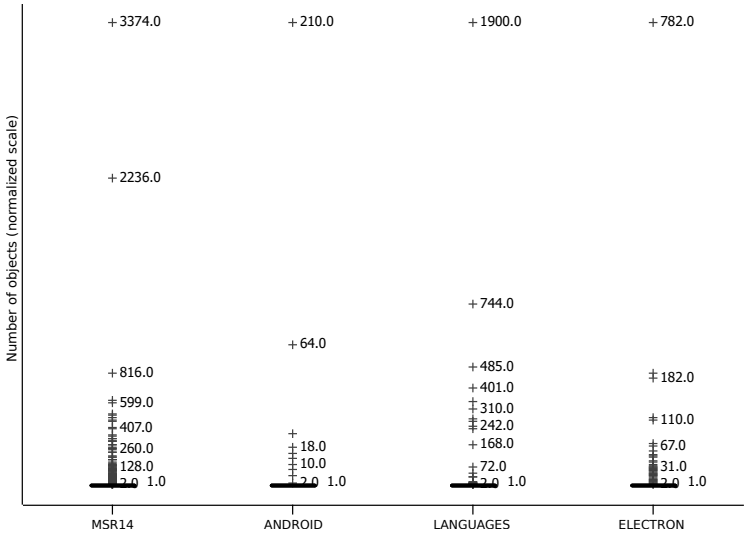
The number of commits per repository and the number of unique users that committed to a repository can be found through a map-reduce, while the number of forks is available in the *repo* collection. Given the fields contained in a commit, however, it is yet unclear which fields should be used to identify the user. Different options are available. For example, the author, the committer and the GitHub login fields are usually available. The GitHub login can uniquely identify the author, but sometimes it is missing. On the other hand, the committer and author data are not validated by GitHub, and the user may have entered incorrect data. For the next steps only the author email address will be used, but an in-depth observation on the problem of identifying the unique authors is discussed in Section 5.3.

The resulting statistics for commits and users and forks numbers, for each dataset, are shown in the graphs of Figure 5.1. As can be seen from the graphs, most of the displayed values are outliers (represented as small crosses), and the actual boxes are just small horizontal lines near the value 0. This apply for all of the three considered parameters: commits, authors and forks. This is because most of the projects contained in the datasets are forks, with much less activities on them w.r.t. the main repositories from which they were forked. Therefore, it is more interesting to have a look at the data from the non-forked repositories.

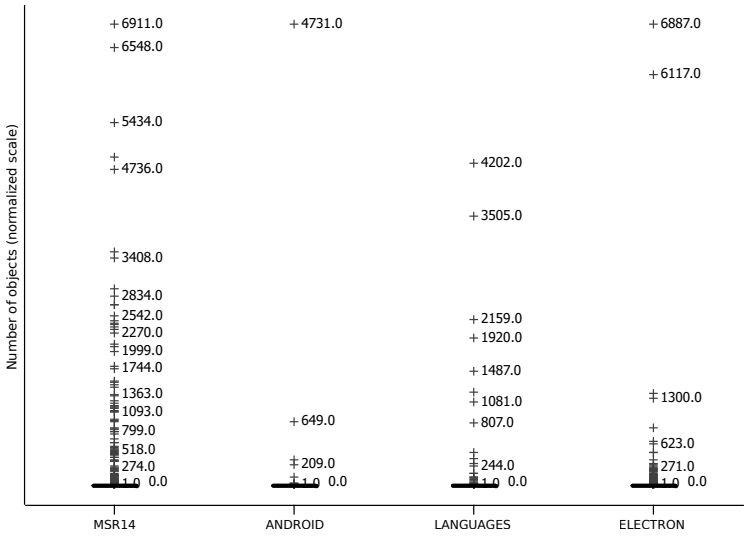
Data from the non-forked repositories is shown in Figure 5.2. From this figure it can be seen that the projects are fairly different among the datasets: the LANGUAGES dataset features repositories with much more commits, on average, than



(a) Commits

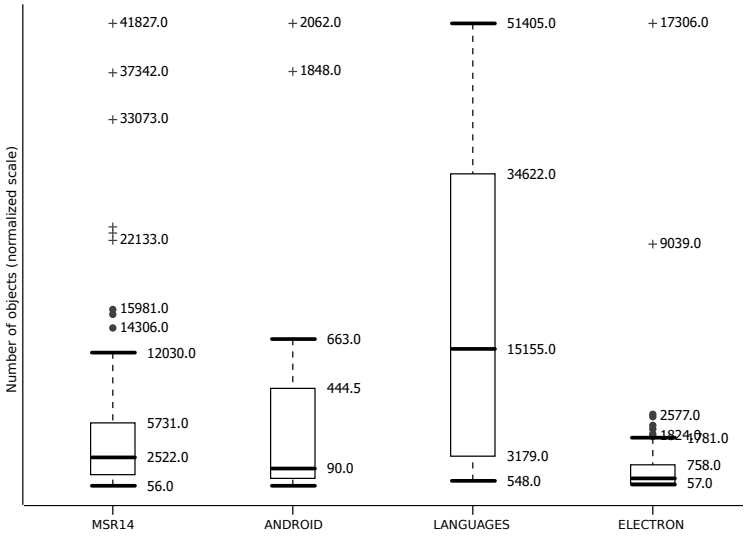


(b) Authors

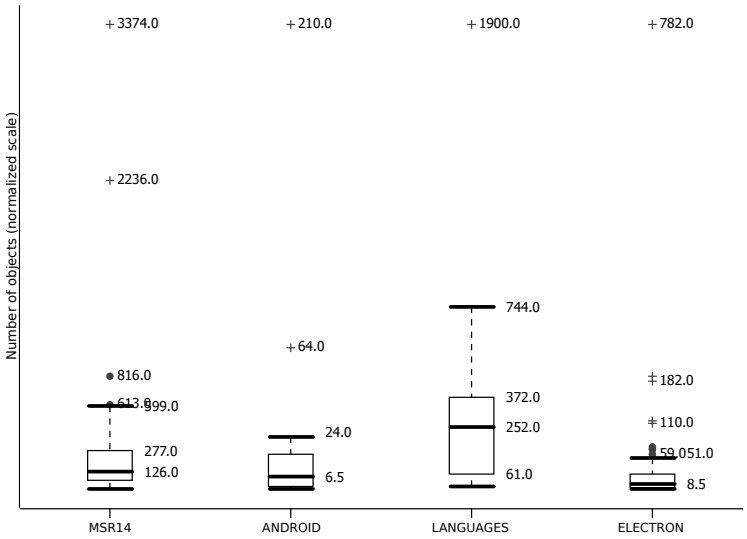


(c) Forks

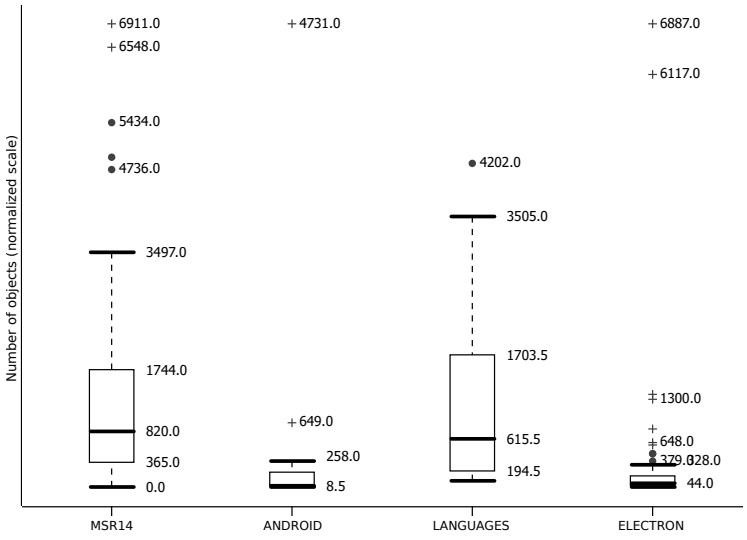
Figure 5.1: Commits, authors and forks counts boxplots for the four datasets. Most of the displayed values are outliers, and the actual boxes appear as flat lines.



(a) Commits (no forks)



(b) Authors (no forks)



(c) Forks (no forks)

Figure 5.2: Commits, authors and forks counts for the four datasets, with forks excluded.

the others, while ANDROID and ELECTRON projects have less commits than the MSR14. The authors for project and forks both follow similar trends. In summary, projects in LANGUAGES seems to be bigger than the ones from MSR14, which in turns have bigger projects than ANDROID and ELECTRON on average.

The next analysis compares the size of the repositories inside a single dataset, which is a result that can be used by a final user to make better decisions when choosing among many alternative repositories. A possible approach to analyze this aspect could be the generation of a 2D representation, i.e., create scatter plots, of pair of parameters. However, this method may not capture possible relationships among multiple parameters, leading to unclear and and poorly usable results. A more effective alternative method is to use the Principal Component Analysis (PCA), a technique to lower the number of features of a dataset, to obtain a bi-dimensional representation on an x-y plane of multiple parameters. PCA necessarily leads to lose details and introduces approximations in the final results, but with the advantage of allowing the representation of elements with similar values for the same parameter that are close to each other. This way, the results become easier to interpret by the user, who can derive useful relationships among projects that are close each others in the representation.

KNIME offers a PCA node, which provides a simple way to apply it on previously computed statistics. Before applying the PCA, it is important to normalize the data to prevent attributes with a bigger scale to have a major impact on the final results. KNIME lacks a native node to create a scatter plot with labels, but a *Python* node with custom script using *pyplot* can be used instead. To further simplify the representation, k-means clustering was used to isolate the bigger cluster and hide labels, to prevent the graph from being unreadable. The repositories from the more dense cluster are represented in red color and without the labels attached.

For the size maps, the three previously considered parameters (commits, authors and forks) are used. Figures 5.3 to 5.6 show the resulting map for sizes of the four datasets. Notice that the two axis, X and Y, cannot be represented as any measurement unit, since they are the projections of the used parameters. In Figure 5.3, the repository *stat-cookbook* is in the left side, forming a cluster with many other projects.

As previously stated, *stat-cookbook* is a small project, therefore the user knows that the cluster of nearby projects is composed by the smaller projects in the dataset.

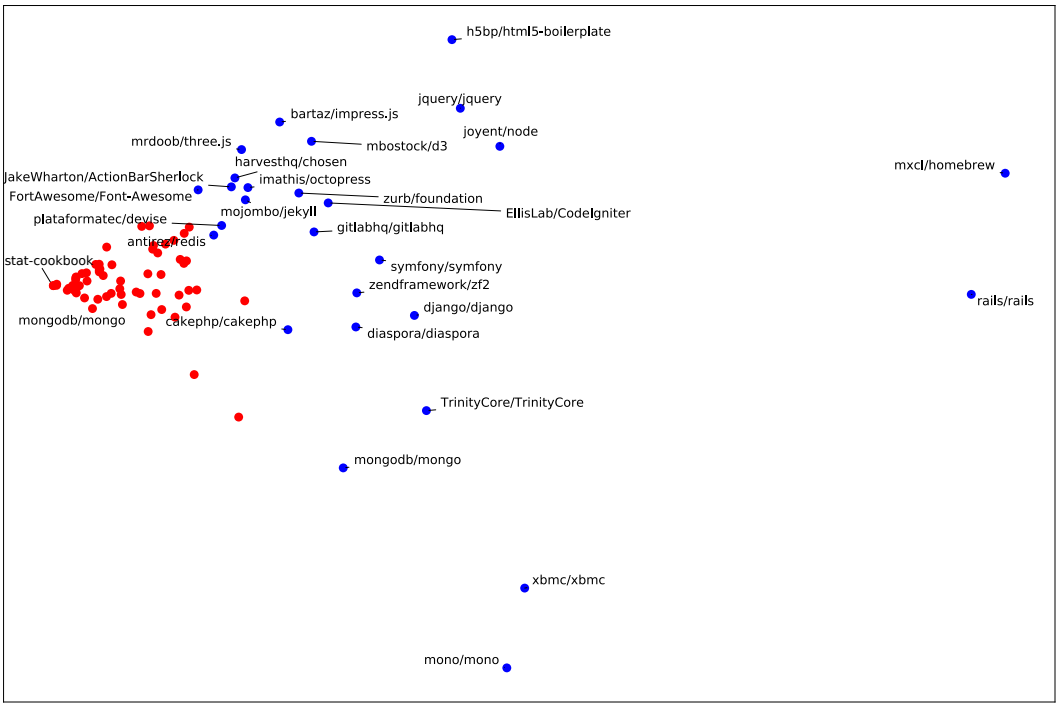


Figure 5.3: Graphical representation of projects sizes for MSR dataset.

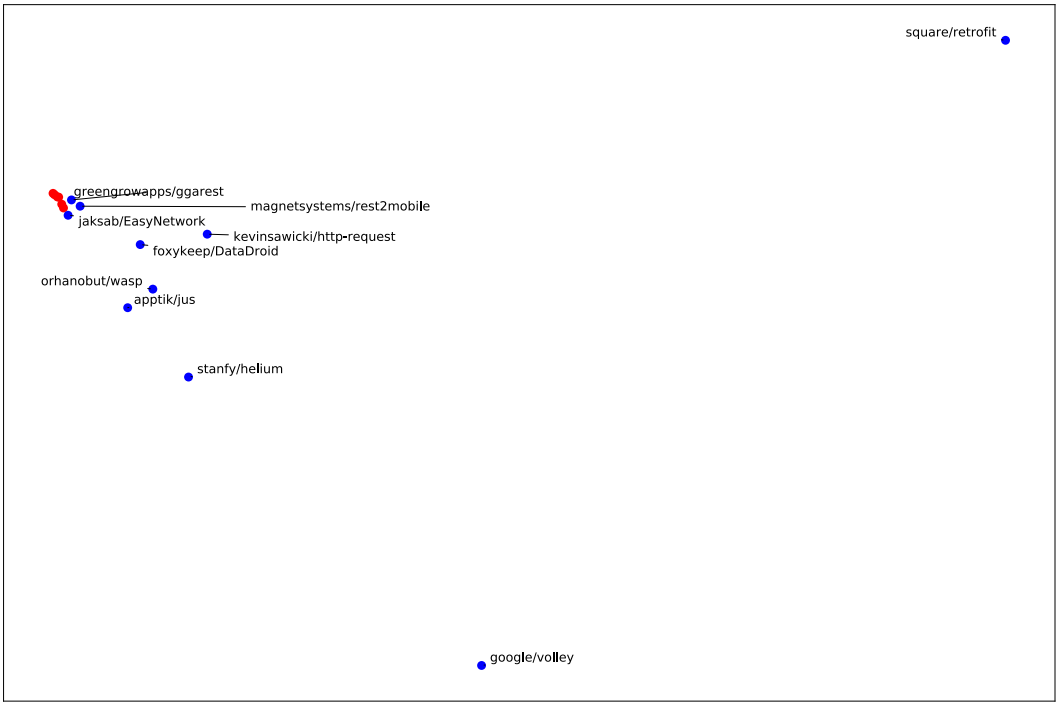


Figure 5.4: Graphical representation of projects sizes for ANDROID dataset.

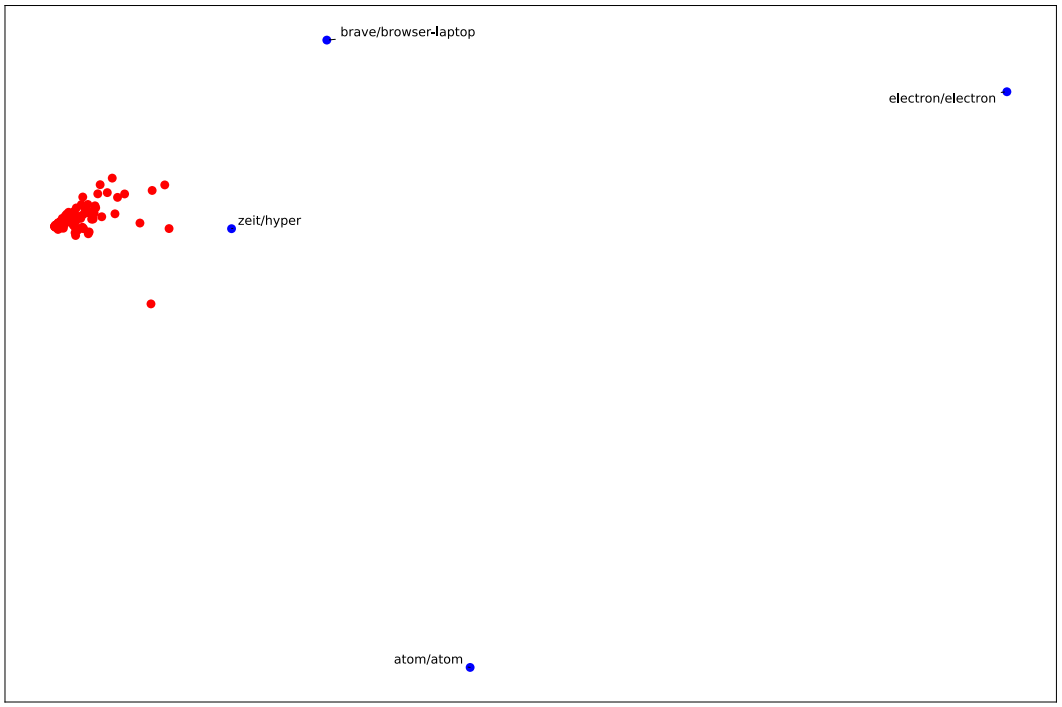


Figure 5.5: Graphical representation of projects sizes for ELECTRON dataset.

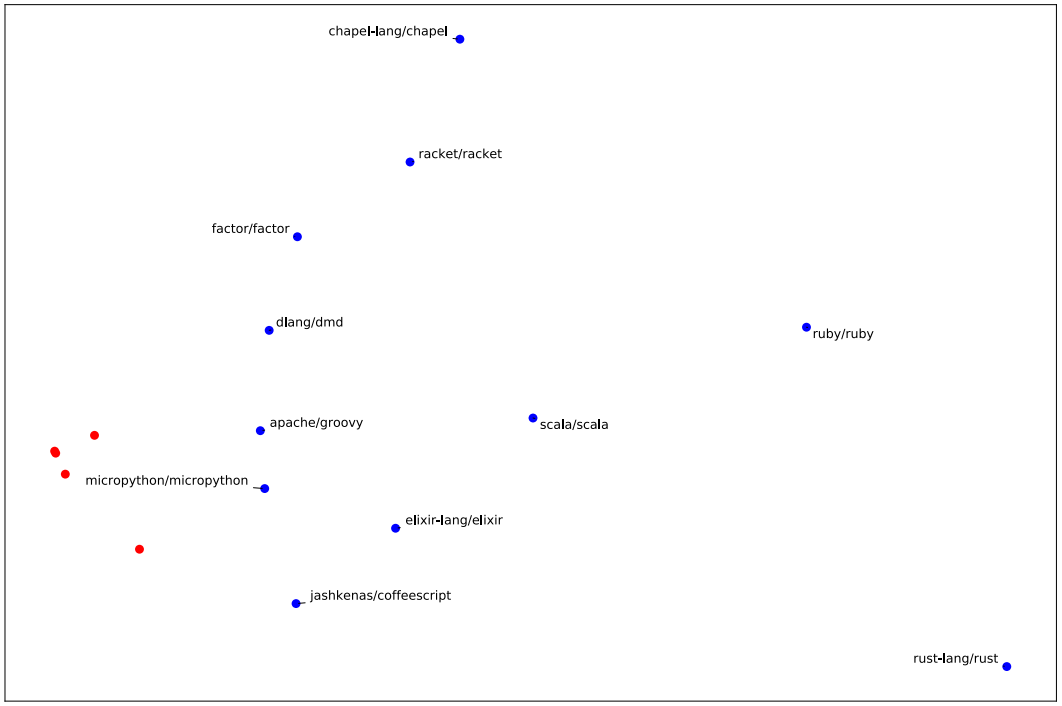


Figure 5.6: Graphical representation of projects sizes for LANGUAGES dataset.

stat-cookbook is also known to be smaller then *jquery*, therefore the user knows that close projects, such as *joyent/node* and *h5bp/html5-boilerplate*, are similarly sized and bigger than *stat-cookbook*. The size for projects that are far away from both *jquery* and *stat-cookbook* remains unknown, until the user reading the figure adds his own evaluations on other projects, which are similarly far away from known ones. For example, in this result, the user may choose to evaluate the characteristics of *mxcl/homebrew* and *mono/mono*, as they are the farthest projects from the cluster of small projects. This hypothetic user would then know that both repositories are big projects. However, while *mono* has much more commits, *homebrew* has more forks and authors. Given the shape of this result, the two axis *stat-cookbook-jquery* and *stat-cookbook-mono* can be used to estimate the sizes of all of the repositories in the dataset: the further the repositories are from *stat-cookbook*, the bigger the repositories, and the more they go towards either *homebrew* or *mono*, the more they tend to have more or less commits with respect to authors and forks.

In Figure 5.4, the *volley* and *retrofit* projects are represented far from each other in the map. Even though they are both big projects, they are not close together. This is not an incorrect result, since *volley* has less forks and authors but similar number of commits to *retrofit*. The two repositories are therefore different enough that it is required for the user to manually check both of them to interpret the results.

In the left side, again, there are many smaller projects clustered together. Among them there are many repositories such as *http-request*, *DataDroid* and *helium*. This information matches an expert knowledge: *DataDroid* is a now deprecated project, which was interesting at the launch time in 2010, and was actively developed for three years; *http-request* is a widely used and known *Java* and *Android* library to wrap the *Java* HttpURLConnection object, and *helium* is a tool to create Domain Specific Languages (DSL) for REST API, capable of creating *Java*/*Android* code. None of this repositories can be classified as small, but neither can be considered as big: *DataDroid* development ceased with the arrival of other Android libraries, while *http-request* and *helium*, even as important project, are much smaller than *retrofit* and *volley*.

Figure 5.5 shows a cluster which includes almost every project in the dataset. As a remainder, this dataset includes *electron/electron*, a framework to build *HTML*, *CSS* and *Javascript* applications, and many projects built on top of it. The framework itself is detached from the left cluster. There are also two projects notably far away

from the cluster: *browser-laptop* and *atom*. The first is actually the repository with most commits, after electron itself, and has a significant amount of authors and forks. *Atom*, on the other hand, is a well known text editor and, while not having many commits in the repository, has by far the higher amount of forks in this dataset.

Finally, Figure 5.6 shows a more distributed map. On the left side there is still a cluster with smaller projects. Unlike previous results, projects do not seem to grow on a single axis/bi dimensional axis, but rather distribute in a radial way from the cluster. Bigger projects distribute on the borders again, with *chapel* and *ruby* having more commits and *rust* having the highest number of authors and forks.

5.2.2 Popularity of the repositories

Popularity is a concept that refers to persons, objects or ideas to indicate how much they are known and appreciated by people, and capable of attracting more people.

Similarly to what happens for size, users often consider repositories as "popular" when they are followed and contributed to by many people and "unpopular" when they are not. The concept of popularity associated to a repository, however, suffers from the same problems of the size: there are no simple metrics to measure popularity, and no clear distinction between different repositories. Therefore, an approach similar to the one explored in Section 5.2.1 for the size can be used to provide a graphical representation that allows the user to estimate projects popularity based on distances from previously known repositories.

Some parameters related to popularity can already be found in a *repo* document. These parameters are the number of forks, which was already used for size, and the number of watchers. The count of subscriber could only be accounted for newer datasets, since this field is missing in the older MSR14 dataset. This latter dataset, in fact, was generated before the introduction of APIs that allow to obtain the value of the parameter. The number of contributors, which was used for projects size, is also relevant here.

Two other parameters related to the popularity of a project are the number of users that wrote comments, either in the issue tracker or in a pull request discussion. Both parameters can be computed with map-reduce, starting from the *issue_comments* and *pull_request_comments* collections, collecting unique logins used and counting them in the finalize step. Since not all of the repositories use the

	non_forked_repos	*missing_issue_tracker*
MSR14	90	2 (2%)
ANDROID	17	3 (18%)
LANGUAGES	33	2 (6%)
ELECTRON	117	13 (11%)

Table 5.7: Missing issue tracker comments.

issue tracker and/or pull requests, a value of 0 is used in case of missing values.

The number of projects which are not using issue trackers and/or pull requests, or is not including any user comment, is reported in Table 5.7. As can be seen in the table, the number of projects without issue tracker data is significant. The MSR14 is the dataset with less missing data, which understandable since it was built explicitly for data mining purposes. Moreover, it features mostly big projects, which are typically managed using the various facilities from GitHub.

The datasets built for this thesis, however, include many smaller projects. A project with no issue tracker data may either be using an external tool or not using any issue tracker entirely, and often small projects happen to belong to the second case. Still, if a project is using an external issue tracker, a lot of activity on issues is missing in GitHub and therefore in the resulting graphs an error is introduced. Once again, this error is mitigated by the aggregation of the wrong resulting count with the other parameters.

Finally, a parameter called “active forks” were introduced in the popularity graph. This parameter is a count of how many unique forks submitted a pull request to the project - without considering whether this was accepted or rejected. This parameter correlates with how well the project attracts active contributors, in opposition to attracting passive users. Active forks are computed using a map-reduce, by counting the unique repository from which a fork was submitted for a pull request in the *pull_request* collection. Missing values are set to 0 without introducing any error, since there were effectively no active external contributor for these projects.

After computing these statistics for the four datasets, an error in the datasets was discovered. All of the projects with no issue tracker data resulted in no active forks and pull request recorded. This may be reasonable for smaller projects in the ANDROID and ELECTRON sets, but seemed suspicious for many of the bigger

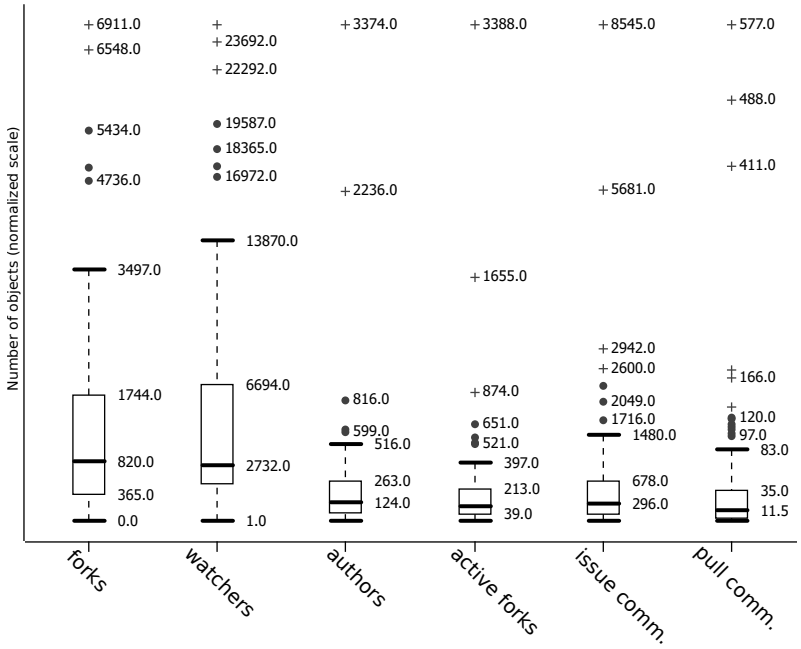


Figure 5.7: How many objects for each parameter of the non forked repositories in MSR14.

projects of LANGUAGES, which are known to be using GitHub's pull requests. The problem is that pull request data is missing entirely from the collected datasets when no issue tracker is used, while still being present on GitHub. It is yet unclear to the author if this is due to a bug in the GitHub API, or a bug in *GHTorrent*, or an incompatibility between the latest version of GitHub APIs and *GHTorrent*. The problem does not seem to occur in the MSR dataset, since there are very few projects with missing issue tracker.

Results based on wrong data would be meaningless, therefore the resulting maps are presented in two forms: one including all of the repositories with pull request commentators and active forks discarded, and one including all of the parameters but discarding projects without the issue tracker.

The results for the four repositories, is shown in Figures 5.7 to 5.10. Out of the four datasets, MSR (Figure 5.7) and LANGUAGES (Figure 5.9) appear as more homogeneous, with outliers which are closer to boxes and most data contained in the central quartiles. ANDROID (Figure 5.8) and ELECTRON (Figure 5.10), on the other hand, are more heterogeneous, with many smaller projects and few outliers which are very distant from most data. For all of the datasets it can be noticed that

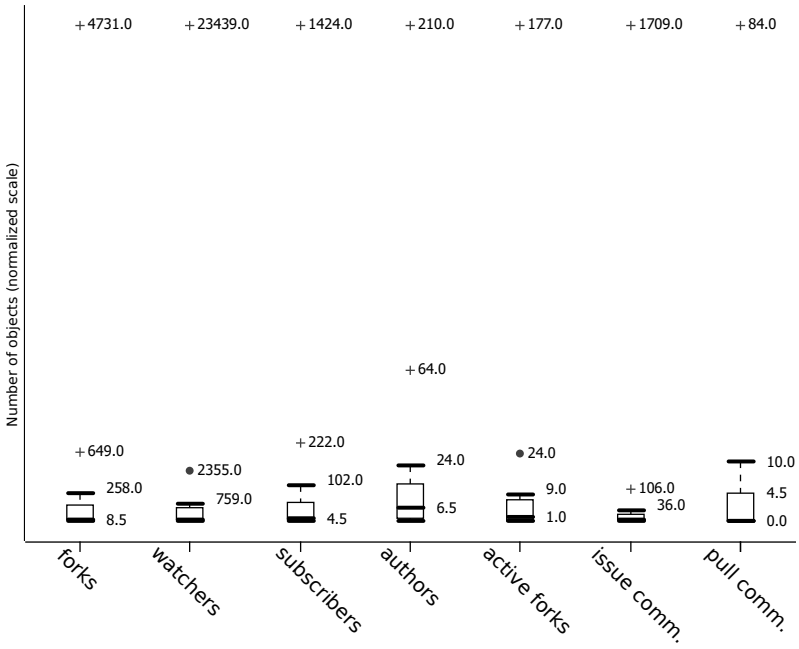


Figure 5.8: How many objects for each parameter of the non forked repositories in ANDROID.

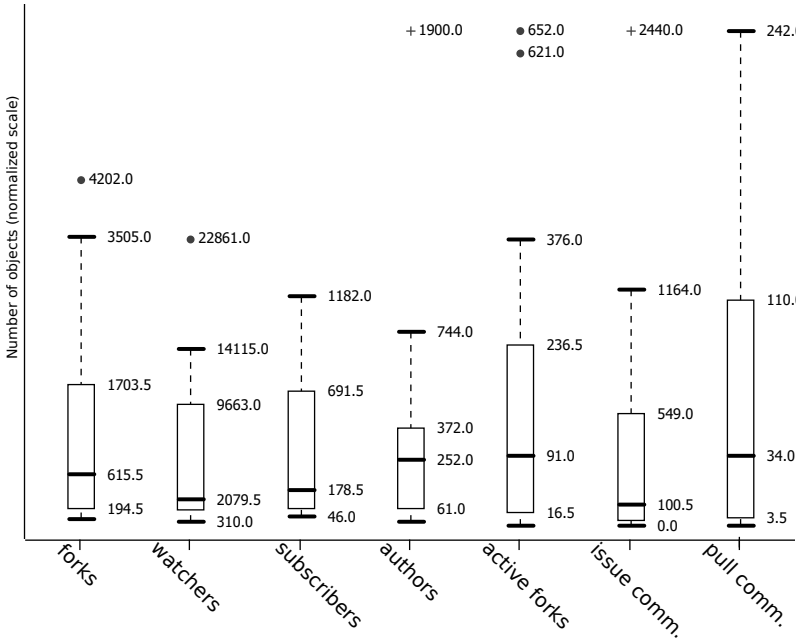


Figure 5.9: How many objects for each parameter of the non forked repositories in LANGUAGES.

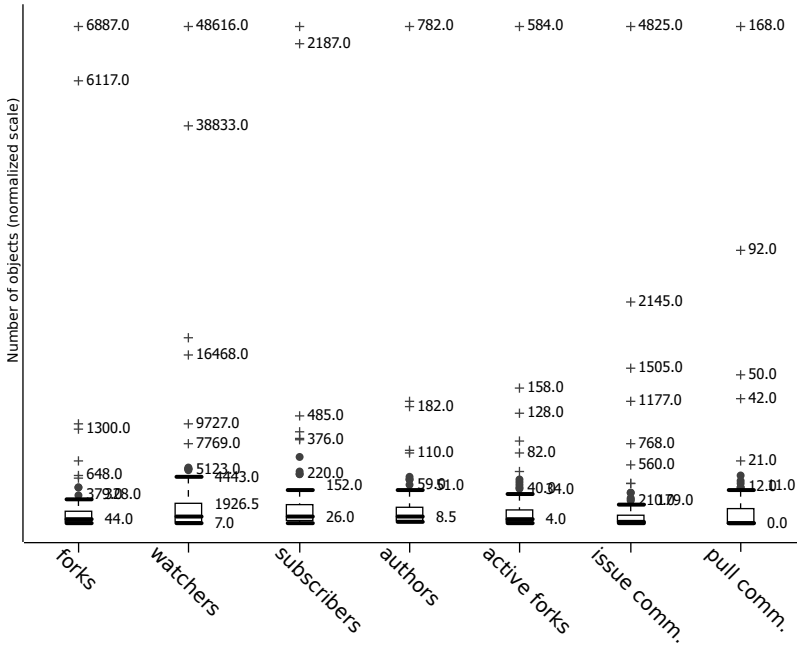


Figure 5.10: How many objects for each parameter of the non forked repositories in ELECTRON.

the number of people posting messages on the issue tracker is on average double the amount of authors, while the number of people discussing the pull requests is much lower. This is interesting since it means that the number of users who are discussing issues or requesting new features is much higher than the number of people who are discussing code contributions. On the other hand, the number of active forks is higher than the number of pull requests commentators, while being still lower than the issue tracker commentators. This means that users on this dataset do propose their contributions for bugs and for new features, but are not discussing others contributions. Notice also how the number of active forks is much lower than the number of total forks.

With the values of the parameters found in Figures 5.11 to 5.14, the popularity map can be found with the same approach used for size: the values are flattened to a bi-dimensional space trough PCA. The resulting maps, with only the repositories with issue tracker entries, are shown in Figures 5.11 to 5.14. At first impression, these maps are similar to the size of the repositories, which is expected since a popular project is more likely to be mature, and therefore grown, than an unpopular one. After a closer look, however, many repositories in Figure 5.11 are different:

big projects such as *mono* and *mongodb* have moved to the cluster of less popular repositories, while most of the remaining repositories are related to web-development. For examples, web development projects that appear as popular are *django*, *symfony*, *d3*, *impress.js*, *jquery*, *node*, *html5-boilerplate*, *rails*, *fundation*, *three.js* and *chosen*. This suggests that web development projects are more popular than the others.

Moreover, in this figure, one can notice that there seems to be two axis coming out of the repository clusters: one ending with *homebrew* and one with *node/jquery/html5-boilerplate*. By looking at the raw, single repository level data, one can conclude that *homebrew* has a high number of "active" users, with more authors, issue commentators, pull request commentators and active forks then the other projects. *html-5-boilerplate* and its neighborhood have less active users, but higher watchers and, in proportion, total forks. Therefore, the two axis may be read here as "active" popularity (projects which attracts more active users) and "passive" popularity (projects with many users who are only observing the project).

Figure 5.12 looks quite similar to the same plot that was done to assess the size (Figure 5.4), which suggests that the smaller projects are also less popular, and that most repositories in the ANDROID dataset are both small and not popular.

In Figure 5.13 the number of repositories is halved with respect to the size map (Figure 5.6). In particular, most of the repositories that were placed on the outer border, far away from the left cluster in the size map are not present since they lack the issue tracker. This may suggest that bigger projects tend to use their issue tracker rather than using GitHub integrated one. *coffeescript* is also notably farther away from the other repositories, possibly due to its number of issue commentators being much higher then the other projects.

Figure 5.14 shows more repositories than the size map. *atom* and *electron* are still farther away from the left cluster, but *browser-laptop*, which was more different in terms of size from the majority of the projects, has moved near the cluster in popularity. The one thing that can be understood from this figure is that many project built on top of *electron*, while not being so big when compared to *electron* itself or *atom*, are still popular among users.

In Figures 5.15 to 5.17 the popularity maps for ANDROID, LANGUAGES and ELECTRON, without the pull requests and active forks parameters, are shown. The popularity map for MSR is not shown since it is not affected by the bug of the missing pull request data. The popularity map for ANDROID, shown in Figure 5.15,

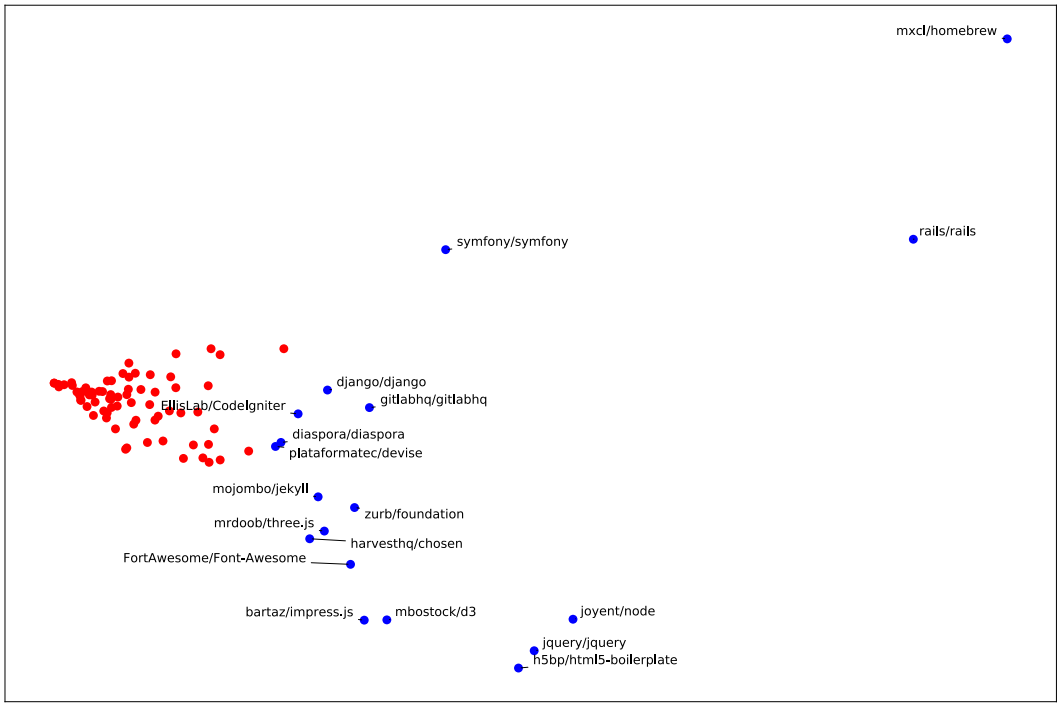


Figure 5.11: Popularity map for non forked repositories and non empty issue tracker repositories for MSR14.

Figure 5.12: Popularity map for non forked repositories and non empty issue tracker repositories for ANDROID.

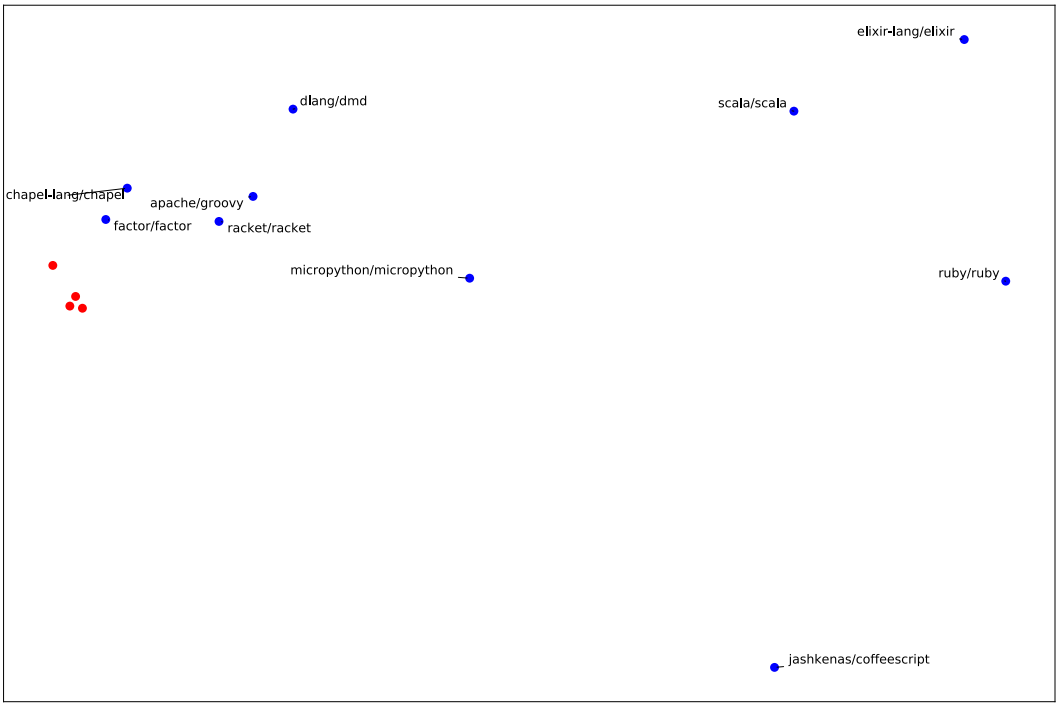


Figure 5.13: Popularity map for non forked repositories and non empty issue tracker repositories for LANGUAGES.

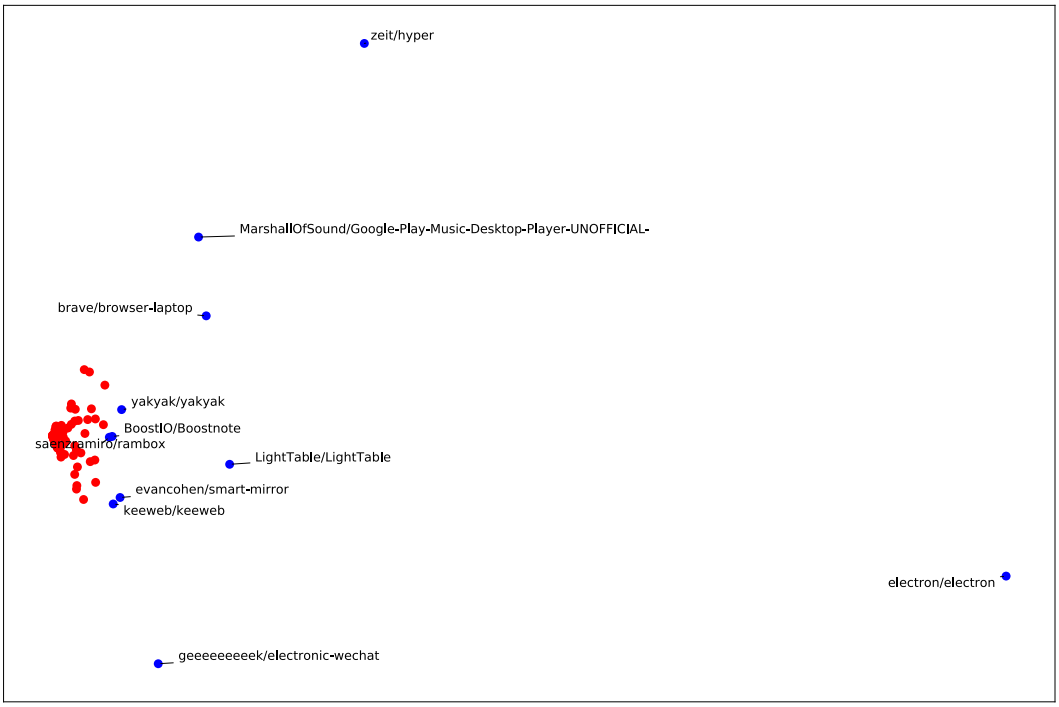


Figure 5.14: Popularity map for non forked repositories and non empty issue tracker repositories for ELECTRON.

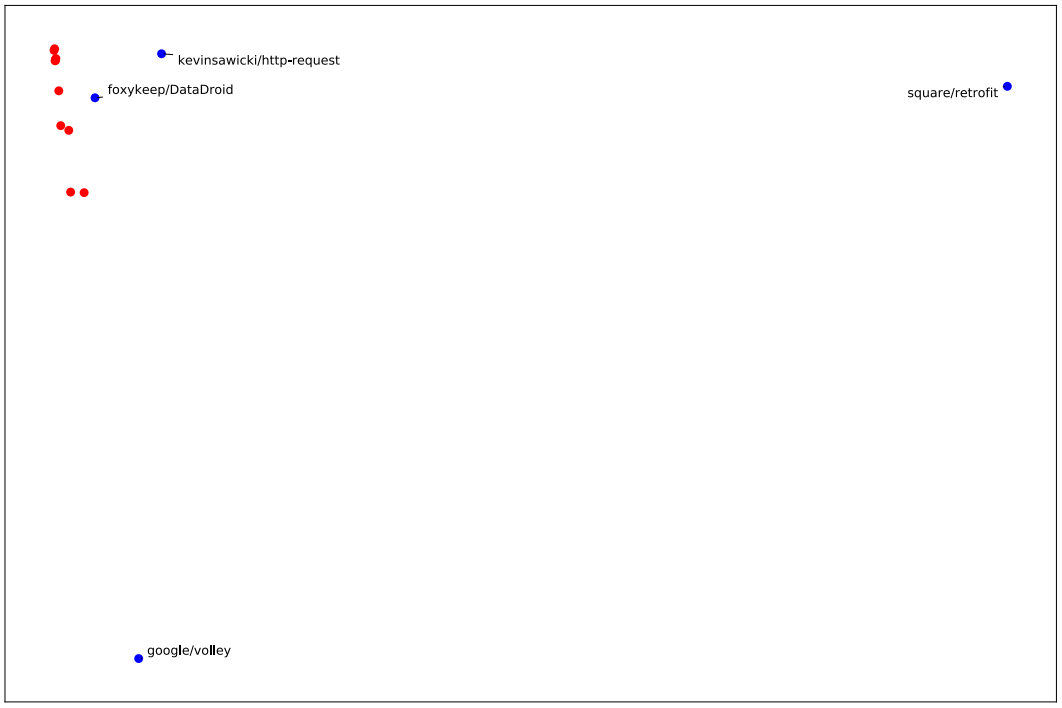


Figure 5.15: Popularity map for non forked repositories, without pull requests and active forks, for ANDROID.

after filtering the repositories without the pull requests, does not change significantly from the one presented in Figure 5.12: the filtered repositories, with the exception of *orhanobut/wasp*, are in the cluster of small projects, and the distances between the repositories do not change.

In Figure 5.16 there are more repositories than the previous version with filtered repositories (Figure 5.13), both in the cluster of similar projects and outside of it. The common nodes seem to maintain similar distances, which may indicate that the additional parameters do not provide a significant differentiation on this dataset.

Figure 5.17 is different from the version of Figure 5.14 with repositories filtered and more parameters used: the minor number of parameters makes repositories closer, and more projects are now in the cluster of similar repositories. Distances are also different among the two versions: *LightTable*, for example, is closer to *browser-laptop* than to *Google-Play-Music-Desktop-Player-UNOFFICIAL* in the map with filtered repositories, while in Figure 5.17 it is the opposite. Still, the two maps appear similar, and there are no significant differences.

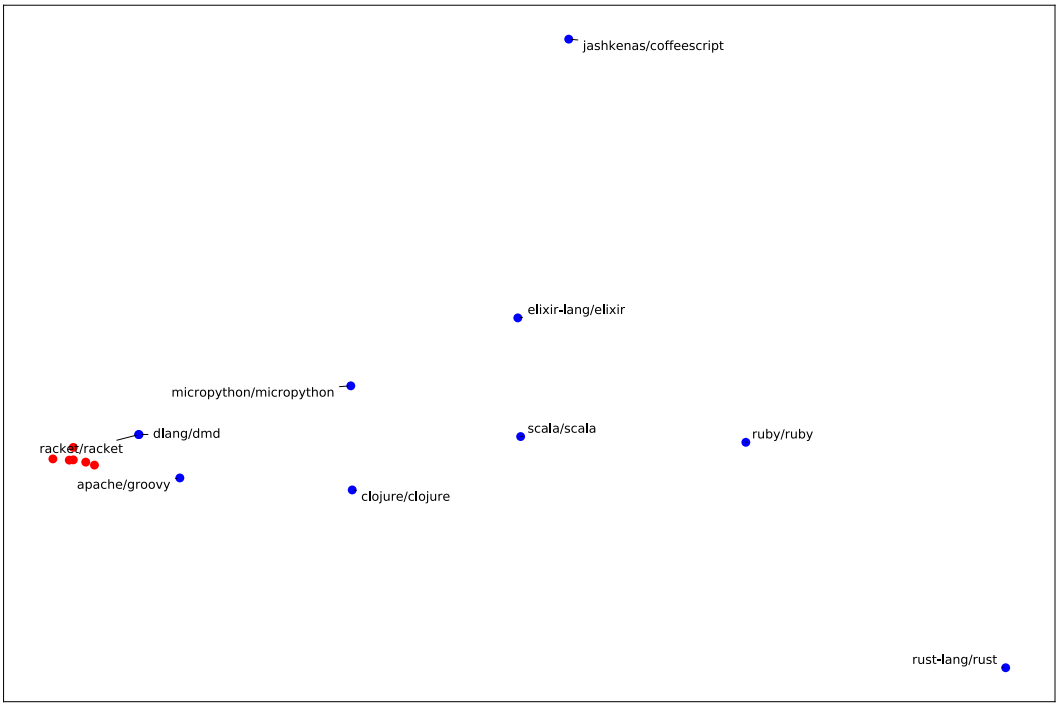


Figure 5.16: Popularity map for non forked repositories, without pull requests and active forks, for LANGUAGES.

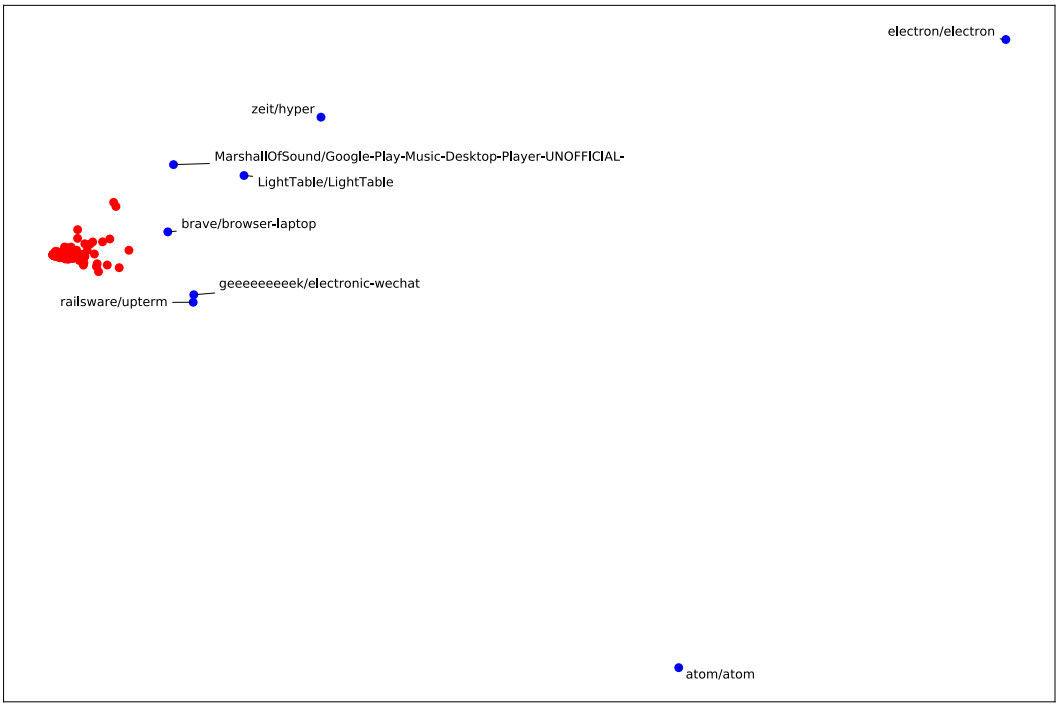


Figure 5.17: Popularity map for non forked repositories, without pull requests and active forks, for ELECTRON.

5.3 Note of commit authors

When counting the number of unique authors, two assumptions were made:

- each author is using a unique email address for his/her commits;
- each email address is used by an unique author.

These assumption are reasonable, however, they proved to be false. Therefore, some degree of error is introduced in the results due to these unsatisfied assumptions. The introduced error is expected to be relatively low for inter-project level statistics. Nevertheless, it is useful to study the data and measure this error for further considerations.

Let's consider, for example, the most active committers for the repository *jquery/jquery*, from the MSR14 dataset. *jquery* is a popular and large repository, and is expected to have many contributors and contributions. Through map-reduce, the unique committer emails and the count of the commits they produced can be extracted from the *commits* collection. In total, 320 unique email addresses are associated to 5,983 commits in the repository. Out of these, only 12 unique emails pushed more than 100 contributions, and only 58 emails wrote more than 5 commits. The distribution of the commits for each email address is shown in Figure 5.18, where it can be noticed that the most active contributor, `jeresig@gmail.com`, made as many commit as all of the users who are not in the top 10 committers. This suggests that most contributions come from few users, while less contributions are coming from a high number of users.

Considering a unique email address as a user, however, introduces two possible distortions in the results:

1. a user may be using more than one email address, in which case a single user is detected as multiple users;
2. the same email address may be used by different users, therefore multiple users may be detected as a single one.

The first error can be seen in Figure 5.19, where the timeline with the commits produced by the top contributors over the project duration is shown. The users *jaubourg* and *timmywill*, in particular, are using different email addresses. Notice also how the

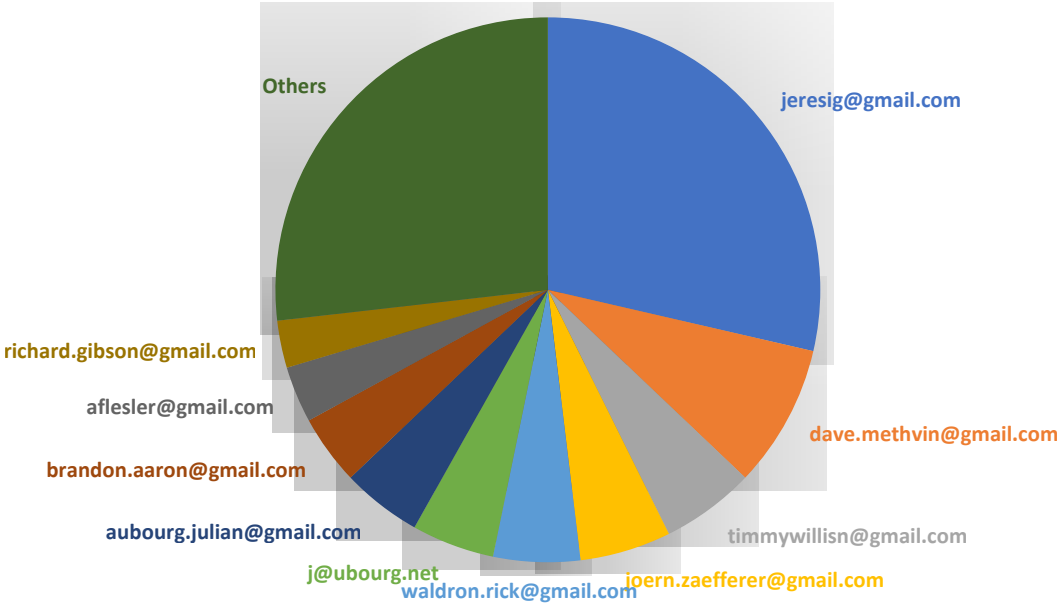


Figure 5.18: Committers for jquery using emails (Top 10 + others).

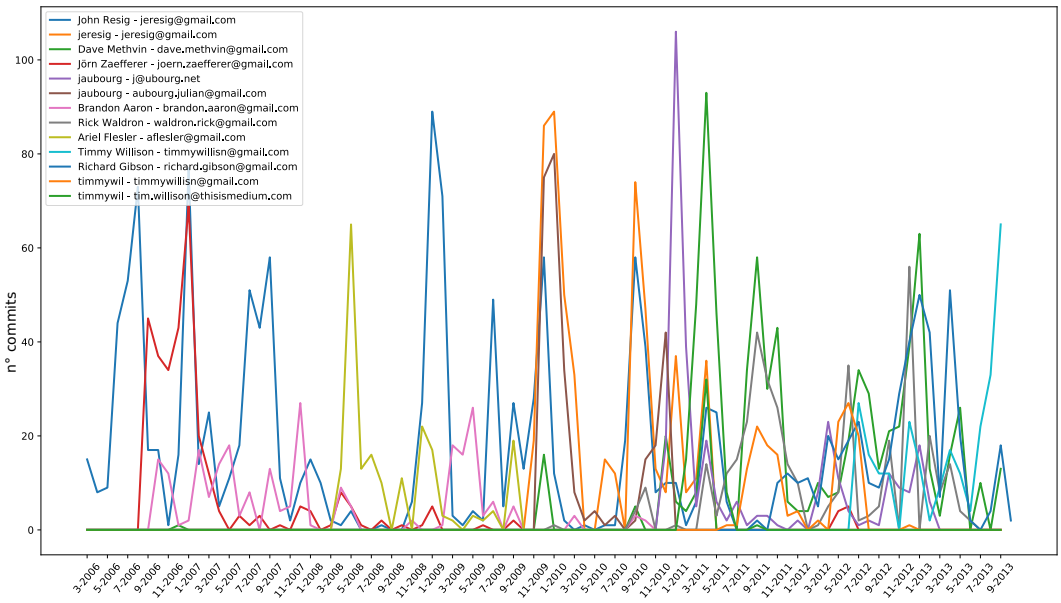


Figure 5.19: The number of commits produced by the top contributors for each month.

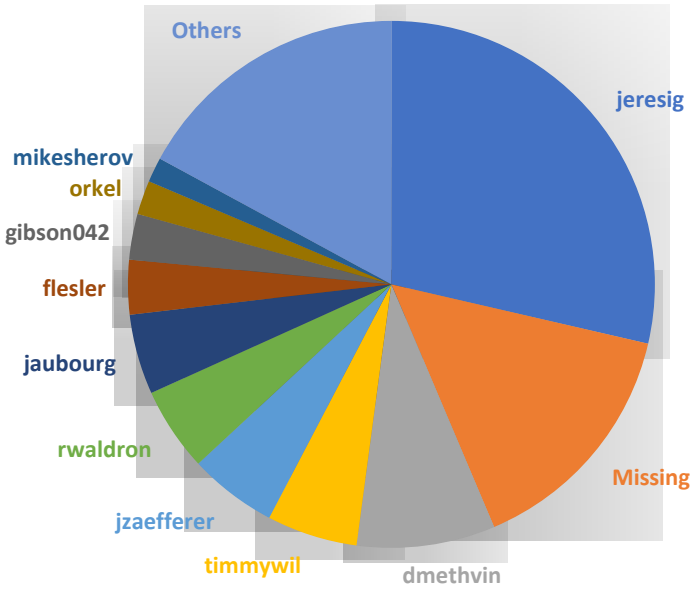


Figure 5.20: Committers for jquery using logins (Top 10 + missing + others).

same email address appears for different author name, such as `jeresig@gmail.com`, which is used by *John Resig* and *jeresig*, and `timmywillisn@gmail.com`, which is used by *timmywill* and *Timmy Willison*. For these two examples the different names seem similar enough, and therefore one can assume that they belong to the same user and thus they could be merged, but this may not always be the case.

Given that the GitHub login can be assumed correct, when present, one can think of using it as a mean to distinguish the authors. Following this approach the results change slightly: only 9 unique logins produced more than 100 commits, with 47 logins producing more than 5. Instead of the previously counted 320 emails, a total of 256 logins were found in total from the commits. On the other hand, 898 commits did not include the login field, and their author is therefore unknown.

The distribution of the commits made by authors considering logins can be seen from Figure 5.20, which is is similar to the previously seen pie chart of commits considering authors by email (Figure 5.18). Again, most commits come from few users. Here, however, it can be seen that the number of commits made by unknown logins is significant. Moreover, while many logins can be imagined to belong to similar emails, the email `brandon.aaron@gmail.com` is not appearing in the top 10 committing logins. This may be either because this email is used by one of the other logins or because the login cannot be associated by GitHub.

In summary, the GitHub login produces reliable data, at the cost of possibly

email	unique logins
flangy@gmail.com	2114.0
jacknagel@gmail.com	736.0
fabien.potencier@gmail.com	338.0
jose.valim@gmail.com	298.0
source@sharpsteen.net	248.0
info@bnoordhuis.nl	236.0
mike@mikemcquaid.com	230.0
ry@tinyclouds.org	188.0
jeremy@bitsweat.net	172.0
pratiknaik@gmail.com	162.0

Table 5.8: Counts of different logins that created commits with the same email address.

having missing data (18% in the considered example). One could think of using the GitHub login when available and some heuristics on username and email addresses when the login is missing. To produce meaningful results, however, it is still required that an email address is not used by more than one person, which seems to be a reasonable assumption. Since the GitHub login can be assumed correct, it can be verified that it does not occur on the used datasets that the same email is used by more than a single login. To do so, a map-reduce can be used to count the unique logins associated to an email address. The results of this evaluation are shown in Table 5.8.

These results may be surprising: either the same user, using the email address `flangy@gmail.com`, created more than 2.000 accounts on GitHub or more than 2.000 users willingly entered the email of someone else in their GIT configuration. Both scenarios are unrealistic, as there is hardly any conceivable reason to have multiple GitHub accounts, and there are no reasons to enter someone else email address as well.

A quick search on GitHub web interface for *flangy* finds a single user, named *Adam Vandenberg*, with its personal profile page with the email of `flangy@gmail.com`. Among his repositories, one, named *dotfiles*, has description *My OS X configuration files and utilities*. Within this repository, a file named *git-config* contains a pre-built

configuration for Git clients for the OSX operating system, with name set as Adam Vandenberg and email as `flangy@gmail.com`. A similar repository can be found for the second email in the list, `jacknagel@gmail.com`, which again features a Git configuration file. Unlike the previous repository, this one has removed the reference to the user and his email, but these data can be seen in the previous versions through the commit history.

While there is no proof to it, a reasonable explanation would be that many users used pre-built configurations without changing the username and email. This would also explain why 405 unique login have produced both commits with the email `flangy@gmail.com` and with the email `jacknagel@gmail.com`. In general, this suggests that one can not assume that user data in commits is correct, and that users can make mistakes in configuring their environment. This also discourages heuristics to match users through the tuple name-email-login: it is not guaranteed that a commit without login, created by a user with the same email and name of a commit with login, is created by the same user of the latter.

Therefore, reasonings must be made either using logins (no errors, missing data) or through email addresses (no missing data but introduce an arbitrary error).

5.4 Sparsity of contributions

In Section 5.3 the contributions to the *jquery/jquery* repository were discussed as an example, and it was shown that most contributions come from few users, while most of the users provide a few contributions. It is thus interesting to evaluate whether this is a general situation that is shared by other repositories in the datasets. This analysis provides a good understanding on how contributions do work among the repositories.

Figure 5.21 shows how many authors for each repository have contributed with a single commit to that repository. The X axis reports all the 230 non forked repositories considered in the four datasets, while the Y axis is the percentage of authors that contributed with exactly 1 contribution to that repository. Since there is a high variance on the total number of authors among different datasets, the number of authors with 1 contribution is shown as a percentage with respect to the total number of authors for the same repositories, according to Equation (5.1).

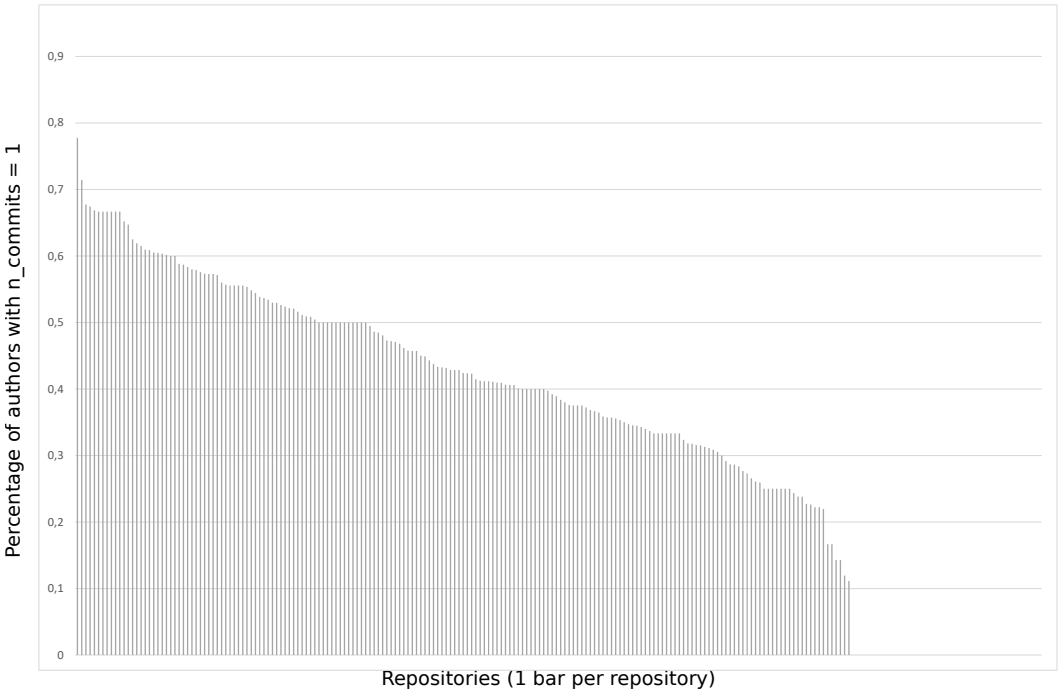


Figure 5.21: Percentage of authors with exactly 1 commit for each repository; 30% of repositories have 50% or more authors with exactly 1 commit.

$$\% \text{ contributors} = \frac{\text{number of contributors with } N \text{ commits}}{\text{total number of contributors}} \tag{5.1}$$

To derive the results depicted in Figure 5.21, it was set $N = 1$. The corresponding figure shows that 30% of the repositories have 50% or more authors that made exactly 1 commit to the project. This result confirms that a significant amount of contributors has only a few contributions made to a project.

In Figure 5.22 the same approach is repeated by setting $N = 5$ in Equation (5.1), i.e., counting those authors who provided 5 or commits less. The results show that, for 80% of repositories, 50% of authors provided only 5 or less commits. It is worth to observe that 5 commits can still be considered a small number of contributions for many projects (see Figure 5.2 for the evaluation of the number of commits in the considered repositories).

Finally, Figure 5.23 depicts the chart derived from Equation (5.1) by setting $N = 10$. It results that 86% of the repositories have 50% or more authors that performed 10 or less commits, and 50% of the repositories have 80% or more contributors who wrote 10 or less contributions. The right side of the graph shows

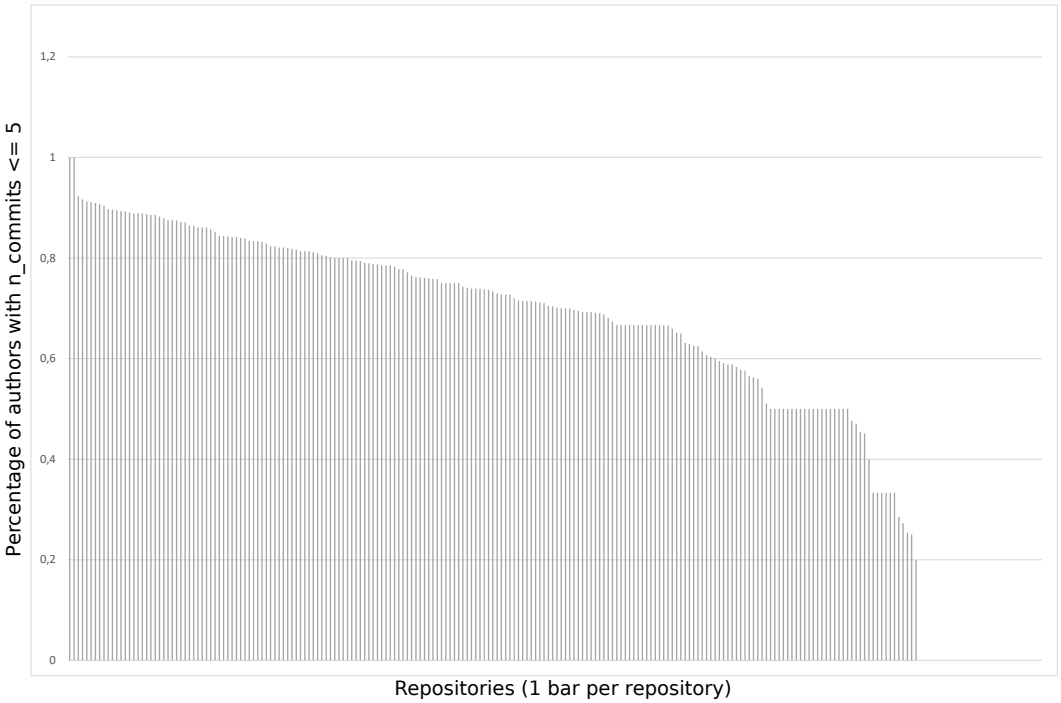


Figure 5.22: Percentage of authors with 5 or less commits for each repository; 80% of repositories have 50% or more authors with 5 or less commits.

that there are 10% of the repositories having no contributors (zero contributors) that wrote 10 or less commits. In other words, every author of these repositories has contributed more than 10 commits.

In summary, for most repositories in the considered datasets, a significant amount of authors, which for many repositories is the majority of the contributors, has written only a very limited amount of commits.

On the other hand, for 13% of the repositories studied this conclusion does not hold. A further manual evaluation of such repositories shows that 27 of 31 projects have only 1 to 3 contributors, who wrote all of the repositories commits. Therefore, it can be concluded that the results on the sparsity of the commits per authors do not hold for small repositories with a limited amount of contributors.

Finally, an interesting aspect that is considered is the number of projects that is contributed by an author. The pie chart in Figure 5.24 shows how many repositories an author contributed to in the datasets used in this thesis. The graph is constructed by counting, for each author, the number of unique repository he/she has contributed. The forked repositories are detected and discarded. The number of authors that

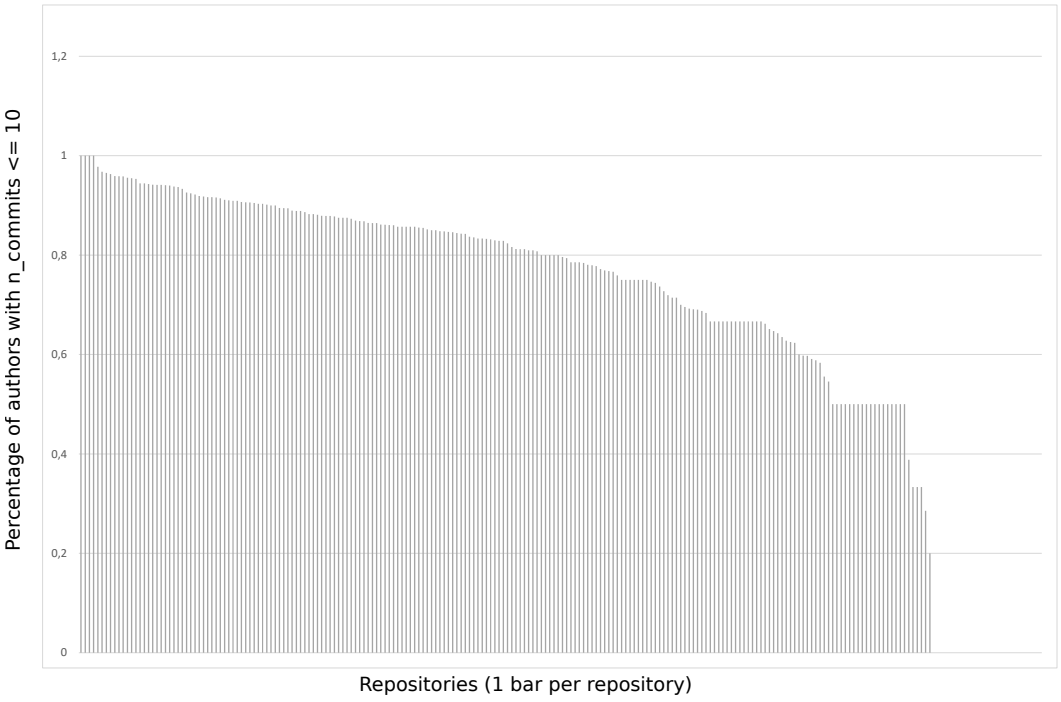


Figure 5.23: Percentage of authors with 10 or less commits for each repository; 86% of repositories have 50% or more authors with 10 or less commits, and 51% of authors have 80% or more authors who contributed with 10 or less commits.

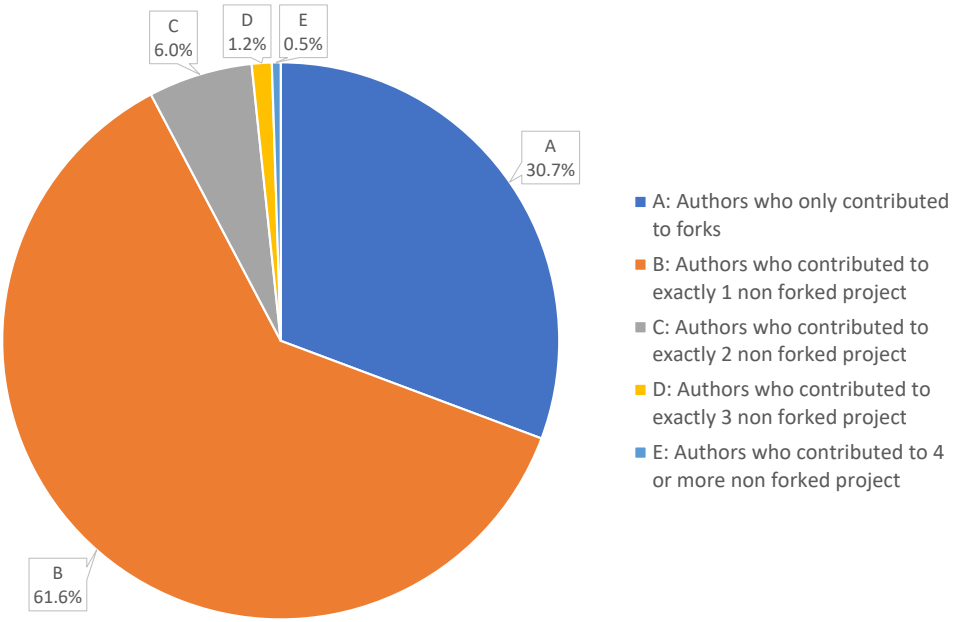


Figure 5.24: Number of repositories that was contributed from each author; 30.7% contributed only to a fork (0 repositories), 61.6% contributed to exactly one non forked repositories, and only 7.7% of authors contributed to two or more non forked repositories.

contributed exactly to X repositories are then counted, with X ranging from 0 (the author only contributed to forked repositories) to 12, the maximum number of unique repositories an author contributed to. The result is that 30.7% of users only contributed to forks (the pie slice 0), but the graph shows that most contributors (61.6%) only contributed to 1 repository. In total, only 7.7% of the authors contributed to 2 or more repositories.

The number of authors who committed to 2 or more repositories is relatively low, as shown in Figure 5.24. Table 5.9, on the other hand, shows that – on

Number of repositories	Average commits
1	28.76
2	82.66
3+	227.51

Table 5.9: Average number of commits per author who committed to exactly 1, exactly 2 and 3 or more repositories.

average – authors who contributed to more than one repository wrote many more contributions than users who contribute to one repository only. In particular, authors who committed to 3 or more repositories have on average almost 10 times the commits count of users who contributed to only 1 repository. The low number of common authors between repositories, therefore, is partially compensated by the fact that, on average, such users write more contributions, and are therefore more interesting to study than "non common" authors. In general, the scarcity of common authors between repositories may be a limitation and must be accounted when applying the presented approach on other datasets.

Chapter 6

The network graph

The bi-dimensional maps presented in Chapter 5 illustrate concepts which can not be captured by simple metrics, and allow for comparisons in a set of projects through graphical representation of distances. Their purpose is mainly to help a user in choosing the better repository for his needs, either by understanding which are the bigger and most popular repositories and by comparing unknown projects to known ones.

This chapter proposes a different approach to characterize repositories: an approach to analyze the interactions of developers who work on multiple repositories. The times in which each contribution was written may either be ignored, to create a situation in which the whole history of the interactions is summarized (*static* analysis) or may be used as a dependent variable, to analyze how the contributions were made in a particular time interval or observe the evolution of the contributions over time (*dynamic* analysis).

The idea is that a developer contributing to a project acquires a certain degree of expertize on the topic, the language, the libraries and the techniques used in that particular project. If the same developer also contributes to other projects, a part of his acquired knowledge is transmitted to those projects through his contributions. It is therefore expected that two projects sharing many contributors acquire a certain amount of common knowledge, and, consequently, that the two projects have similarities in libraries used, techniques and ideas.

As a way to provide a graphical representation of such analysis, this thesis proposes the usage of directed graphs, with nodes representing both repositories and users, and edges representing the commits. To provide a meaningful representation,

the number of commits can be used as weights for the edges, and as the node sizes. This way, a certain degree of information on the repositories, the users and their interactions is captured by the graph.

As mentioned, a dynamic graph adds the time dimension to the final results. This result can be achieved, using Gephi, through a moving window over the time dimension, showing only elements that have non zero value within that time window, and adapting node sizes appropriately. The obtained result resembles the representation provided by Gource [18], a visualization tool for users activities on a repository. Gource, however, provides a representation of a single repository, and does not address the aspects of multiple repositories. Moreover, Gource offers visualization at file level of the repository, while file details are not relevant for the proposed approach.

6.1 Implementation

Three steps are involved in the construction of a network graph: 1) data retrieval, 2) data processing and 3) graph visualization.

6.1.1 Data retrieval

To have a meaningful analysis on projects interactions through contributors, it is necessary for some repositories in the dataset to share common users. Having repositories with no common contributors, would obviously lead to not have any interaction at all to evaluate.

The knowledge that a set of repositories does have few or no contributors in common between projects is a result by itself, and may be further analyzed, e.g., by an in-depth analysis on the projects developers, their history and their goals. The proposed approach does account for this situation to possibly happen, and the resulting graph would show the projects as sets of non connected nodes.

In Figure 6.1 a pseudocode that illustrates the algorithm used is shown. While the pseudocode uses a C-like syntax, to achieve a clear and concise representation some conventions have been established:

- dictionaries are indicated in Python language syntax, with curly brackets for declaration ({}) and *dictionary['field']=value* syntax for additions;

```
repos_to_scan = start_repo;
found_authors = {};
found_repos = [];

for (d=0; d<max_depth; d++){
        next_repos = []

        for (repo in repo_to_scan){
                commits = find_commits_for_repo(repo);
                authors = find_authors_from_commits(commits);

                for (author in authors){
                        commit_counts = find_commit_counts(author,
                            commits);
                        timeline = find_timeline(author, commits);

                        if (! author in found_authors){
                                found_authors[author] = {}
                        }
                        found_author = found_authors[author];
                        found_author[repo] = {'timeline': timeline, '
                            commit_counts': commit_counts};

                        repos_from_author = find_repos_from_author(
                            author);
                        repos_from_author = repos_from_author -
                            found_repos;
                        found_repos += repos_from_author;
                        next_repos += repos_from_authors;
                }
        }
        repo_to_scan = next_repos;
}
```

Figure 6.1: Pseudocode of the algorithm for data retrieval.

- to cycle through an array of objects, the syntax *for (x in array)* is used (again, inspired by the Python language);
- arrays concatenation of unique elements is shown with the += operator (e.g., A+=B adds to the array A each element of B if not already present in A).

The algorithm to generate a network graph starts from a repository name, *start repo* and a max_depth parameter which is used to determine the desired max level of recursion. At the first level of recursion, the algorithm finds only the repositories connected to *start repo*. Two repositories are connected if at least an author is in common with *start repo*. Increasing the level of recursion, the algorithm explores repositories that are not directly connected to *start repo*, but are connected to it along a multi-hop path.

Lines from 1 to 3 are initializations: `repos_to_scan` contains the name of the repositories to be considered in the upcoming iteration, and `found_authors` and `found_repos` will be used to store the algorithm results.

In line 5, a main loop is used to iterate over the recursion level. Line 6 is used to initialize a variable, `next_repos`, which will contain the repository to be considered in the next recursion iteration.

Lines 8-10 start an iteration over repositories in `repo_to_scan`, and, for each repository, finds the commits and, from the commits, the authors. Then, in lines 12-20, for each author in the newly find authors set, the commits from the current repository are found, both as a raw count for the static graph (line 13) and as a timeline for the dynamic graph (line 14). The commit statistics are then added to the dictionary of results for authors (line 20), but only after finding the author element and adding it if not present (lines 16-19).

In line 22, the whole collection of commits in the dataset is scanned to find all the repositories where the author contributions are present. Repositories that were already considered in previous iterations are not scanned again. Therefore, they are removed from the array of found repositories in line 23. In line 24, on the other hand, the newly found repositories are added to the `found_repos` array. Line 25 is used to update `next_repos`, since the newly found repositories will be eventually scanned in the next iteration.

Finally, in line 28, `repo_to_scan` is updated for the next recursion iteration using the values in `next_repos`.

In the implementation of the algorithm, to reduce computational times, 3 map-reduce operations can be used to generate the author list for a repository beforehand, the repository list for an author and the commit timelines and count for each repository of each author.

6.1.2 Data processing

Data collected from the dataset requires a preliminary processing before being used. Part of the pre-processing consists in the removal of data that is not useful for the intended analysis, and part depends on the desired results (user settings). The settings that were implemented in the algorithm are described in the followings, but a summary is provided in Table 6.1.

Setting	Description
`use_login`	If true, GitHub logins are used to detect commit authors. If false, committer emails are used instead
`max_depth`	Max recursion depth for the algorithm
`exclude_forks`	If true, the forked repositories are discarded
`forks_only`	Keeps only forked repositories and starting repository
`keep_only_one_contributors`	If true, users who contributed to a single repository are kept. If false, they are discarded
`min_commits`	Discards user who did not make at least min_commits commits to a repository
`force_include_repositories`	Force to keep repositories in this list, even if they are not linked by contributors to graph results
`force_exclude_repositories`	Discards any repository in this list, without considering these repositories when detecting users

Table 6.1: Settings for the network graph algorithm.

The collected authors must be checked for any data inconsistency. If logins were used to identify authors, there may be null entries for missing commits that must be eliminated. If emails were used instead, either manual or semiautomatic systems can be used to merge different users here, albeit no automatic merging system was implemented in this work. The known list of problematic emails associated to invalid users (see Section 5.3) must also be eliminated, as such data can not be considered reliable.

As shown in Chapter 5, forked repositories present statistical characteristics that are different from non forked ones. Depending on the user's goals, analyzing forks may either be interesting or generate unwanted noise in the resulting graph. Therefore, while processing data, the user needs to choose whether he wants to keep forks, eliminate them or keep only the forks and remove non forked projects from the results. The two settings `exclude_forks` and `forks_only` map the two described behaviors.

Authors that contributed to only one project are not interesting for analyzing common contributions, and may therefore be discarded. On the other hand, showing those nodes with a different graphic style (e.g., a different color) provides a graphical way to get how common contributors and one-repository-only contributors are

distributed in the dataset. The setting `keep_only_one_contributors` can be used to choose whether to keep or discard these users. Similarly, a threshold can be used to filter authors with only a few commits (setting `min_commits`), when graphs are overpopulated by too many users with few contributions. Both settings are strongly dependent on the dataset and the repositories involved, thus the best combination has to be found by means of a trial and error process. In particular, when the resulting graphs are crowded by a high number of user nodes, these settings may be used to remove the ones that provide less information. In the followings of this chapter, many examples are presented and highlight the differences.

If there is the need to include repositories that would not appear in the resulting graph otherwise, for example because these repositories do not share any contributor with the graph results, the setting `force_include_repositories` may be used to define a list of repositories to be included regardless. Similarly, if some repositories must be excluded from the results, for example because of a partially bad dataset with invalid entries, they can be removed by filling the list `force_exclude_repositories` with their names.

The visualization will be done in Gephi (see Section 4.4). The GephiStreamer plugin[1] is used for this purpose. GephiStreamer provides two Python classes, *Node* and *Edge*, to map objects to nodes and edges, respectively. Both classes must have a label (specified by the class constructor) and can have a set of custom properties. Some custom properties, such as *color*, x and y are interpreted by Gephi as node properties and used for rendering. Properties which are not internally used by Gephi are still attached to Node objects, and made available for Node sizing and coloring through ranking or partitioning.

In preparing Node and Edge objects, it is important to use the required data format. Numbers must be extracted from strings, and commit times must be converted from date strings to Unix timestamps, also called *epochs*[2]. Commit timestamps for dynamic graphs should be sent as arrays. However, GephiStreamer does not currently support arrays. The solution is to represent the arrays as strings in the

[1] Gephistreamer plugin on GitHub, last visited on January, 12th 2018: `https://github.com/totetmatt/GephiStreamer`.

[2] Unix time, or epoch time, is a method for describing a point in time. It is defined as the number of seconds elapsed from 00:00:00 UTC, Thursday, 1 January 1970. The count takes into account the number of leap seconds that have taken place since then.

Gephi format: the whole array must be wrapped in the '<' and '>' symbols, and each element must be wrapped in square brackets, separated by a semicolon. Since timestamp intervals with values should be sent, each array element contains the first and last timestamp of the interval, and the commit number. For instance, to represent an author pushing 10 commits between timestamp 0 and 10, and 100 commits between 20 and 30, the string would be formatted as:

```
<[0.0, 10.0, 10]; [20.0, 30.0, 100];>
```

The number of commits performed by a user on a repository is set as an Edge weight to allow proper node positioning (see Section 6.1.3). To help Gephi nodes positioning algorithms, it is convenient to set a custom x and y coordinate for each Node. This is done since Gephi uses force-driven algorithms for positioning, and having the nodes in the same exact coordinates creates an unstable equilibrium. Here, the nodes properties x and y are set randomly in a $[-1, 1]$ interval.

After these steps are performed, the resulting data can be sent to Gephi for visualization.

6.1.3 Graph visualization

When all of the data is received, Gephi represents the results in its interface as shown in Figure 6.2, which depicts the results of the algorithm run for the repository *jkbr/httpie* (`max_depth`=1, `exclude_forks`=True, `min_commits`=1, `keep_only_one_contributors`=False). The graph preview is shown in the center of the figure, and depicts a colorless graph with all of the nodes and the edges almost on the center of the canvas, with no size set.

On the top left of the window, the Appearance tab offers tool to color and size nodes with static values or based on parameters. Node sizes, in particular, are required to be set before node positioning, since node positioning takes this parameter into account. When an animated graph is desired, the node sizes should be set to a pessimistic size (e.g., the biggest number of commits in the history, or the sum of commits in the whole history) to provide better results.

On the bottom left of the window the *Layout* tab should be used for automatic node positioning. ForceAtlas2 (see Section 4.4.1) is the positioning algorithm that provides the best results for these graphs, as it also accounts for the edge weights

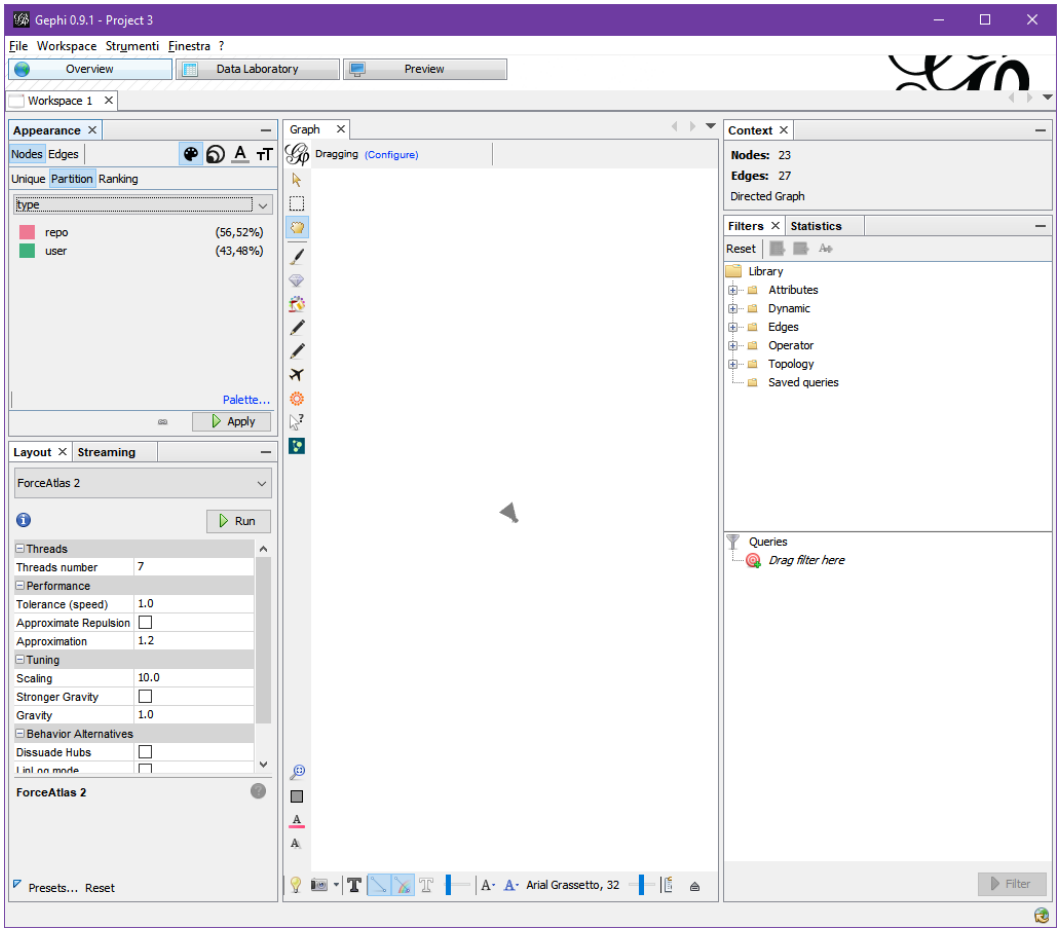


Figure 6.2: Gephi interface when data streaming is complete.

(the number of commits a user performed) to position the nodes. In particular, ForceAtlas2 tries to place a certain user closer to a repository when that user has made more commits to that repository then to other repositories. For example, if a user has 10 commits to repository A and 100 commits to repository B, the user will be placed closer to A. A user with 50 commits to A and 50 commits to B, however, will be placed mid-way between A and B.

The user must still fine tune the scaling (repulsion between nodes) and gravity (how much the nodes are attracted to the center) until a reasonably understandable layout is found. The *LingLog mode* of ForceAtlas2 can be used when clusters have too many nodes, since it tends to make the visualization more compact, albeit the whole graph assumes a circular shape.

In balancing gravity and scaling, it may occur that user and repositories nodes overlap, partially or completely. ForceAtlas2 features a no overlap flag to prevent nodes from overlapping, however it often results in achieving an unstable equilibrium in the nodes positioning, with resulting graph rotating and floating from the center of the map endlessly. Therefore, for the presented graphs, two more node positioning algorithms were run after ForceAtlas2 achieved a reasonable node positioning: Noverlap, which slightly moves nodes to prevent their overlap, and Label adjust, which slightly moves the nodes to prevent the overlap of their labels.

The results after node positioning with ForceAtlas2, coloring based on Node type and sizes based on number of commits are shown in Figure 6.3. In this figure, in particular, the user nodes are colored in red, and the repositories nodes are colored in green. The size of the nodes has been set between 20 and 150, proportional to the number of commits. As a result of edge weighting, user nodes are more attracted by repositories to which they have contributed more. For example, most of the contribution from the user *Southern* were done to *joyent/node*, some contributions were found in *jquery/jquery* and only one contribution was present in *jbkr/httpie*. On the opposite, the user *faulkner* contributed much more to *jkbr/httpie* than he did on *jquery/jquery*.

More details on the interpretation of this graph are presented in Section 6.2.1.

Animated graphs are handled by Gephi through a dynamic filter, which is controlled by a bottom slider in the user interface. To tune the visualization, the user can choose the playback speed and the length of the considered interval over the bottom slider. Moreover, node sizes can be automatically changed during the

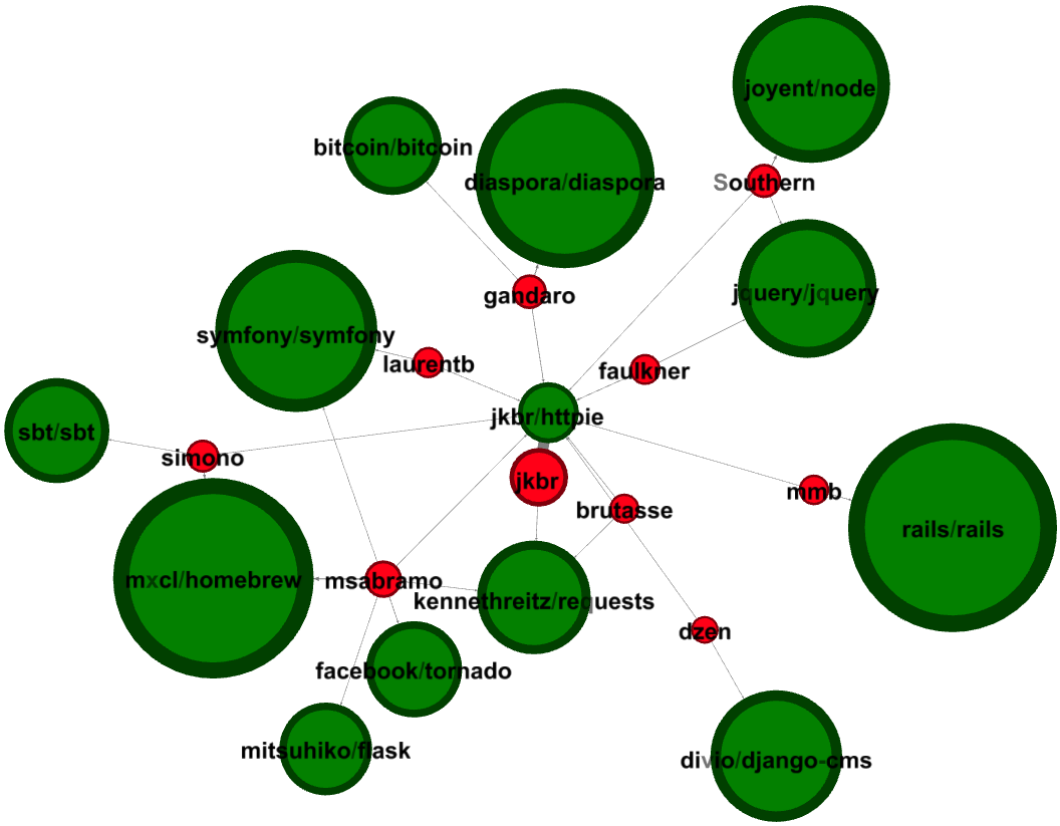


Figure 6.3: Gephi graph for project *HTTPie* after color, size and positions have been set.

playback if the dynamic property is set.

6.2 Interpretation of results

In this section, some examples of applications of the network graph algorithm are shown. The presented projects were manually selected on the basis of specific characteristics and the gathered knowledge on the considered datasets and repositories. The guiding idea in choosing the projects is to use the network graph algorithm to analyze repositories with different contexts and background. It could be expected that, for example, a project born as a debug tool would result in a different graph organization than the one generated when analyzing a web framework project. The interactions between repositories and users in each example are explained and are therefore representative of possible patterns in such interactions.

6.2.1 MSR14: jkbr/httpie

The first presented graph is created from *jkbr/httpie*, which was used as a sample graph in Section 6.1.3 and is shown in Figure 6.3. The *HTTPie* project is a Command Line Interface (CLI) tool to perform HTTP requests, whose development started in 2012.

In the static graph, shown in Figure 6.3, *HTTPie* is the green circle in the center. The red circles around *HTTPie* are the authors who contributed to both *HTTPie* and at least another project. The numerical data that were used to elaborate this graph are reported in Table 6.2. The bigger author circle, *jkbr*, is the first author of the *HTTPie* project, who also contributed to the project *kennethreitz/requests*. The latter is a *Python* library to perform HTTP requests, therefore the two projects are similar: they offer a way to achieve the same goal (performing an HTTP request) through different means (a CLI or a *Python* software).

Most of the other repositories in the graph are web related repositories: *rails/rails* (*Ruby on Rails*), *facebook/tornado* and *mitsuhiko/flask* (*Python* web frameworks), *joyent/node*, *diaspora/diaspora* (a social network) and *jquery/jquery* (*JavaScript* libraries), *divio/django-cms* (strongly related to *Django*, which is a web framework). Other repositories such as *bitcoin/bitcoin* (*Bitcoin* project) and *sbt/sbt* are present in the graph even though it is unclear if they are using HTTP for any reason; however, their inclusion might just depend on having a common author with both the *HTTPie*

Node ID	Node type	Node total commits
rails/rails	repo	33073
mxcl/homebrew	repo	23339
diaspora/diaspora	repo	14306
symfony/symfony	repo	9936
joyent/node	repo	9018
jquery/jquery	repo	5983
divio/django-cms	repo	5068
kennethreitz/requests	repo	3212
sbt/sbt	repo	2522
bitcoin/bitcoin	repo	2064
facebook/tornado	repo	1918
mitsuhiko/flask	repo	1706
jkbr/httpie	repo	410
jkbr	user	351
msabramo	user	35
gandaro	user	19
Southern	user	17
simono	user	12
laurentb	user	6
faulkner	user	5
mmb	user	4
brutasse	user	3
dzen	user	2

Table 6.2: Number of commits for repositories and users in *HTTPie*.

project and another project with common authors. Finally, *mxcl/homebrew* is present in the graph as well, but its presence here and in other graphs resulting in the MSR14 dataset is analyzed in Section 6.2.3.

Since the distance between an author and a repository is dependent on the number of commits performed, one can conclude that most authors that contributed to *HTTPie* are mainly involved in web development, and their contributions on *HTTPie* do not represent their main focus. Given the web oriented nature of both *HTTPie* and of many other projects in the graph, a reasonable supposition is that the *HTTPie* project caught the interest of web developers as a debug/test tools for their main work.

Some more conclusions can be made from the animated version of this graph. For example, given that both *HTTPie* and *Requests* are written in *Python* language, and both serve similar purposes, it may be interesting to know which of the two repositories came first. Since the whole animation can not be represented in a printable document, the video was shared on the YouTube streaming platform[3].

The most notable conclusion that can be made from the animated video is that *HTTPie* was born after *Requests*. The user *jkbr*, in particular, contributed to requests only after the creation of *HTTPie*, albeit starting shortly after its creation, and committed to both repositories for an extended period of time. This suggests a bi-directional interaction between *HTTPie* and *Requests*.

The first frame featuring *HTTPie*, depicted in Figure 6.4, shows that many contributors were already contributing to the project during its first months of existence.

In summary, the following hypothesis can be formulated from both the statics and animated graphs of *HTTPie*: 1) *HTTPie* was possibly created as a development tool to test HTTP calls in web projects, and 2) more web developers were attracted to the project during its evolution.

6.2.2 MSR14: mangos/MaNGOS and TrinityCore/TrinityCore

The Massive Network Game Object Server Suite (*MaNGOS*) project was born in 2005, as an educational project. Its goal is to emulate a Massively Multiplayer Online Role-Playing Game (MMORPG) server, and particularly a World of Warcraft

[3] *HTTPie* animated graph, last visited on January, 12th 2018: `https://youtu.be/AEVKgMz7tbA`.

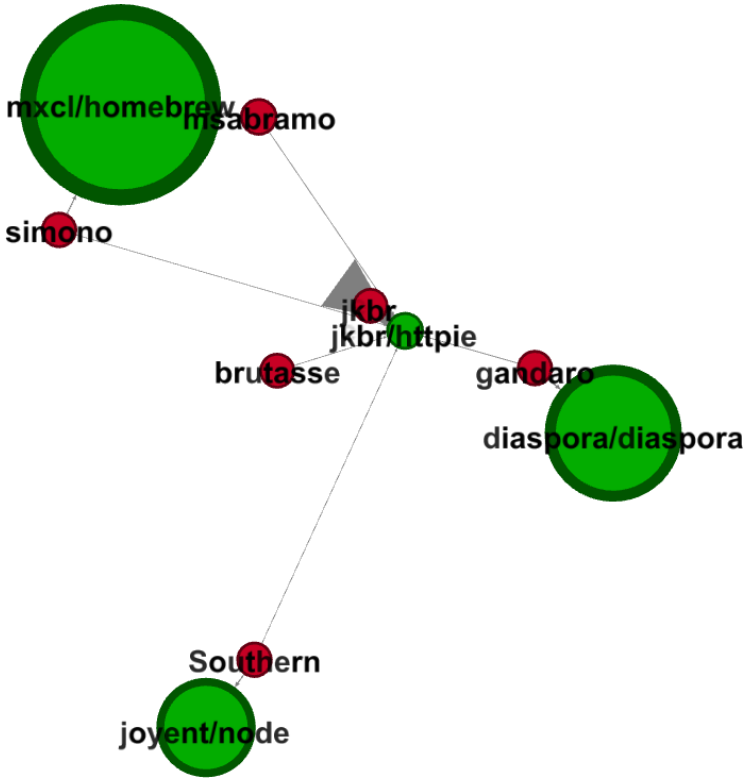


Figure 6.4: The contributions in the first months of *HTTPie*.

(*WoW*) server. While *MaNGOS* was born to be an educational and collaborative project, many users used the project to host their own *WoW* private server. Unlike the official *WoW* server, which is based on a pay-per-month model, the private servers were free projects, often offering premium content as a business model. This attracted a large number of players to the private servers, which were basically acting as beta testers for the *MaNGOS* project. Moreover, the same users hosting the private servers were creating bug-fixing contributions to attract more players, and many of these fixes were sent back to *MaNGOS* as well.

In the year 2008, some developers decided to split from the main project, and created their own fork, *TrinityCore*. This split is said to be happened due to an excessive attention from *MaNGOS* founder and main maintainer, *TheLuda* (GitHub login *danielsreichenbach*), to code style and code stability, which resulted in slower acceptance of code contributions.

A few years later, in 2012, a further split happened in the *MaNGOS* project, with many of the main developers creating the *CMaNGOS* project. *TrinityCore*, in

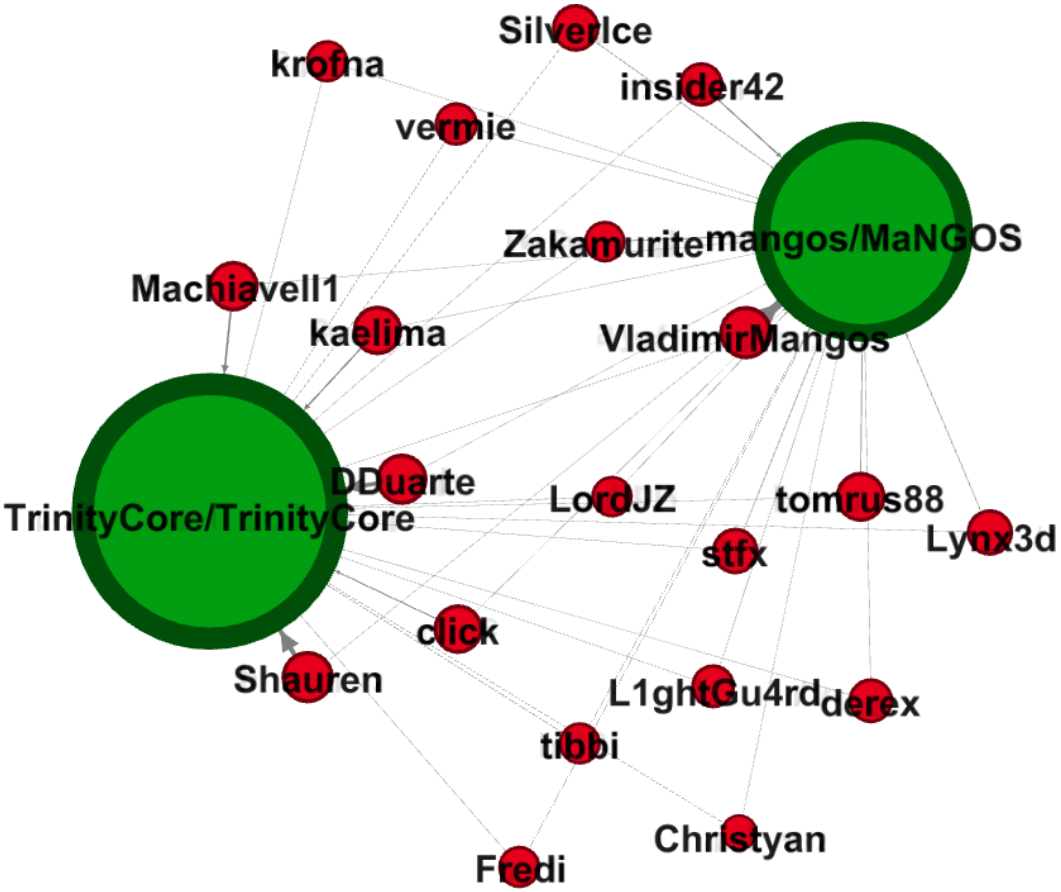


Figure 6.5: Static graph for *MaNGOS* and *TrinityCore*.

the same years, was one of the project with more contributors[4].

From this point onward, the activities on the original *MaNGOS* project are scarse, as contributors prefer either *TrinityCore* or *CMaNGOS*.

The static graph of Figure 6.5, along with the tabular representation of involved data reported in Table 6.3, shows that there are interaction between the two repositories, and that there are contributors which are more involved in one of the two repositories.

The dynamic graph, however, is much more interesting in this case since it is able to represent the moving of the authors between the two repositories during the years, and the different number of contributors attracted by each other. The settings for the dynamic graphs are: `max_depth`, `keep_only_one_contributors`=True, `exclude_forks`=True, `min_commits`=1. Users with contributions to only one repos-

[4]The Octoverse in 2012: `https://github.com/blog/1359-the-octoverse-in-2012`.

User	**Total commits on MaNGOS**	**Total commits on TrinityCore**
VladimirMangos	2163	6
Shauren	1	1206
DDuarte	1	1060
Machiavell1	3	423
kaelima	2	293
insider42	253	1
click	1	195
Zakamurite	147	1
tomrus88	130	1
Lynx3d	74	1
SilverIce	71	1
stfx	55	6
LordJZ	24	11
tibbi	1	26
vermie	18	8
derex	20	4
L1ghtGu4rd	17	1
krofna	2	14
Fredi	1	13
Christyan	1	1

Table 6.3: *MaNGOS* and *TrinityCore* contributors, and their total contributions on either projects.

itory were not filtered, in this scenarios, as it is of interest to capture popularity as well as common users. The common developers between the two repositories are orange-colored, while the users with contributions to only one of the two are colored in light blue. Since it is impossible to represent the animation in this document for obvious reasons, four time snapshots were captured and are shown in Figure 6.6. The full animated graph can be viewed on the online plaftform YouTube[5].

Figure 6.6a shows the situation in the beginning of the year 2009. *TrinityCore* did not have any active users yet, and *MaNGOS* had some contributors working on it. Figure 6.6b, which depicts the end of the year 2009, shows some activities on *TrinityCore*, albeit much less than *MaNGOS*. In Figure 6.6c, taken in the second half of the year 2011, the activities on *TrinityCore* are higher than *MaNGOS*, and this trend continues in Figure 6.6d, where only a few contributions are made on *MaNGOS*.

This particular graph is interesting since it shows how the network graph approach can be used to detect the particular situation of a project being overtaken by another one.

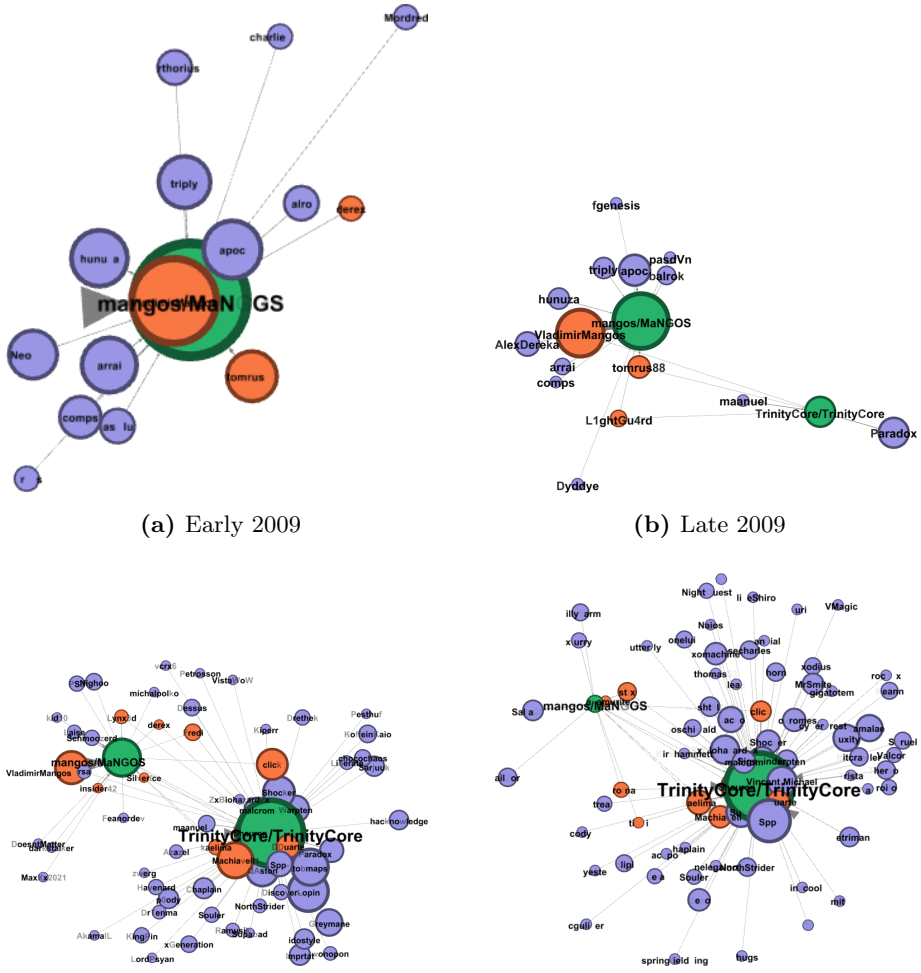
6.2.3 MSR14: mxcl/homebrew

Homebrew is a free and open source packet manager for Apple *macOS* operating system. It is one of the most popular repositories on GitHub: in the year 2012, it was the project with the highest number of unique contributors hosted on the platform.

The static graph of *Homebrew* is shown in Figure 6.7. For this graph, the following settings were used: `exclude_forks`=True, `keep_only_one_contributors`=False, `min_commits`=1. This graph is barely unreadable, as there are too many nodes and almost no label can be read. Even ignoring the unreadable users nodes, the large number of repository nodes makes it difficult to obtain a readable positioning of the nodes in the graph. In fact, this graph includes almost the whole *MSR14* dataset: 73 repositories out of 89 have at least a common contributor with *Homebrew*.

A depth look at the static graph reveals that *ruby/ruby* has the highest number of common contributors with *Homebrew*. Common activities between the two repositories are expected, since *Homebrew* is using *Ruby* internally, and the packages instructions are written as *Ruby* scripts (the so called *formula*s). Most of the projects,

[5] *MaNGOS* vs *TrinityCore* animated graph, last visited on January, 12th 2018: `https://youtu.be/q_ZBtxrWzME`.



(a) Early 2009 **(b)** Late 2009

(c) 2011 **(d)** 2013

Figure 6.6: Four moments in the history of *MaNGOS* and *TrinityCore*.

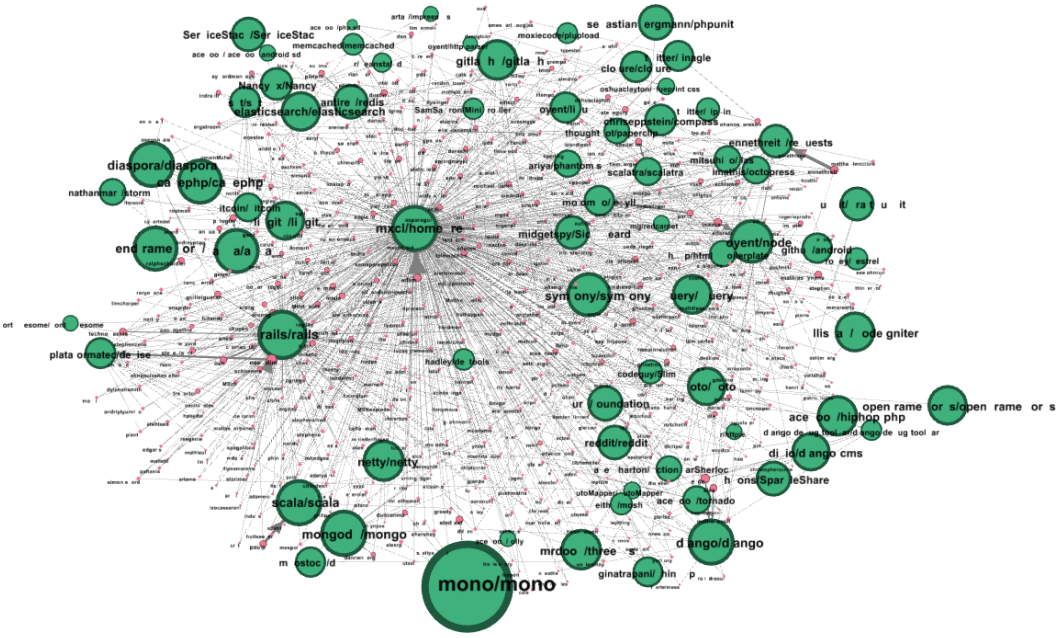


Figure 6.7: Homebrew static graph.

however, have only a low number of contributors in common with *Homebrew*. A realistic explanation for these contributors is that they are contributing with formulas to automate the install of their projects, since *Homebrew* hosts a list of default formulas on its GitHub page. Additionally, since *Homebrew* is widely used by *macOS* programmers, it is likely that developers that are Mac owners were interested in contributing to the project, even if it is hosted in a repository that is different from the repositories where they are usually contributing to (e.g., different language, different technologies involved etc.).

Table 6.4 shows the top 10 contributors on the *Homebrew* project, while Table 6.5 shows the top 25 contributions for the rest of the projects in the resulting graph. As can be seen from these two tables, only 2 users created more than 100 commits on GitHub, while 25 users have more than 100 commits on other repositories. In general, 480 users contributed to *Homebrew*, with an average commit count of 13.56 commits each. Excluding the users *adamv* and *asparagui*, which are the most active on *Homebrew* projects, the average commit count is 4.4.

User	Commits
adamv	3157
asparagui	1247
trevor	125
MindTooth	69
kashif	65
sjonnet19	63
fsouza	44
TylerBrock	32
sferik	32
chrmoritz	31

Table 6.4: Top 10 contributors on *Homebrew*, and the corresponding number of commits on *Homebrew*.

User	Commits	Repository
josevalim	2094	rails/rails
kennethreitz	2075	kennethreitz/requests
paulp	1834	scala/scala
josevalim	1255	plataformatec/devise
isaacs	1253	joyent/node
josh	1032	rails/rails
parkr	991	mojombo/jekyll
jezdez	881	django/django
casualjim	677	scalatra/scalatra
technoweenie	456	rails/rails
pvlugter	390	akka/akka
alex	370	django/django
pietern	343	antirez/redis
ralphschindler	312	zendramework/zf2
guilleiguaran	284	rails/rails
kriswallsmith	260	symfony/symfony
arunagw	231	rails/rails
dustin	214	memcached/memcached
jstedfast	206	mono/mono
parkr	174	imathis/octopress
steveklabnik	169	rails/rails
dannenberg	158	mongodb/mongo
fhemberger	129	imathis/octopress
stevegury	127	twitter/finagle
johanoskarsson	101	twitter/zipkin

Table 6.5: Top 25 contributors to projects other than *Homebrew*, the number of contributed commits and the corresponding repository.

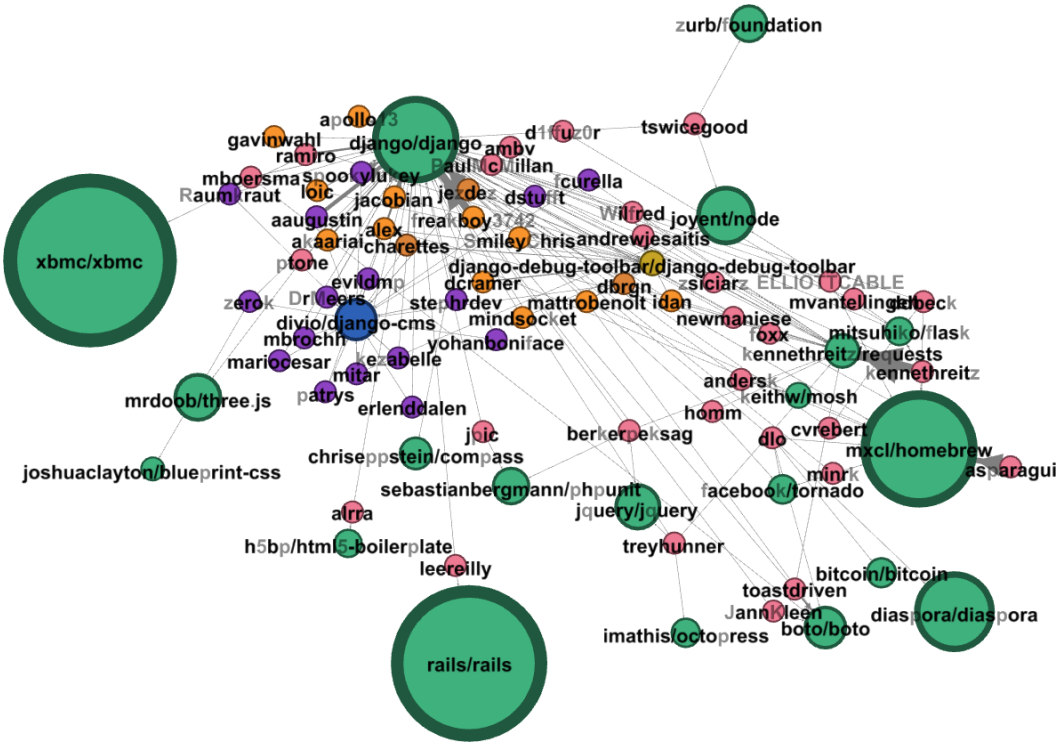


Figure 6.8: *Django* static graph, with *django-cms* and *django-debug-toolbar* highlighted in blue and yellow color.

6.2.4 MSR14: django/django

Django is a *Python* framework for developing web applications using the Model View Controller (MVC) pattern. Many popular websites are built on top of *Django*[6], such as *Disqus*, *Mozilla*, *Instagram*, *Pinterest* and *The Washington Post*[7].

Figure 6.8 shows the static graph of *django/django* (settings: `exclude_forks`=True, `keep_only_one_contributors`=False, `min_commits`=5). Users with few contributions (less then 5) were filtered to achieve a more readable result. In this figure, the two closest repositories to *Django*, which are *divio/django-cms* and *django-debug-toolbar*, are colored in blue and in yellow respectively. The user nodes in common between the three repositories are also colored with a mix of their repositories colors (purple for *django-cms* and orange for *django-debug-toolbar*). As can be seen, most of the contributors in the graph are shared between these three projects. There is also

[6] *Django* official page, including *Django* based projects list, last visited on January, 12th 2018: `https://www.djangoproject.com/start/overview/`.

[7] News on *Django* blog about its adoption for *The Washington Post*, last visited on January, 12th 2018:`https://www.djangoproject.com/weblog/2005/dec/08/congvotes/`.

User	Total commits
freakboy3742	1703
jacobian	890
jezdez	881
aaugustin	814
spookylukey	470
ramiro	410
alex	370
akaariai	215
apollo13	127
loic	87

Table 6.6: Top 10 committers on *Django*.

a high number of repositories with fewer common contributors to *django/django*, which are mainly web-related repositories.

In summary, the static graph shows strong connections, created by a large number of user, between *Django* and the two colored projects *django-cms* and *django-debug-toolbar*, and weaker connections, created by fewer users, between *Django* and a number of web development repositories. The latter suggests that there are some activities in the web development area that are shared even among projects written in different languages and with different purposes.

The existence of a stronger connection between *Django*, *django-cms* and *django-debug-toolbar* is interesting, since the two latter projects are based on *Django* itself: *django-cms* is a Content Management System (CMS), and *django-debug-toolbar* is a set of graphical component to help users who are using *Django* to debug their application.

The most active committers for these repositories are reported in Table 6.6 (*Django* project), Table 6.7 (*django-debug-toolbar* project) and Table 6.8 (*django-cms* project). In Table 6.7 and Table 6.8, the users in common with Table 6.6 are highlighted in *italic font*.

The relationship between these repositories can be further investigated by looking at the animated graph, which can be seen on Youtube[8]. The animated graph shows that *Django* project was developed well before the other two projects, and that *django-debug-toolbar* was created from the most active users on the *Django* project. Once its development started, *django-debug-toolbar* appears with constant activities

[8] *Django* animated graph, last visited on January, 12th 2018: `https://youtu.be/P6f-MG6Chls`.

User	Total commits
dcramer	156
jezdez	88
idan	31
alex	28
mindsocket	7
akaariai	4
jacobian	3
SmileyChris	3
charettes	2
dbrgn	2
mattrobenolt	2
apollo13	1
freakboy3742	1
gavinwahl	1
loic	1

Table 6.7: Committers for the project *django-debug-toolbar*. In *italic* the ones that also appears in Table 6.6.

User	Total commits
evildmp	72
DrMeers	47
jezdez	43
kezabelle	35
stephrdev	20
mitar	14
spookylukey	13
yohanboniface	8
erlenddalen	5
dstufft	4

Table 6.8: Committers for the project *django-cms*.

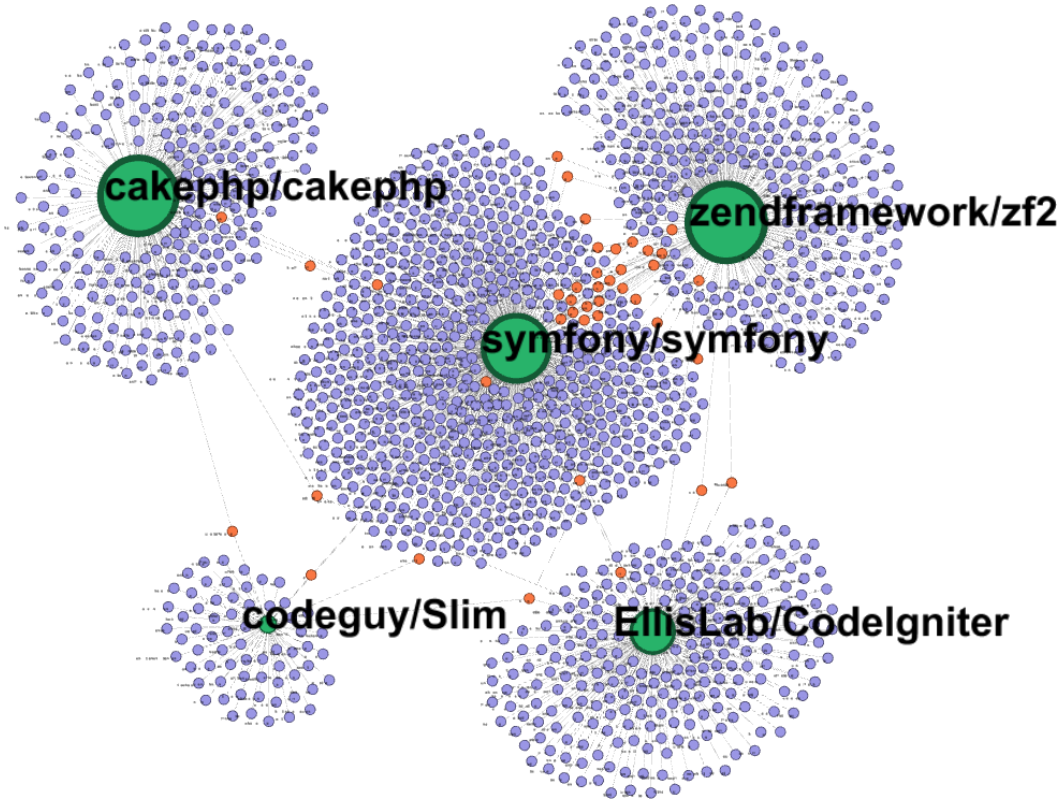


Figure 6.9: Static graph for *PHP* frameworks in MSR14.

by mostly the same user in the animated graph; a reasonable explanation for this is that it was created to debug *Django* itself, and was moved to an ad-hoc repository as it could be used by other users to debug their projects as well. The project *django-cms*, instead, has a different history: it received contributions from *Django* contributors, but for most of its early stages the contributions came from users that only later on contributed to *Django*. This indicates that *django-cms* is more of a standalone project built on *Django*, which only later on resulted in its users having enough skills, knowledge, or interest to contribute back to the original *Django* project.

6.2.5 MSR14: PHP frameworks

The MSR14 includes 5 repositories of different *PHP* frameworks: *symfony/symfony*, *zendframework/zf2*, *cakephp/cakephp*, *EllisLab/CodeIgniter* and *codeguy/Slim*. Out of these 5, only the *Slim* project is nowadays deprecated, while the others are still actively maintained and popular.

The static graph of these five repositories is shown in Figure 6.9. To construct this graph, the following settings were used: `exclude_forks`=True, `min_commits`=1, `keep_only_one_contributors`=True. Moreover, the network graph was forced to ignore any repository outside of the aforementioned ones. Again, the contributions to a single repository have been included to have a graphical representation of the projects popularity, as well as to understand how many contributors contribute to more than one repository.

From the graph, it can be seen that there are are a low amount of users who contributed to more than one repository (41 out of 1,508 total users, less than 3%). Nevertheless, there are some users in common among all of the repositories. In particular, most of the common users can be seen between *symfony/symfony* and *zendramework/zf2*. The author is not currently aware of any event, in the history of these two repositories, that could have caused a high number of contributors to shift from one repository to the other.

The animated graph, which can be found on Youtube[9], can be used to investigate the common users between *Symfony* and *Zend Framework (v2)*. The results show that the activities for the two repositories started at about the same time, in the late 2011-beginning of 2012, and that most common contributors were contributing to both projects at the same time. Moreover, the common contributors are shown mid way from both repositories, meaning that their contribution is mostly equally split to both repositories. This confirms that the situation is not the one seen in Section 6.2.2, where authors move from a repository to another, but it is more like the two repositories had "something" in common.

Both *zendramework/zf2* and *symfony/symfony* are actually the second version of previously existing frameworks, which can now be found on GitHub under the names of *zendframework/zendframework* and *symfony/symfony1*. The projects are based on an updated version of *PHP* (5.3, vs the 5.2 used in their previous version). *PHP 5.3* is not a major update over its previous version, but it is possible that a part of the common users were fixing problems due to *PHP* version change. Another reasonable hypothesis for the common developers is that, since the two projects were developed in the same time frame, the design choices of the two projects were similar. Many *PHP* programmers may have been familiar with the previous versions of both

[9]Animated network graph for *PHP* framework repositories on MSR14 on YouTube, last visited on January, 12th 2018: `https://youtu.be/2wDu3MwPQrA`.

User	Total commits
Maks3w	284
hhamon	71
Ocramius	66
pborreli	59
beberlei	55
samsonasik	55
jmikola	50
Danez	37
tPl0ch	35
stealth35	33

Table 6.9: Top 10 contributors who contributed to more than 1 projects in Section 6.2.5.

framework, and may have influenced the design process of both newer versions. For example, both frameworks offered a renewed version on how dependency injection works, with *Symfony 2* using *the Dependency Injection pattern extensively* and coming with *a built-in Dependency Injection Container*[10] and *Zend Framework 2* providing the Dependency Injection Container[11], a new object to handle dependency injection.

To contextualize the differences between common contributors and contributors who contributed to only one project, Table 6.9 shows the top 10 contributors who contributed to more than 1 project, while Table 6.10 reports the top 10 contributors who contributed to only 1 project. Finally, Table 6.11 shows the total commits for the *PHP* web framework repositories.

6.2.6 Android REST API client libraries

The static graph for the REST API libraries in the ANDROID dataset is shown in Figure 6.10. To construct this graph, `force_include_repositories` were set with the whole list of the repositories in the ANDROID dataset. The graph shows that most of the previously classified as "small" (see Section 5.2.1) and "less popular" (see Section 5.2.2) have few contributors, and no contributions come from authors who are also contributing to other projects.

[10] Symfony official changelog page, last visited on January, 12th 2018: `http://symfony.com/blog/symfony-2-0`.

[11] Zend introduction to DI, last visited on January, 12th 2018: `https://framework.zend.com/manual/2.0/en/modules/zend.di.introduction.html`.

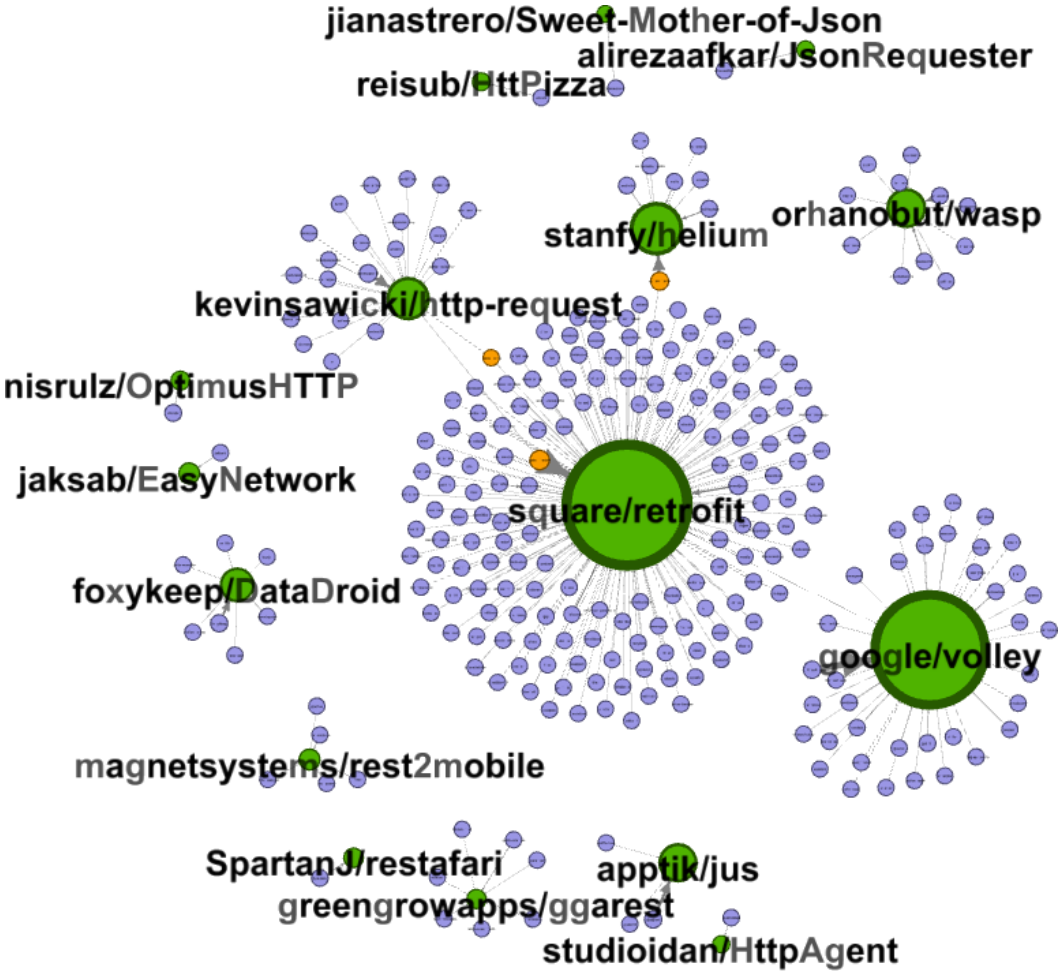


Figure 6.10: Android static graph.

User	Commits
fabpot	5156
weierophinney	3786
markstory	3515
phpnut	1564
narfbg	1492
nateabele	1042
lorenzo	778
derekjones	567
philsturgeon	382
ADmad	358

Table 6.10: Top 10 contributors who contributed to only 1 project in Section 6.2.5.

Repository	Commits
zendframework/zf2	12030
cakephp/cakephp	11844
symfony/symfony	9936
EllisLab/CodeIgniter	5941
codeguy/Slim	1250

Table 6.11: Repositories of Section 6.2.5, and their commits count.

In general, there are only three authors that contributed to more than one repository. Given the limited amount of common authors and their contributions, an in-depth analysis can be made.

Of the three common authors, *JakeWharton*, the orange node close to *RetroFit*, is a famous *Android* developer[12], who created many popular libraries, such as *butterknife*[13] and *ActionBarSherlock*[14]. He is the one who wrote the highest number of commits on *RetroFit* (1001 out of 2062 commits on the whole project), which is not surprising since the *RetroFit* project is developed by Square Open Source[15] company and he has been working there as an *Android* Engineer since 2012[16]. It is surprising, however, that *JakeWharton* contributed to *Volley*, albeit only with a

[12] *JakeWharton* biography on Google developers, last visited on January, 12th 2018: `https://developers.google.com/experts/people/jake-wharton`.

[13] *butterknife* repository on GitHub, last visited on January, 12th 2018: `https://github.com/JakeWharton/butterknife`.

[14] *ActionBarSherlock* on GitHub, last visited on January, 12th 2018: `https://github.com/JakeWharton/ActionBarSherlock`.

[15] Square Open Source website, last visited on January, 12th 2018: `http://square.github.io/`.

[16] *JakeWharton* Linkedin profile with career history, last visited on January, 12th 2018: `https://www.linkedin.com/in/jakewharton/`.

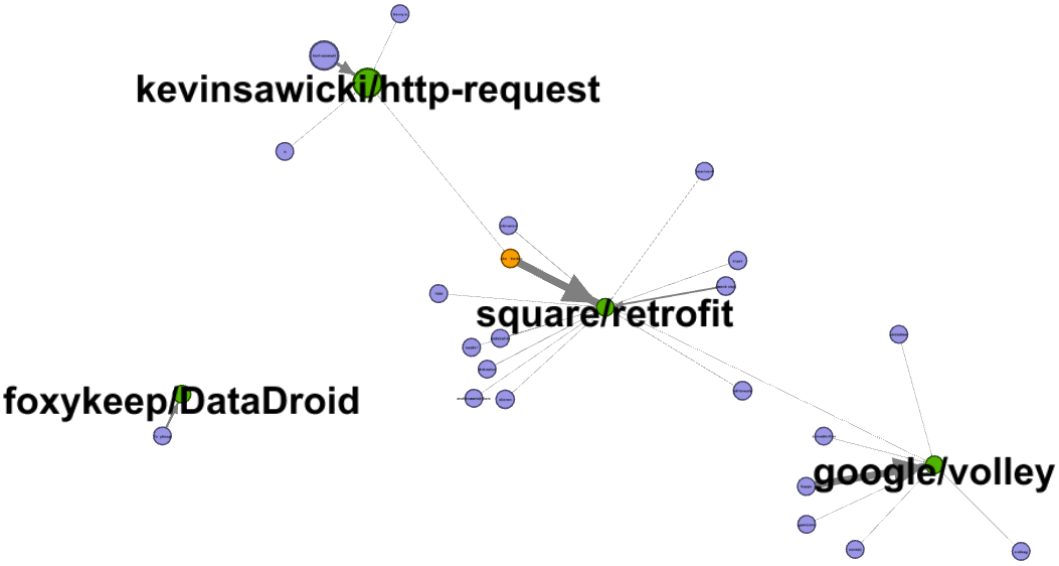


Figure 6.11: Animated graph at the month in which *JakeWharton* contributed to *Volley*.

single commit. Figure 6.11 shows the month in which the contribution was written, and shows that the user was also contributing to *RetroFit* and *http-request* at the same time. The commit that was made on *Volley*[17] has the following commit message: "Allow override point for connection creation. This provides an easy insertion point for alternate implementations of the HttpUrlConnection API (e.g., OkHttp)". Since *OkHTTP* is another library developed by Square Open Source, it makes sense for *JakeWharton* to make that contribution to a competitor of *RetroFit*. This, however, represents a new and unforeseen case where a common contributor between two repositories does not translate in a contribution of "knowledge" between the two repositories. In fact, the knowledge transmitted through the contribution is actually from a repository, *square/okhttp*, which is not included in the dataset. Similarly, *JakeWharton*'s contributions to *http-request* have similar commit messages, delineating a similar situation.

The second user, *EddieRingle*, which can be found in the top left between *RetroFit* and *http-request*, made only a minor contribution (typo-fix) on *http-request*, and his presence in the graph is not particularly impactful. The third user, called *roman-mazur*, is more interesting, since he made many contributions (445 out of 663 of the

[17] *JakeWharton* only commit on *Volley*, last visited on January, 12th 2018: `https://api.github.com/repos/google/volley/git/commits/05a1b0edb25ed84c95523df7a81bd87a44b697d7`.

whole repository) to *Helium*, and three commits on *RetroFit* with technical content.

The animated graph, which can be seen on YouTube[18], is coherent with what can be seen on the static graph: *Volley* and *RetroFit* have a much higher number of users contributing than the other libraries. Moreover, smaller libraries show up from time to time in the graph, while bigger projects are constantly appearing in the representation. This means that the smaller libraries are also not continuously developed.

Since the original goal to collect the ANDROID dataset was to evaluate the usage of the Network Graph tool to choose the best library for the goal, the latter is particularly important: not only by choosing a smaller library the developer would use a project with less commits and less contributors, he would also risk having to wait more time for bugfixes.

6.2.7 ELECTRON: electron/electron

Electron is an open source framework developed by the GitHub company. Its purpose is to build desktop applications using web components (*Chromium* for the front-end and *Node.js* for the backend). The *Electron* project started in 2013. Back than, the project was named *Atom Shell*, and it was part of the *Atom* project. In 2015, the project name changed to *Electron*, and in 2016 the version 1.0 was released.

The ELECTRON dataset was created with the goal of understanding the interactions between the framework and the projects that are based on it, and in particular to investigate on whether the users who created an application using *Electron* also used their acquired knowledge on the project to contribute back to the framework, for example in fixing a bug, or by proposing their own improvements/new features.

Figure 6.12 shows the static graph, with the three bigger green nodes in the center being *Electron*, *Atom* and *browser-laptop*. Due to the high number of nodes, a zoomed in version of the graph that shows the closer projects to *Electron* can be seen in Figure 6.13.

The graphs include 50 repositories. Every repository has at least one contributor in common with *Electron*, and in total the common contributors created 12,745 commits (out of 17,441 total commits of the repository) on *Electron*. This count, however, includes commits made by authors who mainly developed on *Electron*, such

[18] Animated graph for *Android* REST API libraries:`https://youtu.be/QZzicsDft_Y`.

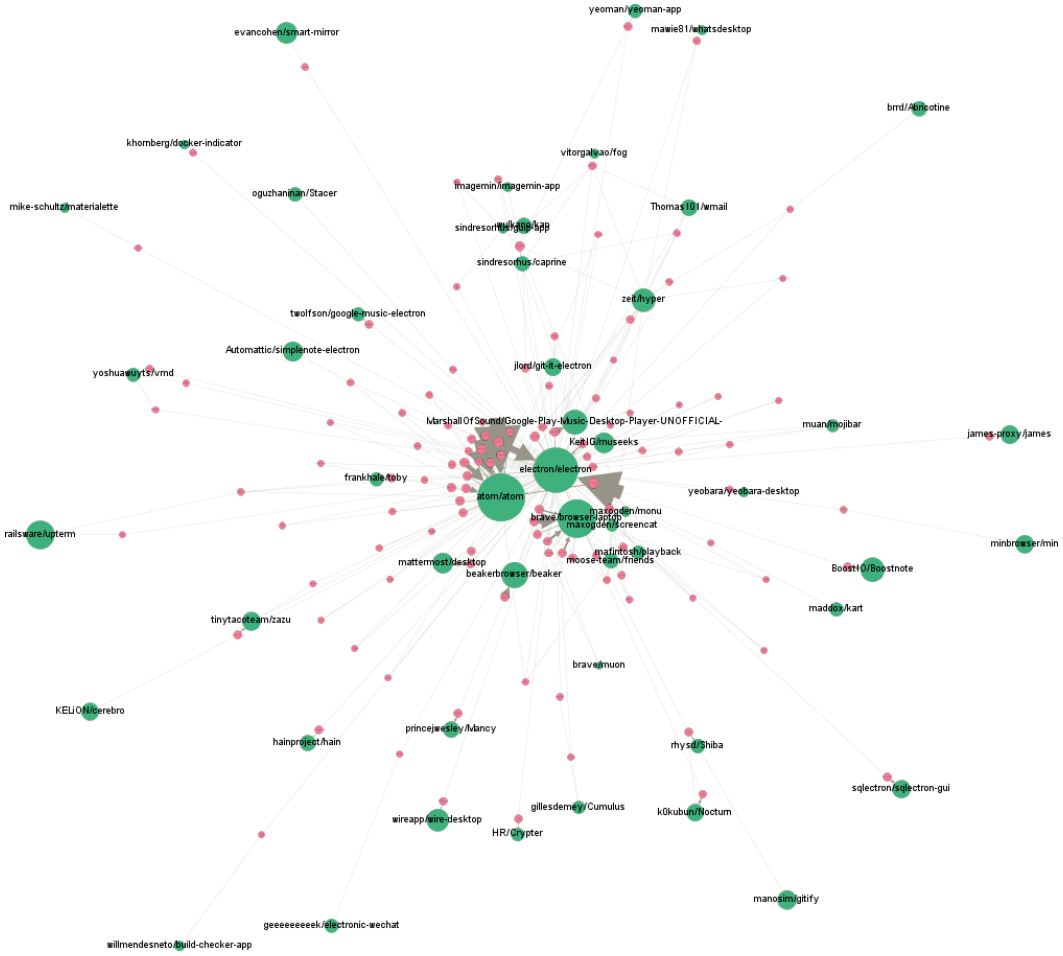


Figure 6.12: Static graph for the *Electron* project (whole view).

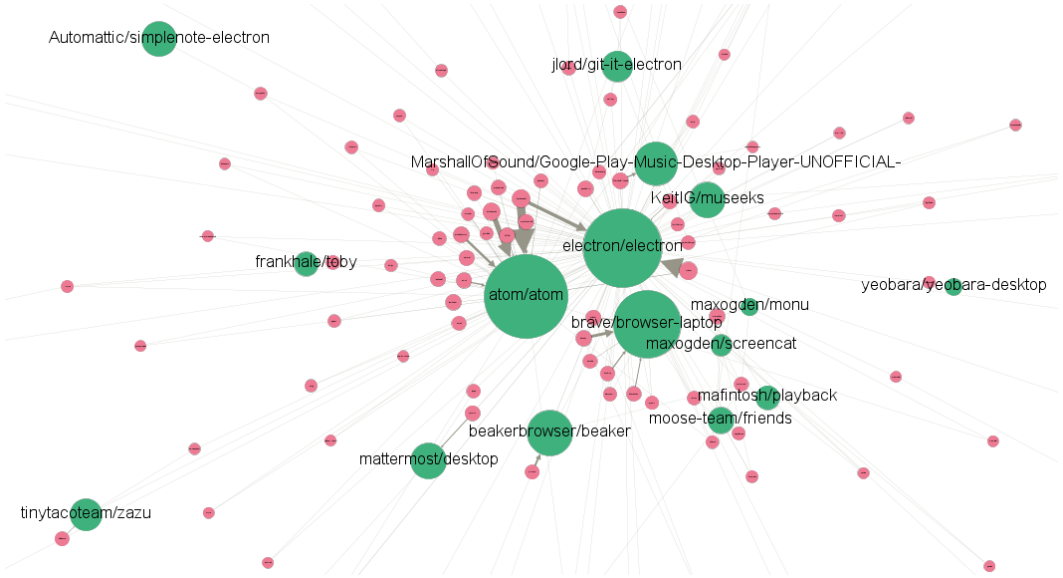


Figure 6.13: Static graph for the *Electron* project (zoomed on center).

user	commit message
mawie81	[ci skip] fix link in screen docs
Rokt33r	fix typoCode block should be ended “
maxcnunes	Improve error handling from remote. This way copy [...]
khornberg	Correct link of debugging UI.Using the link provided [...]
stefanbuck	fix broken links
orderedlist	Get radius working with frameless window.Right now it’s [...]
rhysd	fix TouchBarSpacer class name in doc
Hum4n01d	Fix order of OSs.It was previously inconsistent
appetizermonster	Fix small mistakes in CONTRIBUTING-ko.md
ellerbrock	fixed broken link "how to share data between web pages" [...]

Table 6.12: Commit message for ten randomly selected users who only made few contributions on *Electron*.

as *zcbenz*, who made 7,217 commits on *Electron*. Without counting the commits of the nodes closest to *Electron*, and closer to *Electron* than to other projects, the commits made by common contributors are still about 5,000 (39% of the total commits). Most of these commits are coming from *Atom* developers. It is thus debatable whether commits from the *Atom* project should be considered as from other projects. Discarding the commits from *Atom* developers still leaves about 1,700 commits coming from contributors who mainly developed on other projects, which is about 10% of the total commits of *Electron*.

With either 10% or 39% commits coming from external contributors, *Electron* based projects influenced the *Electron* repository considerably. To further investigate these commits, 10 random commits were chosen from users with only few contributions on *Electron*. These commits, shown in Table 6.12, mostly seem to be related to bug-fixes.

The animated graph, which can be seen on Youtube[19], does not contribute much in understanding the interactions between *Electron* and *Electron*-based repositories. However, it is interesting since it clearly shows the initial developments on *Electron*, and how it was born from *Atom*. Another aspect that can be derived from the

[19] *Electron* animated graph on YouTube, last visited on January, 12th 2018: `https://youtu.be/Xs4VtOJHslQ`.

User	Commits
zcbenz	7217
kevinsawicki	3365
zeke	565
paulcbetts	264
MarshallOfSound	261
bridiver	141
bbondy	125
gerhardberger	105
maxogden	93
joshaber	53

Table 6.13: Top 10 contributors on *Electron* project

User	Commits
kevinsawicki	8329
nathansobo	5013
probablycorey	2379
maxbrunsfeld	1285
as-cii	1261
50Wliu	604
zcbenz	452
joshaber	372
mcolyer	366
simurai	245

Table 6.14: Top 10 contributors on *Atom* project.

animated graph is that the projects based on *Electron* do not start from *Electron* contributors, but rather their authors contribute back to *Electron* after a certain amount of time. This is likely to happen since such developers have possibly discovered bugs in *Electron* and implemented the proposed fixes.

Tables 6.13 to 6.15 provide the number of commits for the top 10 contributors of *Electron*, *Atom* and the other projects in this dataset.

6.2.8 LANGUAGES: collaboration between programming languages

As described in Section 5.1, LANGUAGES includes the repositories for several programming languages. The static graph constructed from this dataset is shown in Figure 6.14. The graph shows a situation where some degree of collaboration is noticeable within all of the repositories.

Rust, *Ruby* and *Elixir*, in particular, are the 3 repositories with most contributors

User	Repository	Commits
bbondy	brave/browser-laptop	2876
pfrazee	beakerbrowser/beaker	1847
MarshallOfSound	MarshallOfSound/Google-Play-Music-Desktop-Player-UNOFFICIAL-	1374
bsclifton	brave/browser-laptop	1198
diracdeltas	brave/browser-laptop	956
yuya-oc	mattermost/desktop	904
Rokt33r	BoostIO/Boostnote	823
bridiver	brave/browser-laptop	727
k0kubun	k0kubun/Nocturn	623
maxcnunes	sqlectron/sqlectron-gui	618

Table 6.15: Top 10 contributions on projects different than both *Electron* and *Atom* in Section 6.2.7.

repository	common contrib.	total contrib.	commits by common contrib.	commits total
rust-lang/rust	80	2070	2456	58715
ruby/ruby	28	401	270	51405
elixir-lang/elixir	35	744	813	15989
scala/scala	16	485	280	30457
jashkenas/coffeescript	19	343	81	6070

Table 6.16: Common contributors and their commits for top 5 projects in LANGUAGES.

who committed on more than one repository. In Table 6.16 the number of common contributors with respect to total contributors are shown, and also the number of commits made by common contributors for each of these 5 projects is shown. When compared to the situation of *Electron* (see Section 6.2.7), this dataset shows less common activities between repositories: the absolute values of both contributors and commits are higher, but in percentage the numbers are lower. This situation is expectable, since different programming languages have much more differences among themselves then framework based applications with the framework itself. Moreover, in *Electron* the common activities and contributions were considered only on the framework itself, while here the contributions are randomly sparse over the programming languages.

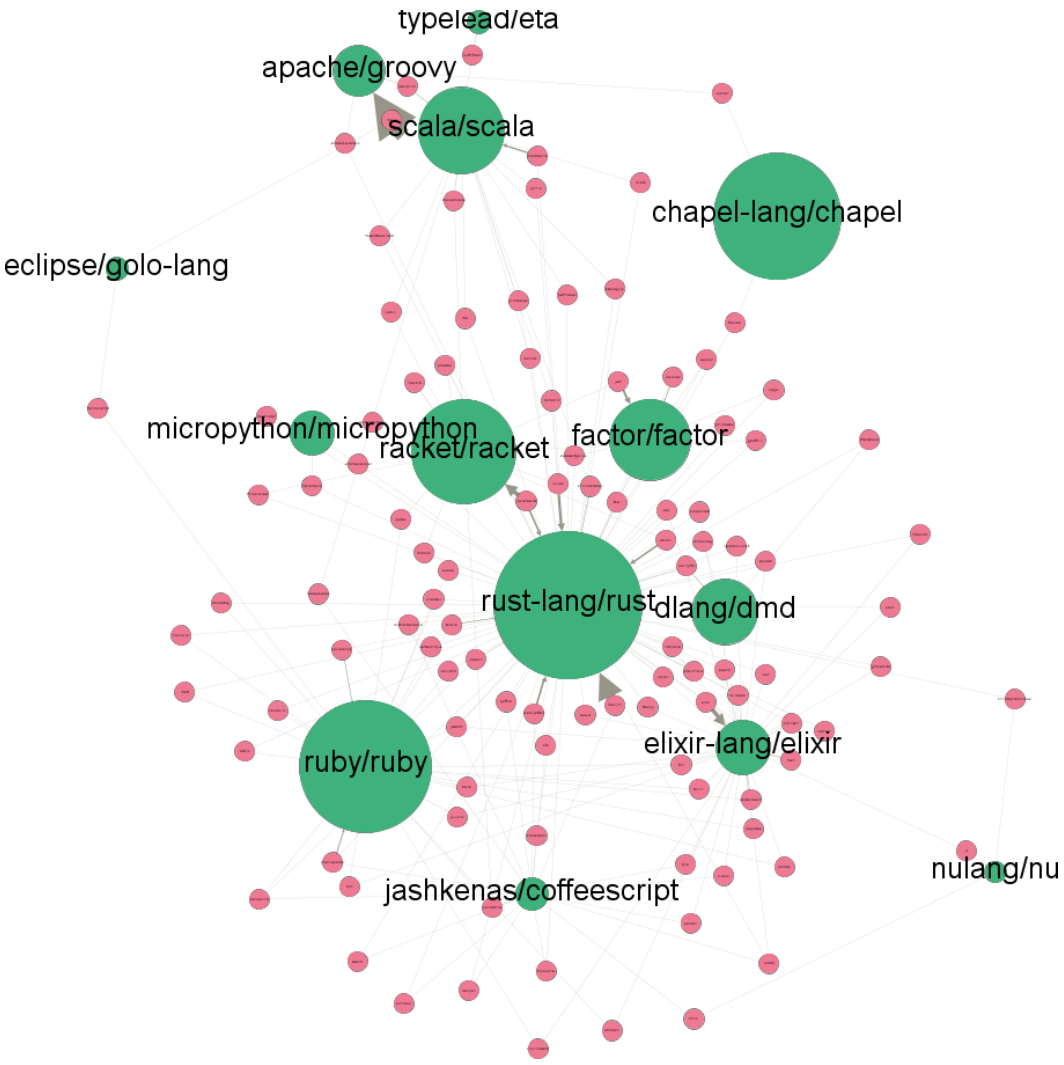


Figure 6.14: Static graph for LANGUAGES.

The animated graph, shown on YouTube[20], shows that there are no contributions between the repositories of the programming languages until *Rust* appearance in the second half of 2010. Starting from that time, most of the common contributors are between *Rust* and another programming language. *Rust* is known for being a very popular programming language.

In terms of "sentiment evaluation", recently *Rust* was voted by developers as the most loved programming language on Stack Overflow[21] (a programmers community) for the second year consecutively.

Rust is popular and loved, but it does not appear in any chart of the most used programming languages. In general, very few projects are actually based on *Rust*. This may be due to the project being still not mature: the version 1.0 was only released in the year 2015[22], and up until that version, code written for previous releases were difficult to adapt and maintain to work with newer versions.

In summary, developers who were interested in *Rust* in the past would rather contribute to the project itself then use the language, and due to the language popularity, many developers from other repositories have decided to offer their contributions.

[20] LANGUAGES animated network graph on YouTube: `https://youtu.be/FPUZLwwlU3s`.

[21] Stack Overflow insights for 2017, last visited on January, 12th 2018: `https://insights.stackoverflow.com/survey/2017`.

[22] *Rust* 1.0 release, last visited on January, 12th 2018:`https://blog.rust-lang.org/2015/05/15/Rust-1.0.html`.

Chapter 7

Conclusion and future works

Version Control Systems (VCS) are widely used both in closed and in open source projects, as they have many features that help working in teams, such as code conflicts management and code history, version control and parallel development of features and bugfixes. The usage of VCS produces huge amount of data, which contains valuable information to characterize a software project and its development. Project managers, for example, may use this information to pro-actively discover conflicts arising in their teams, or to shed the light on patterns that lower productivity in their team's activities. Researchers are interested in data from repositories as well, as it provides insights and historical evolution on the life-cycle of projects, on the adoption of software engineering principles, and on developers activities, locations and discussions. VCS mining involves methods developed in different research fields, such as software engineering, social science, Natural Language Processing (NLP), big data, project management and statistics.

Since it is one of the mostly used VCS, and is widely used by many open source projects, this thesis focused on mining repositories managed using the Git VCS. More specifically, the datasets for this thesis were collected from GitHub, a web platform which integrates the Git repositories with issue trackers and code reviewing tools (see Chapter 2 for more details). The most important works in scientific literature, focused on mining data from Git and Github, are presented in Chapter 3.

This work presented different visualization approaches that provides a graphical overview of different aspects of repositories. Four different datasets were presented in Chapter 5, one of which (the MSR14) is already widely studied by researchers. The other 3 were built on purpose for the use in this thesis, each one referring to a

a specific topic. Chapter 5 also considers the use of Principal Component Analysis (PCA) to provide a compact, bi-dimensional view where projects with similar features appear as close. This technique was used to provide a graphical representation of two concepts, size and popularity, which can not be measured by simple metrics such as threshold based metrics.

Chapter 6 proposed the analysis of common contributors between different repositories as a way to obtain information. The evaluation of common contributors between projects is a novel approach in scientific study of repositories characteristics. The idea behind this study is that a user contributing to a project is also acquiring knowledge on the topic, tools and design patterns used in that project. Therefore, when the same user contributes on other projects, his contributions are influenced by his acquired knowledge. From the standpoint of this analysis, the contributors are hubs through which knowledge moves between repositories. Repositories and users can be represented as nodes in a directed graph, with contributions (commits) being the edges connecting them.

Gephi, a tool to visualize graphs, was chosen to create a visual representation of this graph, for its ability to display dynamic graphs, i.e., graphs with nodes and edge changing over time, with time being the projects life in this work. The resulting graphs were interpreted with expert knowledge on the repositories and their history. When interpreted by a human observer who is expert on the topic, the resulting graphs show different patterns of interactions between users and repositories. Some example cases that may translate into different patterns are presented in this section. A project to make HTTP calls was created and maintained by web projects developers as a tool for their main project (see Section 6.2.1). Albeit this tool is a small project, the main contributors were already major contributors of larger projects with a strong similarity in language and topic (web development). Another interesting case referred to a repository that stopped being contributed to from the community, which moved to a different version of the same project with more open contributions rules (Section 6.2.2). The animated graph has a unique pattern of interactions, as common users clearly stop interacting with one of the two repositories, and focus their efforts on the other. In analyzing the Django web framework, differences were found between a project built as a submodule of Django, and a Django-based project (Section 6.2.4). The amount of initial contributors and the timeline of the contributors actions (both committing and being attracted to the repository) of the two projects

exhibited visual differences. Moreover, while the Django submodule contributors were contributing to the Django repository as well during the project first month of life, the authors of Django-based projects contributed to Django only at later stages of their project life. Finally, it was found that many developers who chose to build projects on top of Electron, a framework for cross platform HTML/Javascript desktop apps, contributed back to the Electron project with bug fixes (Section 6.2.7).

During the elaboration of this work, several critical aspects have emerged that impact on the quality of the analysis and may be addressed in future works.

The first issue is that the study was carried out on the four datasets collected and described in Chapter 5. However, the projects in these datasets represent a small subset of the set of projects that can be found on GitHub. Results may vary considerably with different datasets. This limitation is due to the practical impossibility to operate on much larger sets of repositories, which would require to access online GitHub repositories by an extended and intense use of GitHub API calls. These latter, however, are throttled by GitHub to guarantee the smooth functioning of the platform.

A key assumption to apply the proposed analysis is that repositories should share some common contributors. The finding of a lack of common contributors is by itself a result. However, no useful considerations can be elaborated with the proposed approaches for those sets of projects that are developed in isolated environments. This means that the proposed method may not be suitable for closed source projects.

This work only considered a form of user interaction where users were creating one or more commits. GitHub features different ways for user to contribute to a project, and the usage of the issue tracker to file a bug or to discuss a new feature, or reviewing a submitted code and giving constructive feedback, are forms of contribution that are currently not addressed in the present work. Their inclusion, with the goal to obtain an “holistic” assessment of the interaction of users with software projects, may be evaluated in future works. However, this assessment would require to differentiate between useful and unuseful contributions, which is something that is not necessary for commits: a useless or bad code contribution would not be accepted in the project history by the authors in the first place. Therefore, additional evaluation would be required to integrate these aspects in the evaluation.

Additionally, in the proposed approach all of the commits are considered as having the same important for the development of the corresponding software project.

It is debatable whether a one line contribution to update the project documentation should be considered more or less important than a major commit refactoring many software components of the project. This may be further explored in the future, using software analysis metrics. However, these considerations are highly language dependent, making the evaluation of the quality of the contribution very hard to assess objectively.

List of publications

- Tullio Facchinetti, Guido Benetti, Moses A. Koledoye, Gianluca Roveda, *"Design and implementation of a web-centric remote data acquisition system"*, in Proceedings of the 10th IEEE International Workshop on Service-Oriented Cyber-Physical Systems in Converging Networked Environments (SOCNE)", Berlin, September, 2016.

- Gianluca Roveda, Moses A. Koledoye, Enea Parimbelli, John H. Holmes, *"Predicting Clinical Outcomes in Patients With Traumatic Bleeding: A Secondary Analysis of the CRASH-2 Dataset"*, in 3° International Forum on Research and Technologies for Society and Industry, Modena, Italy, September, 2017

- Daniele De Martini, Gianluca Roveda, Alessandro Bertini, Agnese Marchini and Tullio Facchinetti, *"A Framework for Automatic Generation of Fuzzy Evaluation Systems for Embedded Applications"*, in 9th International Joint Conference on Computational Intelligence, Madeira, Portugal, November 2017

- Inventor of a patent whose deposit will be requested shortly (the title cannot be disclosed due to agreements with the owner company)

Bibliography

[1] Business Software Alliance. Software industry facts and figures, US, 2007.

[2] Business Software Alliance. Economic impact of software, EU, 2014.

[3] Business Software Alliance. Economic impact of software, US, 2016.

[4] O. Arafat and D. Riehle. The commit size distribution of open source software. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–8, Jan 2009.

[5] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *2009 IEEE 31st International Conference on Software Engineering*, pages 298–308, May 2009.

[6] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. How Developers Use Exception Handling in Java? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 516–519, New York, NY, USA, 2016. ACM.

[7] G. Avelino, L. Passos, A. Hora, and M. T. Valente. A novel approach for estimating Truck Factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May 2016.

[8] Ali Sajedi Badashian and Eleni Stroulia. Measuring User Influence in GitHub: The Million Follower Fallacy. In *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering*, CSI-SE '16, pages 15–21, New York, NY, USA, 2016. ACM.

[9] Earl T. Barr, Christian Bird, Peter C. Rigby, Abram Hindle, Daniel M. German, and Premkumar Devanbu. Cohesive and Isolated Development with Branches. In

Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, number 7212 in Lecture Notes in Computer Science, pages 316–331. Springer Berlin Heidelberg, March 2012. DOI: 10.1007/978-3-642-28872-2_22.

[10] S. Bayati, D. Parsons, T. Susnjak, and M. Heidary. Big data analytics on large-scale socio-technical software engineering archives. In *2015 3rd International Conference on Information and Communication Technology, ICoICT 2015*, pages 65–69. Institute of Electrical and Electronics Engineers Inc., 2015.

[11] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. *KNIME: The Konstanz Information Miner*, pages 319–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[12] M. Biazzini, M. Monperrus, and B. Baudry. On Analyzing the Topology of Commit Histories in Decentralized Version Control Systems. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–270, September 2014.

[13] Marco Biazzini and Benoit Baudry. "May the Fork Be with You": Novel Metrics to Analyze Collaboration on GitHub. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, WETSoM 2014, pages 37–43, New York, NY, USA, 2014. ACM.

[14] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, May 2009.

[15] K. Blincoe, J. Sheoran, S. Goggins, E. Petakovic, and D. Damian. Understanding the popular users: Following, affiliation influence and leadership on GitHub. *Information and Software Technology*, 70:30–39, 2016.

[16] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How Do Centralized and Distributed Version Control Systems Impact Software Changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 322–333, New York, NY, USA, 2014. ACM.

[17] G. Burlet and A. Hindle. An empirical study of end-user programmers in the computer music community. In *IEEE International Working Conference*

on Mining Software Repositories, volume 2015-August, pages 292–302. IEEE Computer Society, 2015.

[18] Andrew H. Caudwell. Gource: Visualizing software version control history. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 73–74, New York, NY, USA, 2010. ACM.

[19] T. Chaikalis, E. Ligu, G. Melas, and A. Chatzigeorgiou. SEAgle: Effortless Software Evolution Analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 581–584, September 2014.

[20] K. K. Chaturvedi, V. B. Sing, and P. Singh. Tools in Mining Software Repositories. In *2013 13th International Conference on Computational Science and Its Applications (ICCSA)*, pages 89–98, June 2013.

[21] Michael Cochez, Ville Isomöttönen, Ville Tirronen, and Jonne Itkonen. How Do Computer Science Students Use Distributed Version Control Systems? In Vadim Ermolayev, Heinrich C. Mayr, Mykola Nikitchenko, Aleksander Spivakovsky, and Grygoriy Zholtkevych, editors, *Information and Communication Technologies in Education, Research, and Industrial Applications*, number 412 in Communications in Computer and Information Science, pages 210–228. Springer International Publishing, June 2013. DOI: 10.1007/978-3-319-03998-5_11.

[22] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, and Michiel J. L. de Hoon. Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 2009.

[23] Evans Data Corporation. Global developer population and demographic study vol. 2, 2017.

[24] V. Cosentino, J. L. C. Izquierdo, and J. Cabot. Assessing the bus factor of Git repositories. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499–503, March 2015.

[25] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Gitana: A SQL-Based Git Repository Inspector. In Paul Johannesson, Mong Li Lee,

Stephen W. Liddle, Andreas L. Opdahl, and Óscar Pastor López, editors, *Conceptual Modeling*, number 9381 in Lecture Notes in Computer Science, pages 329–343. Springer International Publishing, October 2015. DOI: 10.1007/978-3-319-25264-3_24.

[26] Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from GitHub: Methods, Datasets and Limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 137–141, New York, NY, USA, 2016. ACM.

[27] J. Eyolfson, L. Tan, and P. Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, 2014.

[28] H. Fang, C. Fu, F. Chen, and Q. Xuan. Analyzing file-motif committed by developers in open source software repository. *Complex Systems and Complexity Science*, 12(2):78–84, 2015.

[29] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[30] M. Foucault, M. Palyart, X. Blanc, G.C. Murphy, and J.-R. Falléri. Impact of developer turnover on quality in open-source software. pages 829–841, 2015. DOI: 10.1145/2786805.2786870.

[31] Minyuan Gao and Chang Liu. TeamWATCH Demonstration: A Web-based 3d Software Source Code Visualization for Education. In *Proceedings of the 1st International Workshop on Code Hunt Workshop on Educational Software Engineering*, CHESE 2015, pages 12–15, New York, NY, USA, 2015. ACM.

[32] M. Goeminne and T. Mens. Towards a survival analysis of database framework usage in Java projects. In *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pages 551–555. Institute of Electrical and Electronics Engineers Inc., 2015.

[33] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 233–236, 2013.

[34] Georgios Gousios and Diomidis Spinellis. Conducting quantitative software engineering studies with Alitheia Core. *Empirical Software Engineering*, 19(4):885–925, February 2013.

[35] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work Practices and Challenges in Pull-based Development: The Contributor's Perspective. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 285–296, New York, NY, USA, 2016. ACM.

[36] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: GitHub Data on Demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 384–387, New York, NY, USA, 2014. ACM.

[37] P. Gyimesi, G. Gyimesi, Z. Tóth, and R. Ferenc. *Characterization of source code defects by data mining conducted on GitHub*, volume 9159 of *15th International Conference on Computational Science and Its Applications, ICCSA 2015*. Springer Verlag, 2015.

[38] V.J. Hellendoorn, P.T. Devanbu, and A. Bacchelli. Will They like this? Evaluating code contributions with language models. In *IEEE International Working Conference on Mining Software Repositories*, volume 2015-August, pages 157–167. IEEE Computer Society, 2015.

[39] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The MSR Cookbook: Mining a decade of research. In *2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 343–352, May 2013.

[40] W. Huang, T. Lu, H. Zhu, G. Li, and N. Gu. Effectiveness of conflict management strategies in peer review process of online collaboration projects. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, volume 27, pages 717–728. Association for Computing Machinery, 2016.

[41] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLOS ONE*, 9(6):1–12, 06 2014.

[42] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki. *Github projects. quality analysis of open-source software*, volume 8851 of *6th International Conference on Social Informatics, SocInfo 2014*. Springer Verlag, 2014.

[43] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering*, pages 1–32, May 2016.

[44] H. Kagdi, M.L. Collard, and J.I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*, 19(2):77–131, 2007.

[45] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. M. German. Open Source-Style Collaborative Development Practices in Commercial Projects Using GitHub. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 574–585, May 2015.

[46] 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. DOI: 10.1007/s10664-015-9393-5, 2015.

[47] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.

[48] M. Klug and J.P. Bagrow. Understanding the group dynamics and success of teams. *Royal Society Open Science*, 3(4), 2016.

[49] Y. Kuwata and H. Miura. A study on growth model of OSS projects to estimate the stage of lifecycle. In Howlett R.J., Ding L., Pang C., Leong M.K., and Jain L.C., editors, *Procedia Computer Science*, volume 60, pages 1004–1013. Elsevier, 2015.

[50] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[51] Justin Longo and Tanya M. Kelley. Use of GitHub As a Platform for Open Collaboration on Text Documents. In *Proceedings of the 11th International Symposium on Open Collaboration*, OpenSym '15, pages 22:1–22:2, New York, NY, USA, 2015. ACM.

[52] P. Louridas. Version control. *IEEE Software*, 23(1):104–107, Jan 2006.

[53] J.F. Low, T. Yathog, and D. Svetinovic. Software analytics study of Open-Source system survivability through social contagion. In *IEEE International Conference on Industrial Engineering and Engineering Management*, volume 2016-January, pages 1213–1217. IEEE Computer Society, 2016.

[54] R. Malhotra, N. Pritam, K. Nagpal, and P. Upmanyu. Defect Collection and Reporting System for Git based Open Source Software. In *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*, pages 1–7, September 2014.

[55] K. J. Millman and M. Aivazis. Python for scientists and engineers. *Computing in Science Engineering*, 13(2):9–12, March 2011.

[56] M. Mirakhorli, H.-M. Chen, and R. Kazman. Mining Big Data for Detecting, Extracting and Recommending Architectural Design Concepts. In *Proceedings - 1st International Workshop on Big Data Software Engineering, BIGDSE 2015*, pages 15–18. Institute of Electrical and Electronics Engineers Inc., 2015.

[57] N. Mostafa and C. Krintz. Tracking performance across software revisions. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009*, pages 162–171, 2009.

[58] M. H. D. d Moura, H. A. D. d Nascimento, and T. C. Rosa. Extracting New Metrics from Version Control System for the Comparison of Software Developers. In *2014 Brazilian Symposium on Software Engineering (SBES)*, pages 41–50, September 2014.

[59] Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. An Empirical Study of Goto in C Code from GitHub Repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 404–414, New York, NY, USA, 2015. ACM.

[60] T. E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, May 2007.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[62] Mohammad Masudur Rahman and Chanchal K. Roy. An Insight into the Pull Requests of GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 364–367, New York, NY, USA, 2014. ACM.

[63] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM.

[64] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, Dec 1975.

[65] Z. Shoroye, W. Yaqub, A.A. Mohammed, Z. Aung, and D. Svetinovic. *Exploring social contagion in open-source communities by mining software repositories*, volume 9492 of *22nd International Conference on Neural Information Processing, ICONIP 2015*. Springer Verlag, 2015.

[66] Vinayak Sinha, Alina Lazar, and Bonita Sharif. Analyzing Developer Sentiment in Commit Logs. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 520–523, New York, NY, USA, 2016. ACM.

[67] D. Spinellis. Version control systems. *IEEE Software*, 22(5):108–109, Sept 2005.

[68] S. Sripada, Y. R. Reddy, and A. Sureka. In Support of Peer Code Review and Inspection in an Undergraduate Software Engineering Course. In *2015 IEEE 28th Conference on Software Engineering Education and Training*, pages 3–6, May 2015.

[69] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Incremental Origin Analysis of Source Code Files. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 42–51, New York, NY, USA, 2014. ACM.

[70] T. Suovuo, J. Holvitie, J. Smed, and V. Leppänen. Mining knowledge on technical debt propagation. In Sievi-Korte O., Makinen E., and Nummenmaa J., editors, *CEUR Workshop Proceedings*, volume 1525, pages 281–295. CEUR-WS, 2015.

[71] Parastou Tourani, Bram Adams, Marco Ortu, and Alessandro Murgia. Do Developers Feel Emotions? An Exploratory Analysis of Emotions in Software Artifacts. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 262–271, New York, NY, USA, 2014. ACM.

[72] V. Uquillas Gómez, S. Ducasse, and T. D'Hondt. Visually characterizing source code changes. *Science of Computer Programming*, 98(P3):376–393, 2015.

[73] J. S. van der Veen, B. van der Waaij, and R. J. Meijer. Sensor data storage performance: Sql or nosql, physical or virtual. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 431–438, June 2012.

[74] B. Vasilescu, A. Serebrenik, and V. Filkov. A data set for social diversity studies of GitHub teams. In *IEEE International Working Conference on Mining Software Repositories*, volume 2015-August, pages 514–517. IEEE Computer Society, 2015.

[75] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M.G.J. Van Den Brand. Continuous integration in a social-coding world: Empirical evidence from GITHUB. In *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pages 401–405. Institute of Electrical and Electronics Engineers Inc., 2014.

[76] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. The Sky is Not the Limit: Multitasking Across GitHub Projects. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 994–1005, New York, NY, USA, 2016. ACM.

[77] W. Wang, G. Poo-Caamaño, E. Wilde, and D.M. German. What is the GIST? Understanding the use of public gists on GitHub. In *IEEE International Working Conference on Mining Software Repositories*, volume 2015-August, pages 314–323. IEEE Computer Society, 2015.

[78] K. Yamashita, S. McIntosh, Y. Kamei, A.E. Hassan, and N. Ubayashi. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In *International Workshop on Principles of Software Evolution (IWPSE)*, volume 30-Aug-2015, pages 46–55. Institute of Electrical and Electronics Engineers Inc., 2015.

[79] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for It: Determinants of pull request evaluation latency on GitHub. In *IEEE International Working Conference on Mining Software Repositories*, volume 2015-August, pages 367–371. IEEE Computer Society, 2015.

[80] Q. Zheng, A. Mockus, and M. Zhou. A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, pages 637–648. Association for Computing Machinery, Inc, 2015.