UNIVERSITÉ DE MONTRÉAL

ON HISTORY-AWARE MULTI-ACTIVITY EXPERTISE MODELS

ALEXANDRE COUROUBLE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
MARS 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ON HISTORY-AWARE MULTI-ACTIVITY EXPERTISE MODELS

présenté par: COUROUBLE Alexandre
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. ANTONIOL Giulio, Ph. D., président
M. ADAMS Bram, Doctorat, membre et directeur de recherche
M. MERLO Ettore, Ph. D., membre

DEDICATION

To my parents,
For pushing me to my fullest potential,
And supporting me all these years.

ACKNOWLEDGEMENTS

In the acknowledgements, the author points out the help that various people have provided, including advice or any other type of contribution as the author carried out their research. As appropriate, this is the section where the candidate must thank their thesis or dissertation supervisor, grant-awarding organizations, or companies that provided bursaries or research funds.

If the thesis or dissertation is in French and students wish to thank someone in particular in English, they must insert a second page and title it in English (i.e., “Acknowledgments”). If the thesis or dissertation is in English and students wish to thank someone in particular in French, they must insert a second page and title it in French (i.e., “Remerciements”).

RÉSUMÉ

Durant l'évolution d'un projet de logiciel, les contributions individuelles d'un developeur present dans le projet vont lentement se faire remplacer par les contributions d'autre dévelopeurs. Ceci engendrera l'érosion de l'empreinte des contributions de ce developeur. Bien que les connaissances de ce dévelopeur n'ont pas disparu du jour au lendemain, pour une personne externe au projet, l'expertise de ce developeur est devenue invisible.

Grace à une étude empirique sur une periode de 5 années de developement de Linux, nous étudions le phénomène de l'érosion de l'expertise en créant un modèle bidimentionnel. La première dimention de notre modèle prend en compte les différentes activités entreprises par les membres de la communauté de développement de Linux, comme les contributions en termes de code, les contributions aux revues de code soumit par d'autre dévelopeurs, ou encore la soumission de code d'autres dévelopeurs en amont. La deuxiéme dimention de notre modèle prend en compte l'historique des contributions citées plus haut pour chaque dévelopeurs.

En applicant ce modèle, nous decouvrons que, bien que les empreintes de contributions de certain dévelopeurs diminuent avec le temps, leurs expertise survit grace à leurs implications dans les divereses activités mentionées plus haut.

ABSTRACT

As software evolves, a maintainer's contributions will gradually vanish as they are being replaced by other developers' code, resulting in a slow erosion of the maintainer's footprint in the software project. Even though this maintainer's knowledge of the file did not disappear overnight, to outsiders, the maintainer and her expertise have become invisible. Through an empirical study on 5 years of Linux development history, this paper analyses this phenomenon of expertise erosion by building a 2-dimensional model of maintainer expertise involving a range of maintainer activity data on more than one release. Using these models, we found that although many Linux maintainers' own coding footprint has regressed over time, their expertise is perpetuated through involvement in other development activities such as patch reviews and committing upstream on behalf of other developers. Considering such activities over time further improves recommendation models.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

LOC	Lines of Code
SCM	Source Code Managment
OSS	Open Source Software
MSR	Mining Software Repositories

CHAPTER 1 INTRODUCTION

Although Linux uses Git as its version control system, most developers cannot commit directly to the repository. In fact, the developers and maintainers use several different repositories in the development process. The main repository (also called the main tree, or the main line) is maintained by Linus Torvalds, the creator of Linux. For a developer to have her code changes integrated into an official release of the Linux Kernel, her changes must be submitted and accepted by Linus Torvalds, as he has the last word on any code being added to the main tree.

Submitting code changes *upstream*, means to submit changes hoping they will be *merged* into the main tree, and thus into an official Linux release. To achieve this, there is a set of guidelines to follow, as the Linux Kernel follows a strict development cadence and has high code quality standards. First, the code submitted must follow the Linux coding style [1]. It is important to impose a strict coding style in projects of the scale of the linux kernel. Code coming from tens of thousands of developers would become very hard to maintain if everyone submitted code with their own coding style.

Second, developers must follow the submission process guidelines. In the vast majority of cases, developers submit their changes to a maintainer. A maintainer is in charge of maintaining a specific subsystem. A subsystem can be represented as a series of files that work together to serve a certain purpose. When a developer is unsure of who to send the changes to, they can consult the `MAINTAINERS` file, or use the script `get_maintainer.pl` to discover person responsible for a certain file. The developer must send her changes in the form of a patch to the modified subsystem's mailing list.

Naturally, maintainers must review the patches to ensure that they are worthy of being integrated into their subsystem tree. These reviews usually occur in the email thread following the email patch. The patch author will probably have to improve the code changes according to the maintainer's reviews. If maintainers are satisfied with a patch, they will *commit* it to their repository. As explained in section 2.3, for each commit, git differentiates between the **commit author** and the **commit committer**.

Before submitting changes to Linus, maintainers usually send the changes acquired in their repository to *linux-next*[2]. They may do so through an email as a patch, although submitting large changes is easier throught Git. Linux-next is where the integration testing occurs. This

[1] `https://01.org/linuxgraphics/gfx-docs/drm/process/coding-style.html`

[2] `https://01.org/linuxgraphics/gfx-docs/drm/process/howto.html#becoming-a-kernel-developer`

is where developers and maintainers make sure new code changes do not interfere with the rest of the code base and do not introduce any bugs. Linus will pull commits that have been in the -next tree for a few weeks [3]. Linux-next ensures important bugs introduced by new commits are discovered and dealt with before being committed to the main tree, as these bugs would drastically slow down the release cycle.

The main tree uses a specific release cycle. After a new version is released, 4.13 for instance, the *merge window* opens for release 4.14. This two-week long merge window is the only opportunity to submit new code changes in hope to have them integrated in release 4.14. Linus pulls most of the changes from linux-next during that time period because these changes are les likely to cause bugs and thus delay the next release. After two weeks, the merge window closes and Linux 4.14-rc1 is released. Then, developers work on ensuring that the kernel is as stable as possible. For approximately 6 to 8 weeks, Linus will only accept patches that address bugs introduced by commits merged during the merge window. The only exception are new drivers. New drivers can be submitted outside of the merge window because bugs introduced by drivers are self contained, and only affect users using the specific driver. A new -rc comes out about every week, until Linus declares that the kernel is stable enough to be released.

Furthermore, large subsystems impose their own release schedule to developers. For instance, the Net subsystem also has two main trees: `net` and `net-next`. Net-next receives all changes submitted to the net subsystem. Once the linus' merge window opens, net-next closes, and all the changes accumulated over the last 10 weeks will be submitted to the main tree. At this point, the net tree will recieve all the fixed related to commits that were committed to the main tree. Once Linus' merge window closes, net-next reopens and developers are free to submit new patches again.

As stated above, the contribution process relies heavily on emails. The reason why the linux community does not use tools such as github or gerrit is that those tools would not scale to the size of the linux community (Armstrong et al., 2017a). And even though the email-based system has been very reliable over the lifetime of the project, there is one drawback. Once a patch is committed to the subsystem tree, and after the commit makes its way upstream to the main tree, there is no easy way to recover the review discussions that took place in the mailing list. The information contained in this discussion is hard to recover manually.

The process described above shows the number of different activities in which Linux developers participate. In particular, maintainers, who are regarded as experts by the rest of the community, tend to participate in a larger array of activities than other developers such

[3] `https://01.org/linuxgraphics/gfx-docs/drm/process/howto.html#becoming-a-kernel-developer`

as upstream committing and patch reviews. In fact, maintainers are often the only people allowed to commit to the subsystem's git repository. Their role being crucial to their subsystem, it is important to be able to find a replacement for them if they decided to give up their maintainer responsibilities.

Employee turnover represents a risk for software organizations. (Joseph et al., 2007) claim that the cost associated with replacing a developer can be up to 1.5 times the developer's salary. As an answer to this risks in software engineering organizations, researchers have attempted to establish models capable of finding experts in certain code areas. By recommending experienced developers, these models would ease the difficulty of replacing current experts as they leave the organization.

Introduced in the early 2000s(Mockus and Herbsleb, 2002)(McDonald and Ackerman, 2000), these models were using the developers' coding activities to assess their expertise. Their activity was quantified in terms of two simple metrics: number of Lines of Code (LOC) and number of commits, which represents the amount of change brought to the source code.

The Linux Kernel, for instance, is a 26-year-old project that has experienced a drastic growth in terms of code base and in terms of developer community. Wanting to keep a high standard of code quality and a fast release system, the community had to implement a hierarchical contribution system, as stated earlier. In this contribution system, some developers are responsible for the maintenance of their subsystems. These developers, or maintainers, are trusted by the rest of the Linux community to be skilled enough to ensure the durability of their subsystem. This trust was acquired by contributing to the subsystem over the years. However, when trying to use traditional expertise determining metrics in the linux kernel, maintainers are rarely recommended. Our results indicate that the number of lines of code only correctly assess less than 50% of experts and number of commits authored only finds 25% of experts.

We believe that previous expertise models fail to take into account some critical aspect of large scale software development. As subsystems become larger and welcome more contributors, maintainers have to dedicate more of their time reviewing other developers' contributions to maintain code quality. Our results indicate that this shift towards a managerial position is usually followed by a drop in code contributions by that person. This implies that, in the eye of a model looking at only LOC and commits, the maintainer will lose expertise.

Additionally, previous models do not take into account the amount of time a developer has been involved in the development of the project. Measuring previous contributions could help quantifying a developer's experience and the expertise she acquired over the years.

We introduce a new expertise model adapted to large software organizations like the Linux community. Our model is better suited for such organizations because it can take into account all the activities undertaken by both maintainers and developers. In addition to quantifying different activities, we use a historical perspective to the model to take into account the previous contributions made by the developers. This thesis provides an in-depth description of the different steps necessary for the creation of this model as well as an evaluation of the accuracy of our model.

1.1 Hypothesis of Thesis

Our main research objective is to create and evaluate an expertise recommender capable of detecting maintainers in a certain subsystem. Our evaluation, provided in chapter 5, indicates that state of the art expertise models are not well suited for the detection of Maintainers, because of the nature of their work. Thus, we based our model on a series of new activities as well as a historical perspective to detect maintainers with higher accuracy.

> *Maintainer detection models targeting large software engineering organizations require a look at (1) all development activities, including code reviews and upstream commits and (2) a look at historical evolution of those activities.*

To explore this hypothesis, we evaluate the relevance of the different activities present in large scale software development for the purpose of detecting maintainers. We build a recommendation model based on metrics quantifying these activities. Moreover, we include a historical dimension to the model to represent the impact of long term involvement with the software project. Finally, we provide an evaluation of the different activities by comparing the maintainer recommendations to the data listed in the `MAINTAINERS` file, which contains an updated list of maintainers for the linux kernel.

We build an expertise-based maintainer recommendation model which considers a wider array of metrics to assess maintainer candidates. Since the Linux contribution process is similar to the one used by other organizations, this model could be accurate in other large software projects.

We were able to make some of the metrics used in the recommendation model available to the community through two open source projects, which are described in Chapter 3.

1.2 Plan of the Thesis

The structure of this thesis is as follows. We conduct a literature review in chapter 2, where we describe previous expertise models and other topics of interest. The previous expertise models will serve as a base line approach to detect maintainers. Chapter 3 gives a detailed overview of the general workflow of the thesis. Chapter 4 gives an in-depth look at Email2git, one of the open source project we created to provide access to part of the metrics acquired. Finally, chapter 5 includes the article we submitted to the 15th International Conference on Mining Software Repositories. Finally, we end the thesis in chapter 6 with a discussion and a conclusion.

CHAPTER 2 CRITICAL LITERATURE REVIEW

Our research project touches upon two areas of software engineering research: *mining software repositories* and *software engineering expertise*. Besides these two research fields, we relied on another type of literature to conduct this research project: Git and Linux documentation. This chapter provides a critical literature review of our academic research areas and a description of the information available in the Linux and Git documentation, as well as how it helped us find solutions for the problems encountered.

2.1 Mining Software Repositories

In addition to providing a contribution platform, Source Code Managment (SCM) systems track and save large amounts of information about changes brought to the source code. During the lifetime of the project, the SCM acquires a large amount of data about the development of various projects. Mining software repositories researchers *mine* this data for their research projects. This data can come from version control systems, mailing list, or bug tracking systems.

Software repositories are not limited to SCM. There are other types of software repositories that researchers mine to gather information about software projects. These repositories include bug tracking systems, mailing lists, source code, and issue tracking systems. Over the years, researchers have used mining software repositories techniques that enabled them to research different topics of software engineering (Bird et al., 2009).

In the scope of our research, we used mining software repositories techniques in each different part of the project. Chapter 3 discusses two open source projects we created during this research project. The data used for both projects came from mining the Linux Kernel git repository and the Linux mailing lists. We eventually used this data for the creation of our maintainer recommendation system.

One of the difficulties often encountered by researchers in mining software repositories is the inability to link data coming from different entities of the software repository. In the case of the linux kernel, the difficulty was to link data from the mailing lists to the data from the git repository. A difficulty we addressed with an algorithm introduced in (Jiang et al., 2013, 2014), as explained in chapter 4.

Furthermore, (Armstrong et al., 2017a) studied the difference between *unicast* and *broadcast* review systems. A unicast review system, like gerrit, provides an environment in which the code reviews are only sent to the author and cced people by default but are still accessible by other developers. On the contrary, broadcast review system, like the email system used by the Linux community, shows the code reviews to each reader of the mailing list. There are advantages and disadvantages to both systems. The authors note, through an empirical study, that unicast reviews lead to less bugs in the future, but that broadcast systems allows for faster review cycles and allow beginers to learn the code base faster.

2.2 Software Engineering Expertise

Many different studies explored the concept of expertise in software engineering. We believe previous work on expertise is crucial to our model because we consider the population of Linux maintainers to be a subset of the population of Linux experts. Hence, we believe we can improve upon existing expertise recommendation techniques to recommend maintainers. In this section, we describe the different expertise models that have been published in the past.

Typically, existing recommenders base their assessment of expertise on **non-historical models**, meaning that they only exploit data of the current snapshot of the software repository, not of earlier versions. These non-historical models use a variety of different metrics to determine expertise.

Implementation expertise. This type of metric implies that a developer gains expertise through implementation, in other words, by making changes to a file, as tracked by a version control system. McDonald McDonald and Ackerman (2000) recommends the last developer who made a change to a file as expert for that file, which could be interpreted as file-level `git blame`. Later, Mockus et al. Mockus and Herbsleb (2002) improved on the approach proposed by McDonald McDonald and Ackerman (2000) by counting the number of changes each developer brought to a file to provide a better expertise recommendation. Girba et al. Girba et al. (2005) later improved Mockus et al.'s method Mockus and Herbsleb (2002) by measuring the size (churn) of each change in terms of lines of code.

Usage Expertise. Developers gain usage expertise by calling ("using") specific methods from within their code. This concept of usage expertise was introduced by Schuler and Zimmermann Schuler and Zimmermann (2008). In later work Ma et al. (2009), the authors compare the accuracy of usage expertise against implementation expertise recommenders. They concluded that usage expertise recommends experts with similar accuracy to implementation expertise models.

Additionally, Fritz et al. Fritz et al. (2007) validate the accuracy of implementation expertise techniques. After a qualitative study consisting of 19 java developers interviews, they were able to confirm a relationship between changes made (commit frequency) and expertise. In addition to that, they found evidence proving that authorship (as obtained from the amount of churn contributed, or through "git blame") is also capable of indicating expertise. In a later study, the authors Fritz et al. (2010) create a degree of knowledge model combining both usage and implementation metrics to recommend experts.

Bhattacharya et al. Bhattacharya et al. (2014) explored the suitability of different implementation expertise metrics depending on a developer's role. They argue that state-of-the-art metrics (lines of code and commits added), being unaware of the developer's role, can lead to inaccurate results. They add that code activity metrics like the number of lines of code added, only describe expertise at a local level and poorly capture global expertise.

Thus far, all models cited above are not **history aware**. Hence, they do not take into consideration the effect of time on developer expertise and assume that developers' memory lasts forever. Based on a survey of psychological literature, Hattori et al. Hattori et al. (2012) create a memory retention model to improve expertise models. Memory retention is computed using a Forgetting function, which reduces the weight of older activities to account for memory loss. The data used in the experiment was acquired by a tool that records information from the developer's IDE. It would be impossible to reproduce this experiment on a project of the scale of the Linux Kernel because it would be impossible to force all Linux developers to use the IDE integrated tool.

During the creation our model, we used methods described in most of these expertise models, as described in chapter 5.

2.3 The Anatomy of a Git Commit

A git commit is a fundamental concept in the scope of this research and for the understanding of git in general. The changes brought to source code by developers are contained in a *commit*. If a developer is tasked to fix a bug or to create a new feature in a project, she will have to modify the source code in order to implement these changes. When a developer feels ready to share these changes, she can apply them to the repository in the form of a git commit. The changes are represented in the *commit diff*, which contains the exact lines to be removed (-) or added (+) in the source code. Git uses the +/- lines to modify the repository of someone who *pulled* the changes from the developer as depicted in Figure 2.1.

Furthermore, commits contain an array of metadata regarding the changes committed, all of which is accessible to anyone with a copy of the repository through a handful of built-in commands. For example, git log returns information about the past commits in the repository. Figure 2.1 shows one commit in the output printed by the following command: `git log --pretty=fuller --patch`, where the `--pretty=fuller` shows more information and `--patch` shows the commit diff. There are three main parts to the commit as seen in the image: the header, the message, and the diff.

The header contains the following data points:

```
commit ee70daaba82d70766d0723b743d9fdeb3b06102a
Author:     Eryu Guan <eguan@redhat.com>
AuthorDate: Thu Sep 21 11:26:18 2017 -0700
Commit:     Darrick J. Wong <darrick.wong@oracle.com>
CommitDate: Tue Sep 26 10:55:19 2017 -0700

    xfs: update i_size after unwritten conversion in dio completion

    Since commit d531d91d6990 ("xfs: always use unwritten extents for
    direct I/O writes"), we start allocating unwritten extents for all
    direct writes to allow appending aio in XFS.

    But for dio writes that could extend file size we update the in-core
    inode size first, then convert the unwritten extents to real
    allocations at dio completion time in xfs_dio_write_end_io(). Thus a
    racing direct read could see the new i_size and find the unwritten
    extents first and read zeros instead of actual data, if the direct
    writer also takes a shared iolock.

    Fix it by updating the in-core inode size after the unwritten extent
    conversion. To do this, introduce a new boolean argument to
    xfs_iomap_write_unwritten() to tell if we want to update in-core
    i_size or not.

    Suggested-by: Brian Foster <bfoster@redhat.com>
    Reviewed-by: Brian Foster <bfoster@redhat.com>
    Signed-off-by: Eryu Guan <eguan@redhat.com>
    Reviewed-by: Darrick J. Wong <darrick.wong@oracle.com>
    Signed-off-by: Darrick J. Wong <darrick.wong@oracle.com>

diff --git a/fs/xfs/xfs_aops.c b/fs/xfs/xfs_aops.c
index 29172609f2a3..f18e5932aec4 100644
--- a/fs/xfs/xfs_aops.c
+++ b/fs/xfs/xfs_aops.c
@@ -343,7 +343,8 @@ xfs_end_io(
                error = xfs_reflink_end_cow(ip, offset, size);
                break;
        case XFS_IO_UNWRITTEN:
-               error = xfs_iomap_write_unwritten(ip, offset, size);
+               /* writeback should never update isize */
+               error = xfs_iomap_write_unwritten(ip, offset, size, false);
                break;
        default:
                ASSERT(!xfs_ioend_is_append(ioend) || ioend->io_append_trans);
```

Figure 2.1 The anatomy of a commit

- **Commit ID:** The commit's unique identifier.
- **Commit Author:** Name and email address of the developer who *wrote*, or *authored* the code change.

- **Author Date:** Time, date, and time zone at which the changes were submitted.
- **Commit Committer:** Name and email address of the person who committed the code to the repository.
- **Commit Date:** Time, date, and timezone at which the commit was committed tot he repository.

In the scope of the Linux Kernel, the Commit Author rarely is the Commit Committer. As explained in chapter 1, the author is the person who wrote the code, and then submitted it for review as a patch in an email. The commit committer is the person that received, accepted, and committed the changes to their repository.

The commit message contains the following data points:

- **Commit summary:** Often called the commit title, a brief explanation of the purpose of the commit.
- **Commit Message:** In depth explanation of the purpose of the commit.
- **Credit Attribution Tags:** List of people who were involved in the commit and the nature of their involvement.

There are many different types of credit attribution tags, each describing the way the person contributed to the commit. The most common ones, and the ones we use in this study are: *Signed-off-by*, *Reviewed-by*, and *Acked-by* (acknowledged by). The tags have the following meanings[1]. *Signed-off-by* indicates the developer assisted in the creation of the patch or that she committed it upstream. *Acked-by* is used by developers who were not involved in the creation of the patch but wanted to record their approval. *Reviewed-by* is used to credit developers who contributed reviews to the submitted patch.

The commit diff, which sits at the end of the git log output, shows the exact files and lines that were modified by the author of the commit. Git uses the commit diffs to apply the changes to the files in the repository. The diff can be perceived as the set of instructions to transform the source code into the desired state.

2.4 Open Source Participation

Previous work studies developers' motivation to participate in Open Source Software (OSS). (Wu et al., 2007a) warn that the loosely organized nature of OSS development could be

[1] `https://www.kernel.org/doc/html/v4.12/process/submitting-patches.html`

associated with a high turnover rate and in unexpected departures. Other work (Rigby et al., 2016), (Foucault et al., 2015), (Daniel Izquierdo-Cortazar and Gonzalez-Barahona, 2009), (Mockus, 2010), (Torchiano et al., 2011), and (Ricca et al., 2011) study the impact of a high turnover on the organization. The authors argue that departing developers leave the project with the knowledge they acquired during their time as a contributor, removing this knowledge from the project. We believe that this implies that OSS projects are at risk of knowledge loss and we believe that accurate expertise modeling could assist in addressing this issue.

CHAPTER 3 GENERAL RESEARCH WORKFLOW

As stated in chapter 1, the goal of our research project is to create an expertise model capable of detecting maintainers in a given subsystem. Although our model could be applied to other projects, we focused our evaluation on the Linux Kernel. Our expertise model takes into account many different activities not only present in the linux contribution process, but also in other large-scale software engineering projects. These activities translate into metrics as we attempt to quantify them. To give back to the Linux community, we made those metrics readily available through two open source tools.

In this chapter, we describe the general workflow of the research. section 3.2 introduces the expertise model we created during our research project, whose approach and evaluation were submitted as a paper to the 15th International Conference on Mining Software Repositories[1]. In section 3.1 and section 3.3, we introduce the tools we created, providing an explanation of how the metrics helped with the creation of our expertise model.

3.1 Srcmap

In the interest of increasing the visibility of the authors of the Linux Kernel, we built a data visualization tool capable of displaying a wide array of information about directories or files found in the linux git repository. This information includes code age and individual developer footprint. We wanted to display the following data points about each file and directory of the source code:

- LOC
- Median age of the LOC within a file/directory
- Number of lines of code modified since 2016
- A list of the 20 developers with the most lines of code
- A bar plot displaying the distribution of line of code age

We needed an interface that would allow the user to navigate the different files and directories of Linux while displaying our list of data points, which is why we chose to base the tool on a treemap. Treemaps, which were introduced by (Bederson et al., 2002) as a solution to display

[1] `https://2018.msrconf.org/`

large hierarchical dataset on a 2 dimensional plane, were a great fit to display the structure of the Linux directory tree.

This tool served two purposes. First, perceived as a contribution to the Linux community, we acquired a lot of contacts and gained goodwill in return. Second, we gained a greater understanding of the contribution process and we discovered the concept of decreasing LOC footprint as explained in subsection 3.1.4.

3.1.1 Srcmap 1.0

In the first version of Srcmap[2], we used the Google Chart treemap implementation[3]. This easy to use library allowed us to create a quick proof of concept.

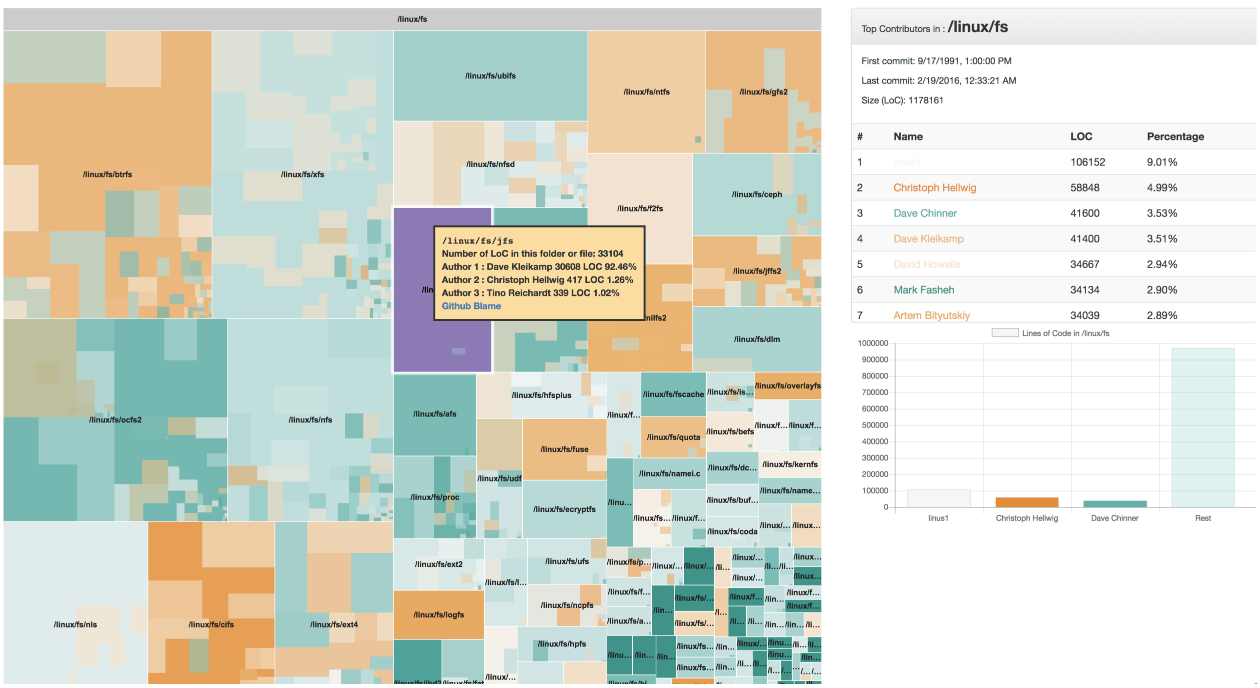

Figure 3.1 First version of srcmap

Figure 3.1 shows the first version of Srcmap. The different boxes represent subdirectories of the Linux Kernel. The different colors within each box give a preview of the content of the box, or the files and subdirectories contained in that directory. In this version of the tool, the color represents the developer having contributed the most lines of code in the contained files. Furthermore, the size of the boxes is proportional to the number of lines of code existing

[2] http://mcis.polymtl.ca/~courouble/linux.html
[3] https://developers.google.com/chart/interactive/docs/gallery/treemap

within the file or directory represented by the box. The panel on the right of the screen and the tooltip contain most of the data: exact number of lines of code, age of the first and last lines of code added, and the top 20 authors and their percentage of lines of code contributed.

3.1.2 Srcmap 2.0

We encountered scalability issues with the first implementation after adding data to the treemap. After some research, we discovered a new treemap implementation[4] capable of handling large datasets and deeply nested structures, which we used for the second version of the tool[5]. This new version, shown in Figure 3.2, introduced three important features:

- Coloring the files and directories according to one out of three metrics:
  - LOC
  - Median age
  - Number of commits since 2016, or 'hot files'
- File search
- A plot displaying the *age* distribution of LOC present in the file/directory.

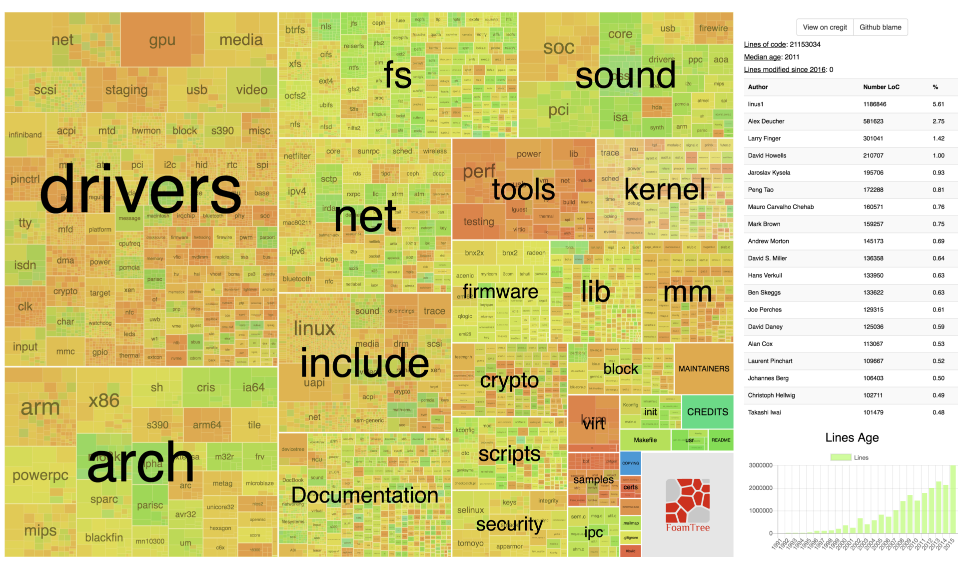

Figure 3.2 Second version of srcmap

[4] https://carrotsearch.com/foamtree/

[5] http://mcis.polymtl.ca/~courouble/dev/

Although the foamtree library had a steeper learning curve than the Google Charts Treemap library, the versatility provided by the foamtree library allow us to provide a much more pleasant user experience and easier access to the data.

The very first Srcmap prototype consisted of a treemap displaying only the `net` subdirectory, which was small enough to provide a smooth user experience. However, Scalability issues started to arise when we attempted displaying the entire Linux Kernel source code. The data would take up to 30 seconds to load, and navigating between each node became very slow.

Furthermore, we discovered the limitations of Google Charts when we tried adding new features to the tool.

This is why we decided to create a new version of the tool with a different treemap library. The new library, Foamtree, provided a richer API and allowed for smooth browsing through deeply nested tree.

3.1.3 Community Engagement

After the creation of the first implementation of Srcmap, we attended LinuxCon 2016 to meet with a series of Linux developers and maintainers. The goal of these informal interviews was to receive early feedback on the tool. In addition to that, We traveled to Santa Fe, New Mexico to present Srcmap at the Linux Plumbers Conference. After discussing srcmap and our research to a series of Linux developers, we deducted the following.

Firstly, experienced linux developers and maintainers are accustomed to their own development workflow. According to our interviews, experienced developers have acquired *muscle memory* from developing in the same editor and terminal over the years. Moreover, experienced maintainers were not interested in a web-based visualization tool, especially since the metrics displayed in Srcmap were accessible from the Linux git repository.

However, the interviewed developers showed interest in combining srcmap with work that was previously done by Jiang et al. (2014): linking Linux git commits to email patches and code reviews. Since our wished to provide a tool that would increase developers' productivity, this became our next goal. Access to the original patches and code reviews would save a lot of time to developers trying to further understand an unknown subsystem.

3.1.4 Lessons learned

We learned many important lesson during the creation of Srcmap. From a community perspective, we now understand the importance of understanding the needs and the habits of the targeted community.

From a research point of view, the creation of srcmap allowed us to understand an important concept in the creation of our expertise model. The data displayed in the tool represents the footprint of the developer in the given file or directory. As we updated srcmap to a newer release of the Linux Kernel, we noticed that the larger footprints were decreasing, which lead us to discover the concept of decreasing footprint, an important aspect of our expertise model.

In conclusion, although Srcmap was not as successful as we originally hoped, we learned many important concepts that helped us in the rest of our research.

3.2 Expertise Based Maintainer Recommendation Model

To address the claim made in our hypothesis, we create an expertise model based on multiple different activities and with a historical dimension. As stated in chapter 1, we discovered that maintainers' contribution frequencies decreased over time. This results in their *LOC footprint* to slowly erode because of the contributions coming from other developers. Wondering whether this decrease in footprint was the result of maintainers reconsidering their involvement with the Linux community, we analyzed other metrics. Starting with the data collected for the creation of Email2git, we take a look at the other aspects of large scale software development. We discovered that as the Linux community becomes larger, maintainers are spending and increasing amount of time review code changes submitted by other developers. With this data available, we were able to improve state of the art expertise model. Chapter 5 presents the paper in which we introduce and evaluate our expertise model.

3.3 Email2git

The linux contribution process has been a reliable way to pipe code contributions (patches) from developers around the world, to the main Linux repository. With a working copy of the Linux Kernel on their computers, developer can modify the source code and, if desired, submit their changes for review, in hope to integrate the main tree. If accepted, the maintainer will *commit* the changes to his local git repository, and submit the changes *upstream* to another maintainer.

Although this system has been very reliable, it has one major drawback: once committed, it is complicated to easily find the email conversation that eventually led to the creation of the patch. We addressed this drawback by implementing an algorithm capable of backtracking the origin of commits in the Linux Git repository introduced by (Jiang et al., 2014). The algorithm and the issues related to scalability are described in chapter 4.

The data generated by the algorithm consists of a list of commit to patch matches. The matches are accessible online through two interfaces: as a commit ID search through the Email2git interface[6], or though the Cregit interface[7]. Chapter 5 provides an in-depth description of our implementation of Email2git.

[6] http://mcis.polymtl.ca/~courouble/email2git/
[7] https://cregit.linuxsources.org/

CHAPTER 4 EMAIL2GIT: FROM ACADEMIC RESEARCH TO OPEN-SOURCE SOFTWARE

As explained in chapter 1, the linux contribution process is a strong email-based system that has proven to be reliable and scalable over the years. Like in many other organizations, the linux development community makes extensive use of code reviews to ensure the quality of contributors' code submissions. These code reviews occur in the mailing lists, where the patch was first introduced. After accepting and integrating the patch to the git repository, it is usually very hard to recover the original patch, the code reviews, and the discussion that took place during the code reviews.

Other tools, such as github and gerrit, have addressed this issue by providing a code review environment and by keeping track of the code review for each commit. However, these environments often do not broadcast the code reviews to everyone. These *unicast* review environments only notify the author to read the code reviews by default. In a quantitative study, (Armstrong et al., 2017a) examined the differences between *unicast* and *broadcast* review system. The authors discovered that broadcast allows for faster review cycle, and provides learning material for new developers. The linux community never used these tool for upstream contributions as they would not scale to the size of the linux community.

Since the kernel's mailing list-based review process is here to stay, we instead address the lack of way to backtrack reviews, we implemented Email2git. The tool is able to find, for a given linux commit, the original patches and the code reviews. This chapter introduce Email2git, from its inception, to its deployement to production.

4.1 Previous Publications and Original Algorithm

The original algorithm capable of backtracking patches from commits was introduced in two papers (Jiang et al., 2013, 2014) published by Jiang, a former member of the MCIS Lab.

The ultimate goal of the algorithm is to find the patch or patches that introduced a single commit. To do that, Jiang creates a SQLite database containing all the patches extracted from the mailing lists. This database contains each patch's +/- lines. The script will then parse a dataset containing all the git commits and find corresponding patches in the database. More specifically, for each commit the script queries the patch database for patches containing the same +/- line contained in the commit. The algorithm then ranks the possible matches by the proportion of lines in common between the commit and the patch. It is important to understand that a patch may have undergone several rounds of reviews before being committed. This means that a single commit will likely be matched to multiple patches.

Although this script was a great proof of concept, I was not able to scale it on our 8-year-long dataset. Hence, we designed and implemented a new algorithm based on the original algorithm capable of scaling to larger dataset. This new implementation is described in section 4.2.

4.2 Scaling the Algorithm

Since we had access to email patches dating back to 2009, we decided to extract git commits from the Linux git repository from the same date until the release of linux v4.11, which represent over *500,000 commits* to analyse. Unfortunatly, this amount of data was too large for the orignal algorithm to parse in a timely fashion.

This called for a new, scalable algorithm that leverages the heuristics mentioned in (Jiang et al., 2013, 2014).

We were able to separate the matching process into multiple different phases. After each phase, the matched commits and patches are removed from the dataset to reduce the workload for the next phase. The different phases are explained in this section.

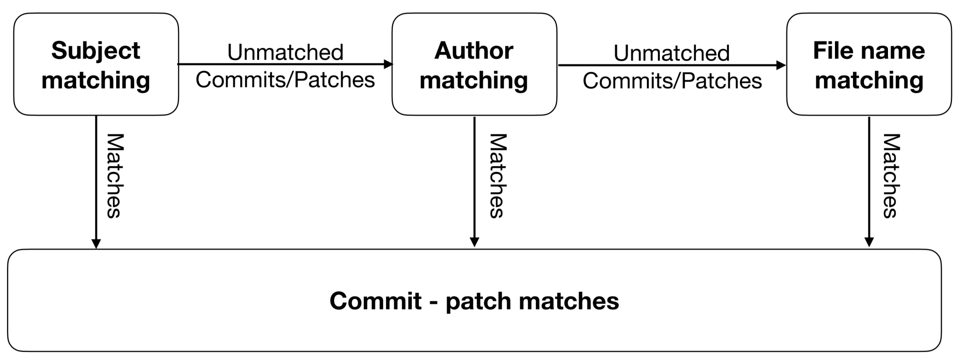

Figure 4.1 The three phases of the new algorithm

4.2.1 Patch Email Subject

The most important heuristic that drastically increased the matching speed is the *email subject - commit summary* concept. The built-in git feature `git format-patch` and `git send-email` allows developers to easily submit their changes to a maintainer by email according to the Linux Kernel Contribution guidelines[1]. The piece of information we are interested in is the email subject. `Git am` automatically saves the email subject and uses it the "commit summary". This commit summary, or commit title, is the first line of the commit message as explained in section 2.3. Comparing both strings of characters allows for a very quick first phase of matching.

Jiang originally used the email subject as oracle to evaluate her original algorithm. We use it as the first phase of the new matching process. We then remove the matched commits and patches from the dataset to reduce the workload of the next steps.

4.2.2 Author and Affected Files

Even though the number of commits was reduced by half after the first phase, the remaining data was still too large to use brute force (comparing each commit against each patch). This is due to the time complexity of the script. For each commit to analyse, the script has to go through every patch in the dataset. Given a time complexity of $O(n^2)$, the time taken to execute the script increases quadratically as the size of the dataset increases. Thus, we had to find a way to use the available meta-data to speed up the matching. The first data point used was the *author name*. As depicted in Figure 4.2, one can use the name and email address of the *commit author* to pin point to the patches that were sent by the same person. In other words, to find a match for each commit, the algorithm has to parse a handful of patches (those by the same author as the commit) instead of hundreds of thousand. After a commit is pointed to a group of patches, it utilises the line-based algorithm used in the original script. Similarly, we use the same technique with the name of the files affected by a commit and a patch can to increase the performance of the +/- line algorithm. Through regular expression and text parsing, we can retrieve the files that are modified in the patch and the *commit diff*. Since the author-based matching is slighty faster and returns more matches than the file-based matching, we start with the former, removing each matched commit and patch to reduce the workload of the next phase.

4.3 The Data

There are two sides to this matching process: the Linux git repository and the archives containing the patches sent in mailing lists over the years. We need to extract the diff (+/- lines), the metadata, and the subject and commit summary from both side. The scripts used for each part of the data extraction are available on the Email2git project's github under the

[1] `https://kernelnewbies.org/FirstKernelPatch`

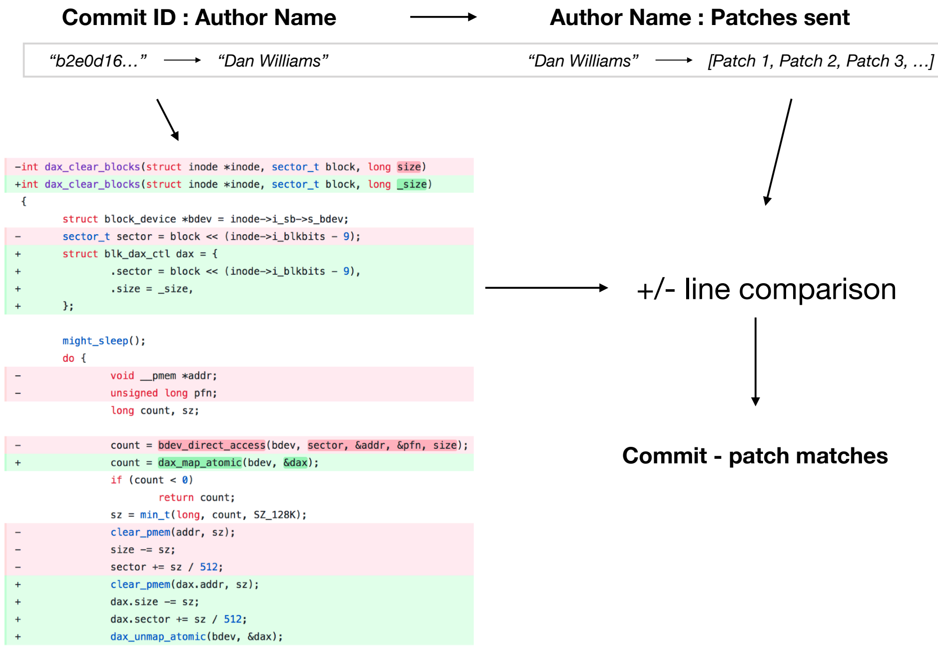

Figure 4.2 Using the patch sender to assist matching

GPL-3.0 license [2].

Because we wanted Email2git to be a usable and practical tool, we needed a way to display the patches and the code reviews in a browser. **Patchwork**[3] is a tool designed to assist maintainers of open source projects using an email-based contribution process. It tracks the mailing lists used by developers to submit patches and receive code reviews. The tool extracts each detected patch as well as its associated reviews, then displays them in a web-based user interface. Additionally, Patchwork stores the patches in a database, along side a unique ID that can be used to generate a URL to that patch.

We were granted read access to the MySQL database behind a patchwork instance hosted on kernel.org[4]. This instance has been tracking 69 of the many linux subsystems mailing lists since 2009, giving us the opportunity to analyse over *1.4 million* patches.

In addition to being a data source, patchwork.kernel.org also represents a way for us to display the patches and the code reviews associated with commits to the users. The only limitation of this patchwork instance is that it does not track some major mailing lists, particularly some of the `Net` mailing lists.

[2] `https://github.com/alexcourouble/email2git`
[3] `https://github.com/getpatchwork/patchwork`
[4] `https://patchwork.kernel.org/`

4.3.1 The Commits

First, we need to extract the commit summary from each commit after 2009. This date is our lower bound because our email data from patchwork.kernel.org only has email patches dating back to 2009. The commit summary is the first line of the commit message, which makes it very easy to retrieve. The script `subject_data_gen/commit_subject_generator.py` reads a git log output and stores the commit summary for each commit in a SQLite3 database. The exact git log command used is the following:

```
git log --no-merges --pretty=format:"%H,%ct,%s" --after={2009-01-01}
```

The `pretty` option formats the output according to the passed parameters.

The next step on this side of the data is to extract the data for the other phases of the matching process. The script `lines_data_prep/git_prep.py` is more complicated, as there is more data to parse and save. This script reads the authors and the files affected by each commit. It will then create two maps: commit ID to author, and commit ID to files affected. These maps, which exist as python dictionaries are then saved to two separate pickle files[5], which make writing, reading, and storing data fast and easy. This script also extracts the +/- lines from the commit diff and stores them in a pickle file as well.

4.3.2 The Patches

The patches are stored on a remote serve in a MySQL database, the same database that hosts the patchwork.kernel.org data. Through the help of SQL queries, we gathered all the necessary data in csv files to avoid complications arising from handling a production database. Once those csv files created, I could parse them with the help of two python scripts available in the Email2git githubg repository. `subject_data_gen/patchwork/pwSubjectFull.py` takes care of the subject data and `lines_data_prep/pw_prep.py` takes care of the authors, file names, and +/- lines of the patches. Here again, the subject data is stored in a SQLite3 database, and the line data is stored in pickle files.

4.4 Providing Access to the Matches

Now that matches have been generated and saved, we need a way to make the information available to linux developers. Each match is composed of four elements: the *commit ID*, the *patchwork permalink ID*, the *date*, and the *phase* that found the match (subject, author,

[5] `https://docs.python.org/2/library/pickle.html`

or file). The patchwork permalink ID is used to point to the patch and conversation on patchwork.kernel.org.

Email2git's original intended contribution was to increase the amount of information available in Srcmap by providing access to the conversation that took place during the creation of the patch. However, since cregit was more popular than Srcmap, we decided to integrate Email2git with Cregit instead. Additionally, we made our data available through a standalone commit lookup interface.

4.4.1 Integrating Email2git with Cregit

Cregit is a project that aims at providing a finer grained approach to *git blame*. The blame option in git returns the name of the developer who last changed a line of code in the source code. It provides a way to quickly unmask the developers responsible for code in the source code. However, it has a serious limitation: git blame assigns a line to a developer even after a small modification to that line. For instance, if developer A writes `print "Hello world"`, this line will then become associated with developer A. However, if developer B modifies the line to read `print "Hello world!"`, git blame will associate the line with devloper B even though developer B only added a character.

Cregit addresses this limitation by tokenizing the source code in a git repository to enables git blame at a token level, instead of a line level. This provides a better understanding of the true authors of the source code. A tokenized version of the Linux kernel source code is available online through the cregit interface[6].

Figure 4.3 shows tokenized linux code as it appears on the Cregit interface. In an effort to ease the access to email2git data, we decided to provide access to the matches through cregit. To this end, I modified the user interface to display a window containing all the available patches after clicking on a token, as shown in Figure 4.4.

4.4.2 Standalone Commit Lookup Tool

Our second access point is a simple commit ID lookup tool. Although both platforms use the same UI and fetching mechanism to display the links to patchwork, the user experience is fundamentally different. On cregit, users navigate the interface by browsing the tokenized files as displayed in Figure 4.3. Once the user clicks on a token, we display a window containing the links to the patches and conversation that introduced that token to the source code. Note that in this case, the user does not need to know the commit ID of the token

[6] `https://cregit.linuxsources.org/`

```
static void dio_bio_end_aio(struct bio *bio)
{
	struct dio *dio = bio->bi_private;
	unsigned long remaining;
	unsigned long flags;

	/* cleanup the bio */
	dio_bio_complete(dio, bio);

	spin_lock_irqsave(&dio->bio_lock, flags);
	remaining = --dio->refcount;
	if (remaining == 1 && dio->waiter)
		wake_up_process(dio->waiter);
	spin_unlock_irqrestore(&dio->bio_lock, flags);

	if (remaining == 0) {
		if (dio->result && dio->defer_completion) {
			INIT_WORK(&dio->complete_work, dio_aio_complete_work);
			queue_work(dio->inode->i_sb->s_dio_done_wq,
				   &dio->complete_work);
		} else {
			dio_complete(dio, 0, true);
		}
	}
}
```

Contributors

Person	Tokens	Prop	Commits	CommitProp
Zach Brown	68	49.28%	4	50.00%
Christoph Hellwig	42	30.43%	2	25.00%
Andrew Morton	27	19.57%	1	12.50%
Neil Brown	1	0.72%	1	12.50%
Total	138	100.00%	8	100.00%

Figure 4.3 Tokenized source code as it appears on Cregit.

of interest. The commit ID, which is necessary to retrieve the patches, is hard-coded in the html element containing the token. The HTML element containing a token looks like this:

```
<a onclick="return
windowpopLinux('2ebda74fd6c9d3fc3b9f0234fc519795e23025a5')">
        include
</a>
```

The onclick event calls a function defined in a global javascript file: `cregit.js`. In the original implementation of the cregit interface, this function would open a new browser window and show the commit associated by the token on github. So I modified `cregit.js` to disable the "popup mechanism" and to instead use the commit ID to fetch the patchwork permalinks IDs from the server. The matches are stored as csv files named after the commit ID they are associated with on the server hosting the interface. The asyncronous request is done through

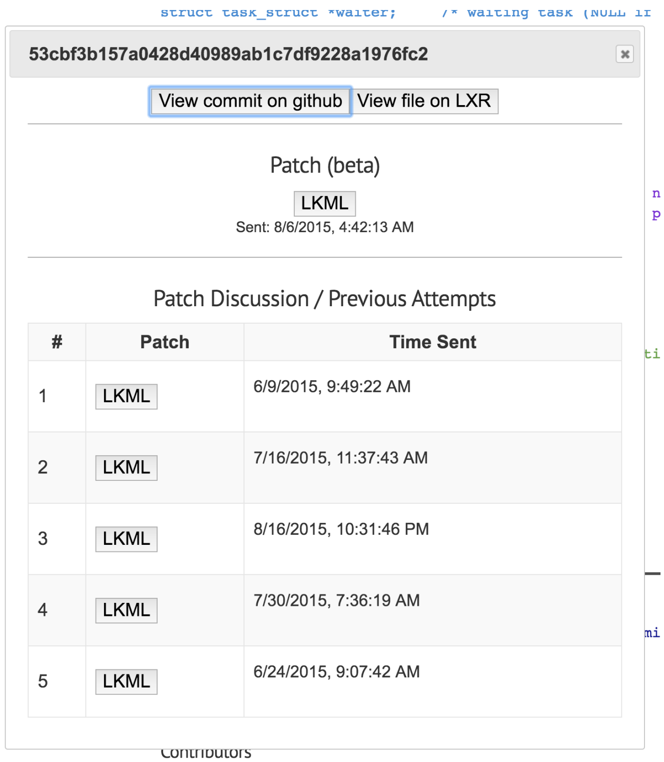

Figure 4.4 Window containing the patches that introduced the commit associated with the clicked token

Papaparse[7], a powerful open source javascript library capable of downloading and parsing csv files from the client. The javascript code that generates the URLs from the permalinks and displays the new window lives in a callback function that executes after the request is complete. We were able to keep the "view commit on github" feature, by showing a button in the new window as displayed in Figure 4.4.

On the standalone commit ID lookup page, Figure 4.5, the mechanism is almost identical, but the user experience is completely different. Instead of clicking on a token, the user knows the commit ID in advance, as they might have encountered it while trying to fix a bug, or read a `git log` output. The user copies and pastes the commit ID in the search bar, and the match window appears with a list of dated links to patchwork order by time sent. The lookup page verifies whether the commit ID is a SHA-1 hash with the following regex:

```
// removing white space
cid = cid.replace(/\s/g, '');

// validating input
```

[7] http://papaparse.com/

```
if (!/\b[0-9a-f]{40,40}\b/.test(cid)){
    window.alert("The input should be a full 40-character SHA1 hash.");
    return;
}
```

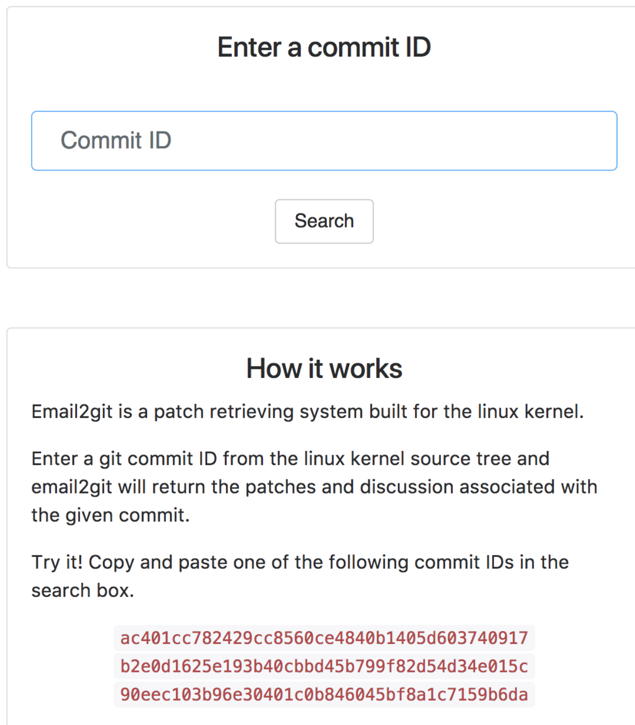

Figure 4.5 Standalone Commit ID Lookup Interface

4.5 Introducing Email2git to the Opensource Community

We undertook various efforts to make our work more visible to the linux and open source community in general. The first effort was a blog post published on linux.com[8]. This blog post discusses email2git and its integration with cregit. This blog post was shared on Facebook and Twitter by the Linux Foundation and by other developers, which helped spread the word about our work. In addition to this blogpost, we gave a refereed talk at the Open Source Summit and the Linux Plumbers Conference in Los Angeles. This gave me the opportunity to give a demo, explain the underlying algorithm and finally discuss the project with developer and receive crucial feedback. An article[9] was published on LWN.net by Jake

[8]https://www.linux.com/blog/email2git-matching-linux-code-its-mailing-list-discussions
[9]https://lwn.net/Articles/734018/

Edge following my talk. It explained the algorithm, the challenges faced, and mentioned some of the questions that were asked during the talk.

The linux developers feedback included the following points:

1. **Including patch zero.** The patch zero is the email that introduces a multi-patch series. A multi-patch series is a large patch that is split over multiple emails. The patch zero contains information regarding the entirety of the patch series. Our current source of matches, lkml.kernel.org, uses patchwork 1.0, which did not track the patch zero. The patch zero feature was implemented in Patchwork 2.0.

2. **Tracking Linux-next.** Linux-next is the branch used for integration testing. Developers would benefit from review discussions to help fix bugs that arised from integrating commits to Linux-next. In its current form, Email2git only contains matches for commits from the main tree, which means that Linux-next commits are not yet included.

3. **Give credit to reviewers.** The `reviewed-by` tag is used to give credit to the developers that contributed during a patch review. However, some reviewers often do not receive the credit they deserve after contributing in reviews. Our tool could automate or help generate the `reviewed-by` tags.

4. **Guidelines for more accurate matching.** A developer suggested the creation of a list of guidelines for developers to follow in their development process. These guidelines would promote the use of the built-in git features that use the email subject as commit-summary. If adopted by the community, those guidelines would increase the proportion of matches found.

4.6 Evaluation

To evaluate the current state of the tool, we look at the number of commits Email2git was able to match. Overall, the algorithm matched 57% of the non-merge commits since 2009. We look at the proportion of commits matched in the largest subdirectories of the Linux Kernel source code. Figure 4.6 presents these proportions. The proportion of matches found is unevenly spread across the different subdirectories of the Linux Kernel source code. For example, the `kernel` and `mm` directories have about **70%** of their commits matched. On the other hand, the `net` directory only has **25%** of their commits matched. This is mainly due to the absence of key mailing lists from our email dataset. One of these missing mailing lists is `netdev`, where many patches related to net development are sent. This is reflected in the low proportion of commits matched in the `net` subdirectory.

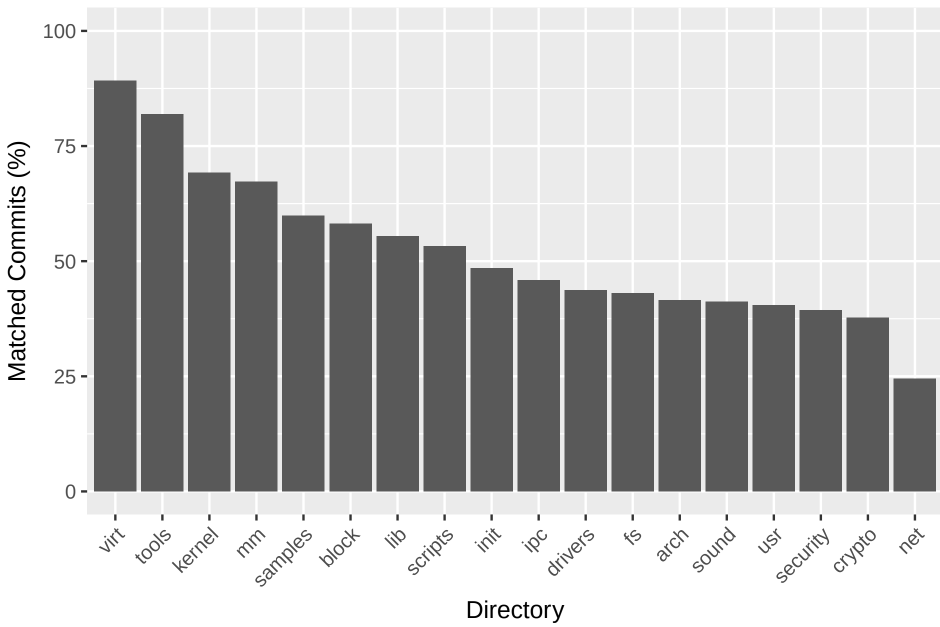

Figure 4.6 Percentage of matched commits in Linux subdirectories, from 2009 to the time of writing this thesis

Furthermore, we can break down the matches by the stage of the algorithm that generated them. The first phase of the algorithm, which uses the email subject, finds about **95%** of the final set of matches. The second part of the algorithm, which uses the author and filename based matching, finds **5%** of the final matches. It is important to note that the second phase of the algorithm is exposed to less commits than the first phase. The commits matched by the first phase are removed from the dataset before starting the second phase. This implies that the second phase will inherently find less matches than the first phase. We tested the performance of the line-based algorithm (both author and filename) on a subset of commit and patches to understand its performance. The algorithm found a *correct* match for *26%* of the analyzed commits.

Finally, we are running an analytics script on the server hosting the matches to understand the usage of the data and to know the proportion of requested commit IDs are not matched. Removing the whitespace and assuring the validity of the string ensures the accuracy of these statistics, by reducing the number of failed requests due to a poorly formatted commit ID.

Figure 4.7 displays the plots created by the analytics script running on the server. We observe two peaks in number of unique IP addresses at two different moments: at the end of July and in mid September. The former date corresponds to the day we introduced Email2git in a blogpost on linux.com and the latter corresponds to the talk we gave at the Open Source

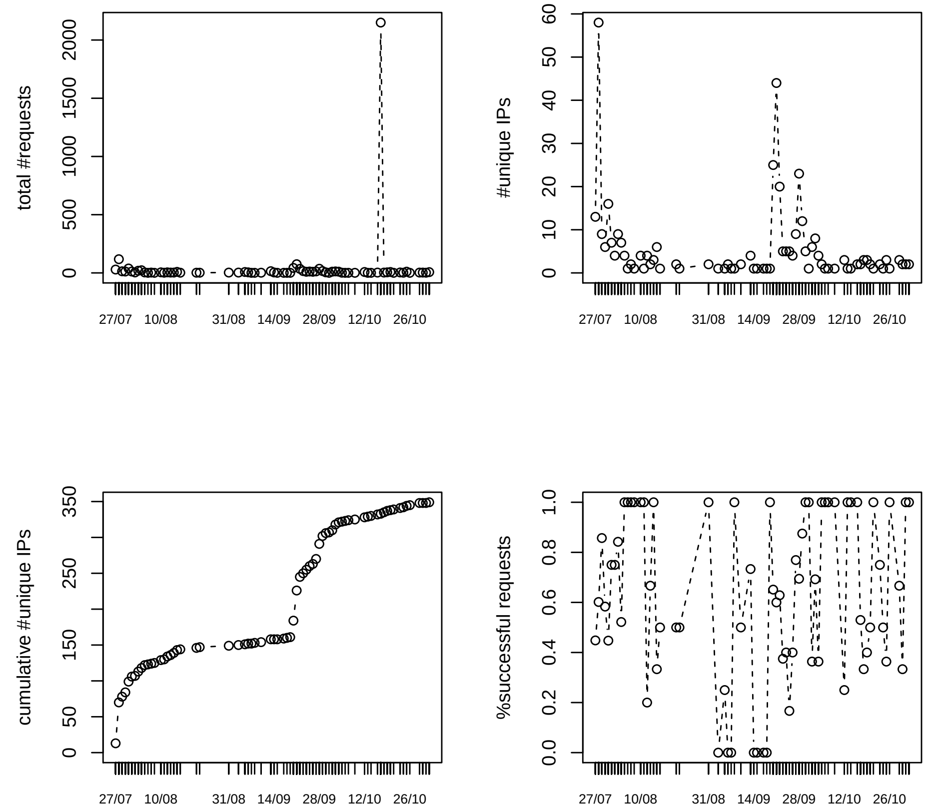

Figure 4.7 Plots created from the analytic

Summit North America in Los Angeles.

4.7 Conclusion

We leveraged the lessons learned from Srcmap while creating Email2git. The purpose of Email2git was to answer a complain coming from multiple developers and maintainers: the difficulty of finding an email discussion about patches that were eventually integrated in the Linux Kernel. Understanding the Linux community allowed us to better introduce our tool when we released it, as explained in Chapter 4. Moreover, the data acquired in the implementation of Email2git served as one of the metrics used by our expertise model as explained in chapter 5.

CHAPTER 5 ARTICLE 1: ON HISTORY-AWARE MULTI-ACTIVITY EXPERTISE MODELS

5.1 Abstract

As software evolves, a maintainer's contributions will gradually vanish as they are being replaced by other developers' code, resulting in a slow erosion of the maintainer's footprint in the software project. Even though this maintainer's knowledge of the file did not disappear overnight, to outsiders, the maintainer and her expertise have become invisible. Through an empirical study on 5 years of Linux development history, this paper analyses this phenomenon of expertise erosion by building a 2-dimensional model of maintainer expertise involving a range of maintainer activity data on more than one release. Using these models, we found that although many Linux maintainers' own coding footprint has regressed over time, their expertise is perpetuated through involvement in other development activities such as patch reviews and committing upstream on behalf of other developers. Considering such activities over time further improves recommendation models.

5.2 Introduction

As reported by Damien et al. Joseph et al. (2007), employee turnover is a major challenge of information technology organizations. Estimations of the cost of losing an employee amount to between 0.5 and 1.5 times her salary, with the cost of replacing a software engineer in particular exceeding $100,000 [1]. These costs are not limited to the software engineer's company, but also spread to open source development. In their 2017 Linux kernel report, Corbet et al. Corbet and Kroah-Hartman (2017) noted that "well over 85 percent of all kernel development is demonstrably done by developers who are being paid for their work". In fact, only 8.2% of all kernel commits were made by volunteer developers. Hence, developer turn-over in companies risks to impact open source development as well!

Apart from improving the working conditions and onboarding procedures, software organizations (both closed and open source) need to invest time in finding the "right" expert to replace a parting maintainer. While it is possible to train newcomers and bring them up to speed (e.g., one third of the kernel contributors in the last 12 months were newcomers, and 60% of those made their contribution as employee), the term "right" refers to having a similar profile, allowing the new maintainer to seamlessly fit in and continue his or her pre-

[1] https://www.economist.com/node/5988

decessor's work, without significant loss of knowledge. Thanks to the widespread adoption of agile development and open source development models, software development has become a collaborative endeavor, in which knowledge is shared across the members of an organization, hence in principle it should be possible to find contributors with similar profiles.

Unfortunately, there is no consensus on how to measure the profile of a developer, and how to determine whether such a profile indicates the developer to be an maintainer. The simplest way to measure someone's development activities is to count the number of code changes (e.g., Git commits) authored. This is for example how Corbet et al. determine the most active developers and organizations in Linux kernel development Corbet and Kroah-Hartman (2017). Yet, at the same time, they note that "The total number of patches signed off by Linus Torvalds (207, or 0.3 percent of the total) continues its long-term decline. That reflects the increasing amount of delegation to subsystem maintainers who do the bulk of the patch review and merging." Worse, developer rankings based on the number of commits differed substantially based on the period during which this metric was measured (e.g., ranking based on last 10 years vs. last year). At a minimum, one needs to be careful interpreting these and other measures such as a developer's code legacy as shown by "git blame" Bhattacharya et al. (2014); Mockus and Herbsleb (2002); McDonald and Ackerman (2000); Fritz et al. (2007). To make developer expertise measures more robust and reliable, this paper proposes a 2-dimensional contribution footprint model, addressing two important issues with current expertise models. First of all, while the amount of code written by a person can be an important indicator of expertise, it does not take into consideration the actions of people who do not directly contribute source code, such as those who review it or discuss it on mailing lists or issue repositories. Our footprint measure combines indicators of multiple kinds of developer activities.

Second, as indicated by Corbet et al., current measures focus on a given software release or development period, basically ignoring the development activities that happened before. While a person who wrote 50% of the code changes of the previous release could be less of an expert than a person who wrote 50% of the code changes of the current release, the former developer might have been ill or absent, or might have been the one mentoring the latter developer. As such, both developers should be considered as experts, not just the latter developer. Our footprint measure allows to consider a developer's activities across different time periods.

We empirically evaluate the footprint model's ability to detect maintainers on 5 years of Linux kernel development history, addressing the following research questions:

RQ1) *How does the role of maintainer evolve in an open source project?*

Almost 1 out of 4 subsystems has seen a change in maintainership during the last 22 Linux kernel releases, with the code footprint of maintainers gradually decreasing over time.

RQ2) *How well does the non-history aware dimension describe maintainers?*

Models involving a maintainers' own code footprint and coordination activities (committing and/or reviewing) perform the best.

RQ3) *How well does the history aware dimension describe maintainers?*

Models considering the last R releases perform better than single-release models.

5.3 Background and Related Work

Maintainers ensure the longevity of a large open source project like the Linux kernel by not only contributing new code, but also by reviewing and integrating code submitted to their subsystems by other developers. Despite their crucial role in open source development, few studies have been done on maintainers Zhou et al. (2017). In particular, a maintainer's departure of her subsystem calls for her immediate replacement. However, only a developer with extensive experience in the subsystem can take on the task of maintainer. Hence, in this section, we aim to understand the different expert detection techniques.

Software expertise and knowledge have been extensively studied in the past Bhattacharya et al. (2014); Mockus and Herbsleb (2002); McDonald and Ackerman (2000); Fritz et al. (2007). Most of this work provides a recommendation system to detect experts among the population of developers. Typically, existing recommenders base their assessment of expertise on **non-historical models**, meaning that they only exploit data of the current snapshot of the software repository, not of earlier versions. These non-historical models use a variety of different metrics to determine expertise.

Implementation expertise. This type of metric implies that a developer gains expertise through implementation, in other words, by making changes to a file, as tracked by a version control system. McDonald McDonald and Ackerman (2000) recommends the last developer who made a change to a file as expert for that file, which could be interpreted as file-level `git blame`. Later, Mockus et al. Mockus and Herbsleb (2002) improved on the approach proposed by McDonald McDonald and Ackerman (2000) by counting the number of changes each developer brought to a file to provide a better expertise recommendation. Girba et al. Girba et al. (2005) later improved Mockus et al.'s method Mockus and Herbsleb (2002) by measuring the size (churn) of each change in terms of lines of code.

Usage Expertise. Developers gain usage expertise by calling ("using") specific methods from within their code. This concept of usage expertise was introduced by Schuler and Zimmermann Schuler and Zimmermann (2008). In later work Ma et al. (2009), the authors compare the accuracy of usage expertise against implementation expertise recommenders. They concluded that usage expertise recommends experts with similar accuracy to implementation expertise models.

Additionally, Fritz et al. Fritz et al. (2007) validate the accuracy of implementation expertise techniques. After a qualitative study consisting of 19 java developers interviews, they were able to confirm a relationship between changes made (commit frequency) and expertise. In addition to that, they found evidence proving that authorship (as obtained from the amount of churn contributed, or through "git blame") is also capable of indicating expertise. In a later study, the authors Fritz et al. (2010) create a degree of knowledge model combining both usage and implementation metrics to recommend experts.

Bhattacharya et al. Bhattacharya et al. (2014) explored the suitability of different implementation expertise metrics depending on a developer's role. They argue that state-of-the-art metrics (lines of code and commits added), being unaware of the developer's role, can lead to inaccurate results. They add that code activity metrics like the number of lines of code added, only describe expertise at a local level and poorly capture global expertise.

Thus far, all models cited above are not **history aware**. Hence, they do not take into consideration the effect of time on developer expertise and assume that developers' memory lasts for ever. Based on a survey of psychological literature, Hattori et al. Hattori et al. (2012) create a memory retention model to improve expertise models. Memory retention is computed using a Forgetting function, which reduces the weight of older activities to account for memory loss. The data used in the experiment was acquired by a tool that records information from the developer's IDE. It would be impossible to reproduce this experiment on a project of the scale of the Linux Kernel because it would be impossible to force all Linux developers to use the IDE integrated tool.

Although these state-of-the-art techniques are well suited to detect experts among regular developers, who specialize in implementation, they ignore most of the daily tasks of maintainers, such as reviewing and committing code upstream, creating an inherent bias in the expertise models. We address this bias by building expertise models based on a variety of metrics capturing the full breadth of software development activities, and also considering the evolution of such activities over time.

5.4 Measuring Developers' Contribution Footprint

This section discusses the two-dimensional contribution footprint model proposed by this paper to enable identification of experienced team members (e.g., developers, testers, etc.) as candidate maintainers. The first dimension of the footprint model considers a wide range of activities performed by a project member, not only focusing on code changes, but also code review or even a developer's code "legacy" (i.e., contributed code that still survives in the code base). The second dimension enhances the first dimension by not only considering the range of activities in the latest release, but *across the last N releases*. As such, accidental lulls or shifts in project activity are accounted for.

Note that the expertise we are interested in is expertise about the *internals* of a particular source code file or component, or implementation expertise. An alternative form of expertise would consider knowledge on how to *use* a particular component (API), or usage expertise. We focus on the former kind of expertise, since it is at the heart of a software organizations needs. For example, it allows to measure the expertise of a particular individual, allowing the organization to better use her skills, evaluate her value to the organization, and assess the risk of her potential departure. Furthermore, it is important for an organization to know—for any section of the system—who are its maintainers, and their level of expertise. Finally, in both cases, it is also important to know how the role of maintainer is changing over time (e.g., the activities where a maintainer is gaining and losing expertise).

5.4.1 Dimension 1: Contributor Activities

The footprint model that we propose explicitly considers a wide range of development activities instead of focusing only on review- or code-related activities. Table 5.1 provides a non-exhaustive list of activities, from very technical to outreach activities. Any activity by a contributor to one of these, can increase (or at least maintain) the contributor's knowledge about the subsystem she is working in. The more measures are considered, the more comprehensive the footprint ranking model becomes, hence the better the expected performance for identification of maintainers in a project under study, provided the activities are weighted based on their relevance for a given project.

This flexibility comes at the expense of additional effort for mining these activity measures. Fortunately, when developers contribute code to an open or closed source project, data about each code change, code review or other activity is automatically stored in the project's software repositories. The most trivial example are the code changes (commits) recorded in a version control system like git. However, information about the contributor's activity

Table 5.1 Non-exhaustive list of activity measures that can be measured for a particular contributor C of a specific subsystem S in a given release R_i.

activity	definition
`legacy`	influential code contributed by C that still survives in R_i
`authored`	code authored by C since R_{i-1}
`committed`	code committed by C since R_{i-1}
`reviewed`	code changes reviewed by C since R_{i-1}
`translated`	involvement in translating/localizing textual strings for R_i
`integrated`	effort spent by C integrating code changes since R_{i-1}
`discussed`	effort spent by C discussing issue reports since R_{i-1}
`represented`	effort spent by C representing S on social media since R_{i-1}
`planned`	effort spent by C planning R_i

in issue report discussions is also readily available from the project's issue repository (e.g., bugzilla or jira), code review activity from the review repository (e.g., gerrit or mailing list) and mailing list activity from the mailing list archive. Of the metrics in Table 5.1, `represented` and `planned` are the hardest to obtain data for.

Given a set of activity measures $\mathbb{A} = \{a_i | a_i \textit{ is activity measure}\}$, we compute the contribution footprint of a release j as:

$$footprint_j(\mathbb{A}) = \sum_i \frac{w_i \times a_i}{a_i^{tot}}$$

, where w_i is a weight factor given to a_i ($\sum_i w_i = 1$) and a_i^{tot} is the total number of activities of a given type (e.g., number of source code lines, commits or reviews) recorded for a given activity and release. In other words, each activity is normalized, and the weighted sum of the normalized activities yields the $footprint_j(\mathbb{A})$ percentage. Hence, to instantiate the generic $footprint_j(\mathbb{A})$ measure, an organization first has to select the activity measures $\mathbb{A}$ relevant to its context, then determine the relative weight w_i of each selected activity.

It is necessary to normalize each activity's measure to provide a better understanding of the true impact of developers' contributions in the subsystem. This is because the studied subsystems differ in size and the heuristic counts are inherently uneven by nature. For example, the value for `legacy` (in number of lines of code) will likely be much larger than the values of `authored` or `reviewed` (in number of commits).

5.4.2 Dimension 2: Historical Perspective

While the definition of $footprint_j(\mathbb{A})$ takes into account a wide range of activities, it only considers a contributor's activity for one specific release j. As such, this measure might still provide misleading information when used to find the most appropriate expert for a given subsystem (e.g., to help debug a coding issue).

First of all, contributors in both closed and open source development evolve according to a particular career path. Even in open source, many contributors start out translating textual strings, before contributing smaller code fixes and ever larger changes until they are trusted enough to be able to review or even accept other contributors' code. This not only implies that a contributor's volume of contributions is scattered across different activities, but also that this scattering (and volume) might change over time. Hence, depending on the release under study, different $footprint_j(\mathbb{A})$ values are obtained, as if a specific contributor suddenly would have "lost" or "gained" a substantial percentage of expertise (footprint). To counter this noise, one should incorporate past experience to obtain a more robust footprint model.

Second, even when a contributor's responsibilities are stable across a time period, accidental life events such as illness or busy periods at work, or project events such as the scope of the upcoming release (major release vs. bug fix release) could lead to increases or decreases for certain activities. Again, if the contributor was an expert in the previous release, she will not have lost all of this expertise in one release due to illness. Hence, a release-specific $footprint_j(\mathbb{A})$ measure again would yield the wrong impression.

For this reason, the second dimension of our footprint model explicitly takes into account history by taking the weighted sum of $footprint_j(\mathbb{A})$ over the last R releases. In particular:

$$footprint_j^R(\mathbb{A}) = \sum_{i=j}^{j-R} W_i \times footprint_i(\mathbb{A})$$

, where W_i is a weight factor given to the specific footprint of release i ($\sum_i W_i = 1$). Note that $footprint_j(\mathbb{A}) = footprint_j^0(\mathbb{A})$, i.e., the footprint model obtained based on the first dimension is a special case of the second dimension ($R = 0$).

While the choice of weights w_i for dimension 1 could be chosen arbitrarily based on relevance of individual activities, the weights W_i typically will be decreasing, since recent activity typically is at least as important as older activity. For example, the weights could be linearly decaying (e.g., $[0.33, 0.27, 0.20, 0.13, 0.07]$ for $R = 4$), giving each older release proportionally less influence on the footprint model. Alternatively, an exponential ($[0.64, 0.23, 0.09, 0.03, 0.01]$) or logarithmic ($[0.34, 0.29, 0.23, 0.14, 0.00]$) decay could be used to give older release less or more influence, or (less likely) even a uniform ($[0.20, 0.20, 0.20, 0.20, 0.20]$) decay to give all considered releases the same importance.

5.4.3 Use Cases for Contribution Footprint Models

Given the footprint models $footprint_j(\mathbb{A})$ and $footprint_j^R(\mathbb{A})$, a number of use cases can be imagined.

The main use case considered in this paper is the identification of maintainers in a software project. When the maintainer of a specific component or library decides to retire, finding a good replacement is not always straightforward for an organization, as important development knowledge (across a range of development activities) risks to be lost.

A less straightforward application was suggested at one of the 2017 OPNFV Summit's panels, where substantial attention went to the issue of non-responsive Linux kernel maintainers. These are experts responsible for a given subsystem who, due to personal events, loss of interest or other reasons, start becoming non-responsive in communication with other developers or management. Having a reliable expertise measure in place would enable monitoring over time of maintainers' activities to spot long-term periods with sub-par performance. Such pro-active detection of issues could also suggest alternative maintainers.

Similarly, a contribution footprint model can help an organization guard itself against accidental loss of manpower. For example, the bus factor Mens et al. (2014) is a known measure of the risk that key personnel might disappear from a project, either out of free will or due to an accident. Organizations with a high bus factor could leverage a contribution footprint to identify backups for key developers, maintainers, or managers. As such, for each subsystem, an organization could have a list of the main people working in it as well as their expertise level.

In order to use the footprint models to find the most appropriate maintainer candidate of a given subsystem, one needs to calculate $footprint_j(\mathbb{A})$ and/or $footprint_j^R(\mathbb{A})$ for each person who contributed at least once to one of the activities in $\mathbb{A}$. Then, the resulting footprint values should be ranked from high to low. Ideally, the contributor with the highest footprint value is recommended as first candidate maintainer, followed by the contributor with the next highest footprint value, etc.

5.5 Case Study Setup

This section presents the design of an empirical study on the Linux kernel to evaluate the 2-dimensional footprint ranking model introduced in the previous section. The study addresses the following research questions:

5.5.1 Subject Data

Our study evaluates the footprint models in the context of the Linux kernel. First of all, the Linux kernel is one of the hallmark open source projects, with a long history, large code base and vast supply of contributors. Second, the kernel is one of the few open source projects in which maintainers are documented explicitly. The code base contains a file named `MAINTAINERS` that lists, for each subsystem, the experts in charge. Just as for source code, changes in maintainership are recorded through regular commits. This provides us with a unique oracle for our footprint ranking based recommender.

Furthermore, the Linux Foundation (who governs and mentors the development of the Linux kernel and related open source initiatives) recently has started up the CHAOSS committee on Community Health Analytics for Open Source Software[2]. Amongst others, the aim of this committee is to identify explicit measures of expertise that can help prospective adopters of open source projects in choosing the right maintainer to contact. As such, our study can help this concrete initiative, and we are in contact with the CHAOSS consortium.

Determining the footprint rankings in the Linux kernel, especially taking into account the second dimension of our measure, requires a large set of historical data. We conducted our analysis on a set of 27 releases of the Linux kernel, spanning releases *v3.5* to *v4.11*, which corresponds to approximately 5 years of development and release history.

5.5.2 Filtering of the Data

Because of the constantly changing nature of the kernel, new subsystems are being added to the Linux kernel in every release to meet the demands associated with new hardware and changes in user expectations. Furthermore, it is not uncommon to see a subsystem disappearing, or, more precisely, becoming obsolete or orphaned.

On the other hand, the importance of the historical aspect of our analysis forces us to choose long-standing subsystems that would best reflect the evolution of expertise of the subsystems' maintainers. Hence, we filter our Linux kernel data set to keep only subsystems that existed

[2]`https://chaoss.community/`

throughout the studied timespan. This subset reduces the number of subsystems from 1,662 to 734 subsystems.

For RQ2 and RQ3, we need further filtering to ensure a data set of subsystems for which there is ongoing activity in each studied release. To achieve this, we parsed, for each release, the `MAINTAINERS` file to extract each *active* subsystem along with its name, list of maintainer names, and the list of files and directories belonging to that subsystem.

We then retrieved the list of commits made to each subsystem, for each release that we considered. This allows us to compute, for each subsystem, the average number of commits across its releases. After matching each commit to its code reviews (see below), we also compute the average proportion of matched commits per release.

We then set minimum thresholds of 50 commits per release and 60% matched commits per release. This filtering reduces the 734 subsystems to a set of 78 subsystems for RQ2 and RQ3. This subset contains well know subsystems like `ARM PORT`, `XEN HYPERVISOR INTERFACE`, `SOUND`, `SCHEDULER`, and `CRYPTO API`.

5.5.3 Instantiation of the Footprint Rankings

Table 5.2 shows the five concrete activity measures considered in our empirical study on the Linux kernel. These measures cover influential source code contributed (`legacy`), the volume of code changes since the last release (`authored` and `committed`), and code review activities since the last release (`attributed` and `reviewed`).

Our $footprint_j(\mathbb{A})$ and $footprint_j^R(\mathbb{A})$ models are calculated based on the above metrics, using $w_i = 0.20$ as weights for dimension 1 and a linear decay with $R = 4$ (i.e., $W_i \in [0.33, 0.27, 0.20, 0.13, 0.07]$) for dimension 2. A more judicious choice of w_i and/or W_i could improve the results in RQ2 and RQ3, however our empirical study aims to provide a lower bound on the expected performance.

5.5.4 Git-related Activity Measures

To calculate the Git-related activity measures of Table 5.2, i.e., all measures excluding `reviewed`, we cloned the official Linux kernel git repository, then checked out the Git tag corresponding to each analyzed release.

Bread-and-butter analysis of the Git log commits in the time span since the previous official release yields `authored` and `committed`, while simple regular expressions of the commit messages in the same logs obtains `attributed`. Finally, a standard git blame command

Table 5.2 Concrete activity measures used for our empirical study on the Linux kernel. Each activity is measured for a particular contributor C of a specific subsystem S in a given release R_i.

activity	source	definition
`legacy`	git blame	#lines of code contributed by C that still survive in R_i
`authored`	git log	#commits authored by C since R_{i-1}
`committed`	git log	#commits committed by C since R_{i-1}
`attributed`	git log	#commits since R_{i-1} for which C is credited in the commit message under "Signed-off-by", "Reviewed-by" or "Acked-by".
`reviewed`	mailing list	#commits since R_{i-1} for which C has written at least one code review email

yields, for each code line in the release under analysis, the last person touching it. This information allows to calculate `legacy`.

In kernel development, tags like "Signed-off-by", "Reviewed-by" and "Acked-by" are used as "a loose indication of review, so [...] need to be regarded as approximations only" Corbet and Kroah-Hartman (2017). Despite the warning of Corbet et al., `attributed` information is straightforward to obtain from commit messages, which is why we included this measure to complement the more strictly defined `reviewed` measure (calculated from reviewing data).

The next step is to lift up each contributor's Git-related activity measures to the subsystem-level, leveraging the file path information for each subsystem in the `MAINTAINERS` file. To do this, we identify for each commit the changed files, then map the commit to the subsystem(s) to which these files belong and aggregate the file-level measures to the subsystem level, for each contributor.

5.5.5 Linking Commits to Review Emails

In contrast to the developer `attributed` data obtained from the Git repository, the `reviewed` metric considers a second repository, i.e., the review environment. For the Linux kernel, code reviews are performed through mailing list discussions Armstrong et al. (2017b); Jiang et al. (2013, 2014). Patches are sent to one or more of the various linux mailing lists (typically one per kernel subsystem), where anyone can step up and provide review comments simply by replying to the patch email. As such, the review comments of a specific patch are spread across one or more email threads.

Jiang et al. Jiang et al. (2013, 2014) have introduced a number of heuristics to link an accepted patch stored as a commit in the official Git repository to the (different versions of the) reviewed patch in the mailing list. We adopted the best performing heuristic of Jiang et al., which uses simple set intersection of the changed code lines between each email patch and Git commit. The heuristic matches a given Git commit C to the email patch P with which the change code line intersection is the largest and exceeds a given threshold of 4%. All emails in P's email thread are said to correspond to the review comments on P (and hence C).

To improve the line-based heuristic of Jiang et al., we have combined it with other heuristics. First of all, we observed that more and more kernel developers are using the commit message summary as the subject of their email threads. This summary is recommended to be between 50 and 72 characters[3] and appears before the body of a commit message. Hence, before applying the line-based matching of Jiang et al., we first check if there is a unique email patch P with subject identical to a commit C's commit summary. If so, we consider P and C to be a match, and do not need to run the more complex line-based matching algorithm.

If there is no such P, or multiple patches P have been found, we extract for each remaining commit the *author* and the *changed files*. We do the same for each remaining email patch. This information is then used to narrow down the search space of the line-based matching, by trying to match a commit only to email patches authored by the same developer and/or touching the same files. This substantially speeds up the matching process.

The remaining commits, i.e., the commits still not matched to a review, introduce noise to our measures. The reason for not finding a code review could be due to the reviews being sent to a mailing list that we did not analyze, or not being reviewed at all. We were granted read access to the database behind the Patchwork mailing list archive hosted by linux.org[4]. This Patchwork instance has been tracking 69 different Linux mailing lists since 2009, providing us with about 1.4 million patches. However, patches that were submitted through untracked mailing lists are not in our dataset, which explains the variability of matched commits across subsystems. Alternatively, the code change in the accepted commit could also have undergone substantial changes compared to the reviewed commit, for example due to rebasing, cherry-picking or squashing Bird et al. (2009).

Figure 5.1 shows the total percentage of matched commits from 2009 to the time of writing this paper, across the largest subdirectories of the kernel. It shows that this percentage varies greatly among the different subdirectories, with a minimum of 25% for the "net" subdirectory. This is why, in subsection 5.5.2, we filter out those subsystems whose average percentage of

[3] https://medium.com/@preslavrachev/what-s-with-the-50-72-rule-8a906f61f09c
[4] https://patchwork.kernel.org/

matched commits across the studied releases is lower than 60%. Note that we do not show the percentage of unmatched *patches* (only unmatched *commits*), since the unmatched email patches include those patches that were rejected during code review, and hence never showed up in the Git repository.

5.6 Case Study Results

This section discusses for each research question its motivation, specific approach and results.

RQ1. How does the role of maintainer evolve in an open source project?

Motivation:

Open source software maintainers are responsible for the health of their subsystem. For example, each Linux kernel maintainer manages the changes proposed by developers to the subsystem they are responsible for, and shepherd those changes upstream towards the Git repository of Linus Torvalds (i.e., the official Linux kernel repository). Hence, their presence is vital to the kernel community.

Unfortunately, due to the unpredictable nature of life in general and open source software development in particular Wu et al. (2007b); Zhou and Mockus (2015), maintainers, for various reasons, one day will be forced to give up their responsibilities. In most cases, this means that another developer will have to take over the responsibility of maintainer.

Hence, this research question aims to analyze how often maintainership changes in kernel development. Furthermore, we are interested in understanding how much of the code base of official releases is “owned” by the subsystem maintainers, i.e., was originally developed by a maintainer. Since “git blame” is a popular means for finding experts Rahman and Devanbu (2011), our results will help us understand to what extent such a measure is reliable to measure expertise.

Approach:

To confirm the presence of changes in maintainership during the evolution of the Linux kernel, we analyzed the maintainers recorded in the `MAINTAINERS` file of releases *v3.5* to *v4.11* to identify how often maintainers (dis)appeared. Furthermore, for each studied release, we measure and plot these maintainers’ `legacy`, which corresponds to the number of surviving code lines of a maintainer, as given by “git blame”. We then validated the statistical significance of the change in `legacy` distribution between the first and last analyzed release using a Wilcoxon paired test. In case of a significant test result, we also provide the Cliff Delta

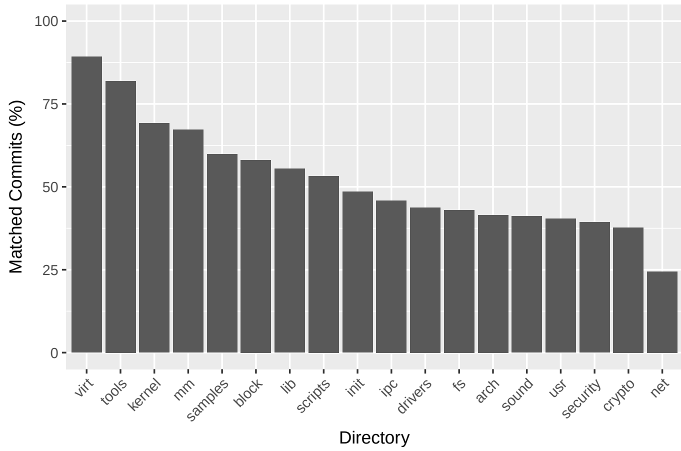

Figure 5.1 Percentage of matched commits in Linux subdirectories, from 2009 to the time of writing this paper.

effect size Romano et al. (2006). An effect size smaller than 0.147 is deemed a “negligible” difference, smaller than 0.33 a “small” difference, smaller than 0.474 a “medium” difference and otherwise a “large” difference.

Results:

23% of the studied subsystems saw changes in maintainership over the last 5 years. Out of the 734 subsystems studied for RQ1, we counted 168 subsystems that experienced some sort of maintainership change. We counted 100 maintainer arrivals, 63 departures, and 88 replacements. These numbers confirm that maintainership change is common, even in mature open source systems like the Linux kernel. Furthermore, the median percentage of developers who are maintainers in the analyzed subsystems is 0.50% (mean of 0.90%), indicating that it is not straightforward to guess the next maintainer. These observations strengthen our case for more advanced expertise measures.

The median maintainer `legacy` significantly decreases over time. Figure 5.2 shows the evolution of the median percentage of maintainer `legacy` across all subsystems in each studied release. The plot shows a clear, steady decrease of this measure across releases in terms of median and variance. We confirm the significance of this decrease with a Wilcoxon paired test ($\alpha = 0.01$) between the first and last studied version, which yields a p-value of 2.2e-16.

Although the Cliff Delta value of 0.07 indicates only a negligible difference, this decreasing trend suggests that, if one limits the measure of expertise to the amount of surviving code originally authored by a maintainer, as was done by earlier work Rahman and Devanbu

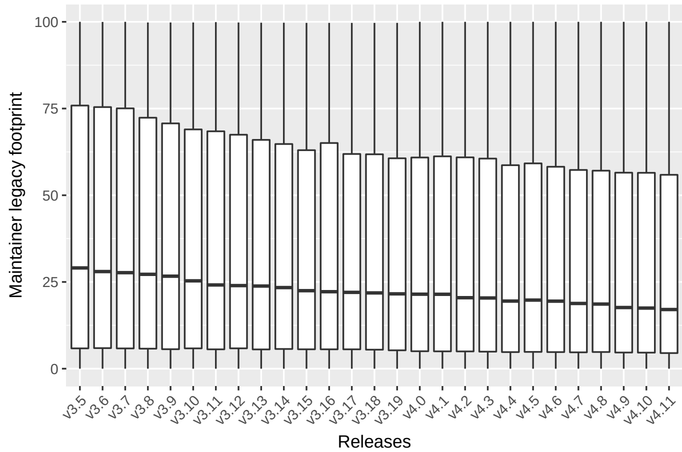

Figure 5.2 Median maintainer `legacy` across releases.

(2011), the expertise of maintainers globally seems to be decreasing over time.

RQ2. How well does the non-history aware dimension describe maintainers?

Motivation:

Prior work on expertise measures Anvik et al. (2006); Bhattacharya et al. (2014); McDonald and Ackerman (2000); Minto and Murphy (2007); Mockus and Herbsleb (2002) primarily are based on code activity, which can be defined in terms of `legacy` and `committed`. As motivated in subsection 5.4.3, we believe that these two metrics do not capture the full breadth of maintainer activities. Indeed, the results in RQ1 indicate that the `legacy` of long-standing maintainers crumbles over time. Unless one assumes that this reflects a real drop in expertise over time, the only explanation is that existing experts reorient their focus to other activities, such as code review and email communication. Hence, this research question evaluates whether considering such additional activities is able to improve the identification of experts.

Approach:

To validate the ability of the measures in Table 5.2 to explain expertise, we evaluate how well the $footprint_j(\mathbb{A})$ measure involving those measures is able to identify the maintainers of Linux kernel subsystems. Those maintainers are the experts listed in the `MAINTAINERS` file of a kernel release.

For a given release and subsystem, we should find the maintainers in the top positions when ranked based on footprint values. The combination of activity measures $\mathbb{A}$ that is able to systematically yield the correct maintainers across subsystems and releases could be assumed to be a better indication of expertise.

In particular, we calculate two performance metrics:

POS_N Percentage Of Subsystems for which at least one maintainer was ranked in the top N recommended candidates

POM_N Percentage Of Maintainers in the *whole* project who were ranked in the top N recommended candidates for their subsystem

For these performance metrics, N is a threshold that can be varied. Our case study uses thresholds ranging from 1 to 5. It is important to note that, if the number of maintainers of a subsystem is larger than N, POM_N could be penalized. To avoid this, we slightly changed the definition of POM_N to be calculated only for the maintainers of all subsystems with at most N maintainers, instead of for all maintainers of the whole project. For example, POM_1 measures the percentage of top-recommended maintainers of subsystems with at most 1 maintainer.

To structure our analysis, we analyzed the performance of maintainer recommender involving only one metric of Table 5.2, then analyzed models involving all combinations of `legacy` with one of the other 4 measures, and finally one model with all measures combined.

Since, for a given release, there is one POS_N value and one POM_N value, we calculate these metrics for each release, then study their distribution across the analyzed releases using boxplots. Figure 5.3 and Figure 5.4 show for each analyzed $\mathbb{A}$ the distributions of POS_N for N=1 and N=3, respectively, across the 22 studied Linux releases, while Figure 5.5 and Figure 5.6 the distributions of POM_N for N=1 and N=3, respectively. We only show the plots for N=1 and N=3, as for higher values of N the plots remain more or less stable.

Results:

`attributed` is the only single-measure model able to keep up with the multi-measure models.

The results in Figure 5.3 and Figure 5.5 indicate that the first two individual measures, i.e., `legacy` and `authored`, are bad indicators of expertise compared to the other studied metrics. For example, in Figure 5.3, `legacy` only reaches a median POS_N value of 47.22%, while `attributed` reaches a median POS_N percentage of 58.4%.

The models combining `legacy` with `committed`, `attributed` and/or `reviewed` perform the best. Figure 5.3 shows indeed how only these four models read median percentages of 69.9%, while the other multi-measure models, especially the one involving only `legacy` and `authored`, are not able to outperform the best individual measure models.

Type	Measure	N = 1	N = 3
POS_N	P-value	4.27e-05	4.28e-05
POS_N	Cliff's delta	0.99	1.0
POM_N	P-value	4.27e-05	4.77e-07
POM_N	Cliff's delta	0.84	1.0

Table 5.3 P-values and Cliff's delta values for the Wilcoxon paired tests ($\alpha = 0.01$) between `attributed` and `legacy + committed` for ranking thresholds N=1 and N=3 and for POS_N and POM_N.

These findings confirm the intuition that maintainers shifted focus from doing development (`authored`) themselves to mentoring others by controlling access to their subsystem's Git repository through committing and/or reviewing. As such, an expertise model only involving their own development (i.e., `legacy` and `authored`) is unable to explain the current kernel maintainers' expertise. In other words, modern expertise models should take into account the time spent reviewing code and pushing changes upstream.

POS_N increases to a median of 87.5% for larger N, with multi-measure models outperforming single-measure models by at least 17%. Comparing Figure 5.4 to Figure 5.3 shows how the top multi-measure models for N=1 are able to increase their distance compared to even the best single-measure models (`attributed`). This, compared with a change in best performing single-measure models, indicates that a larger diversity in activity measures enables better identification of the two additional candidate maintainers. Indeed, by considering top performing contributors across a wider range of activities, there is a larger chance at least one real maintainer is found. Although the percentages of Figure 5.5 and Figure 5.6 cannot be compared directly to each other (cf. modified definition of POM_N), Figure 5.6 (for POM_N) shows a similar ranking of models as Figure 5.3 (for POS_N), confirming the findings for POS_N.

Table 5.3 shows the p-value and effect size of the Wilcoxon test between `attributed` and `legacy + committed` for figures 5.3, 5.4, 5.5, and 5.6. Each effect size being close to 1, we notice a **large** performance increase between `attributed` and `legacy + committed`.

Interesting to note is that, across all analyzed releases, the boxplots show a remarkable small variance, especially for N=3. Although this is partially due to the fact that less than 25% of the subsystems saw at least one maintainer change, it also indicates that our measures are stable across changes in the 5 activity measures used.

RQ3. How well does the history aware dimension describe maintainers?

Motivation:

The metrics analyzed in RQ2 reveal that traditional expertise metrics McDonald and Ackerman (2000); Girba et al. (2005); Mockus and Herbsleb (2002) based solely on a contributor's own development productivity are not well suited to identify maintainers. Expertise models exploiting only the information available for the release under study, are able to obtain median POS_N performance of up to 75% (N=1) and 90% (N=3).

However, we believe that adding a historical dimension considering also the activity in the last R releases would assist the model in two ways. On the one hand, long standing kernel developers' contributions should carry more weight than newcomers' contributions. On the other hand, analyzing data on multiple releases would control for cases where contributors' productivity was lower due to a variety of reasons, such as illness, vacation or work on other projects.

Approach:

For each studied kernel release, we calculated $footprint_j^R(\mathbb{A})$ for R=4, since this covers a time span of 60 to 70 days. For example, when looking at experts in release *v4.11*, we need to take into account data found for releases *v4.7, v4.8, v4.9, v4.10, and v4.11.* We repeat such analysis for each of the 22 releases. In this paper, we use linearly decaying weights W_i to combine the individual $footprint_j(\mathbb{A})$ values across the five considered releases, since this scheme is less extreme than the exponential and logarithmic ones.

Similar to RQ2, we then use the footprint values to create, for each subsystem and release, a ranking of all contributors active in the five considered releases. We also use the same performance metrics as for RQ2, which allows us to compare the results of RQ3 to those of RQ2 to validate whether the historical dimension improves the model.

To save space, and since we found that, similar to RQ2, the combination of `legacy` and `committed` performs the best, we only show the results for this model (the rest of the data will be made available after the double-blind review). In particular, Figure 5.7 and Figure 5.8 show the POS_N performance of the combined `legacy+committed` model without and with the history dimension, for ranking thresholds N ranging from 1 to 5. Figure 5.9 and Figure 5.10 show the corresponding POM_N results.

Table 5.4 contains the results of Wilcoxon paired tests between the POS_N values without and with history, for each N, and (similarly) between the POM_N values without and with history, for each N. For each test, we also provide the Cliff Delta effect size Romano et al. (2006).

Table 5.4 P-values and Cliff's delta values for the Wilcoxon paired tests ($\alpha = 0.01$) between the POS_N results of Figure 5.7 and Figure 5.8, and of Figure 5.9 and Figure 5.10, for ranking thresholds N=1 to N=5. A Cliff delta "-" indicates a non-significant test result, with p-value$> \alpha$

Fig.	Measure	1	2	3	4	5
5.7/5.8	P-value	1.12e-4	8.76e-3	1.15e-2	1.57e-3	7.70e-4
5.7/5.8	Cliff's delta	0.62	0.50	-	0.55	0.51
5.9/5.10	P-value	1.27e-2	4.63e-3	5.13e-4	1.57e-5	1.88e-4
5.9/5.10	Cliff's delta	-	0.46	0.56	0.66	0.69

Results:

The history-aware `legacy+committed` footprint models perform significantly better than the history-unaware models. Figure 5.7 and Figure 5.8 show how, except for N=3, the median performance of the history-aware expertise measure improves upon the history-unaware measure. If one considers only the first recommendation of the measure, there is a median 73.6% chance that at least one maintainer is identified for a history-aware expertise model compared to 69.9% with the single-release model. This difference progressively decreases for higher N, which means that, for higher N, an expertise model considering `legacy+committed` on one release only is robust enough to assess expertise.

We find similar improvements for Figure 5.9 and Figure 5.10, except that the improvements due to history increase for larger N (and for N=1 there is no significant improvement). This is clearly shown by the p-values and effect sizes of the Wilcoxon paired tests in Table 5.4 ($alpha = 0.01$). As an effect size greater than 0.474 indicates a **large** increase in performance, we notice that 7 of the 8 significant differences have an effect size of at least 0.50.

5.7 Discussion

Threats to validity:

Threats to external validity prevent generalization of empirical results to other contexts. In particular, due to the abundant volume of data and presence of an oracle for expertise, our empirical evaluation only focused on 22 releases, or five years of the Linux kernel project. Hence, the study should be expanded to cover not only more kernel releases, but also other open (and closed) source projects. Furthermore, we considered only 5 expertise measures for our footprint models. Other measures, such as those mentioned in Table 5.1, should be studied to understand their impact on expertise.

Threats to construct validity involve risks regarding the measures used in the empirical study. Of the five considered expertise measures, `reviewed` was the only one requiring noisy approximations. Except for cases where the email patch subject was identical to the Git commit message summary, there is a definite risk of false positive and false negative matches, as identified earlier by Jiang et al. Jiang et al. (2013, 2014). This might explain the relatively weak performance of expertise models involving `reviewed`. However, no better alternatives exist for projects that use mailing lists for code review. Projects using web-based review environments like Gerrit do not have this issue, and will have perfect matching between commits and their reviews.

Finally, regarding threats to internal validity (i.e., confounding factors potentially explaining our findings), we mention the limited number of subsystems considered for RQ2 and RQ3. This number was the result of the data filtering in subsection 5.5.2 used to eliminate temporarily inactive subsystems. Furthermore, we used the `MAINTAINERS` file as oracle for expertise. Although this is the known reference in the Linux kernel community for finding the right maintainer to contact, this is a manually maintained text file that hence could contain inconsistencies (even though changes to it are peer-reviewed).

Finally, although maintainership is a form of expertise, there are other forms of expertise that our footprint models be indicators of that were not considered in our empirical study. As such, some of the false positive recommendations of our footprint rankings might actually be correct suggestions based on a different interpretation of expert, in which case our POS_N and POM_N results are lower bounds for the actual performance.

Future work:

Apart from addressing the threats to validity, other future work should consider different weights w_i and W_i. The former weights consider different activities to be more relevant than others, while the latter weights would give more or less weights to older vs. newer releases. For example, comparison of exponential and logarithmic decaying weights to the linear decay used in our study could be interesting. Similarly, different values of R for $footprint_j^R(\mathbb{A})$ should be evaluated.

Finally, whereas we used a top-down approach from expertise model to evaluation on an actual open source project, a bottom-up approach starting from the analysis of a project's or subsystem's maintainers before formulating expertise measures and models could provide complementary insights into different kinds expertise.

5.8 Conclusion

This paper argued about the need for expertise models considering a wide range of developer and other activities, and doing so across different snapshots of a project instead of just for one snapshot. Through an empirical study on 22 releases of the Linux kernel, we empirically showed how measures about a maintainer's own coding footprint (`legacy`) and her involvement in coordinating other project members (e.g., committing their commits and/or reviewing their code changes) significantly improves on coding-only expertise models for the sake of maintianer recommendation. Furthermore, considering those measures across different releases significantly improved performance, with large effect size.

The simplest incarnation of our maintainer recommender that software organizations should consider adopting involves (1) a developer's code `legacy` and number of changes `committed`, which are both readily obtainable from a Git log, calculated across (2) the last 5 releases. In future work, we will consider additional activity measures and empirically analyze other open source projects.

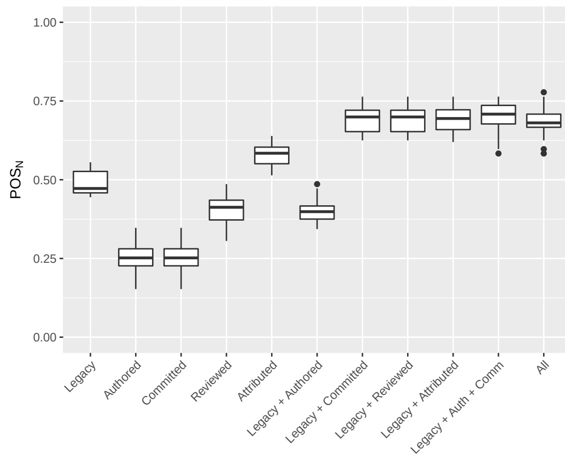

Figure 5.3 Distribution of POS_N for each combination of activity measures, for ranking threshold N = 1.

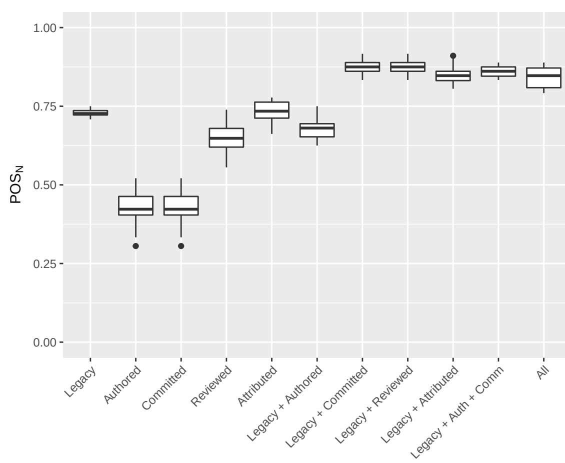

Figure 5.4 Distribution of POS_N for each combination of activity measures, for ranking threshold N = 3.

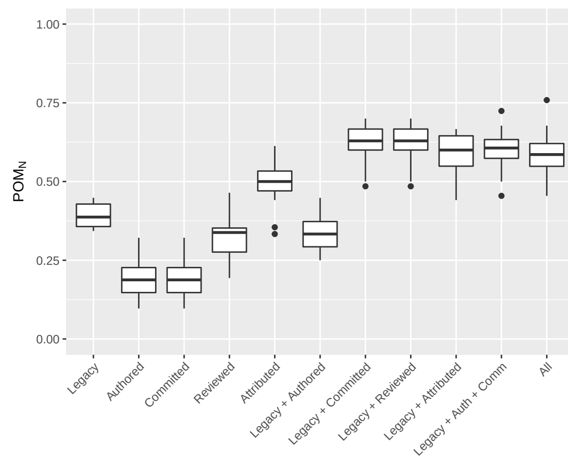

Figure 5.5 Distribution of POM_N for each combination of activity measures, for ranking threshold N = 1. These boxplots only consider subsystems with at most 1 maintainer.

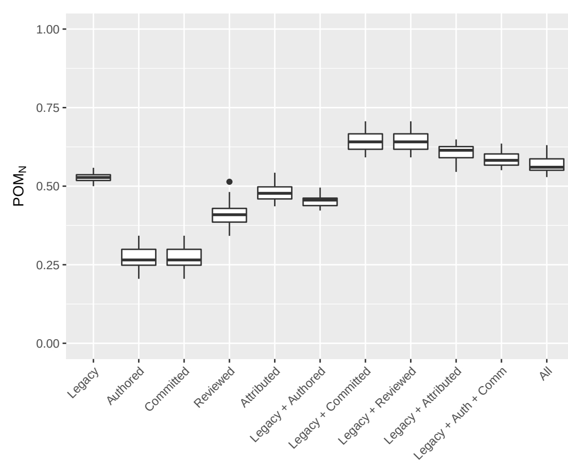

Figure 5.6 Distribution of POM_N for each combination of activity measures, for ranking threshold N = 3. These boxplots only consider subsystems with at most 3 maintainers.

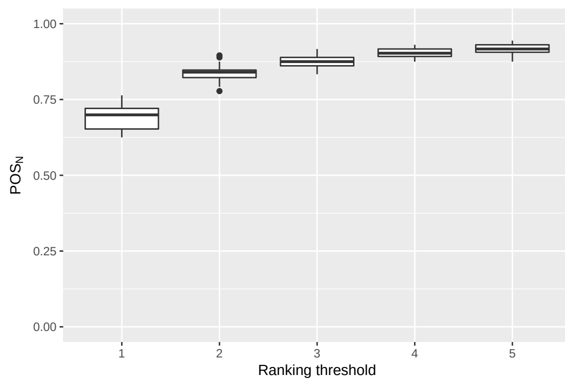

Figure 5.7 Distribution of POS_N for the combined `legacy+committed` model (**without** history dimension), for different N.

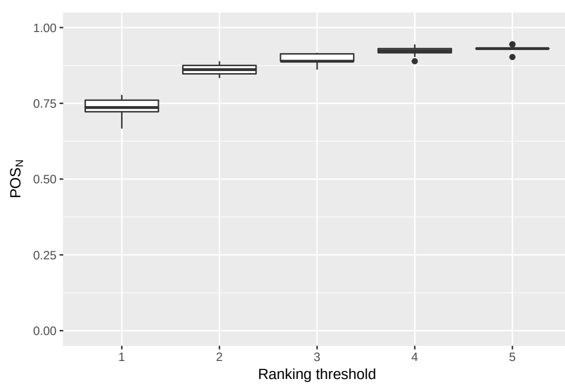

Figure 5.8 Distribution of POS_N for the combined `legacy+committed` model (**with** history dimension), for different N.

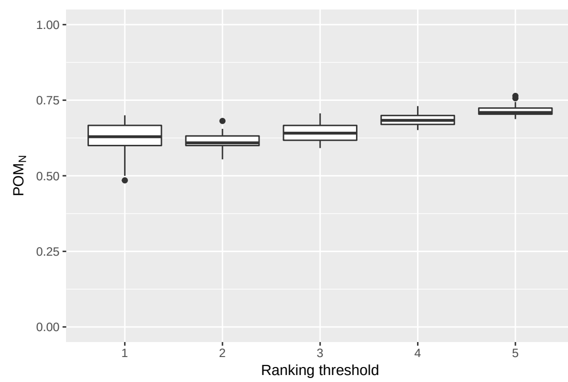

Figure 5.9 Distribution of POM_N for the combined `legacy+committed` model (**without** history dimension), for different N. For each N, the boxplot only considers subsystems with at most N maintainers.

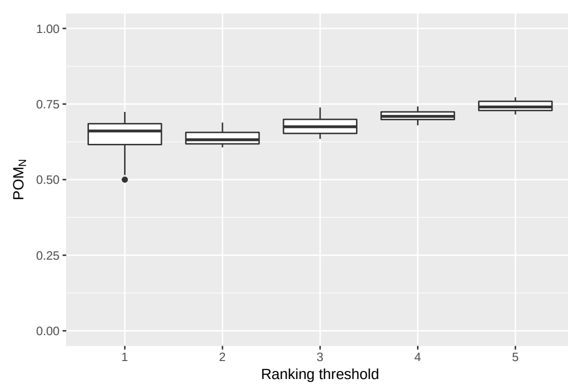

Figure 5.10 Distribution of POM_N for the combined `legacy+committed` model (**with** history dimension), for different N. For each N, the boxplot only considers subsystems with at most N maintainers.

CHAPTER 6 GENERAL DISCUSSION

In this thesis, we explore models capable of expert identification and more specifically, maintainer indentification. Through a review of the existing literature on expertise models, we established that the current models were lacking exposure to the different activities undertaken by Linux developers and lacking exposure to the amount of time the developers have been involved with those activities. To answer this issue, we propose a new model aware of both time and the wide array of activities present in Linux development.

Additionally, we release two open-source projects based on the metrics acquired during the creation of the expertise model. This chapter provides a discussion regarding the two open-source tools: Srcmap and Email2git.

6.1 Srcmap

Srcmap, our visualization of the kernel and its authors, has a few constraints. The main constraint is the lack of a fluid user experience. The amount of data to process in the browser is too high to allow smooth browsing of the main tree. A way to address this issue would be to configure the interface to only download the required data as users browse the visualization. This way, the internet browser uses to display the tool would not have to save the entire dataset in memory and would only process the desired area.

6.2 Email2git

Email2git, our code reivew tracking system, has a few important limitations. The first limitation to consider is the missing mailing lists. Although our patch data source, patchwork.kernel.org, already tracks many mailing lists, some major mailings list like `net-dev` are not tracked. Although this is a minor issue, it reflects in the low number of commits matched in the `net` subdirectory.

We received a lot of valuable feedback from linux developers after our refereed talk the Open Source Summit North America. A developer mentioned the absence of the *Patch 0* from our current implementation of Email2git. The Patch 0 is a summary of the changes submitted, often in multi-patch submissions. Another suggestion was to track *linux-next*. This would allow developers to access discussion behind commits that have not been integrated in the main tree.

For the future of this project, we recommend running our own instance of Patchwork 2.0, which automatically track the Patch 0 of each patch. In addition to ansering the lack of Patch 0, running our own Patchwork would allow to have control on the set of tracked mailing lists. If we have access to old archives of the desired mailing lists, we could be able to create matching data dating to before 2009. We also recommend tracking the linux-next tree, as Email2git could ease the integration debugging process.

6.3 Maintainer Recommendation Model

The article in chapter 5 describes a model used to recommend the maintainer(s) of a given subsystem. To create this recommender, we used thechniques described in previous work to create a modified expertise model. All maintainers are experts, but not all experts are maintainers. Hence we use a different set of metrics chosen to represent the activities undertaken by maintainers on a daily basis, such as upstream committing and code reviews.

As described in section 5.7, there are several threads to the validity of the model. As a threat to *external validity*, we describe that the data set should include data from a longer timespan and from more different projects, proprietary and open source. This would allow our model to be generalized to external contexts.

As a threat to *construct validity*, we identify one of the metrics studies to be noisy. The data related to the `reviewed` activity is often incomplete due to the nature of the Linux code review process. This noise in the data represents a risk regarding one of the measure used in the empirical study.

Additionally, another threat to validity not mentioned in section 5.7 is that we validated the model's recommendations with the maintainers listed in the `MAINTAINERS` file as an oracle. Ideally, our model should be trained to detect or predict the best candidate as a *replacement* to the current maintainer(s). To achieve this, we would have to create an oracle containing the developers that were selected as maintainers, and verify wheter our model is capable to recommend the developer at the moment of the selection.

CHAPTER 7 CONCLUSION

The research project described in this thesis establishes a history-aware model capable of predicting experts in a large software project. We describe the different metrics acquired for the creation of the model as well as the techniques used to mine the necessary data. We also provide an empirical evaluation of the model based on a list of known Linux experts. We were able to make some of the data available to the Linux community through two open source tool available online. We describe the different steps undertaken during the deployment of those tools. We also provide an evaluation of our expertise model.

7.1 Advancement of Knowledge

In addition to communicating our findings regarding expertise models in a submitted scholarly article, we were able to build two interfaces to further share our data with the rest of the Linux community. The main advancement of knowledge carried by our research project are the new dimenssions used in our expertise model. State of the art techniques fail to include other aspects of software engineering, like code reviews and upstream committing. Our model is more appropriate to large organizations where maintainers or managers are usually too busy to continue contributing code to the project.

7.2 Limits, Constraints, and Recommendations

There are two main limitations to the model proposed in our submitted paper. The first limitation is a direct concequence of Email2git's limitation. The missing mailing lists cause an uneven distribution of the matched commits accross the different subdirectories of the kernel. To address this limitation, we only studied subsystems with a certain percentage of matched commits. This ensured homogeneity of the data among the different subsystems.

The other limitation is the validation technique we implement to assess the performance of our model. We use the maintainers currently active in the subsystem for the studied release. The issue with the technique is that our model is partially based on activities usually related to maintainership, such as code reviews and upstream committing. To evaluate our model as **maintainer recommender** instead of an **expert recommender**, we would have to look at the developers that were selected as a replacement for a departing maintainer. A strong model would be capable of detecting the chosen developer for the release before the maintainer change.

At last, we cite the existing link between code stability and knowledge of that code area. Since our model offers a customizable historical weight function, we could choose this weight function according to the stability of the code. For example, we could implement an exponential weight function to a very stable code base, as older contributions should account for more of the expertise measure. Furthermore, a code base undergoing large amounts of changes could require a logarithmic weight function, as older contributions should not affect current expertise as much as recent contributions.

REFERENCES

J. Anvik, L. Hiew, et G. C. Murphy, “Who should fix this bug?” dans *Proceedings of the 28th International Conference on Software Engineering*, série ICSE ’06. New York, NY, USA: ACM, 2006, pp. 361–370. DOI: 10.1145/1134285.1134336. En ligne: http://doi.acm.org/10.1145/1134285.1134336

F. Armstrong, F. Khomh, et B. Adams, “Broadcast vs. unicast review technology: Does it matter?” dans *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 219–229. DOI: 10.1109/ICST.2017.27

F. T. Armstrong, F. Khomh, et B. Adams, “Broadcast vs. unicast review technology: Does it matter?” dans *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, March 2017.

B. B. Bederson, B. Shneiderman, et M. Wattenberg, “Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies”, *ACM Trans. Graph.*, vol. 21, no. 4, pp. 833–854, Oct. 2002. DOI: 10.1145/571647.571649. En ligne: http://doi.acm.org/10.1145/571647.571649

P. Bhattacharya, I. Neamtiu, et M. Faloutsos, “Determining developers’ expertise and role: A graph hierarchy-based approach”, dans *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 11–20. DOI: 10.1109/ICSME.2014.23

C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, et P. Devanbu, “The promises and perils of mining git”, dans *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, série MSR ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. DOI: 10.1109/MSR.2009.5069475. En ligne: http://dx.doi.org/10.1109/MSR.2009.5069475

J. Corbet et G. Kroah-Hartman, “2017 linux kernel development report”, The Linux Foundation, Rapp. tech., 2017.

F. O. Daniel Izquierdo-Cortazar, Gregorio Robles et J. M. Gonzalez-Barahona, “Using software archaeology to measure knowledge loss in software projects due to developer turnover”, dans *Proceedings of the 42Nd Hawaii International Conference on System*

Sciences, série HICSS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. DOI: 10.1109/HICSS.2009.498. En ligne: http://dx.doi.org/10.1109/HICSS.2009.498

M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, et J.-R. Falleri, "Impact of developer turnover on quality in open-source software", dans *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, série ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 829–841. DOI: 10.1145/2786805.2786870. En ligne: http://doi.acm.org/10.1145/2786805.2786870

T. Fritz, G. C. Murphy, et E. Hill, "Does a programmer's activity indicate knowledge of code?" dans *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, série ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 341–350. DOI: 10.1145/1287624.1287673. En ligne: http://doi.acm.org/10.1145/1287624.1287673

T. Fritz, J. Ou, G. C. Murphy, et E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity", dans *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, série ICSE '10. New York, NY, USA: ACM, 2010, pp. 385–394. DOI: 10.1145/1806799.1806856. En ligne: http://doi.acm.org/10.1145/1806799.1806856

T. Girba, A. Kuhn, M. Seeberger, et S. Ducasse, "How developers drive software evolution", dans *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, Sept 2005, pp. 113–122. DOI: 10.1109/IWPSE.2005.21

L. P. Hattori, M. Lanza, et R. Robbes, "Refining code ownership with synchronous changes", *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 467–499, Août 2012. DOI: 10.1007/s10664-010-9145-5. En ligne: http://dx.doi.org/10.1007/s10664-010-9145-5

Y. Jiang, B. Adams, et D. M. German, "Will my patch make it? and how fast? – case study on the linux kernel", dans *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR)*, San Francisco, CA, US, May 2013, pp. 101–110.

Y. Jiang, B. Adams, F. Khomh, et D. M. German, "Tracing back the history of commits in low-tech reviewing environments", dans *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Torino, Italy, September 2014.

D. Joseph, K.-Y. Ng, C. Koh, et S. Ang, "Turnover of information technology professionals: A narrative review, meta-analytic structural equation modeling, and model development", *MIS Q.*, vol. 31, no. 3, pp. 547–577, Sep. 2007. En ligne: http://dl.acm.org/citation.cfm?id=2017336.2017343

D. Ma, D. Schuler, T. Zimmermann, et J. Sillito, "Expert recommendation with usage expertise", dans *2009 IEEE International Conference on Software Maintenance*, Sept 2009, pp. 535–538. DOI: 10.1109/ICSM.2009.5306386

D. W. McDonald et M. S. Ackerman, "Expertise recommender: A flexible recommendation system and architecture", dans *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, série CSCW '00. New York, NY, USA: ACM, 2000, pp. 231–240. DOI: 10.1145/358916.358994. En ligne: http://doi.acm.org/10.1145/358916.358994

T. Mens, M. Claes, P. Grosjean, et A. Serebrenik, *Studying Evolving Software Ecosystems based on Ecological Models*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 297–326. DOI: 10.1007/978-3-642-45398-4_10. En ligne: https://doi.org/10.1007/978-3-642-45398-4_10

S. Minto et G. C. Murphy, "Recommending emergent teams", dans *Proceedings of the Fourth International Workshop on Mining Software Repositories*, série MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 5–. DOI: 10.1109/MSR.2007.27. En ligne: http://dx.doi.org/10.1109/MSR.2007.27

A. Mockus, "Organizational volatility and its effects on software defects", dans *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, série FSE '10. New York, NY, USA: ACM, 2010, pp. 117–126. DOI: 10.1145/1882291.1882311. En ligne: http://doi.acm.org/10.1145/1882291.1882311

A. Mockus et J. D. Herbsleb, "Expertise browser: A quantitative approach to identifying expertise", dans *Proceedings of the 24th International Conference on Software Engineering*, série ICSE '02. New York, NY, USA: ACM, 2002, pp. 503–512. DOI: 10.1145/581339.581401. En ligne: http://doi.acm.org/10.1145/581339.581401

F. Rahman et P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship", dans *Proceedings of the 33rd International Conference on Software*

Engineering, série ICSE '11. New York, NY, USA: ACM, 2011, pp. 491–500. DOI: 10.1145/1985793.1985860. En ligne: http://doi.acm.org/10.1145/1985793.1985860

F. Ricca, A. Marchetto, et M. Torchiano, “On the difficulty of computing the truck factor”, dans *Proceedings of the 12th International Conference on Product-focused Software Process Improvement*, série PROFES'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 337–351. En ligne: http://dl.acm.org/citation.cfm?id=2022348.2022383

P. C. Rigby, Y. C. Zhu, S. M. Donadelli, et A. Mockus, “Quantifying and mitigating turnover-induced knowledge loss: Case studies of chrome and a project at avaya”, dans *Proceedings of the 38th International Conference on Software Engineering*, série ICSE '16. New York, NY, USA: ACM, 2016, pp. 1006–1016. DOI: 10.1145/2884781.2884851. En ligne: http://doi.acm.org/10.1145/2884781.2884851

J. Romano, J. Kromrey, J. Coraggio, et J. Skowronek, “Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?” dans *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–3.

D. Schuler et T. Zimmermann, “Mining usage expertise from version archives”, dans *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, série MSR '08. New York, NY, USA: ACM, 2008, pp. 121–124. DOI: 10.1145/1370750.1370779. En ligne: http://doi.acm.org/10.1145/1370750.1370779

M. Torchiano, F. Ricca, et A. Marchetto, “Is my project's truck factor low?: Theoretical and empirical considerations about the truck factor threshold”, dans *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics*, série WETSoM '11. New York, NY, USA: ACM, 2011, pp. 12–18. DOI: 10.1145/1985374.1985379. En ligne: http://doi.acm.org/10.1145/1985374.1985379

C.-G. Wu, J. H. Gerlach, et C. E. Young, “An empirical analysis of open source software developers' motivations and continuance intentions”, *Inf. Manage.*, vol. 44, no. 3, pp. 253–262, Avr. 2007. DOI: 10.1016/j.im.2006.12.006. En ligne: http://dx.doi.org/10.1016/j.im.2006.12.006

——, “An empirical analysis of open source software developers' motivations and continuance intentions”, *Inf. Manage.*, vol. 44, no. 3, pp. 253–262, Avr. 2007. DOI: 10.1016/j.im.2006.12.006. En ligne: http://dx.doi.org/10.1016/j.im.2006.12.006

M. Zhou et A. Mockus, “Who will stay in the floss community? modeling participant’s initial behavior”, *Software Engineering, IEEE Transactions on*, vol. 41, pp. 82–99, 01 2015.

M. Zhou, Q. Chen, A. Mockus, et F. Wu, “On the scalability of linux kernel maintainers’ work”, dans *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, série ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 27–37. DOI: 10.1145/3106237.3106287. En ligne: http://doi.acm.org/10.1145/3106237.3106287