Mining File Histories: Should We Consider Branches?

Vladimir Kovalenko
Delft University of Technology
Delft, The Netherlands
v.v.kovalenko@tudelft.nl

Fabio Palomba
University of Zurich
Zurich, Switzerland
palomba@ifi.uzh.ch

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

ABSTRACT

Modern distributed version control systems, such as Git, offer support for branching — the possibility to develop parts of software outside the master trunk. Consideration of the repository structure in Mining Software Repository (MSR) studies requires a thorough approach to mining, but there is no well-documented, widespread methodology regarding the handling of merge commits and branches. Moreover, there is still a lack of knowledge of the extent to which considering branches during MSR studies impacts the results of the studies.

In this study, we set out to evaluate the importance of proper handling of branches when calculating file modification histories. We analyze over 1,400 Git repositories of four open source ecosystems and compute modification histories for over two million files, using two different algorithms. One algorithm only follows the first parent of each commit when traversing the repository, the other returns the full modification history of a file across all branches. We show that the two algorithms consistently deliver different results, but the scale of the difference varies across projects and ecosystems. Further, we evaluate the importance of accurate mining of file histories by comparing the performance of common techniques that rely on file modification history — reviewer recommendation, change recommendation, and defect prediction — for two algorithms of file history retrieval. We find that considering full file histories leads to an increase in the techniques' performance that is rather modest.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**;

KEYWORDS

Version Control Systems; Branches; Mining Software Repositories

ACM Reference Format:
Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli. 2018. Mining File Histories: Should We Consider Branches?. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238169

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238169

1 INTRODUCTION

The workflow of modern version control systems (VCS), such as Git, extensively relies on branching. Branching support allows developers to manage multiple isolated versions of the working tree, which can be modified independently of each other. Branch-related operations in Git are by design extremely lightweight compared to older VCSs [27]. Low cost of branching allows branches to be used for development of individual features, for experimenting with design solutions, and for preparing releases [23]. In all these examples use of branches allows teams to keep the main working tree free of questionable code and reduces development overhead related to version conflicts [46].

While being the most popular version control system today [14], Git is quite unfriendly for data mining [27]. In particular, branching features introduce issues for miners: branches can be removed and overwritten, and synchronization with the remote repository can introduce implicit branches with no logical meaning [27].

Despite the difficulties with analysis of Git history, mining of historical data from VCS is still the basis for a variety of studies, which quantitatively explore the development process and suggest approaches to facilitate it [20, 26, 36, 40, 43, 52, 62–64].

History of individual files is a particularly important source of information for prominent practical applications, like (i) defect prediction algorithms, where metrics based on file history are important features [25, 36, 41, 53], (ii) code ownership heuristics [38, 42, 60], which are based on aggregation of individual contributions of all authors of the file, and (iii) code reviewer recommendation [22, 54, 59, 61], where history of prior changes to files serves as a basis for automatic selection of the expert reviewers.

Pitfalls of Git from the data mining perspective pose common threats to validity of every of such studies. Some of these threats, such as mutability of history, are commonly acknowledged by researchers (e.g., [18, 34, 45, 63, 65]). Nevertheless, there is no widespread approach to handling of merge commits and branches during mining. Moreover, MSR studies often do not provide a detailed description of mining algorithms, and handling of the branches in particular, or explicitly focus the analyses on the main branch of the repository [18].

In this study, we aim at making a first step toward the assessment of the threats arising from not considering full information about brenching in mining software repository studies. Specifically, we focus on the impact of the branch handling strategy on extraction of a file modification history. This task requires a traversal of a repository graph to collect individual commits affecting the file. We first perform a preliminary analysis on how the mining of file histories is impacted by branches, by measuring how much *first-parent* (i.e., history extractable when only considering the first parent of each commit when traversing the repository) and *full* (i.e., the history

extractable when considering changes in all branches) file histories differ from each other. Then, we study how performance of three MSR applications (code reviewer recommendation, change recommendation, and defect prediction), that use file modification histories as input data, varies when considering branches.

Our results show that the first-parent and full mining strategies consistently result in different file histories, even though the scale of the difference varies across software ecosystems and repositories within each ecosystem. We find that considering the full file histories leads to an increase in the considered MSR-based techniques' performance that is rather modest. This marginal increase indicates that our findings do not raise any serious questions on the validity of studies that simplify the mining approach. Nevertheless, in our work we devised a method and a tool for efficient mining of full file histories at scale, which we make publicly available [11].

2 BACKGROUND

Several prior studies focus on the use of branching and its added value for developers. Combined, these studies provide strong evidence of importance of branching in modern software development. Appleton et al. [19] explore an extensive set of branching patterns and propose a number of best practices and antipatterns. Buffenbarger and Gruell [32] devise practices and patterns to facilitate efficient parallel development, mitigating the complexity of branching operations in early VCSs. Bird et al. [29] conclude that developers working in a branch represent a virtual team. Barr et al. [23] claim that lightweight branching support is the primary factor in rapid adoption of modern distributed VCSs in OSS projects. Bird and Zimmermann [28] identify common problems from improper branch usage and propose an analysis to assess more efficient alternative branch structures. Shihab et al. [56] find that the excessive use of branches is associated with a decrease in software quality.

Today's most popular version control system — Git — was not designed to preserve a precise history of modifications [10], which implies difficulties with the analysis of these histories [27, 48]. Analysis of software version histories is not only used to study the development practices, but also to facilitate development with data-driven tools. Prominent examples of applications for tools heavily relying on histories of changes of individual files are defect prediction [35, 39, 47], code reviewer recommendation [22, 61], and change prediction [67, 68]. Notably, the complexity of Git, the mutability of its data structure, and the difficulty of figuring out the parent relationships between revisions complicate the work of researchers and prevent some practitioners from using it as their version control system [16]. Being able to accurately retrieve histories of prior changes is vital for efficient use of techniques that are based on histories. Moreover, in some cases histories need to be processed to achieve optimal performance of the techniques: For instance, Kawrykow and Robillard [44] show that removing non-essential changes from modification histories improves the performance of co-change recommendation [68].

2.1 Motivation

Version control repositories are the key data source for a wide variety of software engineering studies [34, 36, 41, 58, 63]. With no widespread high-level mining tool in use, the common way for the researchers to mine the histories of repositories is to use homegrown tools based on low-level libraries, such as JGit [9]. While low-level operations provide greater flexibility of mining, they also undermine the reproducibility of studies, as details of mining are usually not elaborated on in the papers. Reproduction packages, where available, commonly contain information obtained after mining, but not the repository mining scripts.

Restoring the actual change history from a Git repository is challenging and error-prone [27]. To come around the difficulties, some studies (e.g. [18]) focus on the development activity in the main branch, thus omitting part of the changes in the repository. This approach may be sufficiently precise for some applications, because (i) in some repositories most of development activity takes place in the master branch, and (ii) the `rebase` operation is often used to integrate changes from branches into the main branch. However, consideration of branches and careful handling of merge commits might be important for precise calculation of individual file histories, which are the primary source of input data in some contexts, such as code reviewer recommendation [22] and change recommendation [68].

The difference in quality of data between different mining approaches, and the impact of the chosen mining approach on performance of analysis methods driven by historical data, are not clear and have not yet been explored. We conduct this study to quantify the effect of considering the graph structure of the repository (importance of such consideration is reported as one of the perils of mining Git [27]) and to investigate how the difference in the results from different mining approaches impacts performance of MSR applications.

While the file histories are the main input data for a variety of MSR-based techniques, there is no guarantee that more complex and precise mining methods ensure an increase in performance of the techniques notable enough to justify the extra mining effort. With no prior research existing on this topic, with this study we seek not only to identify the impact of branch handling strategy on performance of file-history-based methods, but also to compare the scale of this impact between different techniques. This knowledge could help make a step towards ensuring that MSR studies and their practical applications employ optimal mining strategies to get the most value out of the repository data.

2.2 Challenges of Mining the File Histories

Mining of histories of individual files at large scale is a non-trivial task. Git provides a toolkit for repository operations, including `git log`, which facilitates retrieval of logs of commits. However, Git was not designed to support careful storage and retrieval of history of changes [10, 31, 49, 57], which implies several complications with using `git log` for mining file histories. Specifically:

Performance With no specialized index for file histories in place, retrieval of histories of changes for an individual file requires traversal of the commit graph. Retrieval of histories for every file in the repository tree is very expensive.

Handling of renames A Git repository does not contain any records of renames and moves of files. Such events are detected based on similarity of contents of consecutive versions of a file, with thresholds defined by the settings of the Git client. As a result,

calculated history of the same file in the same repository might appear different on different clients.

Handling of merge commits and branches The `git log` tool, which is often used for analysis of software histories, supports an overwhelming variety of settings, with over 100 argument options [6]. Unless the tool is thoroughly set up, some potentially interesting changes might be implicitly omitted: for example, by default `git log` prunes some side branches. A default approach might not be suitable for some applications.

Difficulty of use A wide variety of settings makes the user experience with `git log` quite complicated. Certain scenarios of retrieval of repository history are even harder: for example, traversing the commits graph forward in time, which might be useful in some contexts, is more difficult. For example, all descendants of a commit can be retrieved with the following command:

```
git rev-list --all --parents | grep ".\{40\}.*SHA.*" |
awk '{print $1}'
```

This command [5], which only lists the commits without providing any information on the structure, is already not trivial.

2.3 Retrieval of File Histories

History of commits in a Git repository can be represented as a directed acyclic graph. Each commit logically represents a state of the repository's file tree. Each version of the state is based on one or several prior *(parent)* version, and only the difference in the state between the parent and the current version is actually stored.[1] In Figure 1 (left), which presents a hypothetical commit tree, the commit parent relationships are represented by black arrows: for example, commit 5 is the parent of commit 6. Merge commits, which integrate changes from multiple branches, have more than one parent. In Figure 1, commit 5 has two parents: 3 and 4. In a merge commit with multiple parents, the list of parents is sorted: if branch *A* is merged into branch *B*, the first parent of the merge commit would be the one that branch *B* was pointing to before the merge. In Figure 1, parent commits are sorted left to right: commit 4 is the first parent of commit 5.

Each commit affects a set of files[2] and defines their new content relative to their content in the parent commit. It is possible to say that a commit *affects* a file, if its content in the revision which is represented by the current commit is different from its content in the parent commit, or if the file was created/removed in the current commit. It is common to think of a commit as simply a set of changes in one or multiple files. This simple model is convenient and is used by Git itself, e.g., in the `git diff` command. In Figure 1, affected files are shown in boxes next to nodes of the commit tree.

Retrieval of a a history of changes for a given file — i.e., list of the commits that affect this file — one needs to traverse the commit graph to identify such commits. A traversal and handling of commits one by one is necessary because Git does not store any auxiliary data that would allow to perform this operation faster. During the traversal, it is possible to either follow all parents of a merge commit, which ensures visiting every transitive parent of the starting point and including all of these commits in the history, or only follow the first parent.

We refer to the traversal strategy that follows all parents and to resulting histories as *full*. Considering the example repository graph in Figure 1, full traversal starting from the latest commit in the repository (12) would include all 12 commits in the repository. A full history for the file *f2* would contain all 9 of these commits that affect *f2*. An alternative strategy is to only follow the first parent of every merge commit during the traversal. We refer to this strategy and resulting file histories as *first-parent*. For the example in Figure 1, such traversal starting from commit (12) would only include commits (12), (11), (9), (6), (5), (4), (2), and (1). A first-parent history of *f2* would only include 6 commits from this traversal that affect *f2*. It is important to note that the first-parent strategy does not omit merge commits that contain changes to the file relative to its first parent: an example of such commit in Figure 1 is (5), and it is included in the first-parent history as well.

The histories for the file *f2*, as retrieved with both strategies, are presented in Figure 1 (right). As the first-parent history contains less commits, some changes to the file are omitted. Thus, using the first-parent strategy to calculate quantitative properties of file histories, such as number of changes or number of contributors, leads to incomplete results. The two cases represented by the two strategies are rather extreme: the simplified strategy omits all changes that were made outside the main branch and did not end up in the main branch after a rebase. There are less radical ways of simplified handling of branches than to omit traversing the branch completely. For example, one way to see the summary of changes in a branch is to inspect the output of `git diff` between two parents of a merge commit. This approach is adopted by some GUI applications on top of Git, such as Sourcetree[3]. While this approach allows to retrieve a summary of changes in a branch, the individual changes, possibly made by different authors, are presented together and are not distinguishable, which makes the approach less applicable for mining tasks: it is impossible to identify individual contributions by count, sizes, or dates of changes per author.

We find the extreme case of comparing full and first-parent histories an appropriate setting to study the impact of the mining strategy on the results of mining and performance of the methods based on these results. With the two extreme cases, we have the highest chance of identifying effects not present in less extreme settings. As this study is the first to explore such effects, we consider this setting fruitful to highlight the directions for future work.

3 METHODOLOGY

3.1 Research Questions

We center our investigation around two research questions. First, we set out to quantify the importance of careful handling of branches during mining. In particular, we (i) explore the repository structure to calculate numbers of commits reachable with and without considering branches and (ii) analyze differences on a lower level of history of individual files. Thus our first research question is:

RQ$_1$. *How does the branch handling strategy impact the results of mining?*

Having identified the magnitude of the difference between mining approaches that consider and do not consider branches, we

[1] Described is a simplified scheme of the storage model of Git, which is in reality more complex.

[2] The set can be empty, e.g., in most of merge commits without conflicts.

[3] https://www.sourcetreeapp.com/

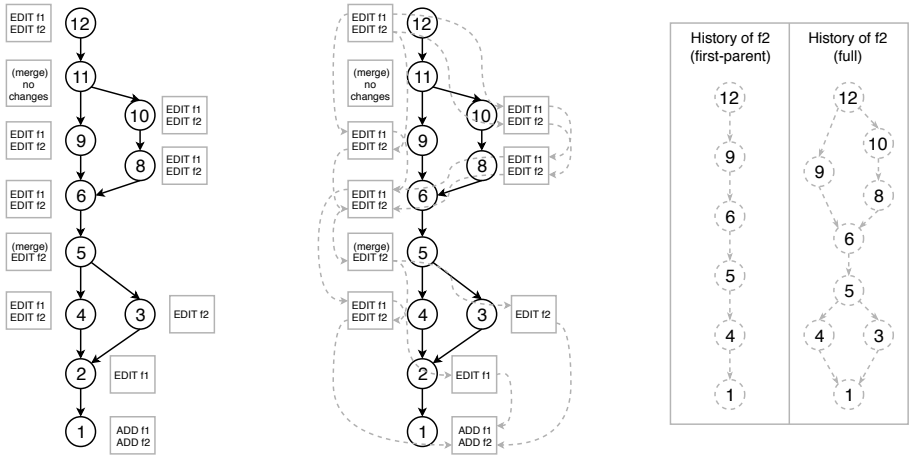

Figure 1: Construction of file parent connections and resulting file histories

investigate the impact of branch handling strategy on the performance of algorithms relying on history of files such as reviewer recommendation, change recommendation, and defect prediction. Thus, our second research question is:

RQ$_2$. *How important is the branch handling strategy for applications?*

3.2 Mining Histories at Large Scale

Traversal of the commit tree can take a significant amount of time: in some repositories the tree contains hundreds of thousands of commits. If this operation has to be repeated for many or even all files in the repository — which is often the case during mining the repository for file histories — it can take a lot of time for larger repositories. Slowness of mining the file histories is a limitation of the storage model of Git, which does not link individual versions of a file to each other, and does not include indices of changes to individual files. To overcome this limitation, we devised an alternative representation of changes in Git, aimed at fast retrieval of file histories. We use a graph database engine[4] to store a representation of the commit graph. To keep the database compact, we only store the commit nodes and records of affected files, excluding their content.

In addition to the commit parent relationship, which is the basis of the commit graph in Git, we introduce the concept of *parent file versions*. A parent version of a file can be defined as a change to the same file in some other commit, which can be reached from the current commit with a unique path over the commit parent graph. In Figure 1 (center), the file parent connections are represented by gray arrows. By processing the repository, we build the graph of parent relationships between file versions, and store it in the graph database aside the commit graph. Once the file parent graph is built, retrieval of prior changes to the file is as simple as retrieving all transitive parents of the current version, which only requires traversing the file parent graph, which is much faster than traversing the whole repository thanks to direct links to parent file versions. For large-scale mining tasks this approach saves time: after processing the repository and building the file parent connections, it is possible to retrieve full modification histories for several thousand files per second. Figure 1 (right) presents the resulting histories from traversal of the file parent graph.

[4]Neo4J: https://neo4j.com/

3.3 Target Systems

We take a number of steps to ensure diversity in our target systems [51]. Our dataset consists of 260 repositories from Github, selected in a stratified manner to include projects of different scale. Using GHTorrent [37], we randomly sample 100 repositories with over 10,000 commits recorded in GHTorrent database for each, and 200 repositories with 1,000 to 10,000 commits. 40 of the 300 repositories turned out not to be publicly available anymore; the rest comprise our Github sample. Counts of commits as retrieved from GHTorrent are not completely accurate, so actual distribution of sizes of repositories in the sample is slighly more disperse. We have attempted to download and mine all repositories from the Apache open source ecosystem, of which we succeeded with 441 repositories of 532. The rest of the repositories were not available, empty, or failed to process with our toolkit.

Finally, we include 395 and 309 repositories from Eclipse and OpenStack respectively, which use Gerrit for code review. The repositories in these two samples belong to the projects concerned by the latest 100,000 reviews in each Gerrit instance, which we have mined to evaluate the performance of a reviewer recommendation algorithm (Section 5.1). We use smaller subsets of repositories for parts of RQ2. For change recommendation and defect prediction, we use samples of 10 repositories from each of Eclipse and Apache ecosystems. These ecosystems were selected based on availability of defect data (explained in Section 5.3.1). We use the same sample for change recommendation (Section 5.2.1). To evaluate reviewer

recommendation performance, we use the repositories of the 20 most active projects from each of Eclipse and OpenStack ecosystems (Section 5.1). For the quantitative analysis of repository structure and file histories (RQ1), we use all 1,405 repositories from the four samples. We report the entire list of repositories in our online appendix [11].

4 RQ1: DIFFERENCE IN MINING RESULTS

Our first research question seeks to quantify the differences resulting from the application of two different mining approaches to retrieval of the history of files in Git.

4.1 Methodology

We devised a set of metrics that quantify the effect of strategy of branch handling on the results of mining. Afterwards, we compared two approaches in terms of these metrics, i.e., the *first-parent* one (which extracts history only considering the first parent of each commit when traversing the repository) and the *full* one (which extracts all the commits by exploiting the approach described in Section 2.3).

For each repository, we first compute descriptive measures of its structure, such as number of commits that are reachable from HEAD (the latest commit in the main branch), number of merge commits (with more than one parent), number of files in the repository, and number of unique contributors to the repository. We use these metrics to compare the ecosystems between each other, and to explore the variation in branching activity within and between them, which is important for mining: for example, differences between the numbers of commits reachable depending on the traversal strategy denote the importance of the strategy for mining: if only the first parent is considered and traversed, some commits are left out, while still contributing to the state of the repository at HEAD. Number of merge commits can be used as a proxy measure of branching activity in the repository.

Beyond repository-wide metrics, the way in which branches are handled also impacts the calculation of histories of individual files: if only the main branch is considered, some changes from file history are omitted. This effect might impact various applications of file histories, ranging from identification of contributors to a file to more complex scenarios such as reviewer recommendation.

To quantify this effect, we calculate histories of every file in the repository, using both the *first parent* and the *full* approaches to retrieve all commits that contribute to a given version of a file. For every repository (all of which contain over 2 million files in total), we calculate repository-wide average length of the history of a file in its tree, when retrieved via first-parent and full method. We compute the ratio of these averages, and fraction of files for which the two methods deliver different histories among all files in the repository. In addition, we calculate numbers of contributors to each file for both methods of file history mining, and compare their repository-wide averages.

4.2 Results

4.2.1 Descriptive Metrics. To display the natural differences between the ecosystems, which are not associated with different mining strategies, we first present the comparison of the ecosystems in terms of natural activity metrics, such as sizes of repositories and number of contributors. The top two rows of Figure 2 present a comparison of descriptive metrics between the repositories in the four subject ecosystems. The numbers of commits in the repositories within each ecosystem greatly vary. A median number of commits in a repository ranges from 172 for Github to 1,039 for OpenStack. An average repository contains several hundred files in the file tree at HEAD and this number varies in all ecosystems, with a lower variation for OpenStack. Projects from GitHub are typically developed by only a few authors – the median number of contributors for a Github repository is 4. This value for OpenStack is 61, with Eclipse and Apache falling in the middle.

For measures of branching activities, repositories from three of the four ecosystems display moderate values: The majority of commits in a typical repository from every ecosystem except OpenStack is reachable from HEAD. OpenStack also stands out in numbers of merge commits. In a median project, almost 30% of commits in the repository that are reachable from HEAD are merges (Figure 2). Along with higher branching activity, OpenStack repositories show the highest difference between numbers of commits reachable from HEAD via the first parent and via all parents.

Notably, repositories from Github are much more diverse in terms of branching activity metrics. We attribute this diversity to the fact that projects in other ecosystems are logically connected, with possibly common engineering guidelines and intersecting development teams. In addition, the branching structure of the repositories is possibly impacted by the strategy of integrating the pull requests, which are a common part of the workflow at Github and can be either merged or rebased.

4.2.2 File History Metrics. Table 1 presents the statistics on the four ecosystems with regard to difference in first-parent and full file histories. Over the four ecosystems, 19% of files display difference in histories retrieved via first parent and full methods. Ecosystem-wide fractions vary from 14% in Eclipse to 55% in OpenStack. 71% of the repositories in our samples contain at least one file with difference in the history. Fraction of such repositories varies between ecosystems from 56% in Eclipse to 97% in OpenStack. 81% of all commits and 72% of all files belong to such repositories, which indicates that the difference between first-parent and full histories is significant in most of the repositories and cannot be ignored as a rare effect.

Distributions of the metrics related to difference in the results of mining file histories are presented in the bottom row of Figure 2. One metric that indicates the importance of the strategy of file history mining for a given repository is number of files for which histories retrieved via the first parent and via the full traversal have different lengths. Such files exist in 305 of 441 repositories (69%) from Apache, 173 out of 260 (67%) from the Github, 220 of 395 (56%) in Eclipse, and in 301 of 309 (97%) OpenStack repositories.

Similarly to the merge activity metrics, fraction of files with difference in history greatly varies within every ecosystem. Median proportion of such files is 8% in Apache and Github, under 1% in Eclipse, and 46% in OpenStack repositories. Distribution of ratio of the length of the two histories across repositories that contain files with the difference in histories (naturally, this metric is only defined for such projects), displays a similar behaviour across ecosystems to fraction of files with the difference in history. This ratio for Eclipse

Table 1: Overview of the target ecosystems

Ecosystem	Projects			Commits			Files				
	total	with difference		total	in projects with difference		total	with difference		in projects with difference	
Github	260	173	(67%)	872,833	257,156	(29%)	322,783	85,170	(26%)	225,861	(70%)
Apache	441	305	(69%)	998,910	938,032	(93%)	861,196	136,692	(16%)	525,997	(61%)
OpenStack	309	301	(97%)	1,317,165	1,317,004	(100%)	87,382	47,634	(55%)	87,166	(100%)
Eclipse	395	220	(56%)	1,000,997	883,318	(88%)	874,044	127,833	(14%)	712,315	(82%)
Total	**1,405**	**999**	**(71%)**	**4,189,905**	**3,395,510**	**(81%)**	**2,145,405**	**397,329**	**(19%)**	**1,551,339**	**(72%)**

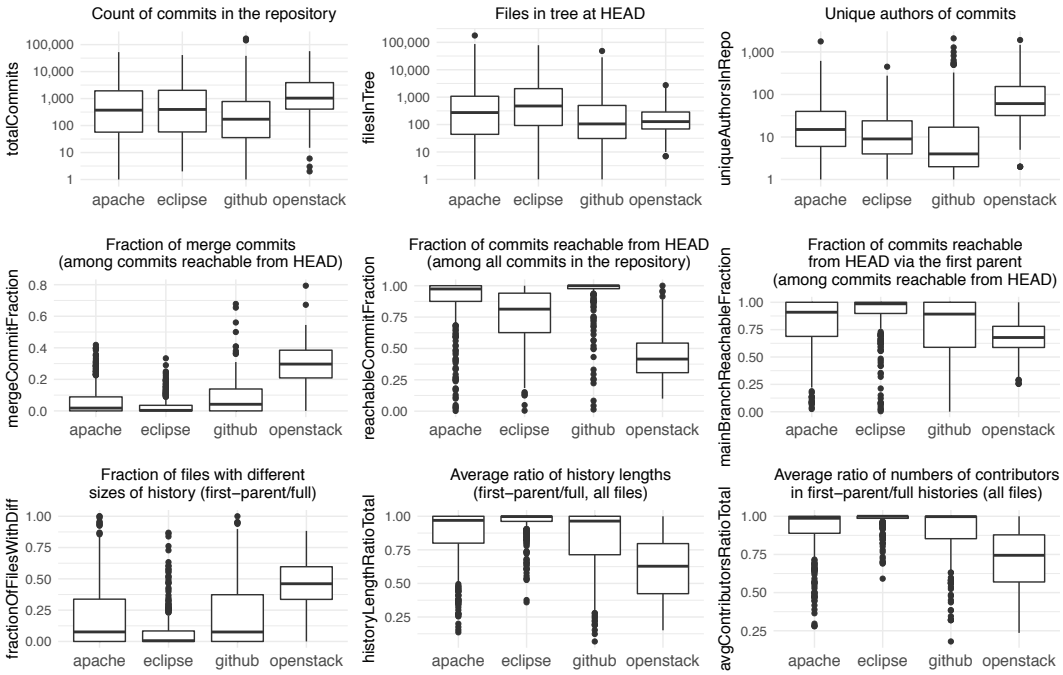

Figure 2: Comparison of descriptive metrics between repositories in different ecosystems

repositories is the highest of the four (the histories are the most similar; median ratio is 0.997) and the lowest for OpenStack (0.63). If we omit the files with identical first-parent and full histories and only consider the files with the difference, an ecosystem-wide median value for average ratio of history lengths in a repository ranges from 0.54 to 0.82 for OpenStack and Eclipse, respectively.

The values for the aforementioned metrics indicate that—in most of the repositories from all four ecosystems—the first-parent strategy leaves out a significant number of changes. The proportion of files affected by the difference and the scale of this difference are the lowest for Eclipse and highest for OpenStack.

One straightforward practical application of file histories is retrieval of contributors to a file. Developer tools with code viewing features, such as Github, display contributors to the current file in the user interface of file content display. Figure 2 displays the ratios of counts of version control user records in first-parent and full histories. While the ratio is close to 1 for most of the repositories in Eclipse, Github and Apache, there are quite a few repositories with large difference in each of these ecosystems. For most of the repositories in OpenStack, retrieval of histories via the first parent of the commit leaves out over 25% of contributor records (median value of the ratio is 0.74). Only considering the files with different histories naturally leads to an even stronger effect.

4.2.3 Summary. The analysis of the results of mining across over 1,400 repositories in the four subject ecosystems reveals that—in most of the considered repositories—the two strategies of mining file histories deliver different results (Figure 2, Table 1). The size of the difference for a typical repository varies across the four ecosystems and is the lowest for Eclipse and the highest for OpenStack. The size of the effect aligns with the proportion of the merge commits in repositories and reachability of commits (Figure 2). The difference between the histories does not only impact their quantitative metrics, but also distorts the observed numbers of contributors

to a file. We explore the impact of the mining technique on the more complex repository analysis techniques relying of file histories in the next research question (RQ2, Section 5).

5 RQ2: IMPORTANCE FOR APPLICATIONS

To evaluate importance of the mining approach for practice, we compare the performance of three prevalent techniques based on histories of changes: (i) recommendation of code reviewers [22], (ii) change recommendation [68], and (iii) defect prediction [41].

It is important to note that, while we compare the numbers of performance of the techniques, we do not perform statistical tests to assess the significance of the difference, because (i) we are merely demonstrating the existence of the difference for individual projects from our samples, rather than trying to generalize the results for a broader population of projects, and (ii) statistical tests, such as Mann-Whitney U, which are commonly used for this purpose, are not applicable in our case, because measures of performance of conceptually similar algorithms on related data from the same projects cannot be considered independent samples [50].

5.1 Code Reviewer Recommendation

5.1.1 Methodology. Recommendation of reviewers for code review has the goal of finding the most qualified reviewer for a new code change committed on a repository [22]. Such recommendation tools usually mine the change history information to identify the developer that is more expert on the piece of code impacted by the change under review. Thus, it represents a prominent example of usage of file histories to assist developers in routine tasks. In the last few years the reviewer recommendation algorithms have been adopted by industrial code review tools, such as Github [7], Gerrit [4] and Upsource [15].

We focus on two open source ecosystems that use Gerrit for code review — Eclipse [2] and OpenStack [12]. For each of the two Gerrit instances [3, 13], we extract the 100,000 most recent code reviews. We choose this number to have a sufficiently large dataset, which would, however, not include all of the reviews in the corresponding instances, but only two sets, similar in size, of the most recent reviews from each instance. We assess the impact of the file history retrieval method on accuracy of recommendations of code reviewers based on the history of changes. We perform the evaluation to find out whether the accuracy of recommendations changes depending on method of file history retrieval, but not to achieve maximum possible accuracy. Thus, we resort to a trivial reviewer recommendation algorithm, based on counts of developers' prior contributions to the files under review. We evaluate the recommendations by comparing a list of recommendations with actual reviewers of a changeset, as recorded in Gerrit. To assess the accuracy of recommendations, we use two commonwise metrics: Mean Reciprocal Rank (MRR) and top-k precision [66].

5.1.2 Results. Table 2 presents the results of evaluation of reviewer recommendation algorithm based on authors of past changes to files under review. We compare the accuracy of the algorithm between the two variations of input data: first-parent and full file histories. For each of the two ecosystems — Eclipse and OpenStack — we compare the accuracy numbers for 20 projects in each ecosystem. The selected projects are the most represented among 100,000 latest code reviews in the corresponding Gerrit instance. We report values of mean reciprocal rank and top-k accuracy for k in $\{1, 2, 3, 5, 10\}$ (average for all reviews in the project) for the recommendation lists based on both first-parent and full histories of files under review, and explore the difference between these values. To keep the table compact, we only report the 5 most active projects from each ecosystem individually, and aggregated values for the top 5 and top 20 projects. To illustrate the scale of difference between file histories, we also report average counts of commits in the union of histories of files under review, for both methods of retrieval of file histories, and the ratio of these numbers for the two methods.

For the Eclipse ecosystem, the difference in recommendation accuracy is subtle: MRR and all 5 top-k precision values only vary very slightly between first-parent and full histories consistently across all projects. For the OpenStack ecosystem the difference is slightly more pronounced. While MRR values only differ slightly, top-k precision values differ increasingly for higher values of k. This difference indicates that the full histories of files may include changes, made by future reviewers of a file, that are not present in the first-parent histories. The increase in the size of the effect with the increase of k suggests that authors of such changes are typically not the main contributors of the file, as they end up around k-th position in the list of past contributors sorted by numbers of contributions, thus starting to affect the top-k precision value for the corresponding k and higher.

The difference in the size of the effect betwen the two ecosystems can be explained by the fact that the difference between the full and first-parent histories is much higher for OpenStack repositories than for Eclipse. In OpenStack, the full histories of commits for all files under review, in union, contain on average 5.26 times as many changes as the first-parent histories. In Eclipse, the average difference between sizes of histories is under 20%. Such a low difference is unlikely to cause a large deviation of the two sorted contributor lists, and is thus not critical for the accuracy of reviewer recommendation based on history of changes. A large difference in OpenStack, however, noticeably impacts the accuracy of reviewer recommendation.

5.2 Change Recommendation

5.2.1 Methodology. Another example of usage of historical data to improve the user experience of development process is recommendation of changes, based on mining of association rules for changes to individual files. A practical application for this technique was described by Zimmermann *et al.* [68]: in particular, given a new code change as input, their technique suggests related changes that the developer might want to apply based on the files that frequently change with the modified file. We perform an experiment to assess the effect of using full histories of changes to a file, compared to using the first-parent histories.

The original design by Zimmermann [68] uses a set of changes from version control to infer file association rules. To apply the approach to our area of focus — difference in results of mining of individual files — we adapt the design of the original tool. We use two different approaches to infer the association rules from past changes, and evaluate their performance in predicting a change of

Table 2: Results of reviewer recommendation evaluation on projects from Eclipse and OpenStack

			MRR			Top 1 precision			Top 2 precision			Top 3 precision			Top 5 precision			Top 10 precision			Changes per review		
Project (OpenStack)	**Reviews**	**Reviews w/ diff**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**ratio**
openstack/tripleo-heat-templates	2,588	2,477	0.16	0.19	**0.03**	0.13	0.10	**-0.02**	0.17	0.18	**0.01**	0.19	0.23	**0.03**	0.21	0.30	**0.09**	0.22	0.38	**0.16**	7.75	76.46	**9.87**
openstack/releases	2,107	1,990	0.16	0.21	**0.05**	0.10	0.10	**0.00**	0.19	0.28	**0.09**	0.22	0.32	**0.11**	0.24	0.35	**0.11**	0.24	0.36	**0.12**	9.81	26.56	**2.71**
openstack/cinder	2,073	2,049	0.02	0.02	**0.00**	0.01	0.01	**0.00**	0.02	0.02	**0.00**	0.02	0.02	**0.00**	0.02	0.03	**0.01**	0.03	0.04	**0.02**	17.41	157.62	**9.05**
openstack/requirements	1,786	1,771	0.05	0.07	**0.03**	0.01	0.01	**0.01**	0.02	0.02	**0.00**	0.02	0.04	**0.02**	0.08	0.12	**0.04**	0.14	0.29	**0.15**	255.40	1,534.32	**6.01**
openstack-infra/zuul	1,367	1,329	0.10	0.13	**0.03**	0.05	0.04	**-0.01**	0.08	0.10	**0.02**	0.11	0.14	**0.03**	0.15	0.23	**0.08**	0.24	0.35	**0.12**	65.85	178.63	**2.71**
Total (OpenStack top 5)	**9,921**	**9,616**	**0.10**	**0.13**	0.03	**0.07**	**0.06**	**-0.01**	**0.10**	**0.13**	**0.03**	**0.12**	**0.16**	**0.04**	**0.15**	**0.21**	**0.06**	**0.17**	**0.29**	**0.11**	**62.79**	**359.34**	**5.72**
Total (OpenStack top 20)	**25,179**	**24,297**	**0.13**	**0.15**	0.02	**0.09**	**0.08**	**-0.01**	**0.14**	**0.15**	**0.01**	**0.16**	**0.19**	**0.03**	**0.19**	**0.24**	**0.05**	**0.21**	**0.31**	**0.10**	**40.05**	**210.50**	**5.26**

Project (Eclipse)	**Reviews**	**Reviews w/ diff**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**ratio**
papyrus/org.eclipse.papyrus	4,884	3,775	0.26	0.26	**0.00**	0.18	0.18	**0.00**	0.27	0.26	**0.00**	0.31	0.31	**0.00**	0.35	0.36	**0.00**	0.41	0.42	**0.02**	48.10	68.46	**1.42**
jgit/jgit	4,842	4,197	0.42	0.42	**0.01**	0.30	0.31	**0.00**	0.43	0.43	**0.00**	0.50	0.51	**0.00**	0.58	0.59	**0.01**	0.64	0.67	**0.03**	44.74	78.38	**1.75**
linuxtools/org.eclipse.linuxtools	4,616	2,910	0.42	0.41	**-0.01**	0.29	0.28	**-0.02**	0.46	0.44	**-0.02**	0.55	0.53	**-0.02**	0.59	0.59	**0.00**	0.62	0.62	**0.01**	42.46	56.11	**1.32**
egit/egit	4,587	4,028	0.35	0.36	**0.01**	0.23	0.23	**-0.01**	0.33	0.33	**0.00**	0.40	0.41	**0.01**	0.51	0.56	**0.05**	0.64	0.67	**0.03**	98.74	140.66	**1.42**
platform/eclipse.platform.ui	4,083	2,486	0.26	0.23	**-0.03**	0.15	0.12	**-0.03**	0.25	0.21	**-0.04**	0.32	0.27	**-0.04**	0.40	0.36	**-0.04**	0.50	0.48	**-0.02**	117.53	145.60	**1.24**
Total (Eclipse top 5)	**23,012**	**17,396**	**0.34**	**0.34**	0.00	**0.24**	**0.22**	**-0.01**	**0.35**	**0.43**	**-0.01**	**0.42**	**0.41**	**-0.01**	**0.49**	**0.50**	**0.01**	**0.56**	**0.57**	**0.01**	**68.67**	**96.15**	**1.40**
Total (Eclipse top 20)	**55,620**	**36,795**	**0.37**	**0.37**	0.00	**0.25**	**0.24**	**-0.01**	**0.38**	**0.37**	**0.00**	**0.46**	**0.46**	**0.00**	**0.55**	**0.55**	**0.01**	**0.62**	**0.63**	**0.01**	**103.37**	**122.57**	**1.19**
Total (Eclipse top 20 + OpenStack top 20)	**80,799**	**61,092**	**0.30**	**0.30**	0.01	**0.20**	**0.19**	**-0.01**	**0.30**	**0.30**	**0.00**	**0.37**	**0.37**	**0.01**	**0.43**	**0.45**	**0.02**	**0.49**	**0.53**	**0.04**	**83.63**	**149.97**	**2.45**

a given file in the commit. The common part of the two algorithms is their context and input data.

Below we use the notation $F \in C$ to denote that the commit C affects the file F, i.e., the file is *modified* in it. Both algorithms try to predict the change to the file $F_{current}$ in the current commit $C_{current}$.

In the first algorithm (*"single-file"*), we infer the association rules from the commits that affected this file in the past: $\{C : C_i \textit{ affects } F_{current}\}$.

In the second algorithm (*"other-files"*) we infer the rules from the commits that affected every of the other files in the commit C: $\{C : C_i \textit{ affects } F_k, F_k \in \{F : F_i \in C_{current}\} \setminus \{F_{current}\}\}$

We evaluate both algorithms and compare their performance depending on the type of file histories in use: first-parent or full. Intuitively, a full history contains more information about past changes, which allows one to infer more association rules, some of which are more likely to match the current change. However, since we are interested in difference between the two methods of file history mining in terms of their capability to provide information to infer the association rules from past changes (quantitative difference is explored by $\mathbf{RQ}_1$), we make adjustments to account for the difference in sizes of the two histories: (1) We only include the predictions where the histories are different (otherwise they perform equally); (2) We trim the full history to the size of the first-parent history, taking the most recent commits into account (to infer rules from the same number of commits); (3) We sort the rules by *support* and trim the larger ruleset to match size with smaller (to account for the possible difference in the number of rules).

In addition, to bring the algorithm closer to a practical approach, we only generate the predictions when the following (empirically derived) criteria are met: (i) We do not consider large commits with more than 10 changes when infering the rules (they rarely represent meaningful changes); (ii) We only execute the algorithm when the smaller of two histories contains at least 5 commits (otherwise the history is too trivial to learn meaningful rules from); (iii) We use at most 100 most recent commits from the history to infer the rules (to capture the current state of logical coupling between files); (iv) We use at most 10 rules with the highest support values (recommendation lists are finite and small in practical contexts). We use an open source implementation [8] of the Apriori algorithm [17] to infer the association rules. A formal definition of the concept of an association rule is available in literature [68].

Imitating a real-life context of recommendation of changes, we derive a recommendation set from the set of rules as a union of all one-item sets of *heads* of the rules, *bodies* of which are fully contained among the other files changed in the commit C.

We use a random sample of 10 projects from each of Apache and Eclipse ecosystems for evaluation. We select these projects to align the sample with the sample used for defect prediction (Section 5.3.1), for which the choice of target systems is restricted by the issue tracker in use.

5.2.2 Results. Table 3 presents the results of evaluation of change recommendation. An "event" corresponds to a single case when association rules have been successfully generated using both first-parent and full histories. In some events, none of the rules match the set of changes in the commit, so no recommendations can be produced. Rate of such events is presented in the last column of Table 3. The setup and the algoritms are described in detail in Section 5.2.1. In the context of this study, we are interested in comparing the performance of full and first-parent histories as the input data for each of the two algorithms.

The "other files" algorithm, which uses histories of the other files in the commit to produce the association rules, produces more events and generates more recommendations on sampled repositories from both Apache and Eclipse. However, only under 20% of

Table 3: Comparison of performance metrics for change recommendation, by mining approach and ecosystem

Apache

Algorithm	History type	Events count	Recommendations (average)	Rules (average)	Success rate	Failure rate	No prediction rate
Single file	Full	8780	0.908	8.848	0.496	0.029	0.474
	First parent	8780	0.858	8.848	0.474	0.031	0.496
Other files	Full	19072	1.565	8.003	0.184	0.528	0.288
	First parent	19072	1.485	8.003	0.170	0.527	0.303

Eclipse

Algorithm	History type	Events count	Recommendations (average)	Rules (average)	Success rate	Failure rate	No prediction rate
Single file	Full	2721	0.772	7.922	0.491	0.016	0.493
	First parent	2721	0.729	7.922	0.483	0.016	0.501
Other files	Full	6661	1.324	7.540	0.163	0.514	0.323
	First parent	6661	1.299	7.540	0.152	0.526	0.322

Table 4: Comparison of performance metrics for defect prediction, by mining approach and ecosystem

		Precision			Recall			F-Measure			AUC-ROC		
Project (Apache)	**% Defects**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**
calcite	42	0.57	0.57	**0.00**	0.65	0.65	**0.00**	0.61	0.61	**0.00**	0.66	0.67	**0.01**
falcon	55	0.61	0.63	**0.02**	0.63	0.63	**0.00**	0.62	0.63	**0.01**	0.63	0.74	**0.11**
james	41	0.48	0.49	**0.01**	0.55	0.58	**0.03**	0.51	0.58	**0.07**	0.55	0.60	**0.00**
lens	63	0.64	0.68	**0.04**	0.69	0.69	**0.00**	0.66	0.68	**0.02**	0.72	0.76	**0.02**
lucy-clownfish	32	0.59	0.59	**0.00**	0.61	0.65	**0.04**	0.60	0.62	**0.02**	0.67	0.67	**0.03**
madlib	35	0.61	0.62	**0.01**	0.67	0.67	**0.00**	0.64	0.64	**0.00**	0.62	0.71	**0.02**
predictionio	44	0.37	0.44	**0.07**	0.48	0.56	**0.08**	0.42	0.49	**0.07**	0.51	0.59	**0.02**
qpid-proton	37	0.55	0.55	**0.00**	0.61	0.61	**0.00**	0.58	0.58	**0.00**	0.59	0.61	**0.02**
ranger	40	0.66	0.67	**0.01**	0.69	0.73	**0.04**	0.67	0.70	**0.03**	0.66	0.67	**0.02**
reef	33	0.60	0.61	**0.01**	0.62	0.62	**0.00**	0.61	0.61	**0.00**	0.72	0.74	**0.01**
Overall (Apache)	-	**0.58**	**0.60**	**0.02**	**0.64**	**0.67**	**0.03**	**0.62**	**0.65**	**0.03**	**0.64**	**0.66**	**0.02**
Project (Eclipse)	**% Defects**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**
acceleo	39	0.64	0.64	**0.00**	0.59	0.63	**0.04**	0.61	0.63	**0.02**	0.67	0.69	**0.02**
chemclipse	33	0.62	0.64	**0.02**	0.61	0.61	**0.00**	0.61	0.62	**0.01**	0.59	0.59	**0.00**
efxclipse	46	0.55	0.55	**0.00**	0.55	0.57	**0.02**	0.55	0.56	**0.01**	0.62	0.65	**0.03**
epp	53	0.72	0.74	**0.02**	0.57	0.62	**0.05**	0.64	0.67	**0.03**	0.67	0.68	**0.01**
hudson	44	0.61	0.64	**0.03**	0.73	0.73	**0.00**	0.66	0.68	**0.02**	0.63	0.68	**0.05**
platform.releng	32	0.66	0.66	**0.00**	0.58	0.58	**0.00**	0.62	0.62	**0.00**	0.64	0.64	**0.00**
recommenders	18	0.65	0.65	**0.00**	0.55	0.59	**0.04**	0.60	0.62	**0.02**	0.55	0.60	**0.05**
swtbot	24	0.60	0.60	**0.00**	0.58	0.59	**0.01**	0.60	0.60	**0.00**	0.60	0.60	**0.00**
tcf	31	0.68	0.69	**0.01**	0.72	0.72	**0.00**	0.70	0.70	**0.00**	0.70	0.75	**0.05**
downloads	35	0.73	0.76	**0.03**	0.55	0.58	**0.03**	0.63	0.66	**0.03**	0.69	0.74	**0.05**
Overall (Eclipse)	-	**0.66**	**0.69**	**0.03**	**0.59**	**0.63**	**0.04**	**0.63**	**0.66**	**0.03**	**0.64**	**0.67**	**0.03**
Overall (Apache + Eclipse)	-	**0.65**	**0.67**	**0.02**	**0.61**	**0.61**	**0.00**	**0.63**	**0.66**	**0.03**	**0.61**	**0.65**	**0.04**

these recommendations are successful. The "single file" algorithm, using rules inferred from the history of a single file, generates less recommendations, of which around a half are successful. The full histories perform slightly better as the input data for both algorithms in both ecosystems: depending on the ecosystem and algorithm, they yield 5 – 8% more recommendations, which are 2 – 8% more likely to be successful (i.e., to match an actual change).

5.3 Defect Prediction

5.3.1 Methodology. The last application aims at recommending developers the files that are more likely to contain defects [39].

In our study, we take into account the Basic Code Change Model (BCCM) prediction model devised by Hassan [41], which is based on the entropy of changes applied by developers in a certain time window and is computed exploiting the concept of Shannon entropy [55]. We consider the model by Hassan [41] rather than more recent ones (e.g., [36]) since (i) we are only interested in models relying on change history information, thus we cannot consider models based on product metrics [24] and (ii) BCCM performs similarly to others proposed in literature, thus still being representative of the field [39]. Our conjecture is that the quantification of the entropy may be more precise when considering the full history of files rather than the single-parent case.

We perform a replication of the study by Hassan [41], considering the BCCM. It splits the change history of a software project into three-month time periods, and adopts a three-month sliding window to train and test a Logistic Regression classifier (that is, the one adopted in the original study by Hassan [41]). In other words, starting from the beginning of the history, it computes the entropy of changes on the files available in a time window and uses such data to train a classifier that predicts the defectiveness of files in the subsequent time window. The process is then repeated until the end of the history. Given the nature of the model, we evaluate its performance in the two scenarios, i.e., single-parent vs full, by considering the mean F-Measure and AUC-ROC [21] achieved when run on each time window.

In this case, we run the experiment over 20 randomly sampled systems belonging to the Apache and Eclipse ecosystems considered in the study (their names are reported in the online appendix [11]). We do so because, to extract actual defects composing the ground truth on which compare the results of the model against, we rely on a issue tracker mining tool based on BugZilla [1], and therefore we limit this study to the ecosystems that use it. Our ground truth is represented by post-release defects marked as solved by developers on the issue tracker. As such, we consider as defective all those files that have encountered a problem in a certain time window, according to the timestamp of the bug report.

5.3.2 Results. Table 4 reports the results achieved when considering the problem of predicting defects using the BCCM model defined by Hassan [41] in case of single-parent or full history configuration. Overall, the delta between the two approaches is not large for any of the evaluation metrics considered. For instance, the F-Measure is only 3% higher in both Apache and Eclipse, respectively. Thus, in a first glance we can claim that considering branches does not improve the defect prediction performance. Nevertheless, finer observations reveal two aspects that it is worth to highlight. In the first place, the model built using the single-parent strategy has *always* performance equal or lower than the one that considers the full history: thus, taking into account the full set of changes is *not detrimental* in the case of defect prediction. Conversely, the full history approach provides benefits in 65% and 80% of the cases when considering F-Measure and AUC-ROC, respectively, meaning that defect prediction performance can generally gain when considering all the changes rather than a subset of them. It is interesting to note the differences are higher when considering the AUC-ROC, i.e., the evaluation metric suggested by previous work [30, 33] to evaluate defect prediction models: specifically, it is up to 11% higher (on the repository of the Apache Falcon project), showing that considering the full history of files can provide strong improvements. Therefore,

based on our findings, we can finally claim that taking the whole history into account when building history-based defect prediction models might provide important benefits in terms of performance.

6 LIMITATIONS AND THREATS TO VALIDITY

A number of limitations affect the results of our study, and pose potential threats to its validity.

(1) When designing the experiments to compare performance, we tried to follow the original approaches as closely as possible, but it was not always possible completely (e.g. with change prediction), thus results on the original approaches may differ.
(2) While we used a diverse population of projects from four independent ecosystems, it is not clear whether and how our results can be generalized further.
(3) Our analysis included an extensive technical work, although we tested it carefully and under several scenarios, we cannot guarantee that code is bug-free. However, we make the code available in the online appendix [11].

7 DISCUSSION AND IMPLICATIONS

7.1 Discussion

Our results highlight several important aspects regarding the choice of the mining strategy.

The technical details of mining can significantly impact the quantitative properties of the retrieved data. In case of our study, such properties are the sizes of file histories and numbers of contributors: In over 2 million files from our composite sample of four ecosystems, the resulting histories differ for 19% of files.

The size of the difference varies across projects and ecosystems. In OpenStack projects, the first-parent and full histories are different for 55% of files, while in Eclipse projects we observe the difference for only 14% of files.

The size of the difference is associated with other properties of a repository. In our case, the ecosystem with the highest difference (i.e., OpenStack) also demonstrates the highest merging activity, most of which can be attributed to code review: after a successful code review in OpenStack, changes are merged back into the trunk by a bot. Thus, high proportion of merges and their impact on file histories can be considered a consequence of their development workflow – namely, code review. Notably, while a similar workflow with code reviews performed with Gerrit is also present in Eclipse, this ecosystem displays the lowest degree of difference: changes in Eclipse are typically rebased, but not merged, after code review. The impact of this particular factor on the results of repository mining deserves a deeper analysis in future work.

The difference in results of mining can influence the performance of techniques based on file histories: reviewer recommendation, change recommendation, and defect prediction. For all of the tested approaches, full histories, when used as input data, perform at least not worse than first-parent histories, in most cases yielding a slight increase in performance.

Some applications are more sensitive to quality of input data than others. In our case, for reviewer recommendation in OpenStack full histories provide better accuracy, especially when longer recommendation lists are evaluated (top-5 and top-10 accuracy). OpenStack also happens to be the ecosystem where the difference between the first-parent and full histories is the highest. At the same time, Eclipse projects show the smallest difference between the first-parent and full histories among all four target ecosystems, and this small difference appears insufficient to influence accuracy of reviewer recommendation.

7.2 Implications

Considering the points above, we see several important implications of our results.

(1) **Software engineering researchers should be aware of the possible impact of the mining technique on the results.** Our study demonstrates that omitting changes outside the main branch during mining of file histories significantly impacts the results of mining, which often leads to a slight decrease in performance of methods that use file histories as input data.
(2) **The choice of the mining technique should account for the context of the mining task.** While using full file histories ensures a better performance, in most cases the difference is only marginal. In many contexts, a small increase in performance may not justify dedicating the extra effort to more precise mining. While we suggest using precise mining methods where possible—and provide a tool for doing that—in many contexts it is not essential.
(3) **Researchers should report the technical details of mining.** We suggest that techniques of repository mining should be described in more detail by authors of MSR studies, as not providing details complicates reproducibility of studies, and oversimplifying the mining potentially undermines performance of methods and validity of studies.

8 CONCLUSION

With the study presented in this paper, we make the following main contributions:

- The first demonstration of the importance of careful handling of merge commits and changes from outside the main branch for calculation of file histories;
- Analysis of impact of a strategy of mining file histories on performance of three techniques relying on them as input data;
- A tool for efficient mining of precise file histories in Git [11].

Our results cover the underrepresented topic of technical details of mining the repositories for file histories, and open opportunities for deeper analysis of associated factors, such as topology of change histories. We hope that this study will inspire other researchers in MSR to apply a more detailed approach to mining, where it is feasible, and to report the technical details of mining to ensure clarity and reproducibility of the studies.

ACKNOWLEDGMENTS

A. Bacchelli and F. Palomba gratefully acknowledge the support of the SNF Project No. PP00P2_170529.

REFERENCES

[1] Bugzilla. https://www.bugzilla.org/. Accessed: 2018-04-26.
[2] Eclipse - the eclipse foundation open source community website. https://www.eclipse.org/. Accessed: 2018-04-16.
[3] Eclipse code review. https://git.eclipse.org/r/. Accessed: 2018-04-16.
[4] Gerrit code review. https://www.gerritcodereview.com/. Accessed: 2018-04-16.
[5] git - find all the direct descendants of a given commit. https://stackoverflow.com/questions/27960605/find-all-the-direct-descendants-of-a-given-commit#27962018. Accessed: 2018-04-24.
[6] git-log(1) manual page. https://mirrors.edge.kernel.org/pub/software/scm/git/docs/git-log.html. Accessed: 2018-04-24.
[7] Github. https://github.com. Accessed: 2018-04-22.
[8] A java implementation of the apriori algorithm for finding frequent item sets and (optionally) generating association rules. https://github.com/michael-rapp/Apriori/. Accessed: 2018-04-22.
[9] Jgit. https://www.eclipse.org/jgit/. Accessed: 2018-04-09.
[10] Lkml: Linus torvalds: Re: git mv (was re: [git pull] acpi & suspend patches for 2.6.29-rc0). https://lkml.org/lkml/2009/1/9/323. Accessed: 2018-04-24.
[11] Mining File Histories: Should We Consider Branches? - Online Appendix. https://github.com/vovak/branches. Accessed: 2018-07-24.
[12] Open source software for creating private and public clouds. https://www.openstack.org/. Accessed: 2018-04-16.
[13] Openstack code review. https://review.openstack.org/. Accessed: 2018-04-16.
[14] Quora Statistics. https://www.quora.com/What-are-the-most-popular-distributed-version-control-systems. Accessed: 2018-04-22.
[15] Upsource. https://www.jetbrains.com/upsource/. Accessed: 2018-04-22.
[16] Why sqlite does not use git. https://sqlite.org/whynotgit.html. Accessed: 2018-04-18.
[17] R. Agrawal et al. Fast algorithms for mining association rules.
[18] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen. Code smells for model-view-controller architectures. *Empirical Software Engineering*, pages 1–37, 2017.
[19] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein. Streamed lines: Branching patterns for parallel software development. 1998.
[20] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 375–384. ACM, 2010.
[21] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.
[22] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 931–940. IEEE, 2013.
[23] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *International Conference on Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2012.
[24] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.
[25] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. The limited impact of individual developer data on software defect prediction. *Empirical Software Engineering*, 18(3):478–505, 2013.
[26] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.
[27] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE, 2009.
[28] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 45. ACM, 2012.
[29] C. Bird, T. Zimmermann, and A. Teterev. A theory of branches as goals and virtual teams. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 53–56. ACM, 2011.
[30] D. Bowes, T. Hall, and D. Gray. Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, pages 109–118. ACM, 2012.
[31] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, pages 322–333. ACM, 2014.
[32] J. Buffenbarger and K. Gruell. A branching/merging strtegy for parallel software development. In *International Symposium on Software Configuration Management*, pages 86–99. Springer, 1999.
[33] C. Catal. Performance evaluation metrics for software fault prediction studies. *Acta Polytechnica Hungarica*, 9(4):193–206, 2012.
[34] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman. Developer-related factors in change prediction: an empirical assessment. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 186–195. IEEE Press, 2017.
[35] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.
[36] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2018.
[37] G. Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.
[38] M. Greiler, K. Herzig, and J. Czerwonka. Code ownership and software quality: a replication study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 2–12. IEEE Press, 2015.
[39] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.
[40] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.
[41] A. E. Hassan. Predicting faults using the complexity of code changes. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 78–88. IEEE, 2009.
[42] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 141–150. IEEE, 2009.
[43] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software: Evolution and Process*, 19(2):77–131, 2007.
[44] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360. ACM, 2011.
[45] E. Kocaguneli, T. Menzies, and J. W. Keung. On the value of ensemble effort estimation. *IEEE Transactions on Software Engineering*, 38(6):1403–1416, 2012.
[46] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.
[47] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.
[48] H. M. Michaud, D. T. Guarnera, M. L. Collard, and J. I. Maletic. Recovering commit branch of origin from github repositories. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 290–300. IEEE, 2016.
[49] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 11–20. IEEE, 2009.
[50] N. Nachar et al. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1):13–20, 2008.
[51] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 466–476. ACM, 2013.
[52] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, 2017.
[53] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 2017.
[54] M. M. Rahman, C. K. Roy, and J. A. Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 222–231. IEEE, 2016.
[55] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.
[56] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 301–310. ACM, 2012.
[57] D. Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.
[58] S. W. Thomas. Mining software repositories using topic models. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1138–1139. ACM, 2011.
[59] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida. Improving code review effectiveness through reviewer recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 119–122. ACM, 2014.

[60] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050. ACM, 2016.

[61] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150. IEEE, 2015.

[62] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15. ACM, 2016.

[63] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[64] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.

[65] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[66] E. M. Voorhees et al. The trec-8 question answering track report. In *Trec*, volume 99, pages 77–82, 1999.

[67] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.

[68] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.