Received: 2 December 2016 | Revised: 5 April 2019 | Accepted: 26 April 2019
DOI: 10.1002/smr.2181

WILEY Journal of Software: Evolution and Process

RESEARCH ARTICLE - EMPIRICAL

Watch out for this commit! A study of influential software changes

Daoyuan Li[1] | Li Li[1] | Dongsun Kim[1] | Tegawendé F. Bissyandé[1] | David Lo[2] | Yves Le Traon[1]

[1]Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg City, Luxembourg
[2]School of Information Systems, Singapore Management University, Singapore

Correspondence
Dongsun Kim and Tegawendé F. Bissyandé, Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg City, Luxembourg.
Email: dongsun.kim@uni.lu; tegawende.bissyande@uni.lu

Funding information
Fonds National de la Recherche Luxembourg, Grant/Award Number: C15/IS/10449467 and C15/IS/9964569

Abstract

One single code change can significantly influence a wide range of software systems and their users. For example, (a) adding a new feature can spread defects in several modules, while (b) changing an API method can improve the performance of all client programs. Unfortunately, developers often may not clearly know whether code changes are influential at commit time. This paper investigates influential software changes and proposes an approach to identify them immediately when they are applied. Our goals are to (a) identify existing influential changes (ICs) in software projects, (b) understand their characteristics, and (c) build a classification model of ICs to help developers find and address them early. We first conduct a post-mortem analysis to discover existing influential changes by using intuitions (eg, changes referred by other changes). Then, we re-categorize all identified changes through an open-card sorting process. Subsequently, we conduct a survey with about 100 developers to finalize a taxonomy. Finally, from our ground truth, we extract features, including metrics such as the complexity of changes and file centrality in co-change graphs to build machine learning classifiers. The experiment results show that our classification model with random samples achieves 86.8% precision, 74% recall, and 80.4% *F*-measure, respectively.

KEYWORDS

change prediction, change risk, influential change, software changes, software evolution

1 | INTRODUCTION

Current development practices heavily rely on version control systems to record and keep track of changes committed in project repositories. While many of the changes may be merely cosmetic or provide minor improvements, others have a wide and long-term influence on the entire system and related systems. Brudaru and Zeller[1] first illustrated examples of changes with long term-influence: (a) changing access privilege (ie, `private` → `public`), (b) changing kernel lock mechanism, and (c) forgetting to check a null return. If we can predict whether an incoming software change is influential or not—either positively or negatively—just after it is committed, it could significantly improve maintenance tasks (eg, easing debugging if a new test harness is added) and provide insights for recommendation systems (eg, code reviewers can focus on fewer changes).

The influence of a software change can, however, be hard to detect immediately since it often does not involve immediate effects to other software elements. Instead, it can constantly affect a large number of aspects in the software over time. Indeed, a software change can be influential not only inside and/or beyond the project repository (eg, new defects in the code base and new API calls from other programs), but also immediately and/or long after the changes have been applied. The following are examples of such influential changes:

Adding a new lock mechanism: `mutex-lock` features were introduced in Linux 2.6 to improve the safe execution of kernel critical code sections. However, after their introduction, the defect density of Linux suddenly increased for several years, largely contributed by erroneous usage of these features. Thus, the influence of the change was not limited to a specific set of modules. Rather, it was a system-wide problem.

Changing build configurations: a small change in configuration files may influence the entire program. In `Spring-framework`, a developer missed file inclusion options when migrating to a new build system (`*.aj` files were missing in `build.gradle`*†). This makes an impact since programs depending on the framework failed occasionally to work. The reason for this failure (missed file) was hard to pinpoint. This bug has been fixed after 6 months after its introduction.

Improving performance for a specific environment: `FastMath.floor()` method in *Apache Commons Math* had a problem with Android applications since it has a static code block that makes an application hang about 5 seconds at the first call. Fixing this issue improves the performance of all applications using the library.

Unfortunately, existing techniques are limited to revealing the *short- term impact* of a certain software change. The short-term impact indicates an immediate effect such as test case failure or coverage deviation. For example, dynamic change analysis techniques[2,3] leverage coverage metrics after running test cases. Differentiating coverage information before/after making a change shows how the change influences other program elements. Other approaches are based on similarity distances.[4,5] These firstly identify clusters of program elements frequently changed together or tightly coupled by analyzing revision histories. Then, they attempt to figure out the best-matching clusters for a given change. Developers can assume that program elements (eg, files or methods) in the cluster may be affected by the given change. Finally, change genealogy[6-8] approaches keep track of dependencies between subsequent changes and can capture some long-term impact of changes. However, it is limited to identifying source code entities and defect density. Overall, all the above techniques may not be successful in predicting a wide and long-term influence of software changes. This was unfortunately inevitable since those existing techniques focus only on explicit dependencies such as method calls.

1.1 | Definition scoping for influential changes

– As hinted by the examples above, an influential change can occur in a different part of the code base, and may relate to a variety of concepts (architecture, API, or algorithm implementations). We scope the definition of an influential change in this paper as a *source code change* which may induce significant (ie, visible) changes in the behaviour/evolution of any of the three entities in the ecosystem of project development: the code base, the end-users, or the developers. Thus, we consider the three following definitions in this paper:

- Code base - An influential change is any change that will *motivate/necessitate other changes* in the code base. Such a change affects the internal development processes of code review, testing, etc. Examples of such changes include domino changes that are necessary to perform collateral evolution when a popular API is invasively modified.
- User base - An influential change is any change that eventually *impacts software adoption or its use*. Such a change can occur in various circumstances, as part of specific effort to attract/retain users (eg, change that focus on improving library API usability) or inadvertently with user-undesired changes (eg, invasive change in a key feature).
- Developer team - An influential change is any change that *impacts the dynamics among developers*. Such a change can positively impact the development team by stimulating development (eg, a fix of a blocking bug[9]) require developers to focus on re-learning about how the code is built (eg, a major structural change) or address a standing issue that left the team in disagreement (eg, reverting a controversial change).

We recognize that these definitions are not exhaustive for identifying all changes that are potentially "influential." Nevertheless, within the scope of this paper, they allow for investigations on the risk of a code change, a topic that is still poorly studied in the literature.

1.2 | Study research questions

In this study, we are interested in investigating the following research questions:

(RQ1) What constitutes an influential software change? Are there developer-approved definitions/descriptions of influential software changes?
(RQ2) What metrics can be used to collect examples of influential software changes?
(RQ3) Can we build a classification model to identify influential software changes immediately after they are applied?

1.3 | Terminology

In this work, we use several terms that require explicit definitions to avoid confusion. The following are the most recurrent terms:

post-mortem analysis: we use this expression to refer to investigations of changes whose impact on the software ecosystem has become apparent (eg, a change that stops all discussion on a specific issue).

Anomaly: we use this term to refer to a metric whose value is out of the ordinary. It is determined via outlier detection in a series of measurements.

Change behaviour: we use this expression to refer to the observable metrics associated with a series of changes in a software project. The average delay between two commits (ie, the rhythm of commits) is part of an example of change behaviour observation.

* Commit `a681e574c3f732d3ac945a1dda4a640ce5514742`
† Bug report https://jira.spring.io/browse/SPR-9576

Prediction: we use this term to refer to a classification decision, based on machine learning methods, on whether a given commit will eventually be revealed as influential or not. The objective of a prediction is to "know" now whether a given change will become apparent later based on observations of previous changes.

1.4 | This study

To automatically figure out whether an incoming software change will be influential, we designed a classification technique based on machine learning. Since the technique requires labeled training instances, we first discovered existing influential changes in several open source projects in order to obtain baseline data. Specifically, we collected 48 272 code commits from 10 open source projects and did post-mortem analysis to identify influential changes. This analysis examined several aspects of influential changes such as controversial changes and breaking behaviors. In addition, we manually analyzed whether those changes actually have a long-term influence on revision histories. As a result, we could discover several influential changes from each subject. Note that change properties that present obvious patterns for recognizing a change as influential with high probabilities are post-mortem properties: the influence of the commit has been established at this point (eg, impact on user adoption is now visible). We further label these changes to build a category definition for influential software changes through an open-card sorting process. These categories are then validated by developers with experience in code review.

On the basis of the influential changes we discovered in the above study, we extracted feature vectors for machine-learning classification. These features include program structural metrics,[10] terms in change logs,[10] and co-change metrics.[11] Then, we built a classification model by leveraging machine learning algorithms such as Naïve Bayes[12,13] and Random Forest.[14] The prediction/classification is about determining when a change is being committed, whether it could be influential (later). We leverage machine learning to build a model based on features available at the time of prediction/classification, but that would not be used to measure "influence" per se. To evaluate the effectiveness of this technique, we conducted experiments that applied the technique to 10 projects. Experimental assessment results, with a representative randomly sampled a subset of our data, show that our classification model achieves good performance. Given a change being applied, our classifier is able to classify it as "influential" with a precision of 86.8%. The classifier is further able to identify 74% (recall) of the influential changes in the dataset. The overall F-measure of the classifier is thus 80.4%.

This paper makes the following contributions:

- Collection of influential software changes in popular open source projects.
- Definition of influential software change categories approved by the software development community.
- Correlation analysis of several program metrics and influential software changes.
- Accurate machine-learning classification model for influential software changes.

The remainder of this paper is organized as follows. After describing motivating examples in Section 2, we present our study results of post-mortem analysis for discovering influential changes in Section 3. Section 4 provides our design of a classification model for influential changes together with a list of features extracted from software changes. In addition, the section reports the evaluation result of experiments in which we applied the classification model to open source projects. Section 5 discusses the limitations of our work as well as threats to validity. After surveying the related work in Section 6, we conclude with directions for future research in Section 7.

2 | MOTIVATING EXAMPLES

Among code changes committed by developers, a specific subset of them may have an extensive impact on the software project. These changes are influential not only to internal modules but also other external projects. The influence temporally varies from an immediate impact to a long-term effect. However, those changes are often recognized in hindsight.

To motivate our study, we consider influential change examples identified from the Linux kernel project.

Linux is an appropriate subject as several changes in the kernel have been influential. These changes are already highlighted in the literature[15,16] as their long-term impact started to be noticed. In this section, we present four different examples of influential changes in the Linux kernel and their impact.

2.1 | Collateral evolution

In the Linux kernel, since driver code—which makes up over 70% of the source code—is heavily dependent on the rest of the OS, any change in the interfaces exported by the kernel and driver support libraries can trigger a large number of adjustments in the dependent drivers.[17]

Such adjustments, known as collateral evolution, can unfortunately be challenging to implement correctly. Starting with Linux 2.5.4, the USB library function *usb_submit_urb* (which implements message passing) takes a second argument for explicitly specifying the context (which was previously inferred in the function definition). The argument can take one of three values: GFP_KERNEL (no constraints), GFP_ATOMIC (blocking is not allowed), or GFP_NOIO (blocking is allowed but not I/O). Developers using this USB library must then check their own code to understand which context it should be as in the example of Figure 1.

```
spin_lock_irqsave(&as->lock, flags);
if (!usbin_retire_desc(u, urb) &&
    u->flags & FLG_RUNNING &&
    !usbin_prepare_desc(u, urb) &&
- (suret = usb_submit_urb(urb)) == 0) {
+ (suret = usb_submit_urb(urb, GFP_ATOMIC)) == 0) {
  u->flags |= mask;
} else {
  u->flags &= ~(mask | FLG_RUNNING);
  wake_up(&u->dma.wait);
  printk(KERN_DEBUG "...", suret);
}
spin_unlock_irqrestore(&as->lock, flags);
```

FIGURE 1 Code patch for adaption to the new definition of *usb_submit_urb*. In this case, when the API function is called, locks are held so the programmer must use GFP_ATOMIC to avoid blocking. Its influence was propagated to most drivers using this library and mostly resulted in defects

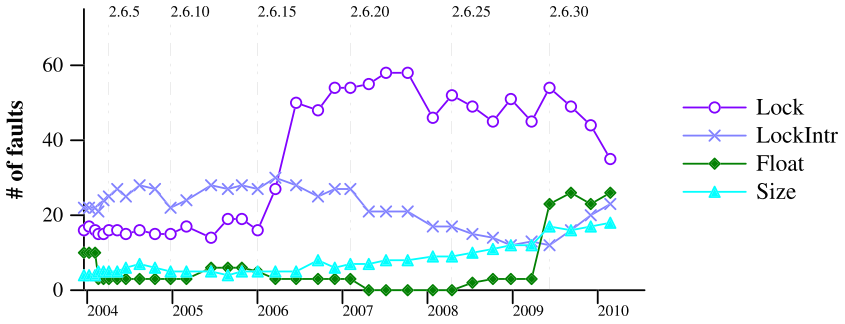

FIGURE 2 Evolution of faults in Linux 2.6 kernel versions for *Lock*, *LockIntr*, *Float*, and *Size* fault categories (see Palix et al[15]). Faults relevant to *Lock* suddenly increased after Version 2.6.16 while other types of faults gradually decreased. In the version, a feature for Lock was replaced and it was influential to many of kernel functions

This leads to bugs that keep occurring. A study by Pallix et al[15] has reported that because of the complexity of the conditions governing the choice of the new argument for *usb_submit_urb*, 71 of the 158 calls to this function were initially transformed incorrectly to use GFP_KERNEL instead of GFP_ATOMIC.

This change is interesting and constantly influential to a large portion of the kernel, as its real impact could only be predicted if the analysis took into account the semantics of the change. However, the extent of influences made by the change is difficult to detect immediately after the commit time since existing techniques[2-5] focus only on the short-term impact.

2.2 | Feature replacement

In general, the number of entries in each fault category (eg, *NULL* or *Lock*) decreases over time in the Linux code base.[15] In Linux 2.6 however, as illustrated in Figure 2, there are some versions in which we can see a sudden rise in the number of faults. This was the case of faults in the Lock[‡] category in Linux 2.6.16 because of a replacement of functionality implementation. In Linux 2.6.16, the functions *mutex_lock* and *mutex_unlock* were introduced to replace mutex-like occurrences of the semaphore functions *down* and *up*. The study of Palix et al again revealed that nine of the 11 Lock faults introduced in Linux 2.6.16 and 23 of the 25 Lock faults introduced in Linux 2.6.17 were in the use of *mutex_lock*.

If the replacement is identified earlier as an influential change to the most of kernel components (and other applications), it may prevent the defects from recurring everywhere since the change is likely to be an API change.[19,20] The developer who committed the new feature did not realize the influence and thus, there was no early heads-up for other developers.

2.3 | Revolutionary feature

An obvious influential change may consist in providing an implementation of a completely new feature, eg, in the form of an API function. In the Linux kernel repository, Git commit `9ac7849e` introduced device resource management API for device drivers. Known as the *devm* functions, the API provides memory management primitives for replacing *kzalloc* functions. This code change is a typical example of influential change with a long-term impact. As depicted in Figure 3, this change has first gone unnoticed before more and more people started using *devm* instead of *kzalloc*. Had the developers recognized this change as highly influential, they could have examine the potential effectiveness of *devm* earlier.

2.4 | Fixes of controversial/popular issues

Some issues in software projects can be intensively discussed or commented longer than others. Code changes that fix them will be influential for the project. The characteristics of a controversial/popular issue are that its resolution is of interest for a large number of developers, and it takes more time to resolve them than the average time-to-fix delay. Thus, we consider that an issue report which is commented on average more than other issues and is fixed very long after it is opened, is about a controversial/popular issue. In Linux, Git commit `bfd36103` resolved Bug #16691 which remained unresolved in the bug tracking system for 9 months and was commented about 150 times.

[‡]To avoid *Lock/LockIntr* faults, release acquired locks, restore disable interrupts, and do not double acquire locks.[15,18]

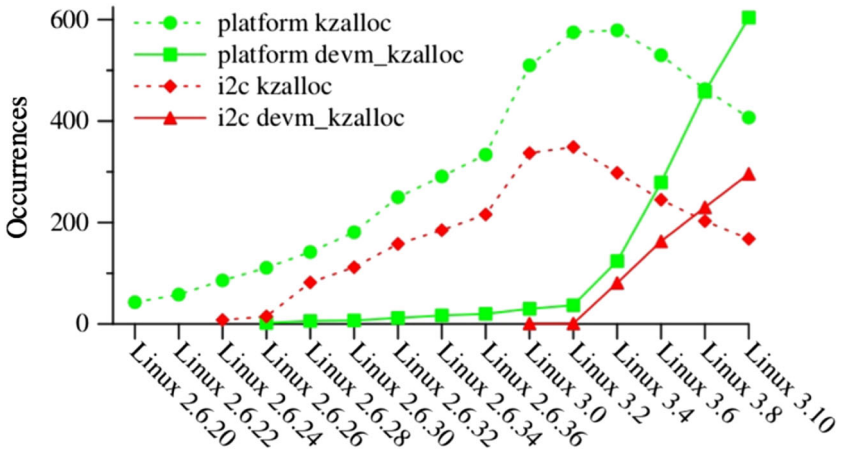

FIGURE 3 Usage of memory allocation primitives in Linux kernel (See Lawall[21]). *kzalloc* is the traditional API for memory allocation, before managed memory (*devm*) was introduced in Linux

TABLE 1 Observational study subjects. Data reflect the state of repositories as of 26 January 2015

Project Name	Description	# Files	# Commits	# Developers	# Issues	# Resolved Issues
`Commons-codec`	General encoding/decoding algorithms	635	1,424	24	195	177
`Commons-collections`	Extension of the Java Collections Framework	2983	2722	47	542	512
`Commons-compress`	Library for working with file compression	619	1716	24	308	272
`Commons-csv`	Extension of the Java Collections Framework	141	956	18	147	119
`Commons-io`	Collection of utilities for CSV file reading/writing	631	1718	33	454	365
`Commons-lang`	Extra-functionality for java.lang	1294	4103	46	1073	933
`Commons-math`	Mathematics and statistics components	4582	5496	37	1194	1085
`Spring-framework`	Application framework for the Java platform	19 721	9748	153	3500	2632
`Storm`	Distributed real-time computation system	2038	3534	189	637	321
`Wildfly`	aka JBoss Application Server	31 699	16 855	307	3710	2993
`Total`		64 388	48 272	878	11 760	9409

3 | POST-MORTEM ANALYSIS FOR ICs

In this study, we focus on systematically discovering influential changes. Although the motivating examples described in Section 2 show some intuitions on influential changes, it is necessary to reveal a larger view to figure out the characteristics of these changes. Therefore, we collected 48 272 changes from 10 popular open-source projects and conducted an observational study.

Since there are too many changes in software repositories and it is not possible for us to inspect all, we get a set of changes that are likely to have a higher density of influential changes. We are able to get this set by leveraging several intuitions obtained from examples described in Section 2.

The study design basically addressed three different criteria to discover influential changes: (a) popular changes in the sense that they have been somehow noticed by other developers and users, (b) anomalies in change behaviors, and (c) changes that are related to controversial/popular issues. These criteria are designed to conduct post-mortem analysis and represent how people can recognize influential changes in hindsight.

For changes in these categories, we manually examine them using the following procedure:

- First of all, the authors of this article ask themselves individually whether a change is really influential. They manually verify that the assumptions behind the specific criteria used to identify a change are supported.
- Then we cross-check the answers to reach a consensus among the authors.
- Afterwards, we double check that these changes are really influential in the eyes of developers by doing card sorting and surveying professional developers.

3.1 | Data collection

The experiment subjects in this study are shown in Table 1. The 10 popular projects were considered since they have a sufficient number of changes in their revision histories. In addition, these projects stably maintained their issue tracking systems so that we could keep track of how developers discussed to make software changes.

For each subject, we collected all available change data (patches and relevant files information) as well as commit metadata (change date and author details) from the source code repository. Additionally, issue reports from the corresponding issue tracking system were collected together. We further mined issue linking information from commit messages and issue reports wherever possible: eg, many commit messages explicitly refer to the unique ID of the issue they are addressing, whether a bug or a feature request.

TABLE 2 Statistics of identified influential changes related to controversial/popular issues

Project Name	# Changes Linked to Controversial/Popular Issues	# Influential Changes
Commons-codec	26	3
Commons-collections	12	8
Commons-compress	7	4
Commons-csv	5	5
Commons-io	10	0
Commons-lang	29	15
Commons-math	38	8
Spring-framework	53	42
Storm	40	3
Wildfly	20	18
Total	240	106

Raw and processed data collected in this work are made available in the following project page: https://github.com/serval-snt-uni-lu/influential-changes

3.2 | Systematic analysis

To systematically discover potential influential changes among the changes collected from the subject projects, we propose to build on common intuitions about how a single change can be influential in the development of a software project.

3.2.1 | Changes that address controversial/popular issues

In software projects, developers use issue tracking systems to track and fix bugs and for planning future improvements. When an issue is reported, developers and/or users may provide insights into how the issue can be investigated. Attempts to resolve the issue are also often recorded in the issue tracking system.

Since an issue tracking system appears as an important place to discuss software quality, we believe it is natural to assume that heated discussions about a certain issue may suggest the importance of this specific issue. Furthermore, when an issue is finally resolved after an exceptionally lengthy discussion, all early fix attempts and the final commit that resolves the issue should be considered to be influential. Indeed, all these software changes have contributed to close the discussion, unlock whatever has been blocking attention from other issues, and satisfy the majority of stakeholders.

To identify controversial/popular issues in projects, we first searched for issues with an overwhelmingly larger number of comments than others within the same project. In this study, we regarded an issue as a controversial/popular issue if the number of its comments is larger than the 99th percentile of issue comment numbers. Applying this simple criterion, we could identify a set of issues that are controversial/popular. Afterwards, we collected all commits that were associated with each of the controversial/popular issues and tagged them as potentially influential.

An example was found in Apache Math. An issue[§] with 62 comments was detected by this analysis. This issue is about a simple glitch of an API method; the API hangs 4 to 5 seconds at the first call on a specific Android device. The corresponding changes[¶] fixed the glitch and closed the issue.

To confirm that a change related to an identified controversial/popular issue (based on the number of comments) is truly influential, we verify that (a) the discussion indeed was about the controversy and (b) the change is a key turning point in the discussion. Table 2 compiles the statistics of changes linked to inferred controversial/popular issues as well as the number of influential changes manually confirmed among those changes.

3.2.2 | Anomalies in change behaviors

During software development, source code modifications are generally made in a consecutive way following a somehow regular rhythm. Break in change behaviors may thus signify abnormality and suggest that a specific commit is relatively more important than others. For instance, consider the following scenario: a certain file within a repository after a period of regular edits remains unchanged for a period of time, then is suddenly updated by a single change commit, and afterwards remains again unchanged for a long time. Such a sudden and abnormal change suggests an urgency to address an issue, eg, a major bug fix. In our observational study, we consider both break in behaviors in the edit rhythm of each file and the edit rhythm of developers. An anomaly in change behavior may be an out-of-norm change that developers do not notice, or a change to stable behavior that many developer/parts of code rely on.

In this study, for each file in the project, we considered all commits that modify the file. For each file of those commits, we computed the time differences from the previous commit and to the next commit. Then, we mapped these two time lags to a two dimensional space and used elliptic envelope outlier detection[22] to identify "isolated commits." The straightforward approach to outlier detection consists of setting a threshold

[§]Bug report https://issues.apache.org/jira/browse/MATH-650
[¶]Commits `52649fda4c9643afcc4f8cbf9f8527893fd129ba` and `0e9a5f40f4602946a2d5b0efdc75817854486cd7`

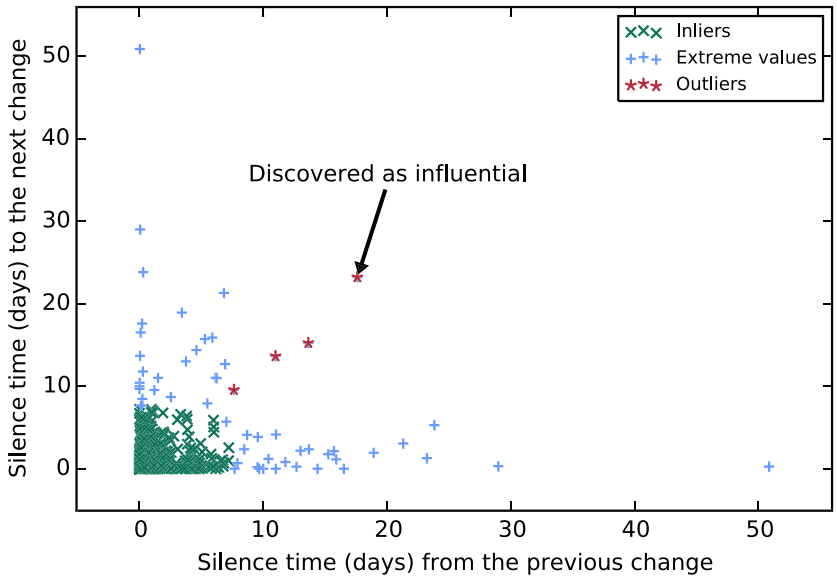

FIGURE 4 Outlier detection to discover isolated commits for *build.gradle* file in `Spring framework`

Project Name	# Isolated Commits	# Influential Changes
`Commons-codec`	7	3
`Commons-collections`	28	9
`Commons-compress`	17	5
`Commons-csv`	13	4
`Commons-io`	18	5
`Commons-lang`	22	7
`Commons-math`	29	5
`Spring-framework`	56	8
`Storm`	48	7
`Wildfly`	213	1
Total	451	54

TABLE 3 Statistics of identified isolated commits and the associated manually confirmed influential changes

value based on the standard deviations to the mean measurements on the data. Such an approach has theoretical (see Leys et al[23]) as well as practical (eg, training required to identify a good threshold value) limitations. We use in this work the elliptic envelope detection approach, which first assumes that the data comes from a known distribution and computes the shape of the data as an ellipse around the central data which are identified based on the covariance estimate to the data. Classical Mahalanobis distances among data points in the ellipse (ie, inliers) are then used to derive a metric for deciding which data point is an outlier. Concretely, we have used the implementation provided in the *scikit-learn*[#] python programming library. In Figure 4, we can visualize the outliers discovered for the changes on the *build.gradle* file from the `Spring` project subject. The highlighted outlier represents a commit[‖] for including AspectJ files in the Spring-sources jar file. This small commit is influential as it fixes a bug that affects the downloadable release packages used by developers leveraging the Spring framework. Its detail is explained in the corresponding bug report.[**]

Individual project contributors also often exhibit abnormal behaviors which may suggest influential code changes. For instance, one core developer constantly contributes to a specific project. If such a developer submits isolated commits (ie, commits that are a long time away from the author's previous commit as well as his/her next commit), this might be recognized as an emergency case where immediate attention is needed.

In this study, we also systematically classified isolated commits based on developer behaviors as potentially influential. For example, from commits by developer *Stefan Bodewig* in `Commons-COMPRESS`, we found an isolated commit[††] where he proposed a major bug fix for the implementation of the *ZipArchiveEntry* API. Before this influential software change, any attempt to create a zip file with a large number of entries was producing a corrupted file.

To confirm that an isolated change is influential, we verify that (a) its importance is clearly stated in the change log and (b) the implication of the change for dependent modules and client applications are apparent. Table 3 provides the statistics on detected isolated commits and the results of our manual analysis on those commits to confirm influential changes.

[#] http://scikit-learn.org/stable/modules/outlier_detection.html$\backslash$#id1
[‖] Commit `a681e574c3f732d3ac945a1dda4a640ce5514742`
[**] Bug report https://jira.spring.io/browse/SPR-9576
[††] Commit `fadbb4cc0e9ca11c371c87ce042fd596b13eb092`

TABLE 4 Statistics of identified referenced commits and influential commits

Project Name	# Referenced Commits	# Influential Changes
Commons-codec	8	3
Commons-collections	3	1
Commons-compress	3	0
Commons-csv	3	2
Commons-io	5	1
Commons-lang	21	2
Commons-math	43	3
Spring-framework	1	1
Storm	1	0
Wildfly	11	9
Total	99	22

TABLE 5 Overall manual assessment results. We compared the percentage of changes that were manually confirmed to be influential from the datasets yielded by our systematic analysis (see Section 3.2) and a random selection in projects. Note that we count unique commits in this table since some commits can be detected by more than one method described in Section 3.2

Project Name	Systematic Analysis Findings			Random Selection		
	Total	Influential	Rate	Total	Influential	Rate
Commons-codec	40	8	20.0%	20	0	0.0%
Commons-collections	42	17	40.5%	20	0	0.0%
Commons-compress	27	9	33.3%	20	1	5.0%
Commons-csv	21	11	52.4%	20	1	5.0%
Commons-io	33	6	18.2%	20	0	0.0%
Commons-lang	72	24	33.3%	20	0	0.0%
Commons-math	108	14	13.0%	20	0	0.0%
Spring-framework	110	51	46.4%	20	1	5.0%
Storm	89	10	11.2%	20	1	5.0%
Wildfly	243	27	11.1%	20	1	5.0%
Total	785	177	**22.5%**	200	5	**2.5%**

3.2.3 | Changes referred to in other changes

We considered that popular changes are potentially influential. These are changes that other developers have somehow noticed (eg, incomplete fix, API change that causes collateral evolution). Indeed, when developers submit software changes to a project, they usually submit also a commit message introducing what their patch does. Occasionally, developers refer to others' contributions in these messages. This kind of behaviors suggests that the referred contribution is influential, at least to a certain extent. For example, in the Commons-CSV project, commit[‡‡] 93089b26 is referred by another commit.[§§] This commit implemented the capability to detect the start of a line, which is undoubtedly an influential change for the implementation of CSV format reading.

Because some of the projects have switched from using Subversion to using Git, we first managed to create a mapping between the Subversion revision numbers (which remain as such in the commit messages) and the newly attributed Git Hash code. To confirm that a change referenced by other changes is influential, we verify that (a) it is indeed referenced by others because it was inducing their changes and (b) the implication of the change for dependent modules and client applications are apparent. Table 4 provides the statistics of influential changes derived with this metric.

3.3 | Manual validation on a sample set

We then set to manually validate the suitability of the metrics used in our observational study. To that end, we must compute the rate of influential changes among the changes that exhibit high scores in the metrics defined about, as well as the rate of influential changes among a randomly selected set of changes within a project history. Concretely, we manually verify whether the potential influential changes yielded by the systematic analysis are indeed acceptable as influential (240, 451, and 99 commits shown in Tables 2, 3, and 4, respectively—ie, 785 distinct commits as shown in Table 5). We further randomly pick change commits from each project and manually check the percentage of changes that can be accepted as influential as well. The comparison between the two types of datasets aimed at validating our choices of postmortem metrics to easily collect influential changes. Table 5 provides the results of the qualitative assessment. For each project, the random dataset size is fixed to 20 commits[¶¶], leading to a manual checking of 200 changes. Our systematic analysis findings produce change datasets with the highest rates of "truly" influential changes (an order of magnitude more than what can be identified in random samples).

[‡‡] Commit 93089b260cd2030d69b3f7113ed643b9af1adcaa
[§§] Commit 05b5c8ef488d5d230d665b9d488ca572bec5dc0c
[¶¶] We selected this number to align with the smallest number of commits found using the systematic analysis technique on a project

The validation procedure, for a commit to be accepted as influential, was implemented to follow a simple but rigorous procedure. Given a commit, we first read the commit log to understand "what" the change is about. At this point, we look for hints that the change is influential (eg, the commit author can suggest it when he enumerates the important issue that the change addresses). Second, we look at the component that the commit touches as well as the amount of changes that are made in the commit (eg, an entire repackaging of a codec code with a new build configuration may be too invasive and requests several changes later, including a potential reverting). Finally, we consider the commit number as well as key terms that we identify in the commit log, and grep other subsequent commits to check whether other commits were necessary follow-up of the commit under study, and thus to measure its "influence" on the development.

Conclusion: *The difference in influential rate values with random shows that our postmortem metrics (isolated changes, popular commits, changes unlocking issues) are indeed good indicators for collecting some influential software changes.*

3.4 | Developer validation

To further validate the influential software changes dataset that we have collected with our intuition-based postmortem metrics, we perform a large-scale developer study. Instead of asking developers to confirm each identified commit, we must summarize the commits into categories. To that end, we leverage the open-card sorting,[24] a well-known, reliable, and user-centered method for building a taxonomy of a system.[25] Card sorting helps explore patterns on how users would expect to find content or functionality. In our case, we use this technique to label influential software changes within categories that are easily differentiable for developers.

We consider open-card sorting where participants are given cards showing the description of identified influential software changes[##] without any pre-established groupings. They are then asked to sort cards into groups that they feel are acceptable and then describe each group. We performed this experiment in several iterations:

- first, two authors of this paper provided individually their group descriptions based on the 177 influential changes identified from the postmortem analysis.
- Second, the two authors then met to perform another open-card sorting with cards containing their agreed group descriptions. Table 6 enumerates the 28 cards that were then summarized.
- Finally, a third author, with more experience in open-card sorting joined for a final group open-card sorting process which yielded 12 categories of influential software changes reported in Table 7.

The influential software changes described in the 12 categories span over four software maintenance categories initially defined by Lientz et al[26] and updated in ISO/IEC 14764:

- Most influential software changes belong to the *corrective changes* category.
- Others are either *preventive changes*, *adaptive changes*, or *perfective changes*.
- Finally, changes in one of our influential change categories can fall into more than one maintenance categories. We refer to them as *cross area changes*.

Developer assessment. We then conduct a developer survey to assess the relevance of the 12 categories of influential changes that we described. The survey participants have been selected from data collected in the GHTorrent project[27] which contains history archives on user activities and repository changes in GitHub. We consider active developers (ie, those who have contributed to the latest changes recorded in GHTorrent) and focus on those who have submitted comments on other's commit. We consider this to be an indication of experience with code review. The study[‖‖] was sent to over 1952 developer email addresses. After one week waiting period, only 800 email owners opened the mail and 144 of them visited the survey link. Finally, 89 developers volunteered to participate in the survey. Of these developers, 66 (ie, 74%) hold a position in a software company or work in freelance. Nine respondents (10%) are undergraduate students and eight (9%) are researchers. The remaining six developers did not indicate their current situation. In total, 78% of the participants confirmed having been involved in code review activities; 26 (29%) developers have between one and five years of experience in software development; 29 (33%) developers have between five and 10 years of experience. The remaining 34 (38%) have over 10 years of experience.

In the survey questionnaire, developers were provided with the name of a category of influential software changes, its description, and an illustrative example from our dataset (we provided the same example for each category to every participant). The participant was then requested to assess the relevance of this category of changes as influential software changes using a Likert scale between 1 (very influential) and 5 (unimportant). Figure 5 summarizes the survey results. For a more detailed description of the categories, we refer the reader to the project web site (see Section "Availability").

[##] We consider all 177 influential software changes from the postmortem analysis.
[‖‖] Survey form at https://goo.gl/V2g8OE

TABLE 6 Cards after the second-round of open-card sorting

Card description	Example Commit Description (Only When not Summary Needs Explanation)
Implementation of a long-awaited feature	
Change which was discussed at length as solving a bug that was very hard to reproduce	
trivial fixes that has high impact on basic requirements for system functionality	(eg, in commons-compress, simple fixes but it can prevent filepath encoding problems)
Bug fix of a key feature in library	
Bug fix of a bug that manifests itself only in corner-cases	(eg, This fix prevents data integrity violations, which cannot be revealed unless accessing missing data.) Bug fix to prevent an infinite loop when API reading input stream encounters an unmappable character
Replacement of key functionality code to improve usability	
Fix for a regression error that is hard to reproduce and that	(eg, the change is about exception handling (whether it should be caught in the program and which place).
occurs in specific, but popular, environment	This is not easy to reproduce. It happens in Google App Engine and the previous version works correctly.)
Fix for an API bug—the API is used pervasively	
New feature that was discussed at length	
New feature that implements as an API a functionality that developers	
implement regularly in an ad hoc manner in their own code	
Non-functional bug fix for speciific popular API—leads to debate	(eg, This change fixes a performance bug which affects Android applications. This change leads to a long debate since people have different opinions on this issue. The repaired method is popular.)
Change reverting a bug fix to a key functionality	
Controversial change that is debated and then reverted	
Fix for a bug that affects many clients/components/applications (ie, core component)	
Fix to a blocking bug	
Fix for non-functional defect	(eg, Performance defect—adding multicores could not improve the performance.)
New API implementation for future applications	
Fix of dependency problems for compilation	
Fix of configuration errors	(eg, New build references in POM files and assembly references)
Fix of a trivial bug, but which appears in several different components	https://github.com/wildfly/wildfly/commit/eea5d5fe34e9e7c67f076cae81fec6ebf06626af
– leads to debate and verification of all source code	
Fix a bug that is not easy to locate	(eg, Yes, this change releases the hang (not failing nor crashing) of a test case. A test case hang can block maven building.
Change that overhauls an important module/file	
Change that add test cases to avoid specific important functional bugs	
Change for improvement of implementation by creation of new exception class	
Change in nightly process management	
Bug fix that finalizes/corrects previous fix in API	
Changes that lead to several collateral changes (including reverting, new API mapping)	

TABLE 7 List of 12 influential changes labeled during open card sorting, with examples of changes

Cat.	Consensus-Based Label	Description of Influence	Example Change
Adaptive/perfective changes	New key feature	Implementation of a long awaited feature, change that implements as an API, a functionality that developers implement regularly in an ad hoc manner in their own code	Implementation of the Kalman filter in the Commons-MATH project was tagged in https://issues.apache.org/jira/browse/MATH-485 as a major feature request, which was resolved by the change commit https://github.com/apache/commons-math/commit/58d18852
Cross-area changes	Domino changes	Change causing many collateral changes (eg, API modification leading to changes of all API method call sites)	Starting with Linux 2.5.4, the USB library function usb_submit_urb (which implements message passing) now takes a second argument for explicitly specifying the context (which was previously inferred in the function definition). The argument can take one of three values: GFP_KERNEL (no constraints), GFP_ATOMIC or GFP_NOIO (blocking is allowed but not I/O). Developers using this USB library must then parse their own code to understand which context it should be. This leads to bugs that can keep occurring. A study of faults in Linux by Pallix et al[15] have reported that because of the complexity of the conditions governing the choice of the new argument for usb submit urb, 71 of the 158 calls to this function were initially transformed incorrectly to use GFP KERNEL instead of GFP_ATOMIC.
Preventive changes	API usability improvement	Replacement of key functionality code to improve usability or introduction of new error management (eg, improvement of implementation by creation of a new exception class or log management for users)	In the Wildfly project, a major feature request (https://issues.jboss.org/browse/WFLY-280) was resolved for providing an operation to retrieve the last 10 errors from the log. Discussions on the issue page clearly shows that the change was solving a major issue as it improved usability substantially (https://github.com/wildfly/wildfly/commit/a22b8d7ccf872b503da8d43f1c29390356d6d5d3)
	Major structural change	Change that overhauls an important module/file	Commit 34658f08 (https://github.com/apache/commons-lang/commit/34658f08) in the commons-LANG project rewrites an entire utils file
	Fix configuration bug	Fix of dependency problems for compilation, change in nightly build process management	In the WILDFLY project a major bug (https://issues.jboss.org/browse/WFLY-2047) was finally resolved by fixing dependencies in the connector module (https://github.com/wildfly/wildfly/commit/88756ddb1061660cb5ca68f5562d7343570dd955)
	Important test case addition	Changes that add test cases to avoid specific important functional bugs	Commit 1f001d06 (https://github.com/apache/commons-lang/commit/1f001d06) in the LANG project specifically added some test cases to avoid regression faults on key functionalities.
	Fix non-functional bug	Non-functional bug fix for a specific API. Performance or security issues usually lead to debate among developers	A change in Commons-math project fixes a performance bug which affects Android applications (https://github.com/apache/commons-math/commit/52649fda4c9643afcc4f8cbf9f8527893fd129ba). This change leads to a long debate since people have different opinions on this issue. Although this does not affect the functionality of the method, the repaired method is popular

(Continues)

TABLE 7 (Continued)

Cat.	Consensus-Based Label	Description of Influence	Example Change
Corrective changes	Fix hard to reproduce/ locate bug	Fix of a bug that manifests itself only in corner-cases, or a bug that is not easy to locate, or a bug that is simply hard to reproduce	A change in the Spring framework fixes a regression fault that is not easy to reproduce (cf.https://github.com/spring-projects/spring-framework/commit/956b66bbd466bb7a68e8499a483139a516572b24).
	Fix blocking bug	Fix a bug that prevents other bugs from being exposed/addressed	In project CASSANDRA, commit d37696ca provides a change that fixes partially a major blocking bug
	Fix pervasive bug	Fix of a bug that affects many clients/components/applications (ie, core component), fix a bug which may also appear in several different components, or fix of an API that is used pervasively	In the Spring framework, a change in the equality operator is pervasively affecting other components (https://github.com/spring-projects/spring-framework/commit/2a05e6afa116ab56378521b5e8c834ba92c25b85).
	Fix key feature bug	Bug fix of a key feature in library or fix that has high impact on basic requirements for system functionality	In Commons-COMPRESS project, a bug fix change (fadbb4cc) was applied to fix the implementation of the zip functionality. Indeed, creating a zip file with many entries was producing a wrong archive
	Correcting controversial change changes in the Spring Framework	Bug fix that finalizes/corrects previous fix in API, change reverting a bug fix to a key functionality or controversial change that is debated and then reverted	(https://github.com/spring-projects/spring-framework/commit/cfc821d1799ca7c64b1bbc53811b712fdaa4776c and https://github.com/spring-projects/spring-framework/commit/0934751d7aa625fd098086ce3a5fb489f2edc7e0) are fixing regression faults that could not be easily reproduced. These changes were reverted several times due to incomplete fixes

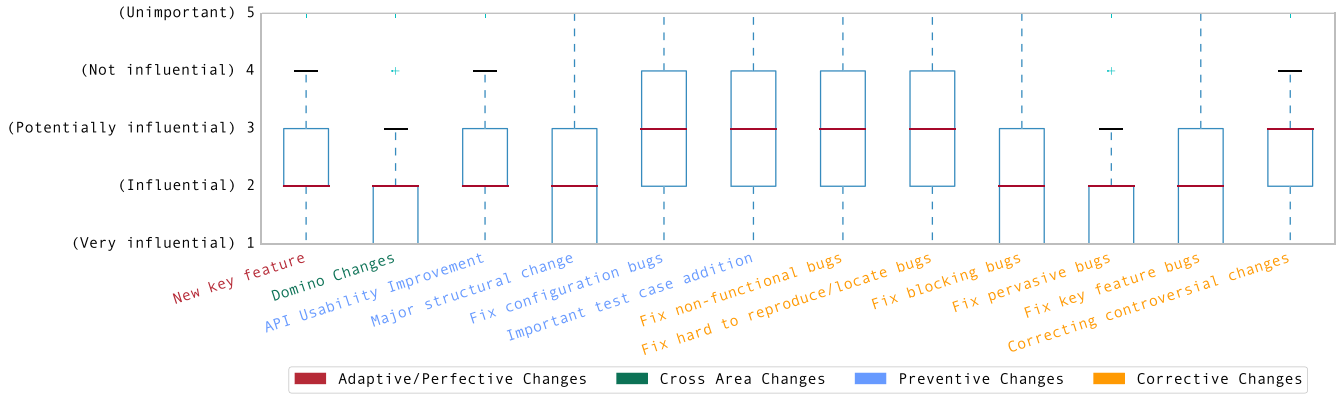

FIGURE 5 Survey results on different categories of Influential Changes

The survey results suggest that:

- According to software developers with code review experience, all 12 categories are about important changes: seven categories have an average agreement of 2 (ie, Influential), the remaining five categories have an average of 3 (ie, potentially influential). Some (eg, "domino changes" and "changes fixing pervasive bugs") are clearly found as more influential than others (eg, "important test case addition").
- Some changes, such as "fixes for hard to reproduce or locate bugs," "fixes non-functional bugs," "important test case addition", and "fixes configuration bugs," are not as influential as one might think.
- Developers also suggested two other categories of influential changes: *Documentation changes* and *Design-phase changes*. The latter, however, is challenging to capture in source code repository artefacts, while the former is not relevant to our study which focuses on source code changes.

With this study, we can increase our confidence in the dataset of influential software changes that we have collected. The examples listed in Section 1 can be categorized: "*Adding a new lock mechanism*" to "new key features," "*Changing build configurations*" to "fix configuration bugs," and "*Improving performance for a specific environment*" to "fix pervasive bugs," or "fix non-functional bugs." We thus consider leveraging on the code characteristics of these identified samples to identify more influential changes.

4 | LEARNING TO CLASSIFY ICs

Beyond the observational study reported in Section 3, we propose an approach to identify influential changes on-the-fly. The objective is to classify, when a change is being submitted, whether it should be considered with care as it could be influential or not (ie, binary prediction: IC or non-IC). Note that this approach does not classify a commit into specific IC types described in Section 3. To that end, the approach leverages machine learning (ML) techniques. In our study, the learning process is performed based on the dataset yielded by the systematic postmortem analysis and whose labels were manually confirmed. In this section, we describe the features that we use to build classifiers as well as the quantitative assessment that was performed.

4.1 | Machine learning features for ICs

A change submitted to a project repository contains information on what the change does (in commit messages), files touched by the change, the number of edits performed by the change, and so on. On the basis of the experience gathered during the manual analysis of influential changes in Section 3, we extract a number of features to feed the ML algorithms. The summary of features we used in this study is listed in Table 8 and the following sections describe more details on the features.

4.1.1 | Structural features

First, we consider common metrics that provide hints on structural characteristics of a change. These metrics include (a) the number of files simultaneously changed in a single commit, (b) the number of lines added to the program repository by the commit, and (c) the number of lines removed from the code base.

4.1.2 | Natural language terms in commit messages

During the observational study, we noted that commit messages already contain good hints on the importance of the change that they propose. We use the *bag-of-words*[12] model to compute the frequency of occurrence of words and use it as a feature. In addition, we noted that developers may be emotional in the description of the change that they propose. Thus, we also compute the subjectivity and polarity of commit messages based on *sentiment analysis*[28-31] techniques.

TABLE 8 Features used in learning a prediction model for ICs

Category	Feature	Type	Description
Structural	# files	Numeric	Number of files simultaneously changed in a single commit.
	# l-added	Numeric	Number of lines added to the program repository by the commit.
	# l-removed	Numeric	number of lines removed from the code base by the commit.
NL Terms	# freq($token_i$)	Numeric	Frequency of occurrence of $token_i$. The number of features varies for each project.
	Subjectivity	Boolean	Binary value indicating whether a commit message is subjective or objective.
	Polarity	Boolean	Binary value indicating whether a commit message is negative or positive
Co-change	Max. PageRank	Numecric	Maximum value of PageRank for each file in a commit. PageRank is computed in a co-change graph built on a revision history of a project.
	Min. PageRank	Numeric	Minimum value of PageRank for each file in a commit.
	Betweenness Centrality	Numeric	Delta of the *betweenness centrality* values of files between current and previous commits.
	Closeness Centrality	Numeric	Delta of the *closeness centrality* values of files between current and previous commits.

4.1.3 | Co-change impact

Finally, we consider that the frequency to which a pair of files are changed together can be an indication of whether a given change commit affecting both files (or not) is influential.

In our experiments, for each commit, we build a co-change graph of the entire project taking into account the history of changes until the time of that commit. Then, considering files that are actually touched by the change commit, we extract common network metrics.

PageRank[32] is a link analysis algorithm for "measuring" the importance of an element, namely, a page, in a hyperlinked set of documents such as the world wide web. In our study, this metric is used for computing the likelihood that a file is changed together with other files. Considering a co-change graph as a linked set, we extract PageRank values for all files. When a commit change is applied, the co-change graph can be changed and then we re-compute new PageRank values. We take (newly computed) PageRank values of changed files in a commit after applying the commit to the co-change graph. Among the values, the minimum and maximum are used for features.

Centrality metrics are commonly used in social network analysis to determine influential people or to identify key nodes in networks. We use these metrics to represent how much a file is well-connected with other changed files in this study. In our experiments, we focus on computing *betweenness centrality*[33] and *closeness centrality*[34] metrics for all files associated with a commit change. We compute betweenness and closeness centrality metrics of each file involved in a commit. After taking the sum of each centrality metric for all files in the commit, we then compute the δ of the sum between the current and previous commits. The δ of each centrality metric is used as our feature.

4.2 | Influential change classification

In this section, we present the parameters of our machine learning classification experiments for predicting influential changes. In these experiments, we assess the quality of our features for accurately classifying influential changes. We perform tests with two popular classifiers, the Naïve Bayes[12,13] and Random Forest.[14]

In the process of our validation tests, we are interested in assessing: (a) whether connectivity on co-change graphs correlates with a probability for a relevant change to be an IC; (b) if natural language information in commit messages are indicative of ICs; (c) if structural information of changes are indicative of ICs; (d) whether combinations of features is best for predicting ICs; (e) if our approach can discover ICs beyond the types of changes discovered with postmortem analysis.

4.2.1 | Experiment setup

To compute the feature vectors for training the classifiers, we used a high-performance computing system[35] to run parallel tasks for building co-change graphs for the various project subjects. After extracting the feature metrics, we preprocessed the data and ran 10-fold cross validation tests to measure the performance of the classification.

Preprocessing. Influential software changes likely constitute a small subset of all changes committed in the project repositories. Our manual analysis yielded very few influential changes leading to a problem of imbalanced datasets in the training data. Since we try to identify influential changes, which constitute the minority classes and learning algorithms are not adapted to imbalanced datasets, we use oversampling techniques to adjust the class distribution. In our experiments, we leverage the Synthetic Minority Over-sampling Techniques (SMOTE).[36]

Evaluation Measures. To quantitatively evaluate the performance of our approach for predicting influential changes, we used standard metrics in ML, namely, Precision, Recall, and F-measure.[10,13,37] *Precision* quantifies the effectiveness of our machine learning-based approach to point to changes that are actually influential. *Recall* on the other hand explores the capability of our approach to identify most of the influential changes in the commits set. Finally, we compute the *F-measure*, the harmonic mean between Recall and Precision. We consider that both Precision and Recall are equally important and thus, they are equally weighted in the computation of F-measure.

TABLE 9 Performance comparison using the Naïve Bayes and Random Forest classifiers

	Algorithm	Commons-codec	Commons-collections	Commons-compress	Commons-csv	Commons-io	Commons-lang	Commons-math	Storm	Average
F-Measure	NB	95.1	92.9	91.5	84.2	98.5	89.2	94.3	86.1	91.5
(Influential Class)	RF	97.4	96.4	98.2	77.8	97.0	95.0	99.1	97.8	94.8
F-Measure	NB	93.5	87.5	83.9	92.7	98.1	79.5	92.6	86.5	89.3
(Non Influential Class)	RF	97.0	93.9	97.1	90.5	96.3	92.9	98.9	97.5	95.5

TABLE 10 Ten-fold cross validation on influential changes using Random Forest with different metrics combinations. CC: co-change features. NL: natural language terms on commit messages. SI: structural features

		Influential Class			Non-Influential Class		
Project Name	Metrics	Precision	Recall	F-Measure	Precision	Recall	F-Measure
Commons-codec	CC	97.5	97.5	97.5	96.9	96.9	96.9
	NL	100.0	92.5	96.1	91.4	100.0	95.5
	SI	81.0	85.0	82.9	80.0	75.0	77.4
	CC NL	100.0	95.0	97.4	94.1	100.0	97.0
	CC SI	95.0	95.0	95.0	93.8	93.8	93.8
	NL SI	100.0	95.0	97.4	94.1	100.0	97.0
	ALL	100.0	95.0	97.4	94.1	100.0	97.0
Commons-collections	CC	90.5	92.7	91.6	87.5	84.0	85.7
	NL	94.9	90.2	92.5	85.2	92.0	88.5
	SI	80.4	90.2	85.1	80.0	64.0	71.1
	CC NL	97.3	87.8	92.3	82.8	96.0	88.9
	CC SI	86.7	95.1	90.7	90.5	76.0	82.6
	NL SI	95.1	95.1	95.1	92.0	92.0	92.0
	ALL	95.2	97.6	96.4	95.8	92.0	93.9
Commons-compress	CC	92.9	96.3	94.5	94.1	88.9	91.4
	NL	100.0	96.3	98.1	94.7	100.0	97.3
	SI	89.7	96.3	92.9	93.8	83.3	88.2
	CC NL	100.0	96.3	98.1	94.7	100.0	97.3
	CC SI	87.1	100.0	93.1	100.0	77.8	87.5
	NL SI	100.0	100.0	100.0	100.0	100.0	100.0
	ALL	96.4	100.0	98.2	100.0	94.4	97.1
Commons-csv	CC	40.0	36.4	38.1	65.0	68.4	66.7
	NL	100.0	63.6	77.8	82.6	100.0	90.5
	SI	100.0	81.8	90.0	90.5	100.0	95.0
	CC NL	100.0	54.5	70.6	79.2	100.0	88.4
	CC SI	66.7	54.5	60.0	76.2	84.2	80.0
	NL SI	100.0	72.7	84.2	86.4	100.0	92.7
	ALL	100.0	63.6	77.8	82.6	100.0	90.5
Commons-io	CC	93.9	93.9	93.9	92.6	92.6	92.6
	NL	100.0	97.0	98.5	96.4	100.0	98.2
	SI	82.5	100.0	90.4	100.0	74.1	85.1
	CC NL	100.0	97.0	98.5	96.4	100.0	98.2
	CC SI	94.1	97.0	95.5	96.2	92.6	94.3
	NL SI	100.0	97.0	98.5	96.4	100.0	98.2
	ALL	97.0	97.0	97.0	96.3	96.3	96.3
Commons-lang	CC	86.5	88.9	87.7	82.6	79.2	80.9
	NL	94.4	93.1	93.7	89.8	91.7	90.7
	SI	72.2	79.2	75.5	63.4	54.2	58.4
	CC NL	95.8	95.8	95.8	93.8	93.8	93.8
	CC SI	91.9	94.4	93.2	91.3	87.5	89.4
	NL SI	98.5	93.1	95.7	90.4	97.9	94.0
	ALL	97.1	93.1	95.0	90.2	95.8	92.9
Commons-math	CC	95.4	96.3	95.9	95.7	94.7	95.2
	NL	100.0	100.0	100.0	100.0	100.0	100.0
	SI	76.3	80.6	78.4	76.1	71.3	73.6
	CC NL	100.0	100.0	100.0	100.0	100.0	100.0
	CC SI	96.4	98.1	97.2	97.8	95.7	96.8
	NL SI	100.0	98.1	99.1	97.9	100.0	98.9
	ALL	100.0	98.1	99.1	97.9	100.0	98.9
Spring-framework	NL	96.2	90.9	93.5	84.6	93.2	88.7
	SI	75.8	88.2	81.5	68.3	47.5	56.0
	NL SI	96.0	86.4	90.9	78.6	93.2	85.3
Storm	CC	97.7	95.5	96.6	95.1	97.5	96.2
	NL	97.8	98.9	98.3	98.7	97.5	98.1
	SI	90.0	80.9	85.2	80.7	89.9	85.0
	CC NL	97.8	97.8	97.8	97.5	97.5	97.5
	CC SI	97.7	95.5	96.6	95.1	97.5	96.2
	NL SI	98.9	98.9	98.9	98.7	98.7	98.7
	ALL	97.8	97.8	97.8	97.5	97.5	97.5
Wildfly	NL	93.7	98.4	96.0	98.0	92.6	95.2
	SI	78.7	82.3	80.5	79.0	75.0	77.0
	NL SI	96.0	99.2	97.6	99.0	95.4	97.2

4.2.2 | Assessment results

In the following paragraphs, we detail the classification results for influential changes using 10-fold cross validation on labeled data. In addition, we present experimental results of the performance of the classification models with new dataset labeled based on developer accepted definitions of influential challenges.

Cross validation is a common model validation in statistics to assess how the results of a statistical analysis will generalize to an independent data set. In machine learning experiments, it is common practice to rely on *k-fold* cross validation where the test is performed *k* times, each time testing on a *kth* portion of the data. We perform 10-fold cross validation on the labeled dataset built in Section 3.

In the first round of experiments, we built feature vectors with all features considered in our study. We then built classifiers using Naïve Bayes and Random Forest. Table 9 depicts the F-measure performance in 10-fold cross validation for the two algorithms. Although Random Forest performs on average better than Naïve Bayes, this difference is relatively small.

Table 10 details the validation results with Random Forest for different combinations of feature groups for the experiments. We considered separately features relevant to co-change metrics, commit messages written in a natural language, and the structural information of changes. We also combined those type of features to assess the potential performance improvement or deterioration.

Co-change metrics, which are the most tedious to extract (hence missing from two projects in Table 10 because of too large graphs) histories, allow to yield an average performance of 87.7% precision, 87.5% recall, and 87.6% F-measure.

Natural language terms in commit messages also allow yielding an average performance of 94.9% precision, 94.4% recall, and 94.4% F-measure for the influential change class on average.

Our experiments also revealed that structural features of changes yield the worst performance rates, although those performances reached 80.5% F-measure on average. For some projects, however, these metrics lead to a performance slightly above 50% (random baseline performance).

The performance results shown in Table 10 also highlight the fact that, on average, combining different features contributes to improving the performance of influential change classification. Combining co-change and natural language terms in commit messages achieves on average a precision, recall, and F-measure performance of 95.6%, 94.5%, and 94.5%, respectively. Similarly, combining co-change and structural features shows the F-measures at 90.1% on average. Combinations of natural language and structural information show 95.6% F-measure. Finally, combining all features leads to an average performance of 96.1% precision, 94.9% recall, and 95.2% F-measure. However, no feature combination achieves the best performance in every project, possibly suggesting these features are specific to projects.

Figure 6 further shows graphically the Area Under the Receiver Operating Characteristic (AUROC) for the predictors including all features. Overall, we note that the performance of the classifiers is high.

4.2.3 | Generalization of influential change features

In previous experiments, we have tested the machine learning classifier with influential change data labeled based on three specific criteria (changes that fix controversial/popular issues, isolated changes and changes referenced by other changes). These categories are however strictly related to our initial intuitions for collecting influential changes in a postmortem analysis study. There are likely many influential changes that do not fit into those categories. Our objective is thus to evaluate whether the *features* that we use for classification of influential changes are still relevant in the wild. In other words, the experiment aims at figuring out the generalizability of the features rather than the models used in previous experiments.

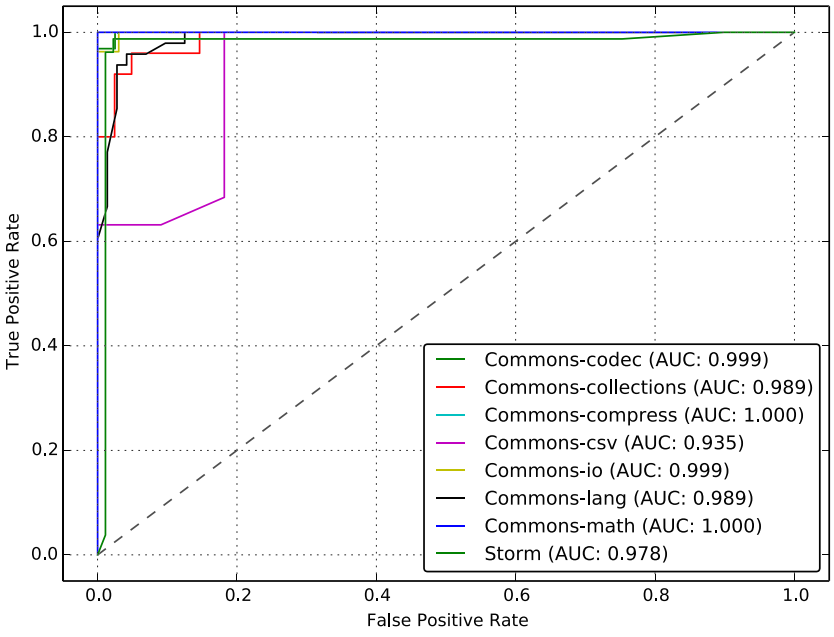

FIGURE 6 Receiver operating characteristic for predictors in all 10 projects

TABLE 11 Ten-fold cross validation on randomly sampled and then manually labelled data. We show results considering all features (NL and SI features in the case of `Spring-framework` and `Wildfly` because of missing CC features)

Project Name	Influential Class			Non-Influential Class		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure
`Commons-codec`	100.0	88.9	94.1	87.5	100.0	93.3
`Commons-collections`	100.0	88.9	94.1	83.3	100.0	90.9
`Commons-compress`	0.0	0.0	0.0	66.7	100.0	80.0
`Commons-io`	86.7	86.7	86.7	75.0	75.0	75.0
`Commons-lang`	97.3	90.0	93.5	85.2	95.8	90.2
`Commons-math`	100.0	31.6	48.0	71.1	100.0	83.1
`Spring-framework`	97.5	96.9	97.2	91.7	93.2	92.4
`Storm`	100.0	88.2	93.8	88.2	100.0	93.8
`Wildfly`	100.0	96.4	98.2	95.8	100.0	97.8

We randomly sample a significant set of changes within our dataset of 10 project commits. Out of the 48 272 commits from the dataset, we randomly consider 381 commits (ie, the exact number provided by the sample size calculator[***] using 95% for the confidence level and 5 for the confidence interval).

Again, we manually label the data based on the categories of influential changes approved by developers (cf Section 3.4). We cross check our labels among authors and perform 10-fold cross validation using the same features presented in Section 4.2.1 for influential change classification. The results are presented in Table 11.

The precision of 10-fold cross validation for influential changes is on average 86.8%, while the average recall is 74%. These results suggest that overall, the features provided in our study are effective even in the wild. For some projects, the performance is especially poor, mainly because (a) their training data is limited (`Commons-CSV` has only one labeled influential change, making it infeasible to even oversample; thus no results are available in the table) and (b) currently, we do not take into account some features of influential changes related to documentation. Developers have already brought up this aspect in the survey.

4.2.4 | Evaluation summary

From our evaluation results we have found that: (a) co-change metrics allow to successfully predict influential changes with an average 87.6% F-measure; (b) features based on terms in commit messages can predict influential changes with high precision (average of 94.9%) and recall (average of 94.4%); (c) Structural features can be leveraged to successfully predict influential changes with an average F-measure performance of 80.5%; (d) overall, combining features often achieves a better classification performance than individual feature groups. For example, combining all features showed 96.1% precision, 94.9% recall, and 95.2% F-measure on average; (e) with the features we collected, our classification model has an acceptable performance in the wild, ie, with different types of influential changes (beyond the ones we relied upon to infer the features).

5 | LIMITATIONS AND THREATS TO VALIDITY

The results of our study support the concept of "risky changes" introduced by Shihab et al.[38] Influential changes can be risky (in the sense that they may have negative effects) but may also be stimulating (in the sense that they may have positive impacts). We have further considered in our work, that the "influence" of the changes can be visible on three entities: the code base evolution, the user base adoption, and the developer team dynamics.

Concretely, the categories that we have inferred and validated with code reviewers, as well as the dataset that we have built are important artefacts that can spark research around the topic of influential changes.

Our work, which we recognize as a preliminary investigation, opens new research directions for recommending changes to the attention of other developers in the team, for dismissing or delaying changes that may affect software adoption, for rewarding/penalizing authors of changes that will have positive/negative effects.

The main limitations of our work include:

- the generality in defining and identifying influential changes. We have opted in this work to not focus on a specific category of influential changes. Instead, we investigate them broadly and leave in-depth characterization to future works.
- our classification model, which shows high performance in predicting influential changes, could be entirely overfitting to the dataset partly because of a small number of instances used in training and testing a model. Future work would require the application of the model to a larger and more diverse dataset.

Our study raises several threats to validity. This section outlines the most salient ones.

[***] http://www.surveysystem.com/sscalc.htm

Internal validity. The authors have manually labeled themselves the influential changes as it was prohibitively costly to request labeling by a large number of developers. We have mitigated this issue by clearly defining criteria for selecting influential changes and by performing cross-checking. Another threat relates to the number of developers who participated in the code developer study for approving the categories of influential changes. We have attempted to mitigate this threat by launching advertisement campaigns targeting thousands of developers. We have further focused on quality and representative developers by targeting those with some code review experience.

Our dataset used for building a prediction model in Section 4 is highly imbalanced, partly because of the nature of ICs. This might affect the results shown in Section 4.2. To mitigate the impact, we applied oversampling and undersampling techniques.

Merging feature data of each project is not eligible since each project has a different number of features because of NL features even though it might alleviate the impact of imbalanced data.

External validity. Although we considered a large dataset of commit changes, this data may not represent the universe of real-world programs. Indeed, the study focused on open-source software projects written in Java. The metrics and features used for predicting influential changes in this context may not be representative for other contexts.

The features used in our study (in Section 4) are limited to three categories. As listed in Shihab et al,[38] there is a more number of feature categories such as time and personnel. Applying more features to building a prediction model might improve the performance. However, some features are not eligible for our subjects. For example, features related to "commit time" is not reliable in open source projects since commit time is often based on several different time zones and the working time of developers are not precisely managed.

Construct validity. Finally, we selected features based on our intuitions on influential changes. Our study may have thus overlooked more discriminative features. To mitigate this threat, first, we have considered several features, many of which are commonly known in the literature; second, we have repeated the experiments based on data labeled following new category labels of influential changes approved by developers.

Furthermore, our experiment focuses only on project-specific classification rather than cross-project setting. Cross-project classification can be useful since some projects may not have a sufficient number of data (ie, commits) to build a training set. However, NL features are project-specific and cannot be applied to a cross-project setting. Our future work includes extracting more features for cross-project IC classification.

6 | RELATED WORK

This section discusses four groups of related work: (a) software evolution, (b) change impact analysis, (c) defect prediction, and (d) developer expertise. These topics address several relevant aspects of our study.

6.1 | Software evolution

Changing any file in a software system implies that the system evolves in a certain direction. Many studies dealt with software evolution in different ways. D'Ambros et al[39] presented *the evolution radar* that visualizes file and module-level coupling information. Although this tool does not directly predict or analyze the change impact, it can show an overview of coupling relationships between files and modules. Chronos[40] provides a narrowed view of history slicing for a specific file. The tool analyzes a line-level history of a file. This reduces the time required to resolve program evolution tasks. Girba et al[41] proposed a metric called *code ownership* to illustrate how developers drive software evolution. We used the metric to examine the influence of a change.

API evolution and deprecation, which often leads to important collateral evolutions can be influential in project development. Several studies[42-48] in the literature have investigated such evolution. For example, Dagenais and Robillard have proposed SemDiff,[42] a framework for recommending replacements of non-trivial changes. Robbes et al[43] have studied the importance in practice of API deprecation for developers. Teyton et al[45] have built a framework for analyzing software library dependencies to support developers in library migrations based on evolution trends.

6.2 | Change impact analysis

Many previous studies revealed a potential impact of software changes. There is a set of techniques that use dynamic analysis to identify change impacts. Ren et al[2] proposed *Chianti*. This tool first runs test cases on two subsequent program revisions (after/before a change) to figure out atomic changes that describe behavioral differences. The authors provided a plug-in for Eclipse, which helps developers browse a change impact set of a certain atomic change. FaultTracer[3] identifies a change impact set by differentiating the results of test case executions on two different revisions. This tool uses the extended call graphs to select test cases affected by a change.

Brudaru and Zeller[1] pointed out that the long-term impact of changes must be identified. To deal with the long-term impact, the authors proposed a change genealogy graph, which keeps track of dependencies between subsequent changes. Change genealogy captures addition/change/ deletion of methods in a program. It can measure the long-term impact on quality, maintainability, and stability.[6] In addition, it can reveal cause-effect chains[7] and predict defects.[8]

Although dynamic analysis and change genealogy can pinpoint a specific element affected by a change in source code, its scope is limited to executed statements by test cases. This can miss many affected elements in source code as well as non-source code files such as build scripts

and configuration settings. Revision histories can be used for figuring out files changed frequently together. Zimmermann et al[49] first studied co-change analysis in which the authors revealed that some files are commonly changed together. Ying et al[50] proposed an approach to predicting files to change together based on revision histories.

There have been cluster-based techniques for change impact analysis. Robillard and Dagenais[4] proposed an approach to building change clusters based on revision histories. Clusters are retrieved by analyzing program elements commonly changed together in change sets. Then, the approach attempts to find matching clusters for a given change. The matching clusters are regarded as the change impact of the given change. Sherriff and Williams[5] presented a technique for change impact analysis using singular value decomposition (SVD). This technique basically figures out clusters of program elements frequently changed together. When clustering changes, the technique performs SVD. The clusters can be used for identifying the change impact of an incoming change.

In addition, Shihab et al studied risky changes[38] in an industrial project. Risky changes are, rather than just a bug, software changes that developers need to pay additional attention to avoid a negative impact on their product. They built a prediction model based on change-relevant features such as change time, code size, and the number of files changed. The model achieves 67% recall and 87% precision. The study explored a different perspective of influential changes; the context was an industrial case, and risky changes are labeled by the developers. On the other hand, our study revealed latent ICs in open source projects. Combining these two perspectives is our future direction.

Identifying high-impact defects[51] is highly-relevant to our work as well. This type of defects includes breakage and surprise defects in detail. Shihab[51] suggested a prediction model to classify the high-impact defect types. They figured out that breakage defects are relevant to the number of pre-release defects and file size, while the time between the latest pre-release change and the release date lead to surprise defects. These findings are helpful to find latent ICs in addition to our methods.

6.3 | Change pattern mining

Discovering common change patterns may reveal influential changes since those patterns appear across different projects.

There have been several empirical studies on change patterns. Pan et al[52] explored common bug fix patterns in Java programs to understand how developers change programs to fix a bug.

Martinez and Monperrus further investigated repair models that can be utilized in program fixing. Zhong and Su[53] conducted a large-scale study on bug fixing changes in open source projects. Tan et al[54] analyzed anti-patterns that may interfere with the process of automated program repair. While most pattern discovery studies focused on statement-level (ie, coarse-grained) change patterns, Liu et al[55] investigated fine-grained (eg, expression-level) code changes to discover better change elements for program repair. Koyuncu et al[56] explored how developers manually or automatically make program changes in the Linux kernel project and their impact; this study showed changes generated by different tools may have different impacts on a software project.

In addition, several studies have proposed techniques to automate change pattern discovery. SYDIT[57] and Lase[58] generate code changes to other code snippets with the extracted edit scripts from examples in the same application. RASE[59] focuses on refactoring code clones with Lase edit scripts.[58] FixMeUp[60] extracts and applies access control templates to protect sensitive operations.

REFAZER implements an algorithm for learning syntactic program transformations for C# programs from examples[61] to correct defects in student submissions.

Genesis[62] heuristically infers application-independent code transform patterns from multiple applications to fix bugs, but its code transform patterns are tightly coupled with the nature and syntax of three kinds of bugs (ie, null pointer, out of bounds, and class cast defects). Koyuncu et al[63] have generalized this approach with FixMiner to mining fix patterns for all types of bugs given a large dataset.

Reudismam et al[64] tried to learn quick fixes by mining code changes to fix PMD violations.[65] Their approach aims at learning code change templates to be systematically applied to refactor code. Li et al[66] leveraged convolutional neural networks, which is one of the deep neural network techniques, to automatically cluster commons fix patterns from more than 88 000 bug-fixing changes. Those fix patterns can successfully fix real bugs.[67]

6.4 | Program repair

Automated program repair (APR) can promote influential changes since most program repair techniques focus on common and recurring bug types. These techniques often locate and fix similar bugs by scanning all files in a project. Thus, applying APR techniques can produce massive program changes at once and this could be influential for the further evolution of the project.

Most of program repair studies focus on automating the entire process of fixing bugs, ie, localizing a bug, generating a patch, and validating the patch. Automated program repair is pioneered by GenProg.[68,69] This approach leverages genetic programming to create a patch for a given buggy program. It is followed by an acceptability study[70] and systematic evaluation.[71] Regarding the acceptability issue, Kim et al[72] advocated GenProg may generate nonsensical patches and proposed PAR to deal with the issue. PAR leverages human-written patches to define fix templates and can generate more acceptable patches. HDRepair[73] leverages bug fixing history of many projects to provide better patch candidates to the random search process. Recently, LSRepair[74] proposes a live search approach to the ingredients of automated repair using code search techniques.

While GenProg relies on randomness, utilizing program synthesis techniques[75-77] can directly generate patches even though they are limited to a certain subset of bugs. Other notable approaches include contract-based fixing,[78] program repair based on behavior models,[79] and conditional statement repair.[80]

While fully automated repair techniques can be heavy and verbose in practice, patch suggestion techniques are light-weight but involve human developers in the loop. MintHint[81] generates repair hints based on statistical analysis. Tao et al[82] investigated how automatically generated patches can be used as debugging aids. Bissyandé suggests patches for bug reports based on the history of patches.[83] Caramel[84] focuses on potential performance defects and suggests specific types of patches to fix those defects.

Many studies have explored properties of program repair. Monperrus[85] criticized issues of patch generation learned from human-written patches.[72] Barr et al discussed the plastic surgery hypothesis[86] that theoretically illustrates graftibility of bugs from a given program. Long and Rinard analyzed the search space issues for population-based patch generation.[87] Smith et al presented an argument of overfitting issues of program repair techniques.[88] Koyuncu et al[56] compared the impact of different patch generation techniques in Linux kernel development. Benchmarks for program repair are proposed for different programming languages.[89,90] On the basis of a benchmark, a large-scale replication study was conducted.[91] TBar[92] dissects the relationships between common fix patterns suggested by existing studies such as PAR[72] and bugs in the Defects4J benchmark.[90]

6.5 | Defect prediction

Changing a program may often introduce faults.[93,94] Thus, fault prediction at an early stage can lead developers to achieve a better software quality. Kim et al[95] proposed a cache-based model to predict whether an incoming change may introduce or not. They used *BugCache* and *FixCache* that record entities and files likely to introduce a bug and fix the bug if they are changed. The results of their empirical study showed that the caches 46% to 95% accuracy in seven open source projects.

Machine learning classification can be used for defect prediction as well. Kim et al[10] presented an approach to classifying software changes into buggy or clean ones. They used several features such as number of lines of added/deleted code, terms in change logs, and cyclomatic complexity. The authors conducted an empirical evaluation on 12 open source projects. The result shows 78% prediction accuracy on average. In addition, Shivaji et al[96] proposed a feature selection technique to improve the prediction performance of defect prediction. Features are not limited to metrics of source code; Jiang et al[97] built a prediction model based on individual developers. Defect prediction techniques are often faced with imbalanced datasets. Bird et al[98] pointed out that unfair and imbalanced datasets can lead to bias in defect prediction.

6.6 | Developer expertise

It is necessary to discuss developer expertise since influential changes imply that the developer who made the changes can be influential to other developers.

As the size of open-source software projects is getting larger, developer networks are naturally constructed and every activity in the network may affect other developers substantially. Hong et al[99] reported a result of observing a developer social network. The authors investigated Mozilla's bug tracking site to construct a developer social network (DSN). In addition, they collected general social networks (GSNs) from ordinary social communities such as Facebook and Amazon. This paper provides a comparison between DSN and GSNs. Findings described in this paper include: (a) DSN does not follow power-law degree distribution while GSNs do and (b) the size of communities in DSNs is smaller than that of GSNs. This paper also reports the result of an evolution analysis on DSNs. DSNs tend to grow over time but not much as GSNs do.

Onoue et al[100] studied and enumerated developer activity data in `Github.com`. It classifies good developers, tries to understand developers, and differentiates types of developers. However, the paper does not provide any further implication. In addition, there is no result of role analysis and social structure.

Pham et al[101] reported the results of a user study which has been conducted to reveal a testing culture in OSS. The authors have interviewed 33 developers of GitHub first and figured out the transparency of testing behaviors. Then, an online questionnaire has been sent to 569 developers of GitHub to find out testing strategies.

7 | CONCLUSION AND FUTURE WORK

In software revision histories, we can find many cases in which a few lines of software changes can positively or negatively influence the whole project, while most changes have only a local impact. In addition, those *influential changes* can constantly affect the quality of software for a long time. Thus, it is necessary to identify the influential changes at an early stage to prevent project-wide quality degradation or immediately take advantage of new software new features.

In this paper, we reported results of a postmortem analysis on 48 272 software changes that are systematically collected from 10 open source projects and labeled based on key quantifiable criteria. We then used open-card sorting to propose categories of influential changes. After developers have validated these categories, we consider examples of influential changes and extract features such as complexity and terms in

change logs in order to build a classification model. We showed that the classification features are efficient beyond the scope of our initial labeled data on influential changes.

The findings presented in this paper may shed light on how to interpret and assess a code change. As our study introduces different ways to detect ICs with a postmortem analysis, developers would recognize the presence and the impact of ICs. In addition, this study investigates the potential characteristics of ICs, which may help follow-up studies formally define ICs. Further investigation can discover more characteristics of ICs, which can be used for building a prediction model of ICs. Earlier identification of ICs can minimize the negative impact of such changes and provide a better understanding of the software system under development.

Our future work will focus on the following topics:

- Influential changes may affect the popularity of projects. We will investigate the correlation between influential changes and popularity metrics such as the number of new developers and new fork events.
- In our study, we used only metrics for source code. However, features of developers can have correlations with influential changes. We will study whether influential changes can make developer influential and vice versa.
- Once influential changes are identified, it is worth finding out who can benefit from the changes. Quantifying the impact of the influential changes to developers and users can significantly encourage further studies.

ACKNOWLEDGEMENT

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.

AVAILABILITY

We make available all our observational study results, extracted feature vectors, and developer survey results in this work. See https://github.com/serval-snt-uni-lu/influential-changes.

ORCID

Dongsun Kim https://orcid.org/0000-0003-0272-6860

REFERENCES

1. Brudaru II, Zeller A. What is the long-term impact of changes? In: Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering; 2008; New York, NY, USA:30-32.
2. Ren X, Shah F, Tip F, Ryder BG, Chesley O. Chianti: a tool for change impact analysis of java programs. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications; 2004; New York, NY, USA:432-448.
3. Zhang L, Kim M, Khurshid S. FaultTracer: a change impact and regression fault analysis tool for evolving java programs. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering; 2012; New York, NY, USA:40:1-40:4.
4. Robillard MP, Dagenais B. Retrieving task-related clusters from change history. In: 15th Working Conference on Reverse Engineering, 2008. WCRE '08. Antwerp, Belgium: 2008:17-26.
5. Sherriff M, Williams L. Empirical software change impact analysis using singular value decomposition. In: 1st International Conference on Software Testing, Verification, and Validation. Lillehammer, Norway: April 2008:268-277.
6. Herzig KS. Capturing the long-term impact of changes. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2. New York, NY, USA: ACM: 2010:393-396.
7. Herzig K, Zeller A. Mining cause-effect-chains from version histories. In: 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE). Hiroshima, Japan: November 2011:60-69.
8. Herzig K, Just S, Rau A, Zeller A. Predicting defects using change genealogies. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). Pasadena, CA, USA: November 2013:118-127.
9. Valdivia Garcia H, Shihab E. Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories. New York, NY, USA: ACM; 2014: 72-81.
10. Kim S, Whitehead EJ, Zhang Y. Classifying software changes: clean or buggy? *IEEE Trans Softw Eng*. March 2008;34(2):181-196.
11. Beyer D, Noack A. Clustering software artifacts based on frequent common changes. In: Proceedings of the 13th international Workshop on Program Comprehension; 2005:259-268.
12. Lewis DD. Naive (Bayes) at forty: the independence assumption in information retrieval. *Machine Learning: ECML-98*. Berlin Heidelberg: Springer; April 1998:4-15en.
13. Alpaydin E. *Introduction to machine learning*. Cambridge, Mass: MIT Press; 2004.
14. Breiman L. Random Forests. *Mach Learn*. 2001;45(1):5-32.
15. Palix N, Saha S, Thomas G, Calvès C, Lawall JL, Muller G. Faults in Linux: ten years later. In: Asplos'11: Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems; 2011; Newport Beach, CA, USA: 305-318.
16. Padioleau Y, Lawall JL, Hansen RR, Muller G. Documenting and automating collateral evolutions in Linux device drivers. In: Eurosys'08: Proceedings of the 2008 ACM Sigops/Eurosys European Conference on Computer Systems; 2008; Glasgow, Scotland:247-260.

17. Padioleau Y, Lawall JL, Muller G. Understanding collateral evolution in linux device drivers. In: Proceedings of the 2006 ACMSIGOPS/EUROSYS European Conference on Computer Systems, Eurosys'06; 2006; Leuven, Belgium:59-71.
18. Bissyandé TF, Revéillère L, Lawall J, Muller G. Diagnosys: automatic generation of a debugging interface to the linux kernel. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2012; Essen, Germany:60-69.
19. Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyvanyk D. API Change and Fault Proneness: A Threat to the Success of Android Apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013; 2013; New York, NY, USA:477-487.
20. Dig D, Johnson R. How do APIs evolve? A story of refactoring. *J Softw Maint Evol Res Pract*. 2006;18(2):83-107.
21. Lawall J. Automating source code evolutions using coccinelle. Kernel Recipes – https://kernel-recipes.org/en/2013/; 2013.
22. Rousseeuw PJ, Driessen KV. A fast algorithm for the minimum covariance determinant estimator. *Technometrics*. 1999;41(3):212-223.
23. Leys C, Ley C, Klein O, Bernard P, Licata L. Detecting outliers: do not use standard deviation around the mean, use absolute deviation around the median. *J Exp Soc Psychol*. 2013;49(4):764-766.
24. Nielsen J. Card sorting to discover the users' model of the information space. NN/g – http://www.nngroup.com/articles/usability-testing-1995-sun-microsystems-website/; 1995.
25. Spencer D. Card sorting: a definitive guide. http://boxesandarrows.com/card-sorting-a-definitive-guide/; 2004.
26. Lientz BP, Swanson EB, Tompkins GE. Characteristics of application software maintenance. *Commun ACM*. June 1978;21(6):466-471. http://doi.acm.org/10.1145/359511.359522
27. Gousios G. The ghtorrent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13. Piscataway, NJ, USA: IEEE Press; 2013:233-236.
28. Hu M, Liu B. Opinion feature extraction using class sequential rules. In: Proceedings of AAAI 2006 Spring Sympoia on Computational Approaches to Analyzing Weblogs (AAAI-CAAW 2006); 2006; Palo Alto, USA:61-66.
29. Ohana B. Opinion mining with the SentWordNet lexical resource. Master's thesis. Technological University Dublin, Dublin; 2009.
30. Liu B. Sentiment analysis and subjectivity. In: Indurkhya N, Damerau FJ, eds. *Handbook of natural language processing, second edition*. Boca Raton, FL: CRC Press, Taylor and Francis Group; 2010: 627-666.
31. Thelwall M, Buckley K, Paltoglou G, Cai D, Kappas A. Sentiment strength detection in short informal text. *J Am Soc Inf Sci Technol*. 2010;61(12):2544-2558.
32. Brin S, Page L. The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the Seventh International Conference on World Wide Web 7, WWW7; 1998; Brisbane, Australia:107-117.
33. Freeman LC. A set of measures of centrality based on betweenness. *Sociometry*. March 1977;40(1):35-41.
34. Sabidussi G. The centrality index of a graph. *Psychometrika*. 1966;31(4):581-603.
35. Varrette S, Bouvry P, Cartiaux H, Georgatos F. Management of an academic HPC cluster: The UL experience. In: Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014). Bologna, Italy: IEEE; 2014.
36. Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP. SMOTE: synthetic minority over-sampling technique. *J Artif Intell Res*. 2002;16:321-357. https://doi.org/10.1613/jair.953
37. Montgomery DC, Runger GC, Hubele NF. *Engineering Statistics*. Hoboken, NJ: Wiley. 2001.
38. Shihab E, Hassan AE, Adams B, Jiang ZM. An Industrial Study on the Risk of Software Changes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York, NY, USA: ACM; 2012:62:1-62:11.
39. D'Ambros M, Lanza M, Lungu M. The evolution radar: Visualizing integrated logical coupling information. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. Shanghai, China: ACM; 2006:26-32.
40. Servant F, Jones JA. History slicing: Assisting code-evolution tasks. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York, NY, USA: ACM; 2012:43:1-43:11.
41. Girba T, Kuhn A, Seeberger M, Ducasse S. How developers drive software evolution. In: Eighth International Workshop on Principles of Software Evolution; 2005; Lisbon, Portugal, Portugal:113-122.
42. Dagenais B, Robillard MP. Semdiff: Analysis and recommendation support for api evolution. In: Proceedings of the 31st International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society; 2009:599-602.
43. Robbes R, Lungu M, Rüthlisberger D.. How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York, NY, USA: ACM; 2012:56:1-56:11.
44. Sawant AA, Robbes R, Bacchelli A. On the Reaction to Deprecation of 25,357 Clients of 4+1 Popular Java APIs. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME); October 2016; Raleigh, NC, USA:400-410.
45. Teyton C, Falleri J-R, Palyart M, Blanc X. A study of library migrations in java. *J Softw Evol Process*. 2014;26(11):1030-1052.
46. Kim E, Kim K, In HP. A multi-view api impact analysis for open spl platform. In: Proceedings of the 12th International Conference on Advanced Communication Technology. Piscataway, NJ, USA: IEEE Press; 2010:686-691.
47. Dagenais B, Robillard MP. Recommending adaptive changes for framework evolution. *ACM Trans Softw Eng Methodol*. 2011;20(4):19:1-19:35.
48. Wu W, Guéhéneuc YG., Antoniol G, Kim M. Aura: A hybrid approach to identify framework evolution. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 1; 2010; Cape Town, South Africa:325-334.
49. Zimmermann T, Weisgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society; 2004:563-572.
50. Ying ATT, Murphy GC, Ng R, Chu-Carroll MC. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*. 2004;30(9):574-586.
51. Shihab E, Mockus A, Kamei Y, Adams B, Hassan AE. High-impact Defects: A Study of Breakage and Surprise Defects. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. New York, NY, USA: ACM; 2011:300-310.
52. Pan K, Kim S, Whitehead EJ. Toward an understanding of bug fix patterns. *Empir Softw Eng*. 2009;14(3):286-315.

53. Zhong H, Su Z. An Empirical Study on Real Bug Fixes. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. ACM; 2015; Florence, Italy:913-923.
54. Tan SH, Yoshida H, Prasad MR, Roychoudhury A. Anti-patterns in Search-based Program Repair. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, NY, USA: ACM; 2016:727-738.
55. Liu K, Kim D, Koyuncu A, Li L, Bissyandé TF, Traon YLe. A Closer Look at Real-World Patches. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME); 2018; Madrid, Spain:275-286.
56. Koyuncu A, Bissyandé TF, Kim D, Klein J, Monperrus M, Le Traon Y. Impact of Tool Support in Patch Construction. In: Proceedings of the 26th ACM Sigsoft International Symposium on Software Testing and Analysis. New York, NY, USA: ACM; 2017:237-248.
57. Meng N, Kim M, McKinley KS. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*. 2011;46(6):329-342.
58. Meng N, Kim M, McKinley KS. LASE: locating and applying systematic edits by learning from examples. In: 2013 35th International Conference on Software Engineering (ICSE). San Francisco, CA, USA: ACM; 2013:502-511.
59. Meng N, Hua L, Kim M, McKinley KS. Does automated refactoring obviate systematic editing? In: Proceedings of the 37th International Conference on Software Engineering, Vol. 1. Piscataway, NJ, USA: ACM; 2015:392-402.
60. Son S, McKinley KS, Shmatikov V. RoleCast: Finding Missing Security Checks when You Do Not Know What Checks Are. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. ACM; 2011; New York, NY, USA:1069-1084.
61. Rolim R, Soares G, D'Antoni L, et al. Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering. Buenos Aires, Argentina: ACM; 2017:404-415.
62. Long F, Amidon P, Rinard M. Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. New York, NY, USA: ACM; 2017:727-739.
63. Koyuncu A, Liu K, FBissyandé T, et al. Fixminer: mining relevant fix patterns for automated program repair. arXiv preprint arXiv:181001791; 2018.
64. Rolim R, Soares G, Gheyi R, D'Antoni L. Learning quick fixes from code repositories. arXiv preprint arXiv:180303806; 2018.
65. Github. PMD: an extensible cross-language static code analyzer. https://pmd.github.io/; Last Accessed: Nov. 2017.
66. Liu K, Kim D, Bissyande TF, Yoo S, Traon YLe. Mining fix patterns for FindBugs violations. *IEEE Trans Softw Eng*. 2019. (to appear). https://doi.org/10.1109/TSE.2018.2884955
67. Liu K, Koyuncu A, Kim D, Bissyandè TF. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER); 2019; Hangzhou, China, China:1-12.
68. Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering. Vancouver, BC, Canada: IEEE; 2009:364-374.
69. Le Goues C, Nguyen T, Forrest S, Weimer W. Genprog: a generic method for automatic software repair. *IEEE Trans Softw Eng*. 2012;38(1):54.
70. Fry ZP, Landau B, Weimer W. A human study of patch maintainability. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. Minneapolis, MN, USA: ACM; 2012:177-187.
71. Le Goues C, Dewey-Vogt M, Forrest S, Weimer W. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering. Zurich, Switzerland: IEEE; 2012:3-13.
72. Kim D, Nam J, Song J, Kim S. Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering. San Francisco, CA, USA: ACM; 2013:802-811.
73. Le XD, Lo D, Le Goues C. History Driven Program Repair. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering. Suita, Japan: IEEE; 2016:213-224.
74. Liu K, Anil K, Kim K, Kim D, Bissyandé TF. Live search of fix ingredients for automated program repair. In: Proceedings of the 25th Asia-Pacific Software Engineering Conference; 2018; Nara, Japan: 658-662.
75. Nguyen HDT, Qi D, Roychoudhury A, Chandra S. Semfix: program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering. San Francisco, CA, USA: IEEE; 2013:772-781.
76. Mechtaev S, Yi J, Roychoudhury A. Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering. Austin, TX, USA: ACM; 2016:691-701.
77. Mechtaev S, Yi J, Roychoudhury A. Directfix: looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering, Vol. 1. Florence, Italy: ACM; 2015:448-458.
78. Wei Y, Pei Y, Furia CA, et al. Automated fixing of programs with contracts. In: Proceedings of the 19th International Symposium on Software Testing and Analysis. Trento, Italy: ACM; 2010:61-72.
79. Dallmeier V, Zeller A, Meyer B. Generating fixes from object behavior anomalies. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. Auckland, New Zealand: ACM; 2009:550-554.
80. Xuan J, Martinez M, Demarco F, et al. Nopol: automatic repair of conditional statement bugs in java programs. *IEEE Trans Softw Eng*. 2017;43(1):34-55.
81. Kaleeswaran S, Tulsian V, Kanade A, Orso A. Minthint: automated synthesis of repair hints. In: Proceedings of the 36th International Conference on Software Engineering. New York, NY, USA: ACM; 2014:266-276.
82. Tao Y, Kim J, Kim S, Xu C. Automatically generated patches as debugging aids: a human study. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering>. Hong Kong: ACM; 2014:64-74.
83. Bissyandé TF. Harvesting fix hints in the history of bugs. arXiv preprint arXiv:150705742; 2015.
84. Nistor A, Chang P-C, Radoi C, Lu S. Caramel: detecting and fixing performance problems that have non-intrusive fixes. In: Proceedings of the 37th International Conference on Software Engineering, Vol. 1. Florence, Italy: IEEE; 2015:902-912.
85. Monperrus M. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th International Conference on Software Engineering. New York, NY, USA: ACM; 2014:234-242.
86. Barr ET, Brun Y, Devanbu P, Harman M, Sarro F. The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM; 2014:306-317.

87. Long F, Rinard M. An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings of the 38th International Conference on Software Engineering. Austin, TX, USA: IEEE; 2016:702-713.
88. Smith EK, Barr ET, Le Goues C, Brun Y. Is the cure worse than the disease? Overfitting in automated program repair. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. New York, NY, USA: ACM; 2015:532-543.
89. Le Goues C, Holtschulte N, Smith EK, et al. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans Softw Eng*. 2015;41(12):1236-1256.
90. Just R, Jalali D, Ernst MD. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. New York, NY, USA: ACM; 2014:437-440.
91. Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empir Softw Eng*. 2017;22(4):1936-1964.
92. Liu K, Koyuncu A, Kim D, Bissyandé TF. TBar: revisiting template-based automated program repair. arXiv:190308409 [cs]; March 2019.
93. Śliwerski J, Zimmermann T, Zeller A. HATARI: raising risk awareness. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, NY, USA: ACM; 2005:107-110.
94. Kim S, Zimmermann T, Pan K, Whitehead EJ. Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE '06; 2006; Tokyo, Japan:81-90.
95. Kim S, Zimmermann T, Whitehead Jr. EJ, Zeller A. Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society; 2007:489-498.
96. Shivaji S, Whitehead Jr. EJ, Akella R, Kim S. Reducing features to improve bug prediction. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. Washington, DC, USA: IEEE Computer Society; 2009:600-604.
97. Jiang T, Tan L, Kim S. Personalized defect prediction. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE); 2013; Silicon Valley, CA, USA:279-289.
98. Bird C, Bachmann A, Aune E, et al. Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. New York, NY, USA: ACM; 2009:121-130.
99. Hong Q, Kim S, Cheung SC, Bird C. Understanding a developer social network and its evolution. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM); 2011; Williamsburg, VI, USA:323-332.
100. Onoue S, Hata H, Matsumoto K-I. A study of the characteristics of developers' activities in GitHub. In: 2013 20th Asia-Pacific Software engineering conference (APSEC); 2013; Bangkok, Thailand: 7-12.
101. Pham R, Singer L, Liskin O, Figueira Filho F, Schneider K. Creating a shared understanding of testing culture on a social coding site. In: Proceedings of the 2013 International Conference on Software Engineering. Piscataway, NJ, USA: IEEE Press; 2013:112-121.

How to cite this article: Li D, Li L, Kim D, Bissyandé TF, Lo D, Le Traon Y. Watch out for this commit! A study of influential software changes. *J Softw Evol Proc*. 2019;e2181. https://doi.org/10.1002/smr.2181