Watch out for This Commit! A Study of Influential Software Changes

Daoyuan Li
University of Luxembourg
daoyuan.li@uni.lu

Li Li
University of Luxembourg
li.li@uni.lu

Dongsun Kim
University of Luxembourg
dongsun.kim@uni.lu

Tegawendé F. Bissyandé
University of Luxembourg
tegawende.bissyande@uni.lu

David Lo
Singapore Management Univ
davidlo@smu.edu.sg

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

ABSTRACT

One single code change can significantly influence a wide range of software systems and their users. For example, 1) adding a new feature can spread defects in several modules, while 2) changing an API method can improve the performance of all client programs. Developers often may not clearly know whether their or others' changes are influential at commit time. Rather, it turns out to be influential after affecting many aspects of a system later.

This paper investigates influential software changes and proposes an approach to identify them early, i.e., immediately when they are applied. We first conduct a post-mortem analysis to discover existing influential changes by using intuitions such as *isolated changes* and *changes referred by other changes* in 10 open source projects. Then we re-categorize all identified changes through an open-card sorting process. Subsequently, we conduct a survey with 89 developers to confirm our influential change categories. Finally, from our ground truth we extract features, including metrics such as the complexity of changes, terms in commit logs and file centrality in co-change graphs, to build machine learning classifiers. The experiment results show that our prediction model achieves overall with random samples 86.8% precision, 74% recall and 80.4% F-measure respectively.

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 simply cosmetic or provide minor improvements, others have a wide and long-term influence to the entire system and related systems. Brudaru and Zeller [7] first illustrated examples of changes with long term-influence: 1) changing access privilege (i.e., `private` → `public`), 2) changing kernel lock mechanism, and 3) 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 (e.g., easing debugging if a new test harness is added) and provide insights for recommendation systems (e.g., 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 (e.g., new defects in 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 of this failure (missed file) was hard to pinpoint.

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 five 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 [36,48] 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 [37,41]. 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 (e.g., files or methods) in the cluster may be affected by the given change. Finally, change genealogy [14–16] 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 den-

sity. 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.

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 prediction model to identify influential software changes immediately after they are applied?

To automatically figure out whether an incoming software change is influential, we designed a prediction technique based on machine learning classification. 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 long-term influence to revision histories. As a result, we could discover several influential changes from each subject. We further label these changes to build category definition for influential software changes through an open-card sorting process. These categories are then validated by developers with experience in code review.

Based on the influential changes we discovered in the above study, we extracted feature vectors for machine-learning classification. These features include program structural metrics [20], terms in change logs [20], and co-change metrics [2]. Then, we built a prediction model by leveraging machine learning algorithms such as Naïve Bayes [1,24] and Random Forest [5]. 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, subset of our data show that our prediction model achieves overall 86.8% precision, 74% recall, and 80.4% F-measure performance.

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 prediction 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 prediction 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 prediction model to open source projects. Section 5 discusses the limitations of our work. After surveying the related work in Section 6, we conclude with directions for future research in Section 7.

2. MOTIVATING EXAMPLES

In the development course of software project, developers regularly commit changes to project repositories. While some of those changes may simply be cosmetic, a few others may be somehow influential not only inside and/or beyond the repositories but also immediately and/or long after they are applied. An influential software change can be recognized as such for various reasons, not all of which are known while the change is being performed.

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 [32,34] as their long-term impact started to be noticed. In this section, we present four different examples of influential changes in 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 [33].

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 parse their own code to understand which context it should be as in the example of Figure 1.

This leads to bugs that keep occurring. A study by Pallix *et al.* [34] has reported that, due to 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 [36, 37, 41, 48] focus only on the short-term impact.

2.2 Feature Replacement

In general, the number of entries in each fault category (e.g., *NULL* or *Lock*) decreases over time in the Linux code base [34]. 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[1] category in Linux 2.6.16 due to a replacement of func-

[1]To avoid *Lock/LockIntr* faults, release acquired locks, restore disable interrupts and do not double acquire locks [4, 34].

```
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.

tionality 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 9 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 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 [9,26]. The developer who committed the new feature did not realize the influence and thus, there was no early heads-up for other developers.

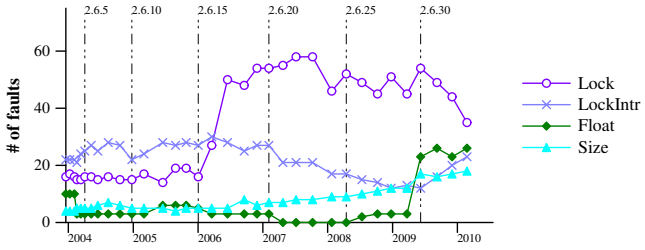

Figure 2: Evolution of faults in Linux 2.6 kernel versions for *Lock*, *LockIntr*, *Float* and *Size* fault categories (see [34]). Faults relevant to *Lock* suddenly increased after Version 2.6.16 while other types of faults gradually decreased. In the version, a feature for Lock was replaced and it was influential to many of kernel functions.

2.3 Revolutionary Feature

An obvious influential change may consist in providing an implementation of a completely new feature, e.g., 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, *devm* could have been adopted earlier and result in less bugs and better performance in driver code.

2.4 Fixes of Controversial/Popular Issues

Some issues in software projects can be abnormally discussed or commented longer than others. Code changes that fix them will be influential for the project. The characteristics of a controversial/popular issue is 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.

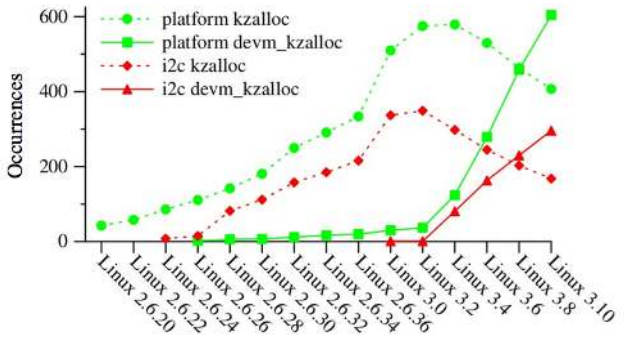

Figure 3: Usage of memory allocation primitives in Linux kernel (See [23]). *kzalloc* is the traditional API for memory allocation, before managed memory (*devm*) was introduced in Linux.

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: 1) popular changes in the sense that they have been somehow noticed by other developers and users, 2) anomalies in change behaviors, and 3) 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, 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 an 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

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	2,983	2,722	47	542	512
Commons-compress	Library for working with file compression	619	1,716	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	1,718	33	454	365
Commons-lang	Extra-functionality for java.lang	1,294	4,103	46	1,073	933
Commons-math	Mathematics & Statistics components	4,582	5,496	37	1,194	1,085
Spring-framework	Application framework for the Java platform	19,721	9,748	153	3,500	2,632
Storm	Distributed real-time computation system	2,038	3,534	189	637	321
Wildfly	aka JBoss Application Server	31,699	16,855	307	3,710	2,993
Total		64,388	48,272	878	11,760	9,409

have 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: e.g., many commit messages explicitly refer to the unique ID of the issue they are addressing, whether a bug or a feature request.

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 of 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 about 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 criteria, we could identify a set of issues that are controversial/popular. Afterwards, we collected all commits that were associated to each of the controversial/popular issues and tag them as potentially influential.

An example was found in Apache Math. An issue[2] with 62 comments was detected by this analysis. This issue is about a simple glitch of an API method; the API hangs 4–5 seconds at the first call on a specific Android device. The corresponding changes[3] 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 1) the discussion indeed was about a controversy and 2) 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 the number of influential changes manually confirmed among those changes.

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

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, e.g., a major bug fix. In our observational study we consider both break in behaviors in the edit rhythm of each files 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 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 [38] to identify "isolated commits". In Figure 4, we can visualize the outliers discovered for the changes on

[2]https://issues.apache.org/jira/browse/MATH-650

[3]Commits `52649fda4c9643afcc4f8cbf9f8527893fd129ba` and `0e9a5f40f4602946a2d5b0efdc75817854486cd7`

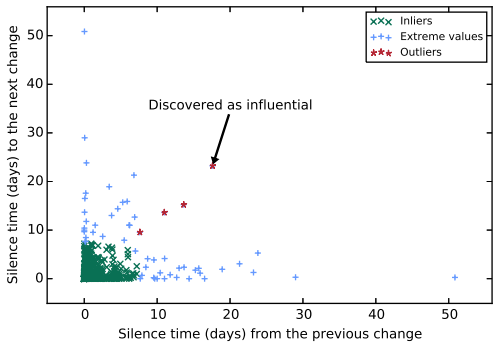

Figure 4: Outlier detection to discover isolated commits for *build.gradle* file in `Spring framework`.

the *build.gradle* file from the `Spring` project subject. The highlighted outlier represents a commit[4] for including AspectJ files in the Spring-sources jar file. This small commit is influential as it fixes the build of Spring-framework.

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 (i.e., 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[5] 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 1) its importance is clearly stated in the change log and 2) the implication of the change for dependent modules and client applications is apparent. Table 3 provides the statistics on detected isolated commits and the results of our manual analysis on those commits to confirm influential changes.

Table 3: Statistics of identified isolated commits and the associated manually confirmed influential changes.

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

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 (e.g., 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[6] `93089b26` is referred by another commit[7]. This commit implemented the capability to detect start of line, which is surely 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 1) it is indeed referenced by others because it was inducing their changes, and 2) 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.

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

3.3 Qualitative Assessment Results

We then set to assess the quality of the metrics used in our observational study. We manually checked all potential influential changes yielded by the systematic analysis. We further randomly pick change commits from each project and manually check the percentage of changes that are influential. The comparison between the two types of datasets aimed at validating our choices of post-mortem metrics to easily collect influential changes. Table 5 provides 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 highest rates of "truly" influential changes (an order of magnitude more than what can be identified in random samples).

Conclusion: *The difference in influential rate values with random shows that our post-mortem 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 post-mortem 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 resorted to open-card sorting [29], a well known, reliable

[4]Commit `a681e574c3f732d3ac945a1dda4a640ce5514742`
[5]Commit `fadbb4cc0e9ca11c371c87ce042fd596b13eb092`
[6]Commit `93089b260cd2030d69b3f7113ed643b9af1adcaa`
[7]Commit `05b5c8ef488d5d230d665b9d488ca572bec5dc0c`

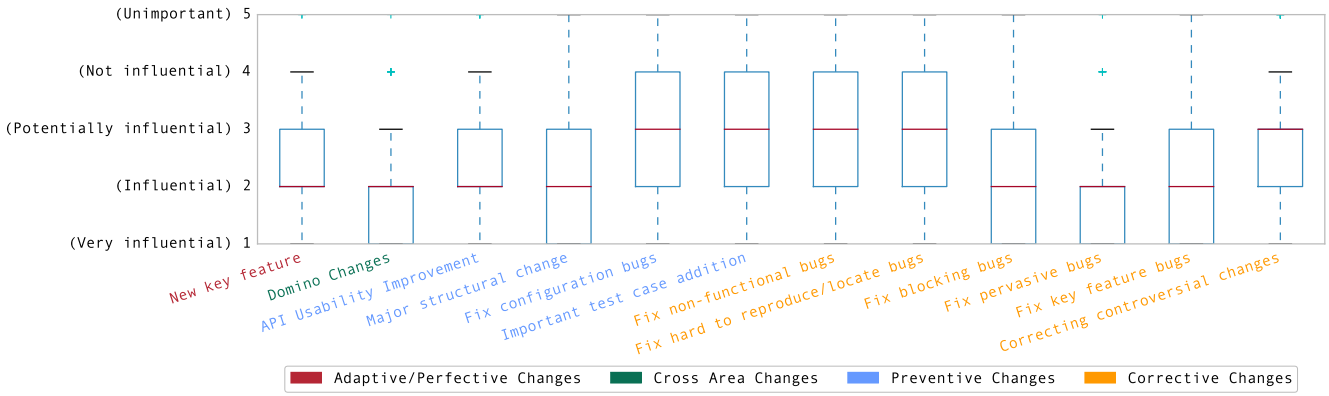

Figure 5: Survey results on different categories of Influential Changes.

Table 5: Qualitative assessment results. We compared the percentage of changes that were actually manually confirmed to be influential from the datasets yielded by our systematic analysis and a random selection in projects. Note that we count unique commits in this table, since some commits fall into more than one categories.

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

and user-centered method for building a taxonomy of a system [44]. 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 description of identified influential software changes[8] 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, then they met to perform another open-card sorting with cards containing their group descriptions. 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.

The influential software changes described in the 12 categories span over four software maintenance categories initially defined by Lientz *et al.* [25] 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 describe. The survey participants have been selected from data collected in the GHTorrent project [13] which contains history archives on user activities and repository changes in GitHub. We consider active developers (i.e., those who have contributed in 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[9] 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. 66 (i.e., 74%) of these developers 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 experience in software development. 29 (33%) developers have between five and ten years of experience. The remaining 34 (38%) have over ten 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. The participant was then requested to assess the relevance of this category of changes as influential software using a Likert scale between `1:very influential` and `5:unimportant`. Figure 5 summarizes the survey results. For more detailed description of the categories, we refer the reader to the project web site (see Section "Availability").

The survey results suggest that:

- According to software developers with code review experience, all 12 categories are about important changes: 7 categories have an average agreement of 2 (i.e., Influential), the remaining 5 categories have an average of 3 (i.e., potentially influential). Some (e.g., "domino changes" and "changes fixing pervasive bugs") are clearly found as more influential than others (e.g., "important test case addition").
- Some changes, such as "fixes for hard to reproduce or locate bugs", are not as influential as one might think.
- Developers also suggested two other categories of influential changes: *Documentation changes* and *Design-*

[8]We consider all 177 influential software changes from the post-mortem analysis.

[9]Survey form at `https://goo.gl/V2g8OE`

phase changes. The latter however are challenging to capture in source code repository artefacts, while the former are 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. We thus consider leveraging on the code characteristics of these identified samples to identify more influential changes.

4. LEARNING TO PREDICT ICs

Beyond the observational study reported in Section 3, we propose an approach to identify influential changes on-the-fly. The objective is to predict, when a change is being submitted, whether it should be considered with care as it could be influential. 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 post-mortem 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, quantity of edits performed by the change and so on. Based on the experience gathered during manual analysis of influential changes in Section 3, we extract a number of features to feed the ML algorithms.

4.1.1 Structural features

First we consider common metrics that provide hints on structural characteristics of a change. These metrics include (1) the number of files simultaneously changed in a single commit, (2) the number of lines added to the program repository by the commit and (3) 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* [24] 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 polarility of commit messages based on *sentiment analysis* [18, 27, 30, 45] techniques.

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 [6] 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. 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 is modified and PageRank values are changed. We build a feature vector taking into account these changes on the minimum and maximum PageRank values.

Centrality metrics are commonly used in social network analysis to determine influential people, or in Internet networks to identify key nodes. In our experiments, we focus on computing *betweeness centrality* [11] and *closeness centrality* [39] metrics for all files associated to a commit change. We build features by computing the deltas in the sum of centrality metrics between the metrics computed for files involved in previous commits and for files involved in current commit.

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 [1, 24] and Random Forest [5].

In the process of our validation tests, we are interested in assessing: 1) Whether connectivity on co-change graphs correlates with a probability for a relevant change to be an IC; 2) If natural language information in commit messages are indicative of ICs; 3) If structural information of changes are indicative of ICs; 4) Whether combinations of features is best for predicting ICs; 5) If our approach can discover ICs beyond the types of changes discovered with post-mortem analysis.

4.2.1 Experiment Setup

To compute the feature vectors for training the classifiers, we used a high-performance computing system [46] to run parallel tasks for building co-change graphs for the various project subjects. After extracting the feature metrics, we preprocess the data and ran ten-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 Oversampling Techniques (SMOTE) [10].

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 [1, 20, 28]. **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 6: Performance comparison using 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 (Influential Class)	NB	95.1	92.9	91.5	84.2	98.5	89.2	94.3	86.1	91.5
	RF	97.4	96.4	98.2	77.8	97.0	95.0	99.1	97.8	94.8
F-Measure (Non Influential Class)	NB	93.5	87.5	83.9	92.7	98.1	79.5	92.6	86.5	89.3
	RF	97.0	93.9	97.1	90.5	96.3	92.9	98.9	97.5	95.5

4.2.2 Assessment Results

In the following paragraphs, we detail the prediction results for influential changes using ten-fold cross validation on labelled data. In addition, this section describes the result of influential change prediction in the wild.

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 k^{th} portion of the data. We perform ten-fold cross validation on the labelled 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 6 depicts the F-measure performance in ten-fold cross validation for the two algorithms. Although Random Forest performs on average better than Naïve Bayes, this difference is relatively small.

Table 7 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, the natural language commit message, 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 7 due to 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 to yield 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 7 also highlight the fact that, on average, combining different features contributes to improve the performance of influential change prediction. 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.

4.2.3 Generalization of Influential Change Features

In previous experiments, we have tested the machine learning classifier with influential change data labelled 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 post-mortem 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.

We randomly sample a significant set of changes within our dataset of 10 projects commits. Out of the 48,272 commits from the dataset, we randomly consider 381 commits (i.e., the exact number provided by the Sample Size Cal-

Table 7: 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.

Project Name	Metrics	Influential Class			Non-Influential Class		
		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

culator[10] 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 ten-fold cross validation using the same features presented in Section 4.2.1 for influential change classification. The results are presented in Table 8.

Table 8: 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

The precision of ten-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 1) 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), 2) 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: 1) Co-change metrics allow to successfully predict influential changes with an average 87.6% F-measure; 2) Features based on terms in commit messages can predict influential changes with high precision (average of 94.9%) and recall (average of 94.4%); 3) Structural features can be leveraged to successfully predict influential changes with an average F-measure performance of 80.5%; 4) Overall, combining features often achieves a better prediction performance than individual feature groups. For example, combining all features showed 96.1% precision, 94.9% recall, and 95.2% F-measure on average; 5) With the features we collected, our prediction approach has an acceptable performance in the wild, i.e., with different types of influential changes (beyond the ones we relied upon to infer the features).

5. THREATS TO VALIDITY

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

Internal validity. The authors have manually labelled themselves the influential changes as it was prohibitively costly to request labelling 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 to 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.

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.

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 labelled following new category labels of influential changes approved by developers.

6. RELATED WORK

This section discusses four groups of related work; 1) software evolution, 2) change impact analysis, 3) defect prediction, and 4) 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. [8] 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. [12] proposed a metric called *code ownership* to illustrate how developers drive software evolution. We used the metric to examine the influence of a change.

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. [36] 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 help developers browse a change impact set of a certain atomic change. FaultTracer [48] 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 [7] 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 long-term impact on quality, maintainability, and stability [16]. In addition, it can reveal cause-effect chains [15] and predict defects [14].

[10] `http://www.surveysystem.com/sscalc.htm`

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. [47] 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 [37] 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 [41] 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.

6.3 Defect Prediction

Changing a program may often introduce faults [21, 43]. Thus, fault prediction at an early stage can lead developers to achieving a better software quality. Kim et al. [22] 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 46-95% accuracy in seven open source projects.

Machine learning classification can be used for defect prediction as well. Kim et al. [20] 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. [42] 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. [19] built a prediction model based on individual developers. Defect prediction techniques are often faced with imbalanced datasets. Bird et al. [3] pointed out that unfair and imbalanced datasets can lead to bias in defect prediction.

6.4 Developer Expertise

It is necessary to discuss developer expertise since influential changes implies 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. [17] 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 the comparison between DSN and GSNs. Findings described in this paper include 1) DSN does not follow power law degree distribution while GSNs do, 2) the size of communities in DSNs is smaller than that of GSNs. This paper also reports the result of evolution analysis on DSNs. DSNs tend to grow overtime but not much as GSNs do.

Onoue et al. [31] studied and enumerates 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 for role analysis and social structure.

Pham et al. [35] reported the results of a user study which has been conducted to reveal 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 post-mortem analysis on 48,272 software changes that are systematically collected from 10 open source projects and labelled based on key quantifiable criteria. We then used open-card sorting to propose categories of influential changes. After developer 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 prediction model. We showed that the classification features are efficient beyond the scope of our initial labeled data on influential changes. 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.

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`.

8. REFERENCES

[1] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, 2004.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 259–268, 2005.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. 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*, ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[4] T. Bissyandé, L. Revéillère, J. Lawall, and G. Muller. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 60–69, Sept 2012.

[5] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, Brisbane, Australia, 1998.

[7] I. I. Brudaru and A. Zeller. What is the long-term impact of changes? In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, RSSE '08, pages 30–32, New York, NY, USA, 2008. ACM.

[8] M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: visualizing integrated logical coupling information. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 26–32, Shanghai, China, 2006. ACM. ACM ID: 1137992.

[9] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, Mar. 2006.

[10] N. V. C. et. al. Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[11] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, Mar. 1977.

[12] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution*, pages 113–122, Sept. 2005.

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

[14] K. Herzig, S. Just, A. Rau, and A. Zeller. Predicting defects using change genealogies. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 118–127, Nov. 2013.

[15] K. Herzig and A. Zeller. Mining cause-effect-chains from version histories. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 60–69, Nov. 2011.

[16] K. S. Herzig. Capturing the long-term impact of changes. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 393–396, New York, NY, USA, 2010. ACM.

[17] Q. Hong, S. Kim, S. Cheung, and C. Bird. Understanding a developer social network and its evolution. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 323–332, Sept. 2011.

[18] M. Hu and B. Liu. Opinion feature extraction using class sequential rules. In *Proceedings of AAAI 2006 Spring Sympoia on Computational Approaches to Analyzing Weblogs (AAAI-CAAW 2006)*, 2006.

[19] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 279–289, Nov. 2013.

[20] S. Kim, E. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, Mar. 2008.

[21] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE '06*, pages 81–90, Sept. 2006.

[22] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[23] J. Lawall. Automating source code evolutions using coccinelle, 2013. Kernel Recipes – `https://kernel-recipes.org/en/2013/`.

[24] D. D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. In C. Nédellec and C. Rouveirol, editors, *Proceedings of 10th European Conference on Machine Learning*, number 1398, pages 4–15. Springer Verlag, Heidelberg, DE, 1998.

[25] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.

[26] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 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, pages 477–487, New York, NY, USA, 2013.

[27] B. Liu. Sentiment analysis and subjectivity. In N. Indurkhya and F. J. Damerau, editors, *Handbook of Natural Language Processing, Second Edition*. CRC Press, Taylor and Francis Group, Boca Raton, FL, 2010. ISBN 978-1420085921.

[28] D. Montgomery, G. Runger, and N. Hubele. *Engineering Statistics*. Wiley, 2001.

[29] J. Nielsen. Card sorting to discover the users' model of the information space, May 1995. NN/g – `http://www.nngroup.com/articles/usability\-testing-1995-sun-microsystems-website/`.

[30] B. Ohana. Opinion mining with the SentWordNet lexical resource. *Dissertations*, Mar. 2009.

[31] S. Onoue, H. Hata, and K.-I. Matsumoto. A study of the characteristics of developers' activities in GitHub. In *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, pages 7–12, Dec. 2013.

[32] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys'08: Proceedings of the 2008 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 247–260, Glasgow, Scotland, 2008.

[33] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *EuroSys'06: Proceedings of the 2006 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–71, Leuven, Belgium, 2006.

[34] N. Palix, S. Saha, G. Thomas, C. Calvès, J. L. Lawall, and G. Muller. Faults in Linux: Ten years later. In *ASPLOS'11: Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, 2011.

[35] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 112–121, Piscataway, NJ, USA, 2013. IEEE Press.

[36] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. 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*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[37] M. Robillard and B. Dagenais. Retrieving task-related clusters from change history. In *15th Working Conference on Reverse Engineering, 2008. WCRE '08*, pages 17–26, 2008.

[38] P. J. Rousseeuw and K. V. Driessen. A fast algorithm for the minimum covariance determinant estimator. *Technometrics*, 41(3):212–223, 1999.

[39] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[40] F. Servant and J. A. Jones. History slicing: Assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 43:1–43:11, New York, NY, USA, 2012. ACM.

[41] M. Sherriff and L. Williams. Empirical software change impact analysis using singular value decomposition. In *1st International Conference on Software Testing, Verification, and Validation*, pages 268–277, Apr. 2008.

[42] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim. Reducing features to improve bug prediction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 600–604, Washington, DC, USA, 2009. IEEE Computer Society.

[43] J. Śliwerski, T. Zimmermann, and A. Zeller. 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*, ESEC/FSE-13, pages 107–110, New York, NY, USA, 2005. ACM.

[44] D. Spencer. Card sorting: a definitive guide, April 2004. `http://boxesandarrows.com/card-sorting-a-definitive-guide/`.

[45] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas. Sentiment strength detection in short informal text. *Journal of the American Society for Information Science and Technology*, 61(12):2544–2558, Dec. 2010.

[46] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. 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, July 2014. IEEE.

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

[48] L. Zhang, M. Kim, and S. Khurshid. 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*, FSE '12, pages 40:1–40:4, New York, NY, USA, 2012. ACM.

[49] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.