Noname manuscript No.
(will be inserted by the editor)

Studying High Impact Fix-Inducing Changes

Ayse Tosun Misirli · Emad Shihab · Yasukata Kamei

the date of receipt and acceptance should be inserted later

Abstract As software systems continue to play an important role in our daily lives, their quality is of paramount importance. Therefore, a plethora of prior research has focused on predicting components of software that are defect-prone. One aspect of this research focuses on predicting software changes that are fix-inducing. Although the prior research on fix-inducing changes has many advantages in terms of highly accurate results, it has one main drawback: It gives the same level of impact to *all* fix-inducing changes. We argue that treating all fix-inducing changes the same is not ideal, since a small typo in a change is easier to address by a developer than a thread synchronization issue.

Therefore, in this paper, we study *high impact fix-inducing changes (HIFCs)*. Since the impact of a change can be measured in different ways, we first propose a measure of impact of the fix-inducing changes, which takes into account the implementation work that needs to be done by developers in later (fixing) changes. Our measure of *impact* for a fix-inducing change uses the amount of churn, the number of files and the number of subsystems modified by developers during an associated *fix* of the fix-inducing change. We perform our study using six large open source projects to build specialized models that identify HIFCs, determine the best indicators of HIFCs and examine the benefits of prioritizing HIFCs. Using change factors, we are able to predict 56% to 77% of HIFCs with an average false alarm (misclassification) rate of 16%. We find that the lines of code added, the number of developers who worked on a change, and the number of prior modifications on the files modified during a change are the best indicators of HIFCs. Lastly, we observe that a specialized model for HIFCs can provide inspection effort savings of 34% over the state-of-the-art models. We believe our results would help practitioners prioritize their efforts towards the most impactful fix-inducing changes and save inspection effort.

Ayse Tosun Misirli
Department of Computer Engineering, Istanbul Technical University, Turkey,
E-mail: tosunmisirli@itu.edu.tr

Emad Shihab
Department of Computer Science and Software Engineering, Concordia University, Canada,
E-mail: eshihab@cse.concordia.ca

Yasutaka Kamei
Graduate School and Faculty of Information Science and Electrical Engineering, Kyushu University, Japan,
E-mail: kamei@ait.kyushu-u.ac.jp

Keywords Change metrics, fix-inducing changes, change impact

1 Introduction

Software quality is of paramount importance to software organizations. However, there always exists a trade-off between cost (including time to release) and quality. Therefore, a large amount of software engineering research has focused on helping software organizations improve the prioritization of their quality assurance efforts. The majority of this prior research has focused on building bug prediction models at different code granularities (i.e., method, file, package) (e.g., [20, 46, 49, 64, 67]). These models are extended by increasing the information content of predictions [9], while some of them are also built for specific types of bugs, e.g., reopened bugs [8, 57], or breakage bugs [59].

Recent work has argued that the usefulness of file- and package-level bug predictions are of limited use [18]. These limitations are mainly due to the fact that the recommendations are too coarse grained (i.e., addressing an entire file or package requires too much effort) [18, 29]. They are also not easily assignable to developers, since many developers may work on the same files or packages [32, 55], or testers, since recommended code parts are not always linked to test interfaces [43]. Therefore, change-level prediction models are proposed, which predict code changes that are more likely to induce bugs, namely fix-inducing changes, at the change/commit level (e.g., [29, 32, 56, 61]). The advantages of change-level predictions are that 1) the predictions are made at a fine granularity, since changes can be mapped to a small area of the code, 2) the predictions can be easily and swiftly assigned, since each change has a specific committer that can address the change immediately after it is committed.

However, one drawback of prior change-level prediction work is the fact that it does not take into consideration the impact of fix-inducing changes. In other words, the prior work treats all fix-inducing changes the same. We argue that all fix-inducing changes do not have the same impact, and that distinguishing the high impact changes will improve the prioritization of implementation activities. For example, a change that fixes a wrong or missing link in the help files is much less impactful than a change that fixes a bug due to incorrect synchronization of threads.

In change-impact analysis research, the impact of a bug can mean different things to different stakeholders. For example, a high impact change to a developer may be the one that requires more time and work to debug and find the root cause, to fix the issue or to run associated tests [34]. The impact of a change done during a debugging, testing or refactoring activity could also be different than that spent during a bug fix depending on the problem (e.g. synchronization problem). On the other hand, a high impact change to a customer may be the one that distorts the core functionality that they depend on. In this paper, we focus on high impact fix-inducing changes (HIFCs) as seen from a *developer's* perspective. In particular, we define impact in terms of *the amount of churn, the number of modified files and the number of subsystems where the modified files span* affected during a fix, and quantify the most impactful *(high impact) fix-inducing changes* for developers.

We define our research questions (RQs) as follows:

- RQ1: Can we accurately identify high impact fix-inducing changes?
- RQ2: What are the best indicators of high impact fix-inducing changes?
- RQ3: How much inspection effort does predicting high impact fix-inducing changes reduce?

To address our RQs, we perform our study on six large open source projects, namely GIMP, Maven 2, Perl, PostgreSQL, Ruby on Rails and Rhino. For each project, we identify fix-inducing changes and their corresponding fixes (i.e., fixing changes). We use churn, the number of files, and the number of subsystems as a proxy for impact and classify fix-inducing changes as high- and low-impact using an unsupervised clustering technique. Then, we build a specialized prediction model to identify HIFCs in RQ1, determine the factors are the best indicators of HIFCs in RQ2 and quantify the inspection effort savings (i.e., reduction in false positives) of focusing on these HIFCs in RQ3. We summarize the contributions of this study as follows:

- **Propose a way to determine high impact fix-inducing changes.** We use the churn, the number of files and the number of subsystems modified during bug-fixing changes in order to determine the impact of those changes. To the best of our knowledge, this is the first work that studies high impact fix-inducing changes.
- **Predict high impact fix-inducing changes.** Using a number of code and process factors extracted at change level, our specialized prediction model is able to identify up to 77% HIFCs with an average false alarm rate (misclassification) of 16%.
- **Identify the best indicators of high impact fix-inducing changes.** Using the MeanDecreaseAccuracy measure computed in a random forest model, we find that the lines of code added, the number of developers worked on a change and the number of prior modifications on the files associated with a change are the best indicators of HIFCs, whereas the latter two metrics are unique to HIFCs, compared to best indicators of low impact fix-inducing changes (LIFCs).
- **Measure the inspection effort savings of a specialized prediction model.** We investigate the usefulness of building a specialized model by measuring the savings in the inspection effort, i.e., number of code changes that need to be inspected, and find that building a specialized model to identify HIFCs provides an average savings of 37% over state-of-the-art models which simply predict fix-inducing changes.

The remainder of the paper is organized as follows. Section 2 provides a motivating example from one project supporting our approach of prioritizing HIFCs. Section 3 discusses related work on bug prediction at source code and change levels, change impact analysis techniques and emphasizes the novelty in this study. Section 4 describes the characterization of HIFCs. Section 5 presents the experimental setup. Section 6 presents our results. Section 7 provides a discussion on the techniques used. Section 8 highlights threats to the validity of our findings, and Section 9 summarizes our work.

2 Motivating Example

To motivate our work on HIFCs, we use two examples of changes taken from the Perl project. (Figures 1,3). We also use their corresponding (bug) fixing changes (Figures 2,4) in order to show the differences in terms of impact. In Figure 1, a change is made to merge a duplicated code. This change was actually buggy, and it needed to be fixed later on, during the fixing change shown in Figure 2. The fix is a simple pointer error, which modified only one file and required four lines to be modified.

On the other hand, another change that causes a later fix, i.e., fix-inducing change, is shown in Figure 3. The change in Figure 3 performs renaming of some variables. However, this change introduces a bug, and later on, the change in Figure 4 was made to fix it. The

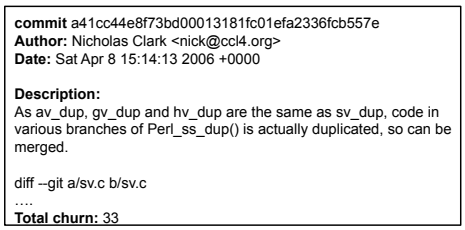


Fig. 1: Low inducing

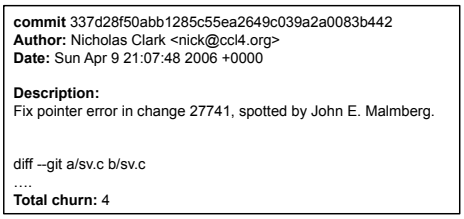


Fig. 2: Low fixing

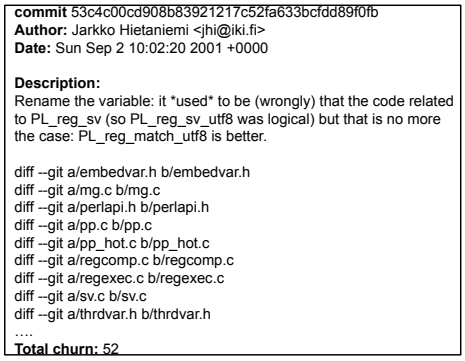


Fig. 3: High inducing

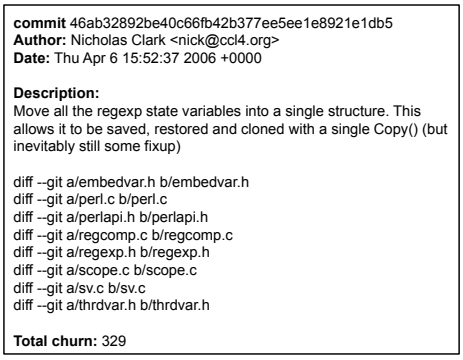


Fig. 4: High fixing

fixing change in Figure 4, on the other hand, requires 329 lines to be modified in eight different files.

Clearly, the fix for the change in Figure 3 has a higher impact than the fix for the change in Figure 1. This is mainly due to the fact that the developer needs to change many more lines in different files. Moreover, the number of subsystems that need to be touched to address each change is different. Similar to prior work [29], we use the root directory name to represent a subsystem. For example, if a change touches a file with the path, "`org.eclipse.jdt.core/jdom/org/eclipse/jdt/core/dom/Node.java`", then the subsystem is considered to be anything at the level of `org.eclipse.jdt.core`. The fix for the change in Figure 1 requires only one subsystem to be modified (which is obvious since only one file is changed). However, the fix for the change in Figure 3 requires nine different subsystems to be modified.

Prior work on change-level prediction (e.g. [29], [33]) would classify both changes in Figures 1 and 3 as fix-inducing with the same level of impact, since both of these changes required future fixes. However, in our work, we distinguish these two changes by ranking the change in Figure 3 as more impactful (i.e., high-impact) than the change in Figure 1 depending on the churn, the number of modified files and the number of subsystems where the modified files span in their corresponding bug fixing changes.

3 Related Work

We consider three areas of research that are most closely related to our research, and summarize recent advancements in each area: file- and method-level bug prediction, change-level bug prediction and change impact analysis techniques.

File- and method-level bug prediction. Boehm and Basili [5] present a software defect reduction top 10 list. They argue that prediction techniques, such as the one presented here and in much of the referenced work, can significantly reduce the cost of software defects. Therefore, a plethora of work has focused on file-level bug prediction. In general, researchers train prediction models to predict buggy locations (e.g., files or directories). Complexity metrics (e.g., McCabe's cyclomatic complexity metric [40]), Halstead operator-operand counts [22], Chidamber and Kemerer (CK) metrics suite [10]), size (measured in lines of code) [20, 25, 35], the number of prior changes and bugs are found as good predictors of bug-prone/buggy locations in software systems [1,3,23,35,46,47,50,67]. Other studies aim to predict bugs at the method level. Giger *et al.* [18] use change and code metrics to predict bug-prone methods. Hata *et al.* [24] also show that method level predictions perform better than file level predictions when the effort is taken into consideration.

Some researchers narrow down the scope of bugs that are predicted based on the type or category of bugs. For example, Shin *et al.* [60] and Zimmermann *et al.* [66] focus on predicting security vulnerabilities which are very few in number and distributed sparsely into the code. They found that classical software measures (complexity, churn, etc.) predict security vulnerabilities with low recall values, whereas the code dependencies predict those with a substantially higher recall [66]. Other studies by Caglayan *et al.* [9], Misirli *et al.* [44] and Shihab *et al.* [59] build specialized models that aim to predict the scope of the bugs, e.g., the phase they are reported. These specialized models have shown some improvements in terms of effort savings, i.e., reducing false alarms compared to traditional models, and increasing the information content of the prediction outcomes. Guo *et al.* [21] perform a study at Microsoft to characterize which bugs get fixed. Their findings showed that bug reported by reputable personnel and bugs that are likely to be fixed by the same team have a better chance of getting fixed. They also build a prediction model that can accurately predict which bugs will get fixed. Kim *et al.* [31] examine the problem of which warnings should be fixed first. They propose an algorithm that recommends warnings based on the project history, which significantly improve the prioritization of warnings.

There are some key differences between the file- and method-level bug prediction research and our study. First, we perform our modelling at the change level instead of the file- or method-level. This difference is important - performing our predictions at change level (e.g., after every commit) makes it easier to address these issues since changes can be flagged while they are still fresh in the developer's mind, and fixed before they are integrated with the rest of the code base. Furthermore, changes can be easily assigned to its owner, i.e., the developer who made the change, in contrast to files in bug prediction, which may be changed by many developers, making it harder to decide to whom files must be assigned. Finally, changes provide a narrower context to address the flagged issue, whereas in bug prediction, in some cases, a bug spans many files that are changed together. Moreover, our work complements prior work that builds specialized prediction models since we also build specialized models that predict fix-inducing changes based on their impact.

Change-level prediction. In addition to file-level bug prediction, there have been studies that focus on predicting fix-inducing changes.

Sliwerski *et al.* [61] study fix-inducing changes in Mozilla and Eclipse, and find that fix-inducing changes tend to be part of large transactions, and that bug fixing changes and

changes done on Fridays have a higher chance of inducing bugs. Kim *et al.* [33] present an approach to accurately identify fix-inducing changes using annotation graphs. They show that their approach can improve the accuracy (i.e., lead to less false positives and false negatives) over the SZZ algorithm. Similarly, Eyolfson *et al.* [14] observe a correlation between a change's bugginess and the time of the day the change was committed, and the experience of the developer making the change in the Linux and PostgreSql projects. Kim *et al.* [32] use change features like the terms in added and deleted deltas, terms in directory/file names, terms in change logs, terms in source code, change metadata and complexity metrics to classify changes as being buggy (i.e., fix-inducing) or clean (i.e, not fix-inducing). Yin *et al.* [65] performed a study that characterizes incorrect bug-fixes in Linux, OpenSolaris, FreeBSD and a commercial operating system. They find that concurrency bugs are the most difficult to correctly fix. Giger *et al.* [19] use social network and object-oriented metrics to predict the type of code changes, such as condition changes, interface modifications and inserts or deletions of methods. Shihab *et al.* [56] study risky (fix-inducing) changes - as deemed by the developers who committed them. They find that lines of code added, bugginess of the touched files, the number of bug reports linked to a commit and the developer experience are the top indicators of risky changes. Herzig *et al.* [26] investigate change genealogies, i.e., graphs of changes that have mutual dependencies. They show that they are able to effectively predict change genealogies using metrics that capture temporal and spatial influences. Madhavan *et al.* [38] present a tool that predicts fix-inducing changes in the IDE. The tool uses change and code metrics extracted from the changes to make its prediction.

There are some key differences between our work and the aforementioned work. First, all of the prior work, including our prior work [56], treated all fix-inducing or risky changes equally. In contrast, this study is based on the fact that all fix-inducing changes are not equal, and hence, the changes that require the highest amount of work to fix are given a higher priority. Furthermore, we perform our study on a number of large open source projects, whereas the most prior work was done on a few open source or commercial projects. This fact is only good for generalizing our results, even though they all are open source. In addition to simply predicting high impact fix-inducing changes, we also investigate what factors are significant indicators of high impact fix-inducing changes, and compare those with the factors of a general model for fix-inducing changes.

There are other studies that focus more on the risk of larger changes (e.g., entire features or service pack updates). Mockus and Weiss [45] assess the risk of Initial Modification Requests, called IMRs, which are groups of code changes, of the 5ESS commercial project. They predict the potential of an IMR to cause a failure (i.e., induce a bug) using IMR diffusion, size, interval, purpose and experience metrics. Czerwonka *et al.* [11] present their experiences with CRANE, a tool used within Microsoft for failure prediction, change risk analysis and test prioritization. The main difference between our work and the work by Mockus and Weiss [45] and Czerwonka *et al.* [11] is that we provide recommendations at a finer granularity (i.e., at the individual change level), whereas the aforementioned work performs their analysis at the IMR or binary level, which is generally made up of hundreds or thousands of files.

Our study is closely related to our prior work by Kamei *et al.* [29], which reported a large scale study on the effectiveness of predicting fix-inducing changes. In the prior work [29], a variety of factors extracted from the commits and bug reports were found as good indicators of fix-inducing changes. This work complements the work in [29] in a number of ways. Our work is similar to [29] in that we use the same set of code and change metrics metrics to perform our prediction. Our work differs from [29] in that 1) we propose metrics to quantify the impact of a fix-inducing change and consider this impact to classify fix-inducing

changes as 'high impact' and 'low impact', 2) we build a specialized model that classifies each change as a HIFC, LIFC or *not* fix-inducing change, 3) we use the Random Forest algorithm, which is inherently suited for a multi-class prediction problem [7], 4) we perform our study on six different projects, 5) we determine the metrics that best indicate HIFCs and compare them to those metrics listed for LIFCs, and finally, 6) we estimate the inspection effort savings of focusing on HIFCs compared to the state-of-the-art models, which simply predicts all fix-inducing changes.

Change impact analysis. Over twenty years, change impact analysis (CIA) techniques are studied for identifying the effects of a change or estimating what needs to be modified to accomplish a change [36]. This type of analysis is necessary when changing or maintaining evolving software systems, since it allows to a) judge the amount of work required to implement a change, b) propose the artifacts that should be changed or c) to identify test cases that should be re-executed to ensure that the change was implemented properly [34]. According to a literature review, most of the CIA literature focus on identifying the impact of a change on the final product - source code [34].

Over 30 papers on code-based CIA techniques, another survey reports that mining software repositories is often preferred over other approaches (e.g. dependency analysis, execution information) in order to identify the impact of a change in terms of its size or scale [36]. In summary, the impact set of a change can be determined using prior code changes (e.g. [53]) or using dynamic call graphs obtained from execution of the tests (e.g. [17]). Given the impact set, the analysis estimates other changes that are linked to the selected change [53] or a subset of tests that should be re-run after the selected change [17].

Recent work by Kawrykow and Robillard [30] argues that important code changes are sometimes accompanied with non-essential modifications, such as type updates, local variable renaming or refactoring, and those changes are unlikely to provide information about the development effort. The authors implemented a tool to discover those non-essential changes and found that up to 15% of a system's method updates were solely due to non-essential differences. They also stated that these non-essential method updates have a significant impact on the recommendations done by change-based analysis tools.

Our study complements the CIA research in a number of ways. First, we also distinguish a specific type of changes, as it is done by Kawrykow and Robillard [30], and focus on fix-inducing changes rather than observing all changes stored in software repositories. Second, using the definition in [53], we define the impact of a fix-inducing change in terms of the amount of churn, the number of modified files and the number of subsystems where the modified files span. Third, we use a set of metrics extracted from prior fix-inducing code changes to quantify the impact of fix-inducing changes. Finally, we aim to improve programmer productivity by providing an accurate estimation of high impact changes responsible for a potential failure and, in turn, reducing the inspection effort during a fixing activity [17].

4 High Impact Fix-Inducing Changes

In this section, we first define our approach to identify fix-inducing changes, then we present the criteria used to quantify the impact of a fix-inducing change, and our approach to further identify HIFCs over all fix-inducing changes. Finally we present the characterization of these HIFCs.

4.1 Identifying Fix-Inducing Changes

To determine whether or not a change induces a fix, we use the SZZ algorithm [61]. The algorithm links each bug fix to the original change, which induced the bug. To do so, the algorithm first identifies a "fixing change" that makes a fix by searching the change comments for keywords (such as "Fixed" or "Bug") and bug identifiers. Second, it searches for a prior change that induces this fix, which is defined as "fix-inducing change", using Git's *diff* command. This command helps to locate the lines that were changed by the associated fixing change. Once the changed lines are identified, we use Git's *blame* command to trace back to the last revision that changed those lines. This way, we have a mapping of each fixing change to a fix-inducing change. Please note that, a fixing change can be mapped to one or more fix-inducing changes depending on the number of changes made on the associated lines. For example, 11940 fix-inducing changes in an open source project (Gimp) were later fixed in a total of 4394 code changes.

The SZZ algorithm assumes the fixing change occurs where a bug is located, i.e., bug was injected into the lines that were modified during a fix-inducing change [61]. It is difficult to define the exact location of a bug since a bug may be located in a file (say A), but it may cause failures in other files (say B, C, D) that are dependent on that file. During its bug fix, developers need to fix all files (A, B, C, D) and commit changes using the associated bug ID. The SZZ algorithm would trace back the last changes done on these files and mark them as fix-inducing if there is no other bug report created between these activities. This procedure is slightly biased, since fix-inducing changes do not really address where the bug is actually injected into the system. However, we do not claim that fix-inducing changes are where bugs were first injected. Our aim is to determine where the earlier changes caused later fixes, and hence it is important to mark both A and the dependent files (B, C, D) that need to be modified during a fix activity.

4.2 How to Quantify the Impact of a Fix-Inducing Change?

Using the SZZ algorithm, we obtain a list of changes that induced a later fix, i.e., fix-inducing changes, and others that did not for the six projects. However, we are still left with the task of determining which of those fix-inducing changes are the most impactful.

We define the impact of a fix-inducing change in terms of *the amount of churn, the number of modified files and the number of subsystems where the modified files span*. As mentioned earlier in Section 3, we focus on the change impact in terms of the amount of churn, the number of modified files and the number of subsystems where the modified files span to fix an issue, rather than the change impact in terms of the work done to debug and refactor a code, or to run associated tests. The amount of work done during a debugging, testing or refactoring activity could be higher than that spent during a fix depending on the problem (e.g. synchronization issue); however defining the amount of work for each activity and comparing those is not the focus of this study

There are many factors that can be used to quantify the impact of a fixing change. Prior work has used different size measures (e.g., lines of code (LOC), number of revisions [41, 58, 62]) and complexity measures (e.g., number of files, modified code chunks

Accordingly, we use three metrics to quantify the impact of a fix-inducing change: 1) the total churn, i.e., number of modified lines of code to fix an issue, 2) the total number of files (modified to fix an issue), and 3) the total number of subsystems (affected during a fix). Using (1) is analogous to LOC, i.e., it accounts for the size of a fixing change, whereas using (2) and (3) are analogous to complexity, i.e., it accounts for how scattered a fixing

change is in terms of the number of touched files and subsystems inside the source code. It is important to note that we do not aim to categorize fix-inducing changes that are more difficult to fix, since the difficulty of a fix cannot be justified by only using the three metrics we use.

Our approach is unique in the sense that we address the impact of fix-inducing changes by measuring the three aforementioned metrics associated with the fixing changes. More specifically, we consider a fix-inducing change as of "higher impact" than other fix-inducing changes, if *more* lines of code (i.e., churn), *more* files and *more* subsystems are modified to fix that issue (i.e., in the fixing change). It is important to note that we chose to classify the changes into two classes, i.e., high- and low-impact since we wanted to balance the trade off between effectiveness and complexity of our approach.

The total churn, number of modified files and number of modified subsystems have been previously used to indicate the risk of a change(e.g., [23, 29]). However, the use of these metrics computed from a fixing change in order to measure the impact of a fix-inducing change is, to the best of our knowledge, a novel contribution of our study. More specifically, we look into the future, i.e., fixing changes, and the amount churn, the number of files modified and the total number of subsystems affected of those changes; then we generate two clusters of current (fix-inducing) changes. The predictions, on the other hand, are performed by extracting a set of factors from current (fix-inducing) changes. Therefore, the clustering metrics to define the impact of fix-inducing changes are not also used as the independent variables of the prediction models.

How to measure *more*, i.e., the quantity of impact, still remains as a question, which we present next.

4.3 High Impact Fix-Inducing Changes

High impact fix-inducing changes cannot be traced manually over thousands of code changes in software projects, e.g., there are 2,955 code changes in Rhino project, and 50,485 code changes in Perl project. We need to use a heuristic or a statistical technique that would automatically categorize fix-inducing changes into 'high impact' and 'low impact' based on the aforementioned metrics (i.e., churn, number of modified files, and number of modified subsystems). An "ad-hoc" approach can be setting a threshold for one of three metrics (e.g. churn above 1,000 lines of code) and classify high impact fix-inducing changes as those exceeding the threshold. This approach is biased since it depends on human judgement in setting the thresholds for each metric and adjusting these thresholds for each project. Thus, we prefer to use an unsupervised clustering technique, Expectation Maximization (EM) [6], that would identify two clusters among fix-inducing changes (high impact and low impact) using three metrics.

EM is an effective clustering technique that assumes the data consists of a mixture of populations (in our case, two clusters) with unknown parameters. The algorithm initially assigns parameters to distributions of these clusters (i.e., Gaussian), and calculates the probability of each data instance belonging to a cluster (E-step). Next (M-step), it re-calculates the parameters that has the maximum log likelihood (locally), based on assignment probabilities calculated in E-step. The algorithm converges when the difference between the old and new parameter estimates are below a certain threshold, or when maximum number of iterations have reached [6]. We set the initial number of populations as two, and ran EM on each project's data (IDs of fix-inducing changes, and their associated metrics) using the `mclust` library in R [15]. We set the initial number of populations as two since we are

Table 1: Statistics of the studied projects

	Domain	Period	Total no. of changes	Median churn of changes	No. of modified files per change	No. of changes per day
GIMP (C)	Graphics application	01/1997 - 06/2013	32,875 (36.3%)	25.0	6.7	5.5
Maven 2 (Java)	Build manager	09/2003 - 05/2012	5,399 (10.2%)	8.5	4.4	1.7
Perl (Perl, C)	Programming language	01/1988 - 06/2013	50,485 (24.1%)	6.5	3.4	5.7
PostgreSQL (C)	DBMS	07/1996 - 06/2013	35,005 (38.6%)	10.5	4.7	5.7
Ruby on Rails (Ruby)	Web app. framework	11/2004 - 06/2013	32,866 (18.9%)	6.0	2.8	10.5
Rhino (Java)	JavaScript engine	04/1999 - 02/2013	2,955 (43.7%)	13.5	3.6	0.6
Median	-	-	32,871 (30.2%)	9.5	4.0	5.6

†The percentage in parentheses shows the percentage of fix-inducing changes to all changes.

interested in splitting the group of fix-inducing changes into high-impact and low-impact fix-inducing changes. Typically, two groups of changes is used in prior work (e.g., [23, 29]), however, and number of populations can be specified if the application arises.

After two clusters are formed, we classify a cluster as a "high impact fix-inducing changes" if the medians for the churn, the number of files and the number of subsystems are higher than the medians of the other cluster, which naturally becomes the "low impact" cluster. The final statistics about the projects used in this study, before and after the clustering approach, are reported in Section 5.

5 Study Setup

In this section, we explain our study setup to answer the following research questions:

- **RQ1:** Can we accurately identify high impact fix-inducing changes?
- **RQ2:** What are the best indicators of high impact fix-inducing changes?
- **RQ3:** How much inspection effort does predicting high impact fix-inducing changes reduce?

First, we present the projects used in this study, and report the statistics about HIFCs and LIFCs in these projects. Second, we discuss the set of factors used in our study. Third, we provide the details of our study setup such as the sampling technique, and the algorithm used for predicting HIFCs, the performance measures used to evaluate our model, and the techniques used to compare with binary change-level prediction models. Finally, we describe the technique used to determine which factors, i.e., metrics, are the best indicators of HIFCs.

Table 2: Multi-class statistics of studied projects

	Total no. of HIFCs		**Total no. of LIFCs**		**NFCs**		**Total no. of changes**
GIMP	7,354	(22.4%)	4,586	(13.9%)	20,935	(63.7%)	32,875
Maven 2	368	(6.8%)	183	(3.4%)	4,848	(89.8%)	5,399
Perl	4,735	(9.4%)	7,437	(14.7%)	38,313	(75.9%)	50,485
PostgreSQL	5,160	(14.7%)	8,351	(23.9%)	21,494	(61.4%)	35,005
Ruby on Rails	2,386	(7.3%)	3,838	(11.7%)	26,642	(81.1%)	32,866
Rhino	321	(10.9%)	970	(32.8%)	1,664	(56.3%)	2,955
Median	3,561	(10.8%)	4,212	(12.8%)	21,215	(64.4%)	32,871

The percentage in parentheses shows the ratio of corresponding changes to all changes.

5.1 Study Context

In order to study HIFCs, we perform our case study on six open source projects (i.e., GIMP[1], Maven 2[2], Perl[3], PostgreSQL[4], Ruby on Rails[5], Rhino[6]). The projects were chosen to achieve diversity in terms of their domain, the language they are mainly written in (C, Java, Ruby, and/or Perl), their size in terms of the number of files and activity (average number of changes per day).

Table 1 shows the statistics of our dataset. For each project, we present the total number of changes of each project and the percentage of these changes that are fix-inducing, shown in parentheses. In addition, to shed light on the activity of these projects, we also present the median over the total number of lines churned per change (which ranges between 6-25 lines), the average number of files touched (which ranges between 2.8-6.7 files) by a change and the average number of changes per day (which ranges between 0.6-10.5 changes). The statistics show that the projects are actively being maintained in terms of LOC being changed, files and number of changes.

We applied the clustering approach on the six projects, and built a dataset containing three classes for all changes in eacn of the six projects: a class of "high impact" (also known as HIFCs), "low impact" fix-inducing changes (LIFCs), and "non-" fix-inducing changes (NFCs), i.e., changes that do not induce a later fix. Table 2 shows the new statistics with three classes on all projects. We see that HIFC is the minority class (on average, 12% of total changes) in all projects except GIMP and Maven 2. These two projects are different than others, as LIFC is the minority class which accounts for nearly half of the total number of HIFCs (14% versus 22%, and 3% versus 7%, respectively).

Figure 5 shows box-plots of Ruby project in terms of the total lines of code churned (total churn), total number of modified files and total number of modified subsystems in two, i.e., high impact and low impact, clusters. The first two box-plots for each clustering metric shows HIFCs and LIFCs, whereas the third one shows both classes for a better comparison. In the figure, we did not draw the outliers as points on the box plots to make the plot easier to interpret. We see that the cluster generated for HIFCs has a greater variance in terms of the metrics used for clustering compared to the cluster generated for LIFCs. We also confirmed

[1] git clone git://git.gnome.org/gimp
[2] git clone git://git.apache.org/maven-2.git
[3] git clone git://perl5.git.perl.org/perl.git
[4] git clone git://git.postgresql.org/git/postgresql.git
[5] git clone https://github.com/rails/rails.git
[6] git clone https://github.com/mozilla/rhino.git

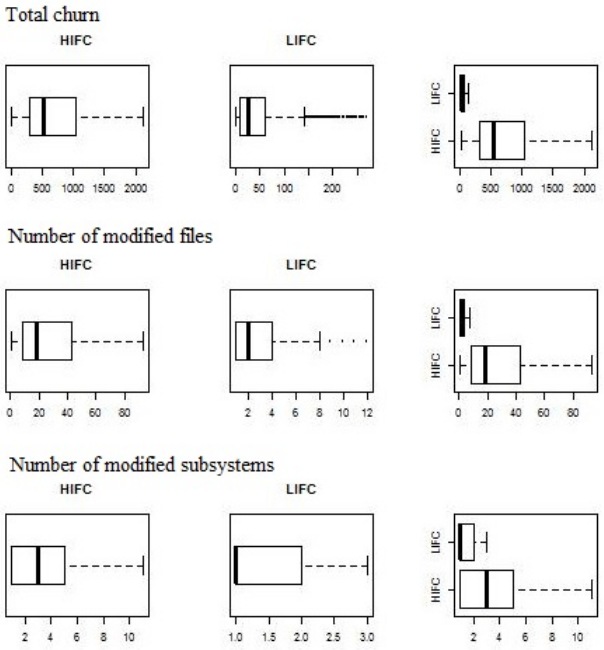


Fig. 5: Box plots depicting the spread of HIFCs and LIFCs in terms of three metrics for Ruby project

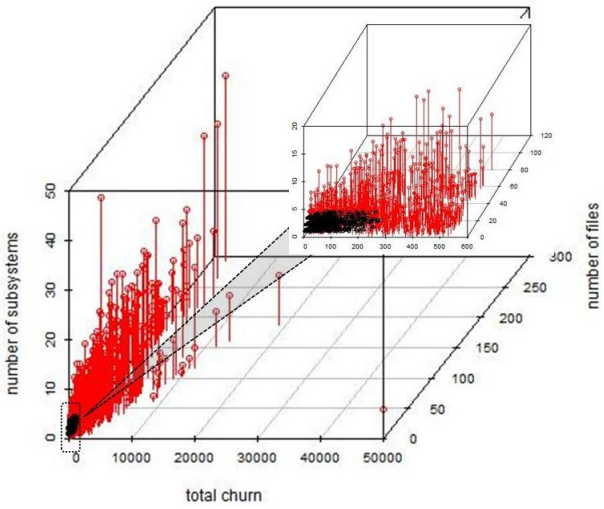


Fig. 6: Three-dimensional scatter diagram presenting the clusters for HIFCs (red) and LIFCs (black) in Ruby project.

using the Mann Whitney U-test that the differences in medians of HIFCs and LIFCs, shown in Figure 5 are statistically significant (p-value $\ll 0.01$) for all projects.

Figure 6 also presents a three-dimensional scatter diagram of Ruby project in terms of three metrics used for clustering fix-inducing changes (X-axis: total churn, Y-axis: number of modified subsystems, Z-axis: number of modified files). The cluster depicting HIFC is colored as red, whereas the cluster for LIFC is colored as black in the scatter diagram. The figure is zoomed to the region where LIFCs are highly concentrated and redrawn to show the differences between two clusters. From this figure, it is clear that LIFCs are concentrated on a smaller region in the graph (up to 300 churn, 5 modified subsystems and 25 modified files) even though they constitute the majority of the fix-inducing changes (63%). This finding also holds for the other projects whose box plots and scatter diagrams are presented in Appendix. Table 12 in Appendix also reports the minimum, 25% quartile, median, 75% quartile and maximum values of three metrics for HIFCs and LIFCs respectively in all projects.

It is important to note here, that we do not aim to create a global model for HIFCs; rather our goal is to examine HIFCs per project. In fact, since all projects are different in terms of size, application domain, user base, process, etc., we expect that the definition of a HIFC will vary from one project to another. Table 12 shows the metric values for HIFCs and LIFCs for each project, and we see that each project has very different values for HIFCs and LIFCs. For example, for Gimp, the median churn for a HIFC is 11,459, whereas, for Maven it is 643.5. Given such results, we do not believe a global definition of HIFCs is possible, hence, we perform our analysis and define HIFCs (and LIFCs) on per project basis.

5.2 Factors Used to Predict HIFCs

In order to predict HIFCs, we use a number of factors extracted from the changes and the code being changed, i.e., change metrics. A number of prior studies that focused on fix-inducing changes proposed different factors that can be used to identify them (e.g., [29, 32, 45,56]). Since coming up with a new set of factors will make it more difficult to compare our results with the prior work, and in order to facilitate replication studies, we decided to use a set of factors there were defined in an earlier study [29]. It is important to note, however, that although we use the same factors, we use data from different projects.

Table 3 shows the change metrics, which we call *change factors*, used in our study. In total, there are 13 factors that make up five different dimensions. Next, we provide a brief description of each dimension and its factors. A more detailed description of each factor can be found in [29].

Diffusion dimension: Prior work has shown that a highly distributed change can be more complex and harder to understand [45]. For example, Mockus and Weiss [45] show that the number of subsystems touched by a change is correlated with its riskiness, whereas Hassan [23] shows that scattered changes in files are good indicators of bugs in these files. We use four different factors to represent the diffusion dimension, as shown in Table 3.

As stated earlier in Section 2, we use the root directory name as the subsystem name (i.e., to measure NS), the directory name to identify directories (i.e., ND) and the file name to identify files (i.e., NF). To illustrate, if a change modifies a file with the path, "`org.eclipse.jdt.core/jdom/org/eclipse/jdt/core/dom/Node.java`", then the subsystem is `org.eclipse.jdt.core`, the directory is `org.eclipse.jdt.core/jdom/.../dom` and the file name is `Node.java`.

Size dimension: In addition to the diffusion of a change, the size of a change can be a good indicator of its potential to induce a bug. For example, Nagappan and Ball [48] and Moser *et al.* [46] show that the size of a change (e.g., the number of lines of code added in a revision)

Table 3: Summary of change metrics [29]

Dim.	Name	Definition	Rationale	Related Work
Diffusion	NS	Number of modified subsystems	Changes modifying many subsystems are more likely to be buggy.	The probability of a buggy change increases with the number of modified subsystems [45].
	ND	Number of modified directories	Changes that modify many directories are more likely to be buggy.	The higher the number of modified directories, the higher the chance that a change will induce a bug [45].
	NF	Number of modified files	Changes touching many files are more likely to be buggy.	The number of classes in a module is a good feature of post-release bugs of a module [49]
	Entropy	Distribution of modified code across each file	Changes with high entropy are more likely to be buggy, because a developer will have to recall and track large numbers of scattered changes across each file.	Scattered changes are more likely to introduce bugs [12, 23].
Size	LA	Lines of code added	The more lines of code added, the more likely a bug is introduced.	Relative code churn measures are good indicators of buggy modules [46, 48].
	LD	Lines of code deleted	The more lines of code deleted, the higher the chance of a bug.	
Purpose	FIX	Whether or not the change is a bug fix	Fixing a bug means that an error was made in an earlier implementation, therefore it may indicate an area where errors are more likely.	Changes that fix bugs are more likely to introduce bugs than changes that implement new functionality [21] [52].
History	NDEV	The number of developers that changed the modified files	The larger the NDEV, the more likely a bug is introduced, because files revised by many developers often contain different design thoughts and coding styles.	Files previously touched by more developers contain more bugs [39].
	AGE	The average time interval between the last and the current change	The lower the AGE (i.e., the more recent the last change), the more likely a bug will be introduced.	More recent changes contribute more bugs than older changes [20].
	NPC	The number of prior changes to the modified files	The higher the NPC, the more likely a bug is introduced, because a developer will have to recall and track many previous changes.	The number of prior changes on modified files contributes to predicting fix-inducing changes [20].
Experience	EXP	Developer experience	More experienced developers are less likely to introduce a bug.	Programmer experience significantly decreases the bug probability [45].
	REXP	Recent developer experience	A developer that has often modified the files in recent months is less likely to introduce a bug, because she will be more familiar with the recent developments in the system.	
	SEXP	Developer experience on a subsystem	Developers that are familiar with the subsystems modified by a change are less likely to introduce a bug.	

is a good indicator of bug-prone modules. The size dimension contains two factors, as shown in Table 3.

Purpose dimension: We also consider the purpose of a change since prior work shows that a change that fixes a bug is more likely to induce a future bug [21] [52]. Similar to prior work, to determine whether or not a change fixes a bug, we look for keywords such as “bug”, “fix”, “defect” or “patch” and bug IDs in the commit logs [32]. The purpose dimension has only one factor.

History dimension: Prior work also shows that the history of changes contain valuable information in determining buggy files [20, 39]. Therefore, we use the history of changes to help us determine high impact fix-inducing changes. The history dimension contains three factors, related to the number of developers, the number of changes and the age of changes.
Experience dimension: Prior work also show conflicting results about the usefulness of using developer information. For example, some work suggests that using information about developers does not help bug prediction [51], whereas other work [45, 56] shows that higher programmer experience significantly decreases a change's bug-proneness. Therefore, we use three factors related to developer's general experience, relevant experience and subsystem experience in our work.

5.3 Sampling Approach

Prior work studying fix-inducing changes has shown that, in general, this type of changes form the minority class in software datasets (i.e., they make up 10-20% of the total changes) [29, 32, 56]. This imbalanced class distribution affects most learners, i.e., classification algorithms, since the algorithm learns the most about the majority class (e.g., not fix-inducing changes), and the least about the minority class (e.g., fix-inducing changes). Researchers proposed sampling techniques, i.e., over- or under-sampling, on software data in order to improve the performance of prediction models [13, 42]. For example, one study conducted on public datasets [42] present that under-sampling works best in software engineering problems. On the other hand, Estabrooks and Japkowicz [13] recommend performing both under- and over-sampling, since under-sampling may lead to useful data being discarded, and over-sampling may lead to over-fitted models.

In our study, the imbalanced data problem is exacerbated due to the fact that we focus on *high impact* fix-inducing changes (HIFCs). Table 2 shows the distribution of HIFCs such that in three out of six datasets, the ratio of HIFC are below 10%. Therefore, we perform sampling on the training data before building the prediction models. We perform both over- and under-sampling on the training data, and predict HIFCs in the test data. It is important to mention that we do not apply any transformation on the ratio of HIFC in test data, because the test dataset represents an actual real-life scenario of software engineering datasets. We find that using under-sampling achieves better prediction results, and hence, we report under-sampling in the results.

During under-sampling, the number of instances which corresponds to the minority class (K) in training data is kept, and other classes are under-sampled by randomly selecting K instances. Thus, the resulting training data consists of 3K instances, i.e., K instances from each class. The test data was not re-sampled and hence, it maintains the same ratio of HIFCs as in the original dataset.

5.4 Classification Approach

The goal of this study is to prioritize fix-inducing changes which have a higher impact than the other changes in terms of the churn, the number of files and the number of subsystems touched in later fixes. To accomplish this, our approach can be defined as a multi-class prediction problem, with the objective of predicting fix-inducing changes, as well as, their impact (i.e., as 'high' or 'low' impact). In a binary classification of changes (i.e., outputs are fix-inducing or not), many algorithms have been utilized, e.g., logistic regression, decision

trees, and random forests [32, 46, 57, 67]. Among these, random forests are found as advantageous with their ability to deal with noisy data, and to adjust few parameters, and hence, they produce robust, highly accurate and stable models [27].

The random forest algorithm is an ensemble of decision trees. It builds a large number of decision trees, where each node in the decision tree is split using a random subset of all the attributes. Doing this random split ensures that all trees have low correlation between them [7]. The dataset is then split into two parts. The first part is used to build the trees (this part of the data is 90% of the dataset) and the remaining data (i.e., 10%), which is called the Out Of Bag (OOB) data, is used to test the prediction accuracy of the created forest. Since there are many decision trees (which may indicate different outcomes), each sample in the OOB is pushed down all the trees in the forest and the final class of the sample is decided by aggregating the votes from all the trees. This ensemble of decision trees is found as inherently suited for multi-class problems [7].

We perform two types of experiments. First, we build a multi-class prediction model, which aims to predict changes as being a HIFC, a LIFC or a non fix-inducing change (NFC). Second, for the sake of comparison, we build a binary model, which simply predicts changes as fix-inducing or not. In each experiment, we randomly divide the dataset into 10-90% stratified (i.e., keeping the class distributions the same) splits as test and training data, respectively. Then, we run our models 20 times, and report the median values over 20 runs.

5.5 Performance Measures

To determine the effectiveness of our models, we use a confusion matrix to store the classification results. For the binary prediction, we use a 2X2 confusion matrix as shown in Table 4. For the multi-class prediction (i.e., HIFC, LIFC and NFC), we use a 3X3 confusion matrix, as shown in Table 5.

For the binary classification, it is straight forward to calculate the true positive rate (TP), the false positive rate (FP), true negative rate (TN) and false negative rate (FN). A change can be classified as fix-inducing when it truly is fix-inducing (TP); it can be classified as fix-inducing when actually it is not fix-inducing (FP); it can be classified as not fix-inducing when it is actually fix-inducing (FN); or it can be classified as not fix-inducing when it is truly not fix-inducing (TN).

For the multi-class prediction, we calculate TP, FP, TN and FN rates per each class (i.e., one for each of HIFC and LIFC). The measures for HIFC class are defined as $TP_H = HH$, $FP_H = HL + HN$, $TN_H = LL + NN$ and $FN_H = LH + NH$. In these formulas, H corresponds to HIFC class, L corresponds to LIFC class, and in turn, HH corresponds to a situation where an instance actually belongs to HIFC class and it is predicted as HIFC (see the leftmost cell in Table 5). For LIFC class,the measures are defined as $TP_L = LL$, $FP_L = LH + LN$, $TN_L = HH + NN$ and $FN_L = HL + NL$.

Using the values of TP, FP, TN, FN, we calculate widely used performance measures, namely precision, recall f-measure, and false positive rate (e.g., in [1, 44, 56]), to evaluate the performance of our models.

The aforementioned measures are defined as follows:

Table 4: 2X2 Confusion Matrix

	True class	
Classified as	Yes	No
Yes	TP	FP
No	FN	TN

Table 5: 3X3 Confusion Matrix

	True class		
Classified as	High	Low	No Bug
High	HH	HL	HN
Low	LH	LL	LN
No Bug	NH	NL	NN

1. **Precision:** Measures the percentage of correctly classified (high impact) fix-inducing changes over all of the changes classified as fix-inducing. The higher precision is, the better the model classifies true class; and hence the ideal value for precision is 1 (i.e., 100%). It is calculated as $PR = \frac{TP}{TP+FP}$.
2. **Recall:** Measures the percentage of correctly classified (high impact) fix-inducing changes over the actual (high impact) number of fix-inducing changes. Similar to precision, the higher the recall rate is, the better the model is at predicting true class. Ideal value for recall is 1 (i.e., 100%). It is calculated as $RE = \frac{TP}{TP+FN}$.
3. **F-measure:** Is a composite measure that measures the weighted harmonic mean of precision and recall. It is measured as $F - measure = \frac{2*PR*RE}{PR+RE}$.
4. **False alarms:** Measures the percentage of wrongly classified (high impact) fix-inducing changes. The lower false alarm rate is, the better the model is at reducing the inspection effort due to misclassification of safe (bug free) changes. Hence, the ideal value is 0 (i.e., 0%). It is calculated as $PF = \frac{FP}{FP+TN}$.

5.6 Determining the Best Indicators of HIFCs

In addition to evaluating the performance of the prediction models, we are interested in determining the factors that are good indicators of high impact fix-inducing changes. To accomplish this, we examine the importance of each change factor during prediction using the mean decrease in accuracy measure (MeanDecreaseAccuracy), as outputted in the R randomForest library [37]. This measure is similar to coefficients in a logistic regression model such that it indicates how important each variable is for improving the performance of a prediction model. The mean decrease in accuracy of a variable is determined during the Out Of Bag error calculation phase. If the accuracy of the random forest decreases during the addition of a single variable, then that variable is considered as being important [37]. The amount of decrease in accuracy indicates the degree of importance of a variable: the more the accuracy decreases, the more important the variable becomes for the model. Therefore, we run the random forest model on the six projects, and identify the factors with the largest mean decrease in accuracy during the classification of HIFCs.

To accumulate all results, and determine the best indicators of HIFCs in all projects, we sort the change factors based on MeanDecreaseAccuracy (in decreasing order) in classifying HIFCs. Then, we use this ranking, and count how many times each factor is listed among

top five as the best indicators. The final list of change factors that are selected in at least four projects are determined as the best indicators of HIFCs.

5.7 Inspection Effort Savings with a Specialized Model

In this study, we investigate the benefits of building a specialized model for predicting HIFCs in two ways: First, we compare the performance measures of our proposed model (specialized model) with those of the state-of-the-art model (general model), in terms of recall, f-measure, false alarms. We conduct this first analysis by following the proposed study setup in Section 5.4. Then, following the approach proposed by [1], we compute the estimated inspection effort, i.e., number of code changes that need to be inspected, to catch HIFCs with a specialized model, and calculate how much inspection effort could be saved in terms of false positives, compared to a general model. To accomplish this, we adopt a similar approach as Shihab *et al.* [59], and examine the confusion matrices of our specialized model and the general model. To facilitate a fair comparison, we make sure to hold the recall of both models to be the same.

An overview of our comparison approach is shown in Figure 7. The approach works as follows; to simulate the use of the state-of-the-art model, which simply predicts fix-inducing changes (without taking the impact into consideration), we train a model using fix-inducing and other changes. Then, we use the trained model to predict HIFCs, and report how many HIFCs are identified. We also build another model, the specialized model, that is trained with HIFCs, LIFCs and NFCs, and is tested on HIFCs. Lastly, as mentioned above, we hold the recall of both models to be the same (by fixing the recall of the specialized model) and compare the false positives of the two models. Lower rates of false positives means less wasted inspection effort. If the general model performs the same or better than the specialized model, then there is no need for the specialized model. If however, the specialized model performs better, then clearly the specialized model is needed and improves over the state-of-the-art models.

This comparison does not imply that we under-estimate the performance of the binary model, but it is an analysis on a ROC curve (e.g. in [63]): We get recall-false positive pairs for each probability threshold and base the comparison on a fixed recall that both models could achieve. Since there is a direct inverse relationship between precision and recall, i.e., the higher the precision, the lower the recall and vice versa, we need to have both models with the same recall so that we can have a fair comparison of false positives (Remember that precision = TP/(TP+FP)). Having same recall means that both models have the same number of TPs and FNs. TPs need to be the same since that directly impacts the precision, whereas FNs need to be the same since that is related to FPs. One thing we could do is compare F1-scores, which is the harmonic mean of precision and recall, however, the F1-score would not tell us if the improvement is due to a reduction in FP or FN. Since we are interested in effort savings, it is most accurate to compare the FPs.

6 Results

In this section, we present the classification performance of a multi-class random forest model on six projects, using the performance measures described in Section 5.5. Then, we compare the performance of our proposed approach with a binary model, in which changes are classified as fix-inducing or not. To present the best indicators of HIFCs, we list all

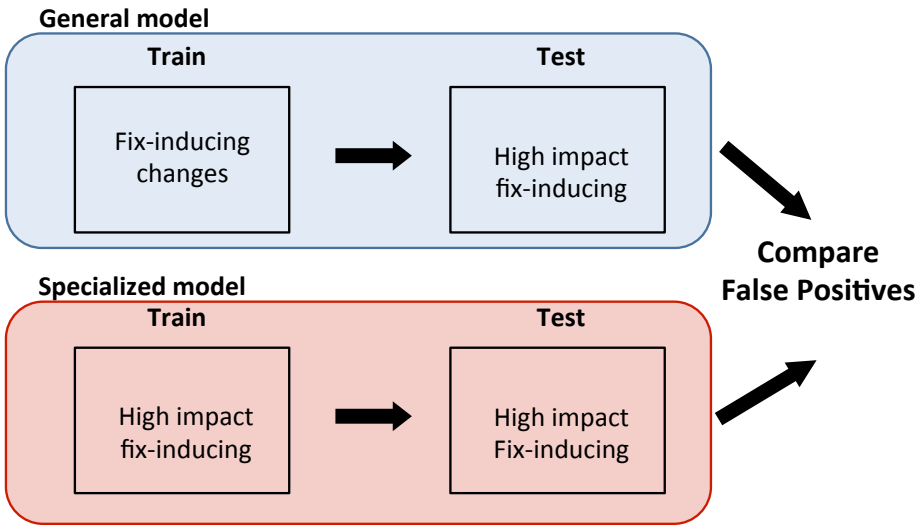


Fig. 7: Overview of the comparison approach used to estimate inspection effort savings

factors, and report the number of their occurrences in the top 5 factors based on the MeanDecreaseAccuracy measure. Finally, we analyze the effort savings of building a specialized model for HIFC.

Table 6: Prediction performance of multi-class prediction model

	HIFC					LIFC				
	RE	**PR**	**F-Meas.**	**PF**	**Base.**	**RE**	**PR**	**F-Meas.**	**PF**	**Base.**
GIMP	0.77	0.78	0.78	0.08	0.28	0.61	0.29	0.39	0.26	0.15
Maven-2	0.56	0.25	0.34	0.16	0.09	0.44	0.06	0.11	0.25	0.04
Perl	0.57	0.31	0.40	0.18	0.12	0.44	0.23	0.30	0.29	0.16
PostgreSQL	0.67	0.47	0.55	0.17	0.19	0.50	0.39	0.44	0.27	0.26
Ruby	0.62	0.27	0.37	0.18	0.09	0.42	0.18	0.25	0.28	0.13
Rhino	0.70	0.34	0.45	0.20	0.13	0.50	0.56	0.53	0.22	0.36

RE=Recall, PR=Precision, PF=False alarms, F-Meas=F-Measure, Base=Baseline model.

RQ1: Can we accurately identify high impact fix-inducing changes?

We build a multi-class model that uses random forest as its prediction algorithm in order to identify HIFCs, LIFCs and NFCs in six open source projects. We use the factors presented in Table 3 and follow the experimental setup explained in Section 5.4 to conduct our experiments. We report performance measures, namely recall, precision, f-measure and false alarms.

Table 6 shows the performance of our multi-class random forest (i.e., specialized) model in predicting HIFC and LIFC classes, separately. Our results show that a multi-class prediction model is able to catch 56% to 77% high-impact fix-inducing changes with a false alarm

Table 7: Predictions from a binary model

	RE	PR	F-Meas.	PF	Baseline
GIMP	0.79	0.59	0.67	0.36	0.39
Maven-2	0.92	0.12	0.21	0.69	0.09
Perl	0.75	0.35	0.48	0.50	0.26
PostgreSQL	0.74	0.58	0.65	0.38	0.41
Ruby	0.70	0.25	0.37	0.56	0.21
Rhino	0.74	0.72	0.73	0.25	0.47

RE=Recall, PR=Precision, PF=False alarms, F-Meas=F-Measure.

rate between 8% and 20% over all datasets. In datasets (Maven-2,Ruby,Perl) where the ratio of HIFCs is less than 10%, the model achieves lower performance measures (F-measure is between 34% and 40%) even though we apply under-sampling to balance the class distributions. Furthermore, the proposed model is better at distinguishing the HIFC than the LIFC class.

We also compare the performance of our proposed model (HIFC) with a baseline model that randomly predicts X% of changes as HIFCs if X% of changes in training set are associated with HIFCs. This comparison shows us whether the specialized model is at least better than a random predictor. The comparisons between our proposed model and a baseline model (the column *Baseline*) also show that the model is, on average, 170% better (from 147% gain in PostgreSQL to 200% in Ruby) in terms of the number of changes that should be inspected for predicting the HIFC class (gain= $(precision - baseline)/(baseline)$).

For the sake of comparison, we also present the performance of a binary random forest (i.e., general) model that predicts fix-inducing changes only. Table 7 shows that this general change-level model is better at predicting fix-inducing changes in terms of recall (from 70% in Ruby to 92% in Maven-2), however considering the trade-off between *recall* and *precision*, F-measure is better in only three out of six projects, namely in Perl (from 40% to 48%), PostGreSQL (from 55% to 65%) and Rhino (from 45% to 73%). Besides, the general model also produces high false alarms (between 25% and 69% as reported in Table 7), which increases the cost even though the objective of such models is just the opposite. We discuss the effects of improving false alarms in the next sections by comparing two models (specialized versus general) on a confusion matrix.

Using our approach, we see that the specialized prediction models can provide more useful and accurate information to developers by prioritizing high impact fix-inducing changes, and reducing false alarms significantly. Mann-Whitney U-tests also confirm that differences in performance measures between the specialized and general model are statistically significant (p-value < 0.05).

6.1 Comparison with Another Change Impact Analysis

In addition to a comparison with a baseline model (i.e., a random predictor), we plan to compare our approach with another change impact analysis using association rule mining [68] and discuss the differences in terms of predicting HIFCs.

In association rule mining, we predict whether or not a change is HIFC based on a rule: If a change modifies all files of an antecedent of a rule and does not modify a file of a consequent of the rule, it is classified as HIFC, because it misses the file that should be modified together. For example, there is one extracted rule (if File A, B and C are modified together, then File D is often modified at same time). If File A, B, C and D are modified in

change 1 of testing set, the change impact analysis predicts that the change 1 is not HIFC, because the change 1 modified all files in antecedent (A, B, C) and consequent (D) in one rule. On the other hand, if File A, B, C and E are modified in change 2 of testing set, the change impact analysis predicts that the change 2 is HIFC. Association rule mining generally extracts more than one rule. In our experiments, if there is at least one rule that classifies a change as HIFC, the change is classified as HIFC. Note that we use 0.01 as minimum support value and 0.5 as minimum confidence value for association rule mining, as it is inspired by previous studies [28, 68]

Table 8 shows the prediction performance of the change impact analysis. The results show that our specialized prediction model provides better F-measures for all datasets compared to the association rule mining. While the minimum F-measure of our prediction model is 0.34 for Maven-2 (in Table 6), the maximum F-measure of the association rule mining is 0.21 for Rhino (in Table 8). Furthermore, we can not find any co-changed patterns for Ruby. Therefore, we conclude that our prediction model outperforms another type of a change impact analysis technique.

Table 8: Prediction performance of change impact analysis for HIFCs

	RE	**PR**	**F-Meas.**	**PF**
GIMP	0.02	0.14	0.03	0.03
Maven-2	0.14	0.20	0.17	0.04
Perl	0.08	0.31	0.13	0.02
PostgreSQL	0.09	0.42	0.15	0.02
Ruby	0.00	0.00	0.00	0.00
Rhino	0.31	0.15	0.21	0.21

RE=Recall, PR=Precision, PF=False alarms, F-Meas=F-Measure

RQ2: What are the best indicators of HIFCs?

We use the MeanDecreaseAccuracy measure (as described in Section 5.6) computed by random forest for each class in order to decide on the most important factors classifying HIFC, and LIFC. We rank all factors according to the MeanDecreaseAccuracy measure, and list the best indicators of HIFCs by calculating top five factors with the highest MeanDecreaseAccuracy values in all projects. We do the same for LIFCs, and make comparison among their best indicators.

Table 9 shows the average rank of each factor (column name: avg. rank) accumulated over six projects for HIFC and LIFC, respectively. Factors are sorted in an increasing order based on their ranking for the HIFC class. The second column of Table 9 shows how many times (j indicates the project) a factor (i) is listed among the top five important indicators when predicting HIFCs and LIFCs. We define a factor as a good indicator (marked in bold) if it is listed in the top five list for more than 3 out of 6 projects.

Our results show that the lines of code added (LA), the number of prior changes to files (NPC), and the number of developers that changed the modified files (NDEV) are the best indicators of HIFCs. The list changes for LIFCs even though LA is still the most important indicator of fix-inducing changes. LA is followed by FIX with an average rank of 4.5, as being among top five metrics in five out of six projects. That is, if a change has been made

Table 9: Importance of change factors

	Avg. Rank		$\Sigma_{i=1}^{6} rank_{ij} \geq 5$	
	HIFC	LIFC	HIFC	LIFC
LA	1.2	1.7	**6**	**6**
NDEV	5.3	6.3	**4**	3
NPC	5.7	6.2	**4**	2
NF	7.2	8.2	3	0
LD	7.7	9.2	2	0
ND	7.7	8.2	1	2
SEXP	7.7	7.7	1	2
EXP	8.3	11.7	2	0
REXP	8.5	10	2	1
AGE	9.3	6.3	0	3
NS	9.5	6.7	2	3
Entropy	9.8	9	0	1
FIX	12.2	4.5	0	**5**

due to a bug fix, then it is more likely that it injects another bug into the code, but that change is more likely to be a LIFC.

Our analysis shows that the number of lines of code added (LA) during a change is, overall, the best indicator of fix-inducing changes. However, as more developers modify the files associated with a change (NDEV), and the higher the number of prior changes on these files (NPC), the more impactful a fix-inducing change becomes.

RQ3: How much inspection effort does predicting high impact fix-inducing changes reduce?

Following the approach in Section 5.7, we build a general model that is trained to predict fix-inducing changes, but it is tested to predict HIFCs. Such a model represents using the state-of-the-art models, which simply predict whether a change is fix-inducing or not. Then, we build our specialized model that is trained and tested to predict HIFCs. To make a fair comparison, we fix the recall rate of both models and compare the number of false positives. Ideally, we want to estimate inspection effort savings (i.e., less false positives) using the specialized models, otherwise, using the simple models would suffice. Notice that we compare the estimated inspection effort savings to identify HIFCs only because it is our main focus in this work; if we would like to observe LIFCs, we should have trained and tested the models for LIFCs in a separate analysis.

We generate a 2x2 confusion matrix of our specialized model (as seen in Table 10 on the left), and a confusion matrix of a general model (on the right) both of which are built using the same training-test splits. The confusion matrix on the left is computed from a 3x3 confusion matrix, which is the original output of our multi-class random forest model. We calculate TP_H, TN_H, FP_H, and FN_H values from the 3x3 confusion matrix by prioritizing HIFC, as explained in Section 5.5.

As seen in Table 10, we fix the recall rates (78%) of both models so that the difference between two models, in terms of the number of changes inspected to predict HIFCs, becomes more visible. In Table 10, the number of false positives (129 vs. 300) produced on the GIMP project shows that the specialized model predicting HIFCs is better at reducing the inspection efforts by 57% $((300 - 129)/300)$. This means, assuming that both models are equally good at predicting HIFCs (same recall), our specialized model manages to decrease the number of changes that should be inspected by 57% (in GIMP) in order to catch

Table 10: Confusion Matrices for GIMP project

	(HIFC → HIFC)			(Fix-inducing → HIFC)		
		Actual			Actual	
		HIFC	No		HIFC	No
Predicted	HIFC	512	**129**	HIFC	512	**300**
	No	148	2169	No	148	1998

Table 11: Estimated Inspection Effort Savings for All Projects

Project	**Savings (%)**
Gimp	57
Maven-2	36
Perl	8
PostgreSQL	20
Ruby	20
Rhino	62
Avg.	34

the same amount of HIFCs. We also performed the same analysis on the other five projects and report the percentage of inspection effort savings in Table 11. We estimate that the specialized model is able to reduce the inspection effort, on average, by 34% compared to the general model.

Thus, we conclude that using the specialized models is advantageous for catching HIFCs since it reduces the amount of inspection effort (i.e., the number of changes that need to be inspected), compared to the state-of-the-art models.

7 Discussion on Co-Factors Affecting Our Study

Throughout our experiments, we made certain decisions regarding the clustering technique, the metrics used for clustering fix-inducing changes into HIFCs and LIFCs, the factors used to predict HIFCs, and the prediction algorithm. In this section, we examine the impact of these choices on our results.

7.1 Impact of the Clustering Technique Used

During our analysis, we used the EM unsupervised clustering technique to distinguish HIFCs from LIFCs. Another widely used unsupervised clustering technique, is the k-Means algorithm. K-Means clustering starts with k (k=2 in our case) random centers for two clusters, and assigns changes to these clusters using a distance-based similarity measure. As more changes are assigned to clusters, centers, i.e., medians of the clusters, would gradually converge to near-optimal places.

We want to examine whether using k-Means instead of EM impacts our results. Thus, we ran both techniques on the datasets of six projects, and saved two different multi-class datasets, each of which has a different number of HIFC, LIFC, and NFC instances. For example, in the GIMP project, the EM algorithm produces a multi-class dataset with 22.4% HIFCs, and 13.9% LIFCs, while k-Means produces a dataset with 26% HIFCs, and 10% LIFCs.

It is difficult to choose between these two algorithms, since the actual high impact changes are not available. However, we can determine which of these clustering algorithms

achieves better performance for the specialized models. We re-ran our experiments on two different multi-class datasets produced by each of these two clustering techniques. Results show that the model trained with HIFC and LIFC clusters generated by 'EM' achieves lower false alarm rates (16% on average), compared to the model trained with the dataset generated after k-Means is used (23% on average). However, the latter model performs better in predicting LIFC class in terms of recall (73% versus 48% on average) and false alarms (13% versus 26% on average). Since our focus is prioritizing HIFCs over LIFCs, we conclude that using EM was the right choice for us.

7.2 Impact of the Clustering Metrics Used

The three metrics used for clustering fix-inducing changes into HIFCs and LIFCs are total churn, number of modified files and number of modified subsystems. We ran correlation analyses on these metrics to see whether they are related, and how that would bias the clustering process, as well as the prediction model. Kendall's tau (non-parametric) test were run between the three metrics used to find clusters of HIFC and LIFC ones.

We have found that there are moderate correlations between total churn and the number of modified files (coefficient is less than 0.7) in some datasets, namely Maven-2 and PostGreSQL, and high correlations between the number of modified files and modified subsystems (0.7 to 0.76) in some datasets, namely Gimp, Rhino, Perl. Therefore, we removed the metric, *number of modified files*, and re-ran the analysis using total churn and the number of modified subsystems. The results show that the model does not perform any better with the new clusters. For example in maven-2 project, the previous prediction model (from Table 6) achieves 58% recall, 24% precision, 34% f-measure and 16% false alarms, whereas the new model (using two metrics for clustering the HIFCs and LIFCs) produces 61% recall, 23% precision, 33% f-measure and 14% false alarms. We see that even though there is a slight improvement in recall, f-measure stays the same (differences are not significant according to Mann-Whitney U-test). So even though in some datasets, we observe high correlations between three clustering metrics, we think it is worth keeping all to have a more accurate prediction model.

7.3 Impact of the Factors Used for Predicting HIFCs

Table 9 shows the top three factors (LA, NDEV, NPC) as the best indicators of fix-inducing changes. We would like to study the prediction capability of LA only, as it is overall the best factor for predicting HIFCs, and compare its prediction performance with a baseline model and our specialized model (using a set of factors listed in Table 3). To accomplish this, we built a multi-class prediction model using the same setup explained in Section 5, but with a single metric (Lines of Code Added). The results of the new specialized model with LA metric only show that 28% to 54% of HIFCs could be successfully identified with a precision rate between 16% and 39%, and a false alarm rate between 17% and 23%. In terms of LIFCs, the new model using LA only could achieve 31% to 48% recall rates with a precision rate between 17% and 49%, and a false alarm rate between 24% and 37%. Compared to the performance of our proposed model (Table 6), we see that a specialized model using LA only is much better than a baseline model in terms of identifying HIFCs. However, only LA metric does not suffice to identify majority of HIFCs in the context of our study. Therefore,

we conclude that using all factors would help to characterize more HIFCs in a specialized model though some factors are major drivers in the model.

7.4 Impact of the Prediction Algorithm Used

As discussed earlier in Section 5.4, we use a random forest algorithm due to its robustness and its ability to handle noise [16, 27]. However, the majority of bug prediction studies employed logistic regression models. Therefore, we would like to examine the effectiveness of logistic regression models on predicting HIFCs.

Experiments using a multi-class regression model show that we can detect, on average, 58% of HIFCs (recall) with 30% precision, 38% F-measure and 23% false alarms for six projects. Performance measures for predicting LIFCs are even worse (32% recall, 25% precision, 27% f-measure and 22% false alarms on average) compared to Table 6. These results are not as good as the random forest algorithm, therefore, for our purpose, we conclude that random forest is better at predicting HIFCs and in turn, more suitable for a specialized model.

8 Threats to Validity

Construct validity. We use the amount of churn, the number of files and the number of subsystems of the bug-fixing change as a proxy of the impact of a fix-inducing change. Clearly, this is a proxy of the impact, and more factors such as the complexity of the modified code, and the type of problem reported for that change, may help to better indicate impact. Our future work will focus on improving this definition of impact. That said, we believe that our current approximation of the impacts a good one, especially since prior work on change-level bug prediction (e.g. [23], [48], [56], [62]) has shown that churn and the number of modified files are good indicators of fix-inducing changes. Prior research on change impact analysis techniques (e.g. [17], [36]) also confirmed that the impact of a change can be defined in terms of the amount of work to debug a code, fix an issue or to run only the associated tests.

Furthermore, using multiple metrics during clustering as a proxy for the impact helps us avoid mono-method bias [54]. We have already discussed the effect of a potential statistical relationship between clustering metrics 7 and mentioned that though there are some datasets in which we observe moderate to high correlations, it is worth keeping all three metrics and preserve a higher accuracy in the prediction models.

Since we focus on the amount of churn, the number of files and the number of subsystems to determine HIFCs, there is a potential bias towards specific types of large changes (e.g., refactoring changes). However, it is important to note that we only focus on fix-inducing changes, hence, even if our approach was more likely to flag refactorings, it would flag fix-inducing refactorings. In the future, we would like to perform more qualitative analysis on HIFCs to determine their types.

In our analysis, we use the number of false positives (code changes that are flagged as high-impact although the actual is the opposite) to estimate the inspection effort savings. Our analysis makes the assumption that each false positive requires equal effort, which may not be the case for all false positives. However, we only compare HIFC false positives. In the future, we would like to develop a method to quantify the cost of each false positive and use this cost in our effort savings analysis.

Internal validity. The link between fix-inducing and bug-fixing changes is formed using the SZZ algorithm [61]. Even though the algorithm has been popularly used in numerous bug prediction research (e.g., [29]), the algorithm is only able to link a fix-inducing change with its fix, if there is a unique keyword in comment logs of fixing changes. In some cases, this approximate linking method may introduce bias, and therefore, impact our findings [2, 4]. Furthermore, the SZZ algorithm also assumes that the previous code change touching a fixed line is the fix-inducing one. Thus, we are also limited with the capabilities of SZZ during identification of fix-inducing changes in this study. As future work, this study can be improved by extending SZZ algorithm, or by investigating commercial projects in which there are clear links between bugs and commits.

External validity. We use six open source projects, each of which has different characteristics in terms of development period, programming language, domain, as well as ratio of HIFCs over LIFCs. The reason is that we would like to observe how specialized models for predicting HIFCs perform in different settings. Results show that specialized models are worth the effort to prioritize high impact changes and to reduce false alarms of general models. However, our study should be replicated on commercial settings in which actual HIFCs can be used to evaluate the prediction accuracy of and to make generalizations on the benefits of specialized models. Thus, as a future work our plan is to extend this study with commercial projects, for repeating/refuting our findings.

Conclusion validity. During our experimental setup, we compare different classifiers (random forest versus logistic regression), clustering techniques (kMeans versus EM), and prediction models (binary versus multi-class) with non-parametric significance tests, i.e., Mann Whitney U-Test, and report significantly better results. We also avoid sampling effects by generating multiple runs of the algorithm on different training-test splits.

9 Conclusion

In this study, we consider one drawback of change-level prediction models: All fix-inducing changes are treated equally in terms of their impact. We define the impact of a change in terms of the amount of churn, the number of files and the number of subsystems required to address a fix-inducing change. We conduct a study on six open source projects, and prioritize high impact fix-inducing changes using a specialized model. Results accumulated over six projects show that a specialized model for predicting HIFCs performs better than a typical binary model in terms of false alarms, and in turn, the specialized model saves the inspection efforts by 34%. We have also observed that there are unique factors characterizing HIFCs, namely number of developers, number of prior changes in the associated files, and unique factors characterizing LIFCs, namely whether the change is a fix for a prior bug.

In the future, we plan to extend this study in a number of ways:

1. Knowing whether a fixing change is actually high impact according to developers would be great to prove the accuracy of such prediction models. Hence, we would like to extend this work by adding commercial projects, and by getting feedback from practitioners on the benefits of these specialized models.
2. Defining the impact of a change for different activities, such as debugging, re-factoring, and bug fixing, and for different stakeholders, such as developers or customers, and compare the amount of implementation work done for each of the activities.

We believe one possible way to improve these specialized models is to train them for each team/developer, and to provide instant feedback during their commit activities. Thus, a

future research direction may be to build developer-specific models which would give a recommended list of (high impact) fix-inducing changes on their latest development activities.

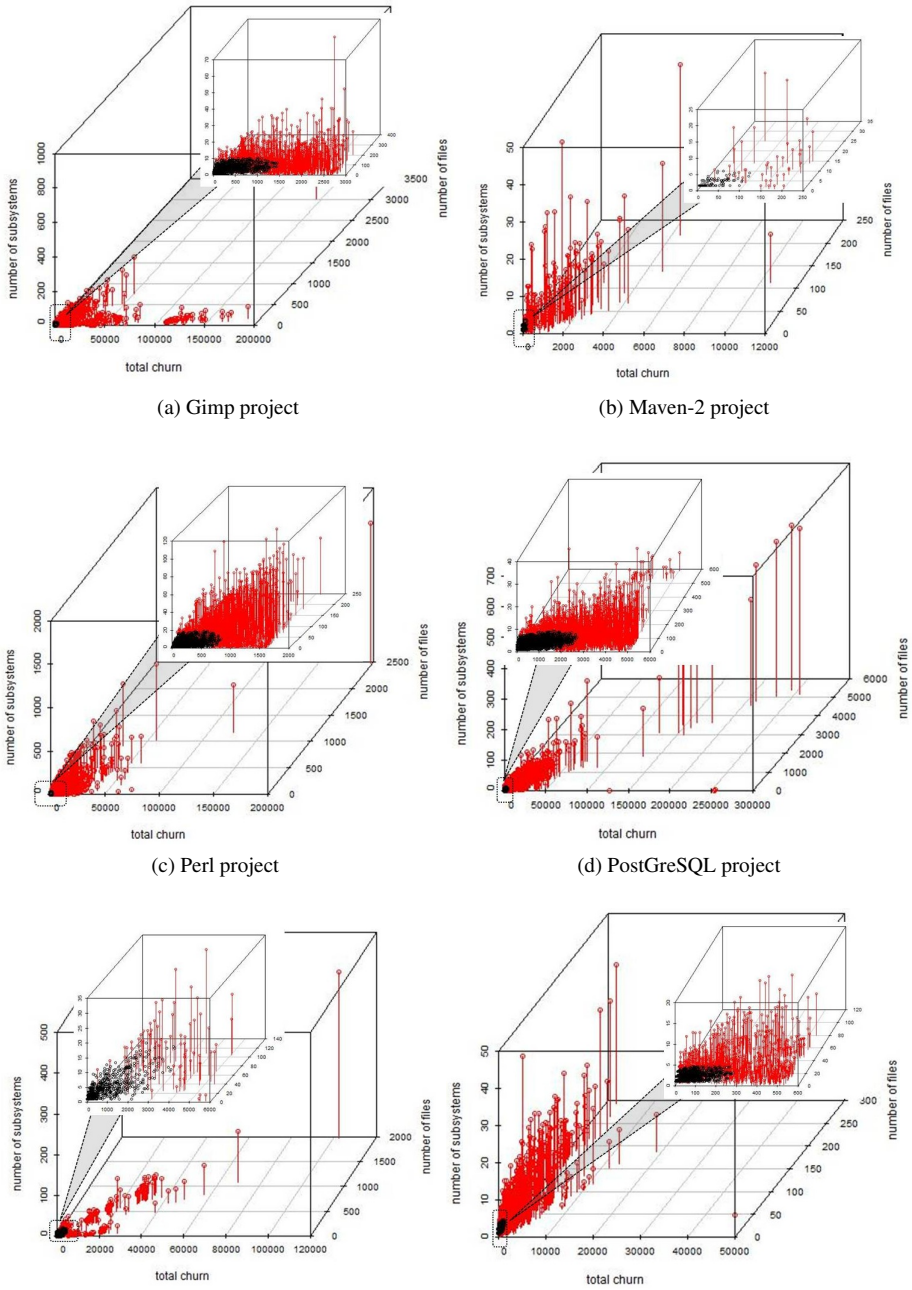
10 Appendix

Table 12 presents the minimum, 25% quartile, median, 75% quartile and maximum values of three metrics for HIFCs and LIFCs respectively in all projects. Figure 8 presents three-dimensional scatter diagrams of all projects in terms of the total churn, total number of modified files and total number of modified subsystems used for clustering fix-inducing changes. The clusters depicting HIFC are colored as red, whereas the clusters for LIFC are colored as black in the scatter diagrams. The figures are zoomed to the region where LIFCs are highly concentrated and redrawn to show the differences between two clusters. Figure 9 also shows box-plots of all projects in terms of the total churn, total number of modified files and total number of modified subsystems in two, i.e., high impact and low impact, clusters.

Table 12: Descriptive statistics for two data clusters (HIFCs and LIFCs) in the studied projects

Project	Metric*	Minimum		1^{st} quartile (25%)		Median		3^{rd} quartile (75%)		Maximum	
		HIFC	LIFC	HIFC	LIFC	HIFC	LIFC	HIFC	LIFC	HIFC	LIFC
Gimp	Churn	28	1	3864	23	11459	105	36045	321	186379	1408
	NF	1	1	1	2	5	3	39	7	3017	26
	NS	1	1	1	1	3	2	7	3	818	9
Maven	Churn	26	1	210.5	6	643.5	20	1073.25	48.5	10545	108
	NF	1	1	9	1	17.5	1	25	2	217	5
	NS	1	1	1	1	3	1	6.25	2	46	3
Perl	Churn	20	1	526	13	1080	47	3491	156	197923	814
	NF	1	1	12	1	23	2	46	4	2479	17
	NS	1	1	3	1	9	2	28	3	1609	15
PostgreSQL	Churn	94	1	2466.25	34	4490	171	8036.25	544	256245	2560
	NF	1	1	42	2	70	5	125	13	5373	48
	NS	1	1	3	1	5	1	10	2	641	7
Ruby	Churn	12	1	306	8	530	25	1036	62	47635	271
	NF	1	1	9	1	18	2	43	4	294	12
	NS	1	1	1	1	3	1	5	2	42	3
Rhino	Churn	243	1	5462	88.5	12603	399	20047	957.5	102727	3483
	NF	2	1	33	3	93	6	436	15	1965	67
	NS	1	1	5	1	12	2	20	4	409	10

*Churn: LA+LD, NF: Number of modified files, NS: Number of modified subsystems



(a) Gimp project

(b) Maven-2 project

(c) Perl project

(d) PostGreSQL project

(e) Rhino project

(f) Ruby project

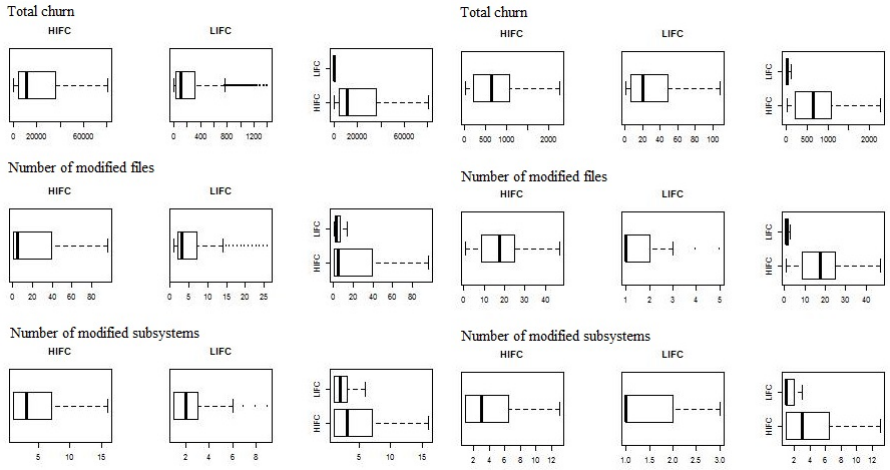
Fig. 8: Three-dimensional scatter diagrams presenting the clusters for HIFCs (red) and LIFCs (black).

References

1. Arisholm, E., Briand, L.C.: Predicting fault-prone components in a Java legacy system. In: Proc.of Int'l Sym. on Empirical Software Engineering (ISESE), pp. 8–17 (2006)
2. Bachmann, A., Bird, C., Rahman, F., Devanbu, P., Bernstein, A.: The missing links: Bugs and bug-fix commits. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pp. 97–106 (2010)
3. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)
4. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced? bias in bug-fix datasets. In: in European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE (2009)
5. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Foundations of empirical software engineering: the legacy of Victor R. Basili **426** (2005)
6. Bradley, P., Fayyad, U., Reina, C.: Scaling em (expectation-maximization) clustering to large databases. Tech. rep., Microsoft Research, MSR-TR-98-35 (1999)
7. Breiman, L.: Random forests. Machine learning **45**, 5–32 (2001)
8. Caglayan, B., Misirli, A.T., Miranskyy, A.V., Turhan, B., Bener, A.: Factors characterizing reopened issues: a case study. In: Proc. of Int'l Conf. on Predictive Models in Software Engineering (PROMISE), pp. 1–10 (2012)
9. Caglayan, B., Tosun, A., Miranskyy, A., Bener, A., Ruffolo, N.: Usage of multiple prediction models based on defect categories. In: Proc. of Int'l Conf. on Predictive Models in Software Engineering (PROMISE), pp. 8:1–8:9 (2010)
10. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994). DOI http://dx.doi.org.proxy.queensu.ca/10.1109/32.295895
11. Czerwonka, J., Das, R., Nagappan, N., Tarvo, A., Teterev, A.: Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In: Proc. of Int'l Conf. on Software Testing, Verification and Validation (ICST), pp. 357–366 (2011)
12. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: Proc. of Int'l Working Conf. on Mining Software Repositories (MSR), pp. 31–41 (2010)
13. Estabrooks, A., Japkowicz, N.: A mixture-of-experts framework for learning from imbalanced data sets. In: Proc. of Int'l Conf. on Advances in Intelligent Data Analysis (IDA), pp. 34–43 (2001)
14. Eyolfson, J., Tan, L., Lam, P.: Do time of day and developer experience affect commit bugginess. In: Proc. of Int'l Working Conf. on Mining Software Repositories (MSR), pp. 153–162 (2011)
15. Fraley, C., Raftery, A., Murphy, B., Scrucca, L.: mclust version 4 for R: Normal mixture modeling for model-based clustering, classification, and density estimation. Tech. rep., Department of Statistics, University of Washington (2012)
16. Gall, J., Razavi, N., Van Gool, L.: An introduction to random forests for multi-class object detection. In: Proc. of Int'l Conf. on Theoretical Foundations of Computer Vision: outdoor and large-scale real-world scene analysis, pp. 243–263 (2012)
17. Gethers, M., Dit, B., Kagdi, H., Poshyvanyk, D.: Integrated impact analysis for managing software changes. In: International Conference on Software Engineering (2012)
18. Giger, E., D'Ambros, M., Pinzger, M., Gall, H.C.: Method-level bug prediction. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12, pp. 171–180 (2012)
19. Giger, E., Pinzger, M., Gall, H.: Can we predict types of code changes? an empirical analysis. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pp. 217–226 (2012)
20. Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. **26**(7), 653–661 (2000)
21. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In: Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 495–504 (2010)
22. Halstead, M.: Elements of Software Science. Amsterdam: Elsevier North-Holland (1977)
23. Hassan, A.E.: Predicting faults using the complexity of code changes. In: Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 78–88 (2009)
24. Hata, H., Mizuno, O., Kikuno, T.: Bug prediction based on fine-grained module histories. In: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 200–210 (2012)
25. Herraiz, I., Gonzalez-Barahona, J.M., Robles, G.: Towards a theoretical model for software growth. In: Proc. of Int'l Workshop on Mining Software Repositories (MSR), p. 21 (2007)
26. Herzig, K., Just, S., Rau, A., Zeller, A.: Predicting defects using change genealogies. In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, pp. 118–127. IEEE (2013)

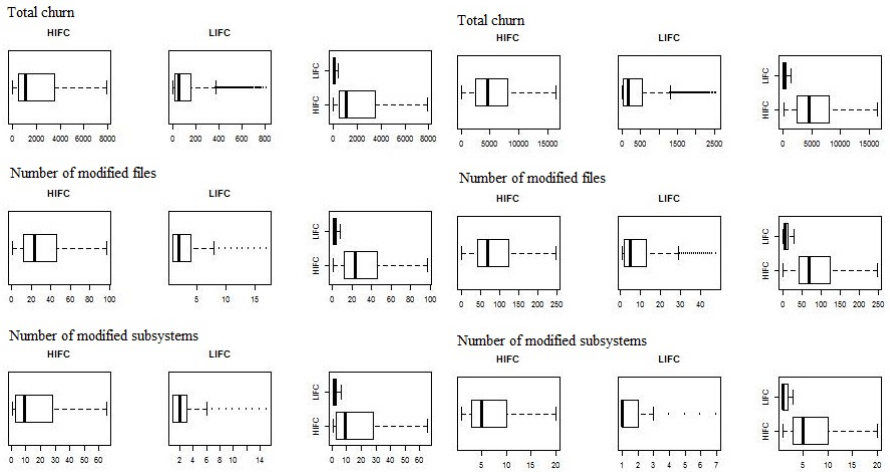
27. Jiang, Y., Cukic, B., Menzies, T.: Can data transformation help in the detection of fault-prone modules. In: Proc. of Int'l Workshop on Defects in Large Software Systems (DEFECTS), pp. 16–20 (2008)
28. Kamei, Y., Monden, A., Morisaki, S., Matsumoto, K.i.: A hybrid faulty module prediction using association rule mining and logistic regression analysis. In: Proc. of Int'l Symposium on Empirical Software Engineering and Measurement (EMSE), pp. 279–281 (2008)
29. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. **39**(6), 757–773 (2013)
30. Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: ICSE, pp. 351–360 (2011)
31. Kim, S., Ernst, M.D.: Which warnings should i fix first? In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 45–54. ACM (2007)
32. Kim, S., Whitehead Jr., E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. **34**(2), 181–196 (2008)
33. Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.: Automatic identification of bug-introducing changes. In: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on, pp. 81–90. IEEE (2006)
34. Lehnert, S.: A review of software change impact analysis. Tech. rep., Technische Universitt Ilmenau (2011)
35. Leszak, M., Perry, D.E., Stoll, D.: Classification and evaluation of defects in a project retrospective. J. Syst. Softw. **61**(3), 173–187 (2002)
36. Li, B., Sun, X., Leung, H., Zhang, S.: A survey of code-based change impact analysis techniques. SOFTWARE TESTING, VERIFICATION AND RELIABILITY **23**, 613–646 (2013)
37. Liaw, A., Wiener, M.: randomforest: Breiman and cutler's random forests for classification and regression. Online: http://cran.r-project.org/web/packages/randomForest/index.html (Accessed September 2013)
38. Madhavan, J.T., Whitehead Jr, E.J.: Predicting buggy changes inside an integrated development environment. In: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, pp. 36–40. ACM (2007)
39. Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K., Nakamura, M.: An analysis of developer metrics for fault prediction. In: Proc. of Int'l Conf. on Predictive Models in Software Engineering (PROMISE), pp. 18:1–18:9 (2010)
40. McCabe, T.J.: A complexity measure. In: Proc. of Int'l Conf. on Software Engineering (ICSE), p. 407 (1976)
41. Mende, T., Koschke, R.: Effort-aware defect prediction models. In: Proc.of European Conf. on Software Maintenance and Reengineering (CSMR), pp. 107–116 (2010)
42. Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., Jiang, Y.: Implications of ceiling effects in defect predictors. In: Proc. of Int'l Conf. on Predictive Models in Software Engineering (PROMISE), pp. 47–54 (2008)
43. Misirli, A.T., Caglayan, B., Bener, A., Turhan, B.: A retrospective study of software analytics projects: In-depth interviews with practitioners. IEEE Software **30**, 54 – 61 (2013)
44. Misirli, A.T., Caglayan, B., Miranskyy, A., Bener, A., Ruffolo, N.: Different strokes for different folks: A case study on software metrics for different defect categories. In: Proc of Int'l Workshop on Emerging Trends in Software Metrics (WETSOM) (2011)
45. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal **5**(2), 169–180 (2000)
46. Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 181–190 (2008)
47. Nagappan, N., Ball, T.: Static analysis tools as early indicators of pre-release defect density. In: Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 580–586 (2005)
48. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 284–292 (2005)
49. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 452–461 (2006)
50. Ohlsson, N., Alberg, H.: Predicting fault-prone software modules in telephone switches. IEEE Trans. Softw. Eng. **22**(12), 886–894 (1996)
51. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Programmer-based fault prediction. In: Proc. of Int'l Conf. on Predictive Models in Software Engineering (PROMISE), pp. 19:1–19:10 (2010)
52. Purushothaman, R., Perry, D.: Toward understanding the rhetoric of small source code changes. IEEE Trans. Softw. Eng. **31**(6), 511–526 (2005)

53. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: A tool for change impact analysis of java programs. In: OOPSLA. Vancouver, Canada (2004)
54. Shadish, W., Cook, T., Campbell, D.: Experimental and Quasi Experimental Designs for Generilized Causal Inference. Boston:Houghton Mifflin (2002)
55. Shihab, E.: Pragmatic prioritization of software quality assurance efforts. In: Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 1106–1109 (2011)
56. Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M.: An industrial study on the risk of software changes. In: Proc. of Int'l Sym. on Foundations of Software Engineering (FSE), pp. 62:1–62:11 (2012)
57. Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W., Ohira, M., Adams, B., Hassan, A., Matsumoto, K.i.: Predicting re-opened bugs: A case study on the eclipse project. pp. 249–258 (2010)
58. Shihab, E., Kamei, Y., Adams, B., Hassan, A.E.: Is lines of code a good measure of effort in effort-aware models? Information and Software Technology **55**(11), 1981 – 1993 (2013)
59. Shihab, E., Mockus, A., Kamei, Y., Adams, B., Hassan, A.E.: High-impact defects: a study of breakage and surprise defects. In: Proc.of Sym. and European Conf. on Foundations of Software Engineering (ESEC-FSE), pp. 300–310 (2011)
60. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. **37**(6), 772–787 (2011)
61. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proc. of Int'l Workshop on Mining Software Repositories (MSR), pp. 1–5 (2005)
62. Thung, F., Wang, S., Lo, D., Jiang, L.: An empirical study of bugs in machine learning systems. In: Proc. of Int'l Sym. on Software Reliability Engineering (ISSRE), pp. 271–280 (2012)
63. Tosun, A., Bener, A.: Reducing false alarms in software defect prediction by decision threshold optimization. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM '09, pp. 477–480. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/ESEM.2009.5316006. URL http://dx.doi.org/10.1109/ESEM.2009.5316006
64. Tosun, A., Bener, A.B., Turhan, B., Menzies, T.: Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. Information & Software Technology **52**(11), 1242–1257 (2010)
65. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? In: Proc.of Sym. and European Conf. on Foundations of Software Engineering (ESEC-FSE), pp. 26–36 (2011)
66. Zimmermann, T., Nagappan, N., Williams, L.: Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: Proc. of Int'l Conf. on Software Testing, Verification and Validation (ICST) (2010)
67. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for Eclipse. In: Proc. of Int'l Conf. on Predictive Models in Software Engineering (PROMISE), pp. 9–15 (2007)
68. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 563–572 (2004)



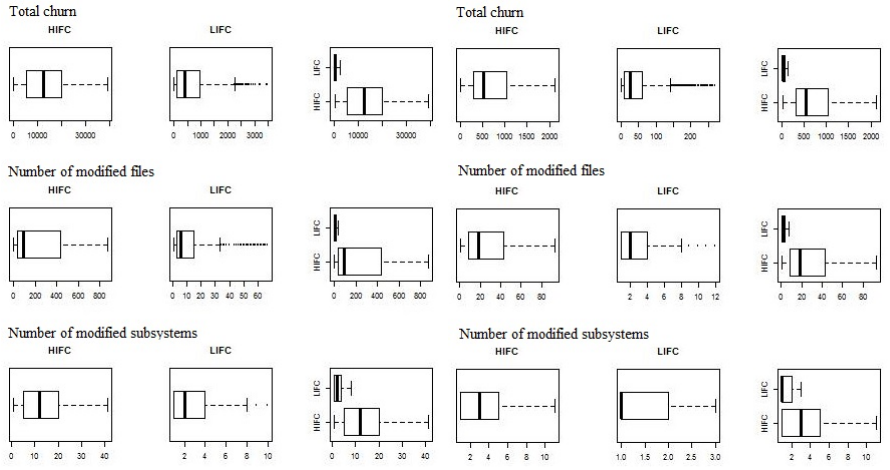
(a) Gimp project

(b) Maven-2 project



(c) Perl project

(d) PostGreSQL project



(e) Rhino project

(f) Ruby project

Fig. 9: Box plots of all projects in terms of three metrics used for identifying the clusters, HIFCs and LIFCs.