Enhanced-SZZ: an improved code change labeling algorithm

Mina Remeli

Supervisors:
Dr Horváth Gábor
BME-HIT
Kollár Nándor
Cloudera,Inc.

2019

Contents

Kivonat

A szoftverhibák elképesztően költségesek - körülbelül egy billió hétszáz milliárd dollárba kerültek 2017-ben - mutatta be egy kutatásában a Tricentis, egy szoftver teszteléssel foglalkozó cég. Nem is csoda, hogy a fejlesztői közösség különböző minőségbiztosító (QA - Quality Assurance) folyamatokkal próbálja ezen költségeket minimalizálni. QA folyamat lehet például tesztek, dokumentáció írása és az ún. peer-reviewing (szakértői felülvizsgálat).

Míg a tesztelés és dokumentáció folyamata részben automatizálható, a peer-reviewing továbbra is olyan szűk keresztmetszet maradt, ami kizárólag humán erőforrásoktól függ. Emiatt számos publikáció született abból a célból, hogy ezen felülvizsgáló folyamatokat felgyorsítsák, és a nagy valószínűséggel hibát (ún. bug-ot) tartalmazó változtatások felülvizsgálatát priorizálják. Ennek egy módja a change classification - egy olyan modell építése, ami egy kódváltoztatásról (code change-ről) megmondja, hogy tartalmaz-e hibát, vagy sem. A change classification több szempontból is előnyt jelentene a fejlesztők számára - egyrészt hamarabb tudnának potenciálisan hibás kódot azonosítani - másrészt a hibakeresést az adott kódváltoztatásra tudnák szűkíteni.

Hogy megépíthessük a change classifier tanítóhalmazát - előbb a múltbéli kódváltoztatásokat kell ellátnunk "tartalmaz hibát" vagy "nem tartalmaz hibát" címkével. Ezt egy SZZ nevű algoritmussal érhetjük el. Ugyan az SZZ algoritmus a legelterjedtebb módszer arra, hogy címkével lássuk el az adatunkat, van néhány gyenge pontja is. Ezen kifogásolható tulajdonságai miatt gyakran fals pozitív illetve fals negatív címkéket produkál.

Ezért is kihangsúlyoznám, hogy milyen fontos egy change classifier megbízhatósága szempontjából, hogy az adat, amin tanul, a lehető legjobb reprezentációja legyen a valóságnak. Az én célom ezen munka keretében az, hogy bemutassam az SZZ-nek egy olyan variánsát, ami megoldást nyújt az eredeti algoritmusban felfedezett gyengeségek nagy részére. Továbbá még a szakirodalomban nem említett, ám általam jelentősnek vélt hiányosságait is javítottam az algoritmusnak. Mindezeket a javításokat kombinálva egy jobb kódváltoztatás-címkéző algoritmust kapunk, aminek az Enhanced-SZZ nevet adtam.

Abstract

Failures are extremely expensive - one study performed by Tricentis[1], a software testing company, shows that the cost of software bugs in 2017 was approximately one-trillion seven hundred billion dollars, which is an astounding amount. To minimize such costs software developer communities follow numerous QA (Quality Assurance) processes that begin in the design phase and follow the software through its various life-cycles. Quality Assurance includes, but is not limited to writing tests, documentation and peer review.

While testing and documentation can be (at least partially) automated, peer review is still one of the core processes relying purely on human intelligence and resources. To speed up and prioritize the review of potentially bug-inducing commits, there have been numerous studies focused on classifying software changes. One of the many benefits of change classification is that it helps developers discover bug-prone code faster (as soon as they make a commit) and more effectively by limiting the location of a bug to the change.

To build the training dataset of the code change classifier, we first have to label past commits as "bug-inducing" or "not bug-inducing". To achieve this, we use an algorithm called SZZ. In essence, the algorithm finds bug inducing commits by looking for changes made before a bug-fixing commit. While SZZ is the most established way of labeling changes as bug-inducing, it also calls for a few comments. The main issue with this algorithm is that it provides us with many unwanted false positive and false negative samples.

I would like to emphasize, that it is of utmost importance to build a reliable training dataset (an accurate ground truth) for change classifiers, otherwise the correctness of the predictions become questionable. My aim in this work is to present a new variant of SZZ, that combines the correction of most, if not all of its criticized attributes. I also fix further shortcomings of SZZ, not handled so far by the literature. By combining all of these improvements, I present Enhanced-SZZ, an improved code change labeling tool.

[1] `https://www.tricentis.com/resources/software-fail-watch-5th-edition/`

1 Introduction

1.1 Motivation

To ensure code quality, software developer communities follow numerous QA (Quality Assurance) processes that begin in the design phase and follow the software through its various lifecycles. They could include designing and running tests, writing documentation, etc. in a predefined, regulated way. One of the most commonly applied QA processes (almost everywhere) is peer code review.

Peer code review is a process where "peers" look at a proposed code change and decide whether it can be merged into the codebase. For example, the Apache Software Foundation has its own predefined set of voting rules regarding code modifications[2]. Members of the Apache community can vote for and against code changes. Rules dictate that three positive votes and no negative votes are needed for accepting a code change.

While testing and documentation can be (at least partially) automated, peer review is still one of the core processes relying purely on human intelligence and resources. Based on a case study, developers at Google spend on average 3.2 hours/week reviewing commits [1]. Another study found that the self-reported time of developers spent on reviewing open-source projects is 6.4 hours/week on average [2].

To aid and partially automate the review process, several commit-level bug-prediction methods are actively researched. Most related research is centered around building classifiers, that group commits into two categories: 'bug-inducing' or 'not bug-inducing'. These are commonly called *code change classifiers*. A good code change classifier can save us time with commits that are classified as safe, and gives us a chance to mitigate possible risks by looking into commits that were classified as bug-inducing.

In order to build such a classifier, we need a dataset that labels code changes as bug-inducing or not bug-inducing. Ideally, one would create a domain-expert curated dataset, where the sources of bugs are pinpointed by the developers themselves. Nonetheless, this is practically impossible for multiple reasons. Not only is it a very time consuming task to label possibly thousands of commits, but they would also need to revisit commits that date back years, which again makes it more difficult to find the origin of a bug. In lack of such time and resources we are forced to use methods that are less accurate - but infinitely faster. Currently there is one preferred family of algorithms used in practice for labeling commits, called SZZ [3].

Since the usefulness of the prediction depends directly on the quality of the underlying training dataset, we need to ensure that the *code change labeling algorithm* we use provides labels as close to the truth as possible.

I felt that this issue is not addressed properly in any of the proposed code change classifiers found in the literature - even though their proposed models are only as good as the ground truth that their datasets offer. I argue that this issue has been glossed over long enough, and **more research should be focused on the algorithm that creates the dataset, rather than the algorithm that learns from it**.

[2] "Apache Voting Process - The Apache Software Foundation!." `https://www.apache.org/foundation/voting.html`

1.2 Background

Code change labeling algorithm

Code change labeling algorithms have been around since 2005, when Sliwerswki, Zimmerman and Zeller introduced an algorithm [3] for finding the source of a bug based on a bug-fix change. Nowadays it is often referred to as SZZ (where the letters stand for the first letter of the authors' names). Its popularity can be attributed to its simplicity and intuitiveness. The algorithm first searches for bug-fix commits (eg. for commits that fixed bugs), and then links them to the commit that induced the bug (eg. the bug-inducing commit) by going back in the history of bug-fixing lines to find the commit that introduced the bug.

Five years later a different approach was proposed by Sinha et al. [4], which analyzes changed dependences instead of changed lines. The authors implement their solution partially (on an *intraprocedural* level) and compare it to SZZ. While inspecting data and control dependences might yield better results, the full implementation of this algorithm is non-trivial and time-consuming. The computational cost of this approach is also much higher, taking on average 7.2 times longer when tested on 4 projects, therefore making SZZ a more preferable candidate.

SZZ has many variants [5, 6, 7] that address various aspects of the algoritm's shortcomings. Further details on the original SZZ algorithm and its variants are presented in Section 2.

Code change classifiers

Code change classifiers are used to classify code changes as "bug-inducing" or "not bug-inducing", and are of equal interest to the research community [8, 9, 10, 11] as well as to the industry [12]. First, they build a dataset by labeling past commits of a selected project using the SZZ algorithm. Subsequently they train a machine learning model on this dataset, which is then used to classify new, incoming commits. They use code change related features as input to their models, like the changed code itself or related metadata (like commit time, commit author or number of changed files).

Details on the different implementations and their reported accuracies are elaborated in Section 2.

1.3 Contributions

My main contributions are as follows:

- I implemented a scalable, customizable variant of SZZ that handles most if not all weak points pointed out by previous implementations. This includes ignoring comments and blank lines like Kim et al. [9], ignoring merge commits similarly to Costa et al. [13] and excluding refactoring changes as seen in Neto et al. [7].
- Furthermore, I extend my implementation by including bug-fix hunks in the search that only added lines. There have been no implementations handling additions thus far, but it has been noted as a possible future improvement to the already existing algorithms [14, 13].

- I evaluate my implementation based on the framework proposed by Costa et al. [13], and some additional metrics proposed by me. Furthermore I present its advantages on a simple change labeling algorithm, that was trained on commits labeled by E-SZZ.

Details on my implementation can be found in Section 3, evaluations are in Section 4.1.

2 Related Work

2.1 The original SZZ algorithm

SZZ is a popular algorithm for finding bug-inducing commits. It was first introduced by Śliwerski, Jacek, et al [3]. The original algorithm has been developed using software versioning systems like CVS or SVN, and bug tracking systems like Bugzilla. The main idea of the algorithm is very simple - it first searches for bug-fix commits (eg. for commits that fixed bugs), and then links them to the commit that induced the bug (eg. the bug-inducing commit).

Now for a more in-depth explanation: it iterates over the commits and searches for tokens possibly indicating a bugfix commit (like "fix", or "bug") or a number (possibly an identifier of a bug in the bug tracking system). If there was a number in the commit message, there is a potential link to a bug. If the commit can be successfully linked to a bug report (having met one of the required conditions of the algorithm), then it is deemed as a bug-fixing commit.

The locating of bug-inducing changes is as follows: we iterate over the bug-fixing commits and call the CVS `diff` command. We note the deleted/modified lines in our bug-fixing revision (SZZ does not handle additions). Then, the CVS `annotate` command is called on the last revision not containing the fix. This outputs the most recent revisions that touched each line, followed by the date and author. We exclude revisions that were created after the bug report (since the bug must have been introduced before it got reported). The remaining revisions are marked as bug-inducing. The core concept of the original SZZ algorithm can be observed in Figure 1, where the bug-fix change modifies both `a()` and `b()`, revision 1.3 could not have introduced the bug because it was created after the bug report.

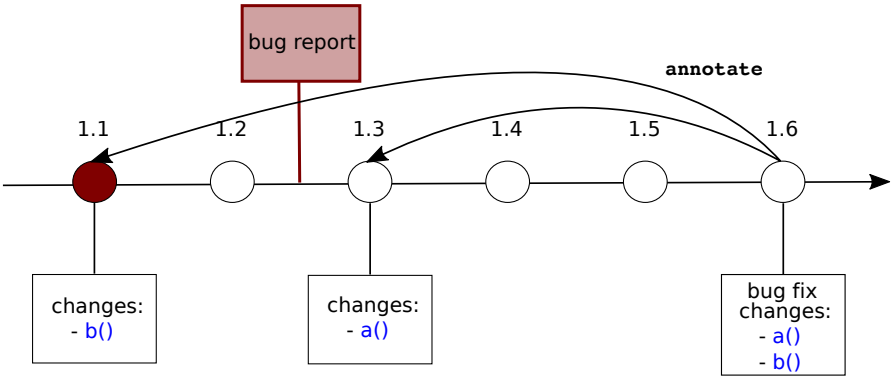


Figure 1: The original SZZ algorithm. The change made in revision 1.1 is marked as bug inducing.

2.2 SZZ variants

Maybe the most well-known variant of SZZ (besides the original one) is the one proposed by Kim et al [5]. It verified the benefits of their approach over the original SZZ by way of demonstrating improvements on the commits of Columba and Eclipse. It claims, that their algorithm can remove about 38%-51% of false positives and 14%-15% of false negatives compared to the previous algorithm.

They apply five steps which achieve these results: (1) They use annotation graphs to link lines of one revision to the other. (2) They ignore comments and blank-line changes. (3) They ignore the blamed revisions that only introduced formatting changes in the code (like moving brackets in a new line). (4) They ignore outlier bug-fix revisions in which too many files were changed. (5) They manually verify all hunks identified as bug-fix changes. Costa et al. [13] later proposed an improvement to this implementation, which included a final step which filtered all meta-changes (eg. all changes that are not related to source code modifications, like merges).

Subsequent work by Williams and Spacco [6] reviewed the SZZ algorithm improved by Kim et al. [5], and compared it to their variant that uses line-number maps instead of annotation graphs, and DiffJ, a Java syntax-aware diff tool that ignores comments and formatting changes. Finally, they manually inspected 25 random bug fixing commits to see if the commits marked as bug-inducing were actually bug inducing. The 25 bug fixing commits contained a total of 50 changed lines, that were mapped back to a bug-inducing commit. Only 43 of the 50 lines were actual bug-fixing changes. 33 changes out of 43 were actual bug-inducing commits. 4 false positives were due to the bug inducing code being injected before them. The rest of the false positives stem from DiffJ not quite producing an accurate set of changes and the source lines being lost by the line mapper.

SZZ has also been criticized for not taking into account refactoring changes - changes that impact the input (bug-fixing changes) and also the output (bug-introducing changes) of the algorithm. Refactoring changes can not fix nor induce bugs since they do not change how the code works. The effect of removing refactoring changes from both the input and the output of the SZZ algorithm was investigated by Neto et al. [7], and resulted in a new variant of SZZ, called RA-SZZ (Refactoring-Aware SZZ). They use an existing tool that automatically detects refactorings in code revisions, called RefDiff [15]. Their observations that are based on 10 projects are as follows: 6.5% of lines that are flagged as bug-introducing are in fact refactoring changes. Regarding bug-fix changes, they observe that 19.9% of lines that are removed during a fix are related to refactorings and, therefore, their respective inducing changes are false positives.

Given that the algorithm can link multiple bug-inducing changes to one bug-fix change, there have been two proposals to reduce the number of candidate bug-inducing changes to one. Davies et al. [14] choose to select either the largest (in terms of changed lines of code, LOC) or the most recent of the candidate bug-inducing changes. This reduces the number of false-positives, as well as the number of true positives unfortunately. They also showcase examples of this approach failing where there are indeed multiple sources being held responsible for the existence of a bug. Based on their manual analysis of the approaches, they conclude that filtering the candidate bug-inducing changes by looking at either the largest or most recent revision does not produce any clear benefits.

Since added lines do not have a history like deleted and modified lines do, the original SZZ algorithm does not know how to find the source of a bug fixed by an addition. Davies et al. [14] are also the first ones to evaluate an extended version of SZZ that handles added lines as well (since the original algorithm is only able to trace back deleted/modified lines). They found, that 70 out of the 301 observed commits in Eclipse had no bug-inducing commits assigned to them (when running the original SZZ algorithm), which introduced a

significant amount of false negatives. This motivated them to treat the enclosing block of additions the same way as if they were the subject of the bug-fixing change. Because of the noticeable reduction of false negatives in their manually analyzed set of commits, they infer that the results clearly benefit from this feature. However, Davies et al. did not implement their proposed changes. Instead, the steps of each variant were conducted manually. This restricted them in their evaluation of the proposed algorithms: they had to limit their scope of analysis to only 15 surveyed changes.

2.3 Dependency based change labeling

Sinha et al. [4] propose a dependency-based code change labeling algorithm instead of SZZ. They compare revisions using *program dependence graphs* (PDG). A PDG is a directed graph describing control and data dependence, where each statement is represented as a node. A control dependence between nodes A and B ($A \rightarrow B$) means that the execution of node B depends on the output of node A. A data dependence on the other hand would indicate that B references a variable that A defined.

First, they analyse what dependences changed in the bug-fix revision, by comparing the PDG of the bug-fix with the PDG of its parent. Then they follow this by searching for the revision that created the removed dependences, and stop the search at the first revision that they find (multiple dependences might be removed, but they only go back as far as the introduction of any one of the removed dependences). In case there are no removed dependences, they analyze the statements connected to the added statements, and search for the last revision that touched those statements.

Thus far, this algorithm has only been implemented to detect bugs inside one method (*intraprocedural approach*) using PDGs, but Sinha et al. made a suggestion on how the *interprocedural approach* could be implemented as well.

An advantage of dependency based change labeling is that it takes semantic information into consideration while searching for the bug-inducing changes, and also tends to generate far less false positives compared to SZZ [14]. However there are several feasibility limitations to using this algorithm - reportedly this algorithm takes 7.2 times longer to run than SZZ. The reason for this is the amount of time it takes to build the dependence graphs - and the further apart the bug-fix and bug-inducing commits are the longer it takes to run the algorithm. Also, there is no readily available open source implementation for building build PDGs. This would force us to implement it ourselves, which due to its complexity would presumably take much longer than implementing SZZ.

2.4 Use case on real projects: Software Change Classification

There are numerous aspects and research areas regarding bug detection. Some studies focus on predicting fault prone classes [16] or modules [17, 18], while others study defect prediction, where the goal is to predict the number of induced bugs on a binary / class / package level [19]. Yet another mention-worthy area is the prediction of the number of defects in a software over a defined period of time using complexity information of past code changes [20].

However, the studies of interest to us address the problem of *change classification*. One of the many benefits of change classification is that it helps developers discover bug-prone code faster (as soon as they make a commit) and more effectively by limiting the location of a bug to the change.

The first study to predict bugs on a change level is the work by Aversano et al. [8]. It uses the modified version of SZZ proposed by Kim et al. [5] for change labeling. It tests its models on two systems - JHotDraw and DNSJava, on 132 and 1204 changes respectively. The tested models are the following: KNN (K-Nearest Neighbor), LR (Logistic Regression), Multi-boosting, C4.5, and SVM (Support Vector Machine) classifiers. The input of the models is the weighted terms vector representation of code changes (where the code change is the vector difference between two subsequent code snapshots). The achieved results present high precision and recall when classifying non-bug-inducing commits across both projects. For bug inducing commits on the other hand the recall was not as high as the precision. In the case of JHotDraw the LR model performed best (with a precision of 80%) and in the case of DNSJava KNN performed best (with a precision of 69.4%).

The second study I would like to talk about is presented by Kim et al [9]. It makes predictions using an SVM (Support Vector Machine) classifier on a wide range of projects that are open source, and achieves an accuracy reaching from 64%-92% (depending on the project). He used a multitude of features as a base for classification, including change log metadata, complexity metrics, change log messages, source code and file names. Surprisingly enough, for identifying bug-inducing changes he doesn't use the modified SZZ algorithm [5] proposed by himself and one of the original authors of SZZ, but the original one. However, he conducts an extensive amount of experiments, playing with the different combinations of features, noting their effect on accuracy and recall. He concludes that there is no definitive feature set performing well on all of the projects. He noted that one of the challenges of the study was to ensure that the SVM model fits into memory. SVM models were a conscious choice for his study seeing that he had a copious amount of features - the number of extracted features in his work ranged from 6k to more than 23k features! The reason why he chose SVMs is because the memory need of SVM models only increases with the number of data points. Kim et al. used features from 500 (sometimes 250) revisions to train and evaluate an SVM classifier.

Kamei et al. [10] has also achieved notable results on the data of 6 open source and 5 commercial projects. Bug inducing commits are identified using the SZZ algorithm. He chose a feature set of 14 features which he divided into 5 dimensions based on the type of information the feature conveys: (1) diffusion, (2) size, (3) purpose, (4) history and (5) experience. He produces an average accuracy of 68% and an average recall of 64% using a Logistic Regression (LR) model. He found that risk increasing factors include the higher number of changed files in a commit and the fact that the commit itself is bug-fixing. A risk decreasing factor is the average time interval since the previous change (the older a file's last change is, the lower the chance that the current change will induce a defect). The results of this work were taken further in creating a commit-risk analysis tool called Commit Guru [11], which is available for the public to try out.

Industrial players are also displaying interest in the field of bug prediction. Ubisoft, a well known video game company has collaborated with researchers

to create CLEVER [12], that can detect risky commits with 79% precision and 65% recall. In addition this tool can also recommend qualitative fixes in 66.7% of the cases. They use the modified version of SZZ proposed by Kim et al [5] for building their dataset of defect and bug-fixing commits. A pre-commit hook was used to catch new commits and calculate the features used in Commit-guru. Then they feed their data into a Random Forest classifier. If the classifier marks the commit as risky, then the change code block is extracted and is matched with a past bug-inducing commit using text-based clone detection techniques. The fix of the matched bug-inducing commit will be the recommended fix.

3 Enhanced SZZ

In this section I present my proposed algorithm, Enhanced-SZZ (or E-SZZ for short). It combines the improvements proposed so far by the literature (aimed to reduce the false positives produced by the original algoritm), and even includes a novel improvement that reduces false negatives. This new improvement includes the processing of additions (which were not handled so far in the literature), that represent a substantial 28.91% of the total bugfix hunks in the project examined by me (as pointed out in Section 3.2). The significance of handling additions was acknowledged both by Davies et al. [14] and Costa et al. [13], but was not yet implemented, nor extensively evaluated. The new algorithm I propose is also designed in a way that is scalable, and easily configurable.

This introduction will include a short insight on the project it was evaluated on, and the issue tracking and version control systems it relies on in Section 3.1. Subsequently in Section 3.2, I discuss the shortcomings of the original algorithm as pointed out by the literature and include some of my own critiques. Then I proceed on introducing the algorithm that is designed to solve the discussed issues in Section 3.3.

We used `Python 3.6`, and E-SZZ was run on a server with a `Intel(R) Xeon(R) E5-2666 v3 @ 2.90GHz` 32-core CPU. Some of the more important libraries were: `GitPython`[3], `JIRA`[4] and `jira-cache`[5]. Implementation and evaluations are available at `https://github.com/minaremeli/TDK_19`.

3.1 Study setting

My proposed algorithm was applied on a real-world, large, active project. First, I will introduce the project that I chose to work on, then I will go on to explain how the tools used almost 15 years ago changed, and how this affected my implementation of SZZ.

HIVE

HIVE[6] is one of the projects of the Apache Software Foundation[7], that facilitates the reading, writing, and managing large datasets via SQL that reside in distributed storage. Currently, it boasts a code base consisting of approximately 18k files and 13k commits, where the oldest commit dates back to 2008. HIVE has close to 200 contributors, and the project is mainly written in Java. I used the HIVE project to create my very first labeled dataset using Enhanced SZZ.

E-SZZ integration with JIRA and Git

It is important to note, that the original SZZ algorithm dates back to 2005, when the version control system of choice was SVN[8] (which happens to be an Apache project as well). At that time there were no issue tracking systems, only

[3] `https://pypi.org/project/GitPython/`
[4] `https://pypi.org/project/jira/`
[5] `https://pypi.org/project/jira-cache/`
[6] `https://github.com/apache/hive`
[7] `https://www.apache.org/`
[8] `https://subversion.apache.org/`

bug tracking systems like Bugzilla[9], which occasionally tracked feature requests and improvements as well, but had no labeling system to distinguish between the different categories.

Issue tracking system: JIRA Most projects in Apache use issue tracking systems like JIRA, where each issue is assigned a label (by the creator of the issue) such as "Bug", "Improvement", "New Feature", "Sub-Task", etc. The main improvement over systems such as Bugzilla is, that it allows issues to be categorized, whereas Bugzilla had no way of distinguishing bug reports from new feature requests, etc. This allows us to completely forego the original, rather fragile way of identifying bug-fixing commits where we searched for bug-fixing issues by parsing certain keywords (like "bug" and "fix").

HIVE uses JIRA[10] as its primary issue tracking system as well. Linking commits to issues, identifying bug fixing changes and finding bug inducing changes became both simpler and more efficient: nowadays more and more projects have a predefined workflow which requires (or at least strongly advises) to link every commit to an issue ticket. This applies to all of the Apache Software Foundation projects as well. I have verified that indeed only 2.8% of HIVE commits could not be linked to any issue ticket, which is a negligible loss.

Version control system: git HIVE currently uses `git` as its version control system. There are many advantages of using `git` over SVN: past works had to build annotation graphs with SVN `annotate` to connect lines from a newer revision to the lines of an older one [5]. Connecting lines across revisions becomes a much simpler task using `git log` that doesn't require an annotation graph.

Here are some short explanations of the `git` commands that I frequently used:

- `git diff <commit1_sha> <commit2_sha>`
  Shows changes between two commits. Some useful flags I used with it: `--ignore-cr-at-eol` (ignores carry at end of line), `--ignore-space-change` (ignores change in number of spaces), `-ignore-blank-lines` (ignores blank lines) and `-diff-filter=MRC` (used for filtering files that have undergone a specific change type, marked with the appropriate flag: M - modified, R - renamed, C - copied etc.).

- `git log -L <range>:<file> <rev>`
  Outputs the history (aka. all the revisions that touched it) of the given line in given file starting with `<rev>`. Some useful flags I used with it: `--ignore-cr-at-eol` (ignores carry at end of line), `--ignore-space-change` (ignores change in number of spaces), `-ignore-blank-lines` (ignores blank lines). This command discontinues the need to build annotation graphs to track the history of a line as seen in Kim et al. [5] and Neto et al. [7].

Integration Our implementation of the algorithm achieves the same goals as the original SZZ implementation, but with different tools. In Figure 2 we can

[9] `https://www.bugzilla.org/`
[10] `https://issues.apache.org/jira/projects/HIVE`

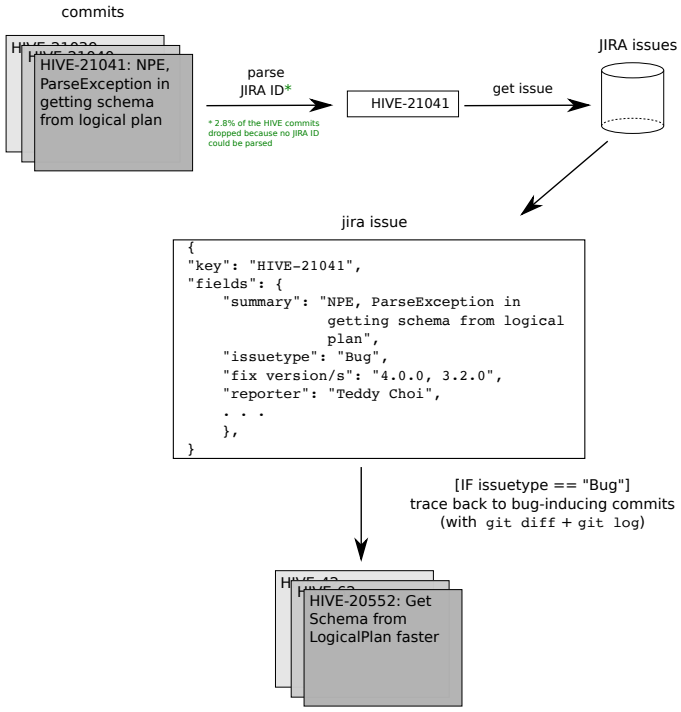


Figure 2: E-SZZ integration with JIRA and git.

see how E-SZZ integrates with JIRA and git. First, we link commits to JIRA tickets by parsing the JIRA IDs in commit messages. Then, if the linked JIRA issue is of type "Bug" (e.g. the linked commit fixes a bug), we use `git diff` to see which lines were changed. Then we find the corresponding bug-inducing commits by calling `git log` on the changed lines.

3.2 Shortcomings of SZZ

The motivation behind creating Enhanced SZZ is to address most of its shortcomings pointed out by the literature, as well as some additional shortcomings pointed out by me.

Multiple marked commits

Due to how SZZ works and traces back all lines belonging to a bug-fix, it tends to not only find one potential bug-inducing commit, but multiple. Many of these can be false positives, for which reason multiple suggestions have been made to select the most probable candidate. Davies et al. [14] select either the largest (in terms of changed LOC) or the most recent of the candidate bug-inducing changes. They evaluate the effect of these suggested modifications by manually analysing 15 commits, and conclude that filtering the candidate bug-inducing changes by looking at either the largest or most recent revision does not produce any clear benefits. They also point out that not every bug is necessarily caused by one commit and that in such cases by blaming only one commit can introduce false negatives.

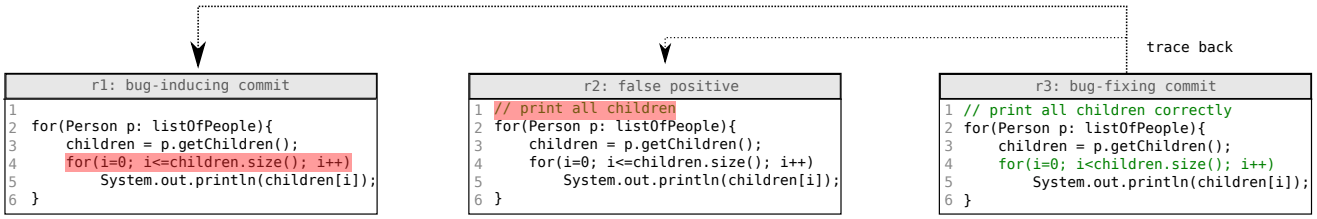


Figure 3: An example of an introduced false positive. **r3** fixes a bug but also alters the comment added in **r2**. Nevertheless the original SZZ algorithm marks both **r1** and **r2** as bug-inducing.

Changes with no effect on functionality

Bug-fix commits do not always modify lines exclusively related to the bug they are fixing - more often than not they include opportunistic refactorings, added test cases (for preventing future occurences of similar bugs) or comments to improve code readability. This however results in more false positives when tracing back the origin of those lines. I present an example of an introduced false positive in Figure 3.

The most obvious flaw pointed out by all previous SZZ variants is SZZ's inability to detect changes that have no effect on functionality whatsoever. Kim et al. [5] filter comments, blank lines and formatting changes. Williams and Spacco [6] use DiffJ, a Java syntax-aware tool that ignores comments and formatting changes. Neto et al. [7] present RA-SZZ (Refactoring-Aware SZZ), that detects refactoring changes using RefDiff. They show, that using the RA-SZZ a significant amount of false positives can be filtered (they observe a 20.8% decrease in lines that are flagged as bug-introducing).

Additions

While research made important contributions to improving the original SZZ algorithm, they fail to address one issue that might be just as important as filtering out changes that have no effect on functionality. And that is the handling of additions in bug-fix changes. *SZZ is only able to trace back bug-fix lines that have a history*, which does not hold true for newly added lines. In Figure 4 I present a common bug-fixing example, the handling of a null-pointer-exception, to justify the incorporation of additions in my proposed algorithm.

Furthermore, I found that a substantial amount of bug-fixing commits contain changes that are purely additions. If one compares a revision with its parent using `git diff`, then they get back all of its changes (called *hunks*) made across all files, with respect to the parent revision. Each hunk shows one area where the files differ. For each of them the `git diff` output starts with some header lines, followed by the actual modifications (an example of such an output can be seen in Figure 7). Lines that were removed from the first revision of the file are prefixed with a `-`, while lines that were added are prefixed with a `+`. Hunks that only have lines starting with a `+`/`-` prefix are purely *additions*/*deletions*, and hunks that have both are *modifications*. I examined the bug-fixing commits of HIVE, and found that **28.91% of the hunks are in fact purely additions**.

Our findings are in accordance to those of Davies et al. [14], who find that out of the 301 changes labeled with SZZ, 70 had no bug-inducing commits

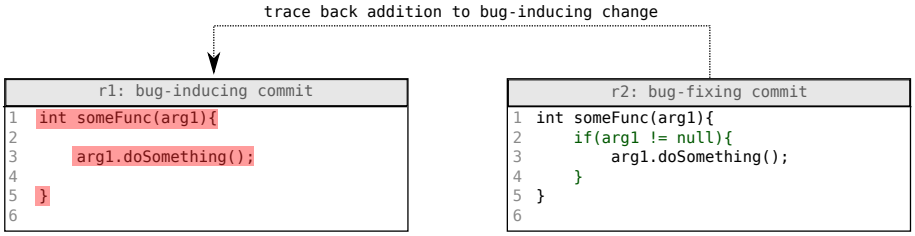


Figure 4: A classic example of a bug: a null pointer exception. This example demonstrates the importance of tracing back additions as well. `r2` is the revision where the null pointer exception is fixed, and by calling `git log` on the surrounding lines we can trace the bug back to `r1`.

linked to them owing to the fact that these changes were purely additions. This gives us all the more justification to consider bug-fixing additions as a valuable source for finding bugs, that should not be ignored. By including the handling of additions in our algorithm, we can eliminate a considerable amount of possibly false-negative samples that the original algoritm does not address. We handled additions by calling `git log` on the lines *surrounding* the change (by including one line above and under the change), as shown in Figure 4.

3.3 Detailed description of E-SZZ

The main steps of my algorithm are explained in Figure 5. First, I find the bugfix commits as shown in Figure 2, then I filter them. The output of the filter are the bug-fixing lines whose history we have to trace back using `git log` to obtain the candidate bug-inducing commits. They are also passed through a filter and last but not least we repeat the trace back step as long as the candidates are refactors.

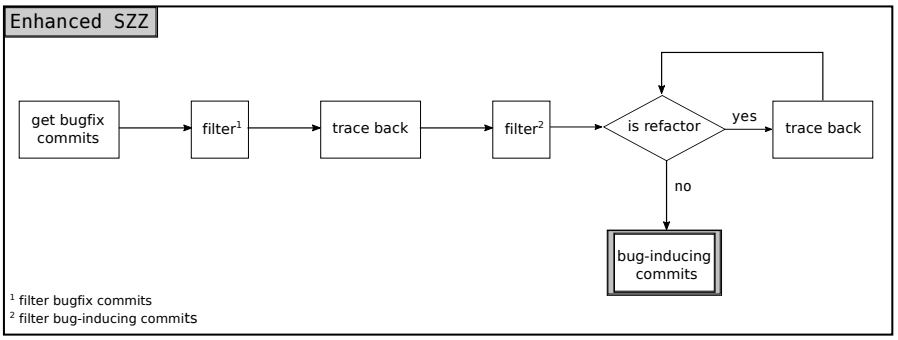
Before running the Enhanced SZZ algorithm, we first need to retrieve/build the data sources that are used later on for some of the steps in our algorithm (Figure 6).

Extracting refactoring data using RefDiff I begin by building our refactoring data, which stores all past refactorings on a line-level granularity. This is done in exactly the same way as shown in Neto et al.'s work [7], where RefDiff [15], an open source[11] tool to mine refactorings is modified so it saves all past refactoring-related information (like revision, filename and line range).

RefDiff is a tool that identifies refactorings performed between two revisions in a git repository, and it supports three languages: Java, JavaScript, and C. It combines static code analysis and code similarity to detect 13 well-known refactoring types out of the 63 from Fowler's catalog [21]. RefDiff takes as input two versions of a system, and outputs a list of refactorings found. They use a model to represent high level source code entities (like types, methods and fields) similar to ASTs (Abstract Syntax Trees). They compare the models of two such revisions and create a bipartite graph by connecting the entities with relationships describing the change.

[11] `https://github.com/aserg-ufmg/RefDiff`

Figure 5: The Enhanced SZZ algorithm.



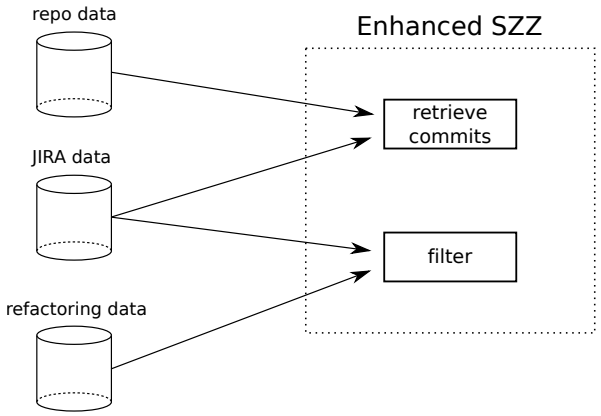


Figure 6: How different data sources contribute to some of the steps in the Enhanced SZZ algorithm.

The tool already provides an interface for extracting information on which line the refactorings start. It was not difficult therefore to extend it to find the ending line as well. I created a script which iterates over each revision of a project and uses RefDiff to extract all introduced refactor changes. For each matched refactor relationship I saved the revision, file, start and ending line associated with it.

Repository and issue tracking data I clone the examined project locally and use `GitPython` to extract repository-related information. Finally, I download all issue tickets using `JIRA`[12], and save them using `jira-cache`[13].

Next I am going to delve into more detail on the pipe-and-filter architecture and the individual filters that I implemented.

Pipe-and-Filter architecture

The pipe-and-filter architecture was created to streamline the processing of commits. There are four types of filters that I created: commit filters, file unit filters,

[12] `https://pypi.org/project/jira/`
[13] `https://pypi.org/project/jira-cache/`

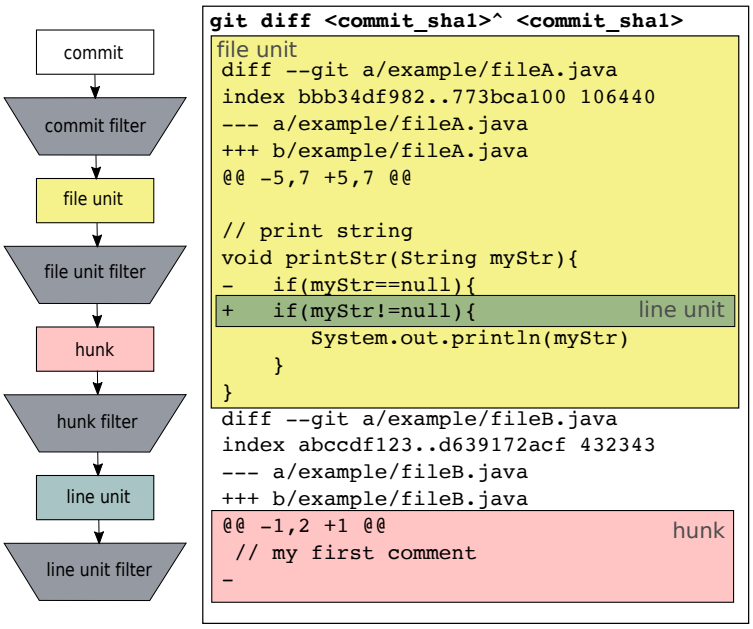


Figure 7: Pipe and filter architecture.

hunk filters and line unit filters (Figure 7). Commits are commit objects, while file/hunk/line units are parts of the outputs of the `git diff` command. I filtered these using regular expressions. This solution provides us with a very flexible, and **customizable** implementation, that can be easily manipulated to fit different project/programming languages as well. Since every commit goes through the same pipeline, it enables us to parallelize these steps, thus making my solution **scalable** as well.

The pipe-and-filter architecture is only one part of the design that I have implemented. Some filtering functionalities were simply achieved by using the right flag with `git diff`, as shown in Section 3.1. For a complete list of flags and filters and their motivation see Table 1.

Bugfix commit filters The following are the main filters that I used to filter the bugfix commits:

- commit filters
  1. *has jira*
  2. *is bugfix*
- file unit
  1. *test files*
  2. *filetypes*
- line unit
  1. [`if hunk != addition`] *context*
  2. *comments*
  3. *blank lines*
  4. *imports*

5. *refactor*

Most of the filters are quite self-explanatory - like checking if a commit has a corresponding JIRA ticket (*has jira*) or if the modified file is a test file (*test files* excludes modifications made to test files). The *file creation* filters changes where new files were created (since they have no history to trace back). The *filetypes* file unit filter filters all changes where the changed files do not have a ".java", ".g" or ".g4" extension (the latter two are ANTLR file extensions). We focused our search of bugs on source files based on the suggestion of professional developers and because of the language-specific nature of the line unit filters. This filter can be easily modified in case we wanted to apply E-SZZ on a different project, written in a different language. The first line filter is a conditional line unit filter (*context*), that is only applied if the hunk (where the line originates from) is not an addition. *Refactor* discards all lines that didn't alter the behaviour of the program. And finally we filter comments (single and multiline), blank lines and lines that import modules.

After passing through these filters, we get the resulting bug-fixing lines that we can trace back to find their respective bug-inducing lines (and their revision).

Bug-inducing commit filters The following are the main filters that I used on potentially bug-inducing commits:

- commit filters
  1. *committed before bug report*
  2. *refactor*
  3. *has jira*

These filters are aimed at reducing false positives produced by our algorithm. The first one filters commits that were created *after* the bug got reported (seeing as commits that are created after the bug report can not be the cause of the bug). Then the *refactor* filter checks whether the traced back line is a refactor-type change. And as the final step we make sure that we are able to link the commit to a JIRA ticket (*has jira*).

name (affects)	**type**	**motivation**
`--diff-filter=MRC` (bug-fix)	`git diff` flag	Excludes (A)dded files from search since they have no history tracing back to potential bugs.
`--ignore-cr-at-eol` (both)	`git diff` flag, `git log` flag	Added carriage-return to end of line can not be the origin, nor the fix of a bug.
`--ignore-space-at-eol` `--ignore-space-change` (both)	`git diff` flag, `git log` flag	Changes in the number of spaces and added spaces to end of line can not be the origin, nor the fix of a bug.
`--ignore-blank-lines` (both)	`git diff` flag, `git log` flag	Blank lines can not be the origin, nor the fix of a bug.
`--no-merges` (bug-inducing)	`git log` flag	Metachanges do not introduce bugs.
testfiles (bug-fix)	file unit filter	Bug-fix changes often include added test cases to eliminate future bugs of similar origin. Since our algorithm traces back additions as well, I wanted to eliminate possible false positives with this filter.
filetypes (bug-fix)	file unit filter	Since the line unit filters are specific to language, we limit the scope of the bug-search to source files.
context (bug-fix)	conditional line unit filter	The context of hunks that are not additions are not the source of the bug.
comments (bug-fix)	line unit filter	Comments do not fix bugs.
blank lines (bug-fix)	line unit filter	`--ignore-blank-lines` only filters blank lines where the code was changed. The *context* of the resulting hunks therefore can still contain blank lines, which need to be filtered.
imports (bug-fix)	line unit filter	On the suggestion of professional developers, we excluded changes in imports from our search.
refactor (both)	line unit filter commit filter	Refactor lines can not be the origin, of a bug. Refactoring change in bug-inducing commit can not be the cause of the bug.
committed before bug report (bug-inducing)	commit filter	Commits coming after the bug report can not be bug-inducing.

Table 1: A complete list of all filtering techniques used in E-SZZ, along with the motivation of usage. Under the name of said filters the affected type of commit is listed (bug-fix/bug-inducing/both). Type indicates the nature of the filter (flag of a used command, or a custom filter).

4 Evaluation and application

4.1 Evaluation

Limitations

Evaluating and comparing different SZZ implementations is a rather challenging task, given that there is no complete baseline dataset for doing so. To create a dataset with labeled revisions of a project, one would need to hire domain experts, preferably developers that are already familiar with the code base. This could include analyzing hundreds, possibly thousands of commits, some of which date back years.

The reason why SZZ is used in so many cases is because of how laborious and time consuming manual labeling can be. But then again, a manually labeled dataset would be required to evaluate these different implementations, which makes us circle back to the original question at hand. **How do we evaluate an SZZ implementation without the ground truth?**

Prior evaluation methods

The evaluation methods of prior work can be grouped into three different approaches. Since all suggested variants are created with the goal of reducing false positives, the first obvious approach would be to observe the decrease in bug-inducing commits compared to different implementations [5, 7]. The second approach involves evaluating the algorithms on a small set of manually labeled commits [14].

Last but not least, there has been one evaluation framework proposed by Costa et al. [13] that was used to evaluate 5 different SZZ implementations. The framework evaluates the following criteria: (1) the *earliest bug appearance*, (2) the *future impact of changes*, and (3) the *realism of bug introduction*. The explanations and used evaluation metrics proposed by them are shown in Table 2.

name	explanation	metric
Earliest Bug Appearance	Developers estimate the version when the bug got introduced with the *affected version* field. If the version of a found bug-inducing commit comes after the estimated version, it counts as a disagreement.	Disagreement Ratio, $\frac{D}{B}$, where D is the number of disagreements and B is the total number of bugs. A lower value is preferred.
Future Impact of a Change	Evaluates the number of bugs that were fixed more than once (e.g. introduced future bugs).	Count of Future Bugs (% of multiple future bugs). A lower value is preferred.
		Timespan of Future Bugs (median of days) A lower value is preferred.
Realism of Bug Introduction	Evaluates the feasibility of the candidate bug-inducing changes (linked to the same bug-fix) by looking at the time gone by between the earliest and latest bug-inducing changes.	Timespan of Bug-introducing Changes (median of days) A lower value is preferred.

Table 2: Evaluation metrics proposed by Costa et al. [13]. The *affected version* field is a non-mandatory field in the JIRA issue tracking system.

Proposed changes to the evaluation framework

The evaluation metrics proposed in the previous paragraph are a great first intuitive step towards analysing and comparing SZZ variants. However, I do find that they do have some limitations. The *earliest bug appearance* relies on the fact that developers document the *affected version* JIRA field accurately. This is aknowledged by the authors as well, but in their evaluation another interesting occurence can be observed. In their comparison of 5 SZZ implementation, the baseline SZZ implementation outperformed the others! They counted very little to none disagreements, compared to the number of bugs detected by the algorithm. Since SZZ is an algorithm that produces the most candidate bug-inducing commits out of all, this certainly plays a role in the disagreement ratio being so low.

Found Bug Inducing Commits To counter the shortcomings, I propose a different evaluation metric called *found bug inducing commits*. Sometimes when the exact origin of a bug is known, JIRA offers an opportunity to link the cause to the fix. Even though this is not too common, it offers a perfect way for evaluating how many bug inducing commits we managed to locate correctly on a small, labeled dataset curated by the developers themselves. In HIVE, we found 174 such commits. We can quantify this feature with the *agreement ratio* and *average linked bugs* metrics. The ***agreement ratio*** is the number of

correctly identified bugs divided by the number of labeled bugs, while ***average linked bugs*** gives us an idea how many false positives the algorithm produces in the process (average of the total number of bug-inducing commits belonging to the same bug-fixing commit).

Bug-fix Coverage Ultimately, I would like to suggest one last metric: *bug-fix coverage*. It looks at how many bug-fix commits were linked to a potential bug-inducing commit. The metric used to describe this feature is ***ratio of linked bugs***, which is the number of bug-fix commits that have a bug-inducing commit linked to them divided by all the bug-fix commits.

name (affects)	**type**	**motivation**
`--diff-filter=MRC` (bug-fix)	`git diff` flag	(A)dded files have no history tracing back to potential bugs.
*test files** (bug-fix)	file unit filter	This filter was not removed due to comparability reasons.
*filetypes** (bug-fix)	file unit filter	This filter was not removed due to comparability reasons.
additions (bug-fix)	hunk filter	Additions have no history tracing back to potential bugs.
committed before bug report (bug-inducing)	commit filter	Commits coming after the bug report can not be bug-inducing.

Table 3: A complete list of all filtering techniques used in SZZ, along with the motivation of usage. Under the name of said filters the affected type of commit is listed (bug-fix/bug-inducing/both). Type indicates the nature of the filter (flag of a used command, or a custom filter). The filters marked with a * are not part of the original algorithm.

A comparative analysis with the original SZZ algorithm

By combining the metrics proposed by Costa et al. [13] (see Table 2) and my own metrics, I have evaluated and compared my proposed version of SZZ and the base implementation. As previously pointed out in Table 1, E-SZZ filters files using the *test files* and *filetypes* filters. This focused our search for bug-inducing commits to the source files that were of interest to us, and made sense for our application. However to make the comparison fair, I altered the original algorithm in such a way that it traced back changes from the same set of files as E-SZZ. A complete list of filters used on SZZ (similar to Table 1) can be seen in Table 3.

Given that the changes proposed by past work are aimed at reducing the number of false positives, whereas the handling of additions possibly increases them, I decided to iteratively evaluate my proposed algorithm. First I evaluate a variant of E-SZZ that does not handle additions to showcase the benefits of the filters, then I proceed onto the benefits of E-SZZ. Results of the evaluations can be seen in Table 4.

At first glance, one can see the obvious benefits of combining all filtering techiques that were so far proposed in the literature. The *disagreement ratio*, *count of future bugs* and *timespan of future bugs* are the lowest in this implementation out of the three. The longest *timespan of bug-introducing changes*

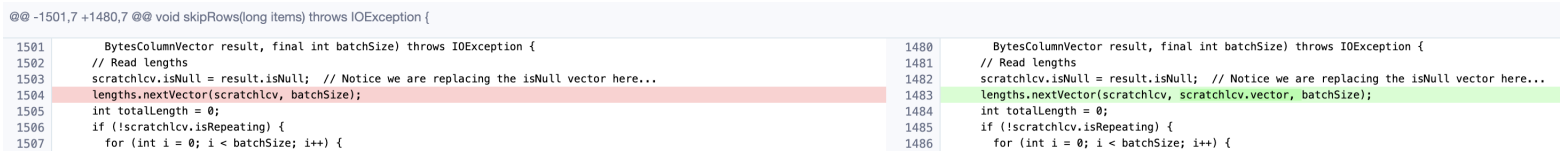
value can be attributed to the refactor-filtering feature - the further the algorithm goes back to find the potential bug-inducing commit, the farther apart will the first and last proposed bug-inducing commit be. However, based on my metrics we can see a decrease in the *agreement ratio* compared to the base implementation - which means that even though many false positives are filtered out, some true positives are lost in the process.

In conclusion, while on one hand the naive approach of E-SZZ introduced slightly more false positives than the base implementation, it also managed to find more of the known bugs (higher *agreement ratio*). On manual inspection of the commits that E-SZZ managed to correctly label I found a commit that beautifully demonstrates the usefulness of tracing back the context of additions - the example is presented in Figure 8. To top it off, E-SZZ also manages to increase the bug-fix commit coverage (it links more bug-reports to bugs). In total, SZZ flagged 4667, while E-SZZ flagged 5282 commits as bug-inducing.

		SZZ	**E-SZZ** (w/o additions)	**E-SZZ**
Costa et al.	Disagreement Ratio	20.25%	**19.08%**	23.82%
	Count of Future Bugs	57.01%	**53.86%**	60.13%
	Timespan of Future Bugs (median)	837.39	**788.08**	829.69
	Timespan of Bug-introducing Changes (median)	**1834**	1926	1911
Proposed metrics	Agreement Ratio	40.23%	37.36%	**42.53%**
	Average Linked Bugs	**2.17**	2.29	2.79
	Ratio of Linked Bugs	67.08%	62.67%	**74.16%**

Table 4: SZZ and two variants of E-SZZ evaluated on metrics proposed by Costa et al. [13] and me. The middle column contains the results of an E-SZZ implementation that does not handle additions. Agreement ratio was calculated based on 174 labeled commits.

(a) Bug-fix commit: HIVE-14483



(b) Bug-inducing commit: HIVE-12159

Figure 8: An example from the 174 labeled commits (Section 4.1: Found Bug Inducing Commits) where E-SZZ found the bug, but SZZ did not, because the bug-fix commit included only the addition of a single line.

4.2 Application

The purpose of this section is to demonstrate on simple code change classifiers that even though E-SZZ introduces more false positives than the original algorithm does, it might be more advantageous to catch as many suspicious commits as possible, regardless of the higher number of false positives. Whether our goal is to build a classifier that has higher precision or higher recall, it undeniably depends on the circumstance and use-case what one might consider advantageous or not. In case of code change classifiers, the primary goal would be to catch as many bugs as possible, because undetected bugs (False Negatives) cost more to companies than some extra time of the developer spent on re-checking commits that were labeled as bug-inducing (False Positives). First I start by introducing what features and machine learning model I used to build my classifier, then I will proceed onto the manual analysis of a commit that was marked as "bug-inducing" by the classifier trained on E-SZZ data, but marked "non bug-inducing" by the classifier trained on SZZ data.

Building a code change classifier

In this section, I want to show an application of our algorithm. I took the labeled commits produced by E-SZZ and SZZ, and used them as an input to train a basic code change labeling algorithm. I extracted 16 features from the available metadata provided by JIRA and git. 2 out of the 16 features are categorical features (*components* and *affects version*), and it has some complexity-related features (like *number of insertions*) and classic metadata features (like *day of week*). A complete list of features can be seen in Table 5. My model of choice for building a classifier was a `RandomForestClassifier`, with `n_estimators` set to 200. I built two classifiers: one that was trained on data labeled by E-SZZ, the other that was trained on data labeled by SZZ. 80% of the commits were used to train the models, while the remaining 20% were used to evaluate their predictions. Both models achieve identical precision scores (72.6%), but the model trained on E-SZZ generated labels has a sligtly higher recall (62.9% instead of 59.3%). Both models achieve almost identical accuracies - the SZZ variant achieves 77.8% while E-SZZ achieves 78.6%.

feature name	explanation
number of insertions	Number of added lines.
number of deletions	Number of deleted lines.
num. of changed files	Number of changed files.
day of week	Day of week it was committed.
hour of commit	Hour of commit time.
solve time	Time of JIRA creation - commit time.
resolution time	Time of JIRA creation - JIRA resolution time.
solve res. diff.	Solve time - resolution time.
number of comments	Number of commits on JIRA ticket.
number of patches	Number of JIRA patches. Patches are created using `git diff`.
patch size mean *patch size variance* *patch size rel. variance*	Different patch size related metrics.
filepath contains test	1 if any of the filepaths contain 'test' as a string. Otherwise 0.
components	The components that the change affects.
affects version	Affected version described by the ticket.

Table 5: Features used in code change classification.

Figure 9: HIVE-2676: An example commit where the code change classifier trained on E-SZZ generated labels predicts that this commit is bug-inducing, while the classifier trained on SZZ generated labels does not.

Manual analysis

In total, there were 949 disagreements on the prediction of test samples out of 2672. 560 E-SZZ based predictions marked a commit as bug-inducing (where SZZ based predictions did not), and 389 predictions were marked as bug-inducing by the SZZ based approach, where the E-SZZ based approach did not. I would like to show a typical example where the classifier trained on E-SZZ labels deemed a commit risky while the other did not.

The commit I would like to present is the modification of the `progress()` method in `HadoopJobExecHelper.java` (see HIVE-2676 in Figure 9). While it might not seem risky at first glance, the justification for the predicted risk is quite obvious on further inquiry. After searching HIVE's issue tracking system with the keywords "HadoopJobExecHelper" and "progress", and narrowed the search to bug reports, I immediately got 14 different bug reports related to the same `progress()` method modified by this commit! While on one hand the commit itself might not introduce any bugs, it still deserves further attention than other commits based on the number of related bugs found.

Yet another reason for commits such as these deserving more attention is related to the type of change it represents. We conducted a survey amongst Cloudera developers in Budapest, and asked them to rate specific changes on a scale 1-5, in terms of likelihood of introducing bugs. Based on 38 answers, the *Modification of existing class / interface / method behaviour* category got the highest average (4.08), as well as lowest variance (0.9968) score! This means that the developers highly agree that modifying the behaviour of existing methods/interfaces is extremely risky.

4.3 Future work

My proposal of handling additions is an important first step to improve SZZ-based code change labeling algorithms. Nevertheless it still is a slightly naive approach to assume that every addition's surrounding lines could be accused of breaking the code. To reduce the number of introduced false positives, one could consider filtering the context, based on the type of the addition we are examining. For example, the surrounding lines of a newly added class or test case most likely do not contain a bug. Notwithstanding, the surroundings of added control statements like `if` and `try-catch` blocks are quite likely the source of a bug! There are many promising researches dedicated to understanding bug-fix patterns [22, 23, 24], that could be used to narrow down bug-fix changes to the lines truly related to the bug.

There is one more approach that could effectively reduce the number of false positives - limiting the number of proposed bug-inducing changes to the one that most likely caused the bug. Some naive approaches have already been proposed - like selecting the most recent bug-inducing revision as proposed by Sinha et al. [4], or by selecting the longest change as proposed by Davies et al. [14]. But these approaches could be further improved by applying the idea of IR-based (Information Retrieval based) bug localization to our problem. The core idea behind IR-based bug localization is to find the source file that needs to be fixed based on the description given in the bug report. This can be done by calculating the textual similarity scores between the candidate source files and the given bug-report. The algorithm marks source files as the probable source of a bug

that are most similar to the bug report. IR-based file-level bug localization is backed by extensive research [25, 26, 27]. Nonetheless, more recent research has also proposed to localize bugs on a change level [28], which is of greater interest to us. By combining the finite set of candidate bug-inducing changes proposed by E-SZZ and the methods of IR-based change-level bug localization, we could effectively find the true source of our bug.

4.4 Conclusion

My implementation of the E-SZZ is the first stepping stone to improving the original SZZ algorithm, by applying filters already proposed by the literature. Furthermore, I am the first in the literature to include an extensive evaluation on the effect of handling additions. By implementing a naive approach (tracing back the context of additions), we get an implementation that finds more bugs (77 out of 174 labeled bugs, compared to 70 out of 174) than the original implementation. Nonetheless, a weakness of this implementation is that it also introduces many false positives in the process. Future research could focus on reducing introduced false positives due to handled additions, by identifying which parts of a bug-fix change are actually responsible for fixing a bug, or by selecting the most likely candidate from the proposed bug-inducing changes.

References

[1] Caitlin Sadowski et al. “Modern code review: a case study at google”. In: *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP ’18*. Gothenburg, Sweden: ACM Press, 2018, pp. 181–190.

[2] Amiangshu Bosu and Jeffrey C. Carver. “Impact of Peer Code Review on Peer Impression Formation: A Survey”. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. Baltimore, Maryland: IEEE, Oct. 2013, pp. 133–142.

[3] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. “When Do Changes Induce Fixes?” In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. MSR ’05. St. Louis, Missouri: ACM, 2005, pp. 1–5.

[4] Vibha Singhal Sinha, Saurabh Sinha, and Swathi Rao. “BUGINNINGS: Identifying the Origins of a Bug”. In: *Proceedings of the 3rd India Software Engineering Conference*. ISEC ’10. event-place: Mysore, India. New York, NY, USA: ACM, 2010, pp. 3–12.

[5] S. Kim et al. “Automatic Identification of Bug-Introducing Changes”. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*. Sept. 2006, pp. 81–90.

[6] Chadd Williams and Jaime Spacco. “SZZ Revisited: Verifying when Changes Induce Fixes”. In: *Proceedings of the 2008 Workshop on Defects in Large Software Systems*. DEFECTS ’08. event-place: Seattle, Washington. New York, NY, USA: ACM, 2008, pp. 32–36.

[7] E. C. Neto, D. A. da Costa, and U. Kulesza. “The impact of refactoring changes on the SZZ algorithm: An empirical study”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pp. 380–390.

[8] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. “Learning from Bug-introducing Changes to Prevent Fault Prone Code”. In: *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*. IWPSE ’07. event-place: Dubrovnik, Croatia. New York, NY, USA: ACM, 2007, pp. 19–26.

[9] Sunghun Kim, E James Whitehead, and Yi Zhang. “Classifying Software Changes: Clean or Buggy?” In: *Software Engineering, IEEE Transactions on* 34 (Apr. 2008), pp. 181–196.

[10] Y. Kamei et al. “A large-scale empirical study of just-in-time quality assurance”. In: *IEEE Transactions on Software Engineering* 39.6 (June 2013), pp. 757–773.

[11] Christoffer Rosen, Ben Grawi, and Emad Shihab. “Commit Guru: Analytics and Risk Prediction of Software Commits”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. event-place: Bergamo, Italy. New York, NY, USA: ACM, 2015, pp. 966–969.

[12] M. Nayrolles and A. Hamou-Lhadj. "CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects". In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. May 2018, pp. 153–164.

[13] D. A. da Costa et al. "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes". In: *IEEE Transactions on Software Engineering* 43.7 (July 2017), pp. 641–657.

[14] Steven Davies, Marc Roper, and Murray Wood. "Comparing text-based and dependence-based approaches for determining the origins of bugs". In: *Journal of Software: Evolution and Process* 26.1 (2014), pp. 107–139.

[15] Danilo Silva and Marco Tulio Valente. "RefDiff: Detecting Refactorings in Version Histories". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). Buenos Aires, Argentina: IEEE, May 2017, pp. 269–279.

[16] Khaled El Emam, Walcelio Melo, and Javam C. Machado. "The Prediction of Faulty Classes Using Object-oriented Design Metrics". In: *J. Syst. Softw.* 56.1 (Feb. 2001), pp. 63–75.

[17] N. Ohlsson and H. Alberg. "Predicting fault-prone software modules in telephone switches". In: *IEEE Transactions on Software Engineering* 22.12 (Dec. 1996), pp. 886–894.

[18] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. "Mining Metrics to Predict Component Failures". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. event-place: Shanghai, China. New York, NY, USA: ACM, 2006, pp. 452–461.

[19] T. Zimmermann and N. Nagappan. "Predicting defects using network analysis on dependency graphs". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. May 2008, pp. 531–540.

[20] A. E. Hassan. "Predicting faults using the complexity of code changes". In: *2009 IEEE 31st International Conference on Software Engineering*. May 2009, pp. 78–88.

[21] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Aug. 2002, p. 256.

[22] Kai Pan, Sunghun Kim, and E. James Whitehead. "Toward an understanding of bug fix patterns". In: *Empirical Software Engineering* 14.3 (June 1, 2009), pp. 286–315.

[23] Matias Martinez, Laurence Duchien, and Martin Monperrus. "Automatically Extracting Instances of Code Change Patterns with AST Analysis". In: *2013 IEEE International Conference on Software Maintenance*. 2013 IEEE International Conference on Software Maintenance (ICSM). Eindhoven, Netherlands: IEEE, Sept. 2013, pp. 388–391.

[24] Yungbum Jung, Hakjoo Oh, and Kwangkeun Yi. "Identifying Static Analysis Techniques for Finding Non-fix Hunks in Fix Revisions". In: *Proceedings of the ACM First International Workshop on Data-intensive Software Management and Mining*. DSMM '09. event-place: Hong Kong, China. New York, NY, USA: ACM, 2009, pp. 13–18.

[25] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation". In: *2008 15th Working Conference on Reverse Engineering*. 2008 15th Working Conference on Reverse Engineering. Oct. 2008, pp. 155–164.

[26] Shivani Rao and Avinash Kak. "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. event-place: Waikiki, Honolulu, HI, USA. New York, NY, USA: ACM, 2011, pp. 43–52.

[27] Jian Zhou, Hongyu Zhang, and David Lo. "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE). June 2012, pp. 14–24.

[28] M. Wen, R. Wu, and S. Cheung. "Locus: Locating bugs from software changes". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Sept. 2016, pp. 262–273.