MASTER'S THESIS | LUND UNIVERSITY 2018

SZZ Unleashed: Bug Prediction on the Jenkins Core Repository

Kristian Berg, Oscar Svensson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-04

SZZ Unleashed: Bug Prediction on the Jenkins Core Repository

(Open Source
Implementations of Bug Prediction Tools on Commit Level)

Kristian Berg
ine13kbe@student.lu.se

Oscar Svensson
elt11osv@student.lu.se

June 26, 2018

Master's thesis work carried out at Axis Communications AB.

Supervisors: Sven Selberg, sven.selberg@axis.com
Markus Borg, markus.borg@ri.se

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Bug prediction is a research area of great interest to software development. Done at the level of commits it has the potential to improve the efficiency of code reviews. However, the progress of adapting bug prediction in practice and furthering research is hampered by lack of available open source implementations. In this study we have provided several such implementations. These include the SZZ algorithm, Online Change Classification and feature mining scripts. We applied our implementations to the Jenkins core repository and trained a machine learning model to identify bug introducing commits in the resulting dataset. The model achieved an F1-score of 15.4% with stratified k-fold cross validation and 12.7% with Online Change Classification, where the proportion of bug introducing commits was 3.6%.

Keywords: MSc, Report, Git, Jenkins, Machine Learning, Defect Prediction, Mining Software Repositories, SZZ, Online Change Classification

Acknowledgements

We would like to thank our supervisors Markus and Sven, as well as our examiner Emelie for helping us out and providing valuable feedback.

Contents

Chapter 1
Introduction

Code reviewing is a widely established practice among software companies where code commits written by developers are put through a peer-reviewing process. One of the main purposes of the reviews is to identify bugs at an early stage, specifically bugs which are not caught during automated testing. An overview of the process is illustrated in Figure 1.1.

A tool which helps reviewers prioritize what commits to focus their efforts on could help improve the efficiency of code reviews. Specifically, it would be preferable to focus reviewing efforts on commits that are more likely to introduce bugs. This leads to a classification problem, namely the classification of code as either clean or buggy. This area of research is referred to as either bug prediction or defect prediction. In this thesis, the unit of code that we are examining is the commit, which is a collection of changes to one or more files. In this thesis we attempt to build a dataset of labeled commits and train a machine learning model to predict instances of bug introducing commits.

In order to build a prediction model one needs data to train it. This data can be collected by using the so called SZZ algorithm. The algorithm builds a dataset of bug introducing commits from the issues of a software repository's bug tracking system, or BTS. This dataset can then be used as ground truth to train a machine learning algorithm, or model, to identify such commits.

In addition to implementing the SZZ algorithm, we have applied the algorithm to the Jenkins core repository [4] and trained a model on the resulting data. In order to assess this model realistically, we implemented Tan et al.'s [30] Online Change Classification, which is a time-sensitive method of splitting data into training and test sets. Finally, we also implemented the necessary scripts for extracting features from each commit in the dataset.

Our machine learning model achieved an F1-score of 15.4% for k-fold stratified cross validation and 12.7% for Online Change Classification. While this is insufficient to be usable in a real life setting, we believe that there is room for improvement.

To summarize, these are the contributions made in this paper:

- A complete open source implementation of the original SZZ algorithm, along with the improvements described by Williams and Spacco [33] and related code infrastructure.
- An open source implementation of a collection of scripts for extracting machine learning features relevant for a software repository.

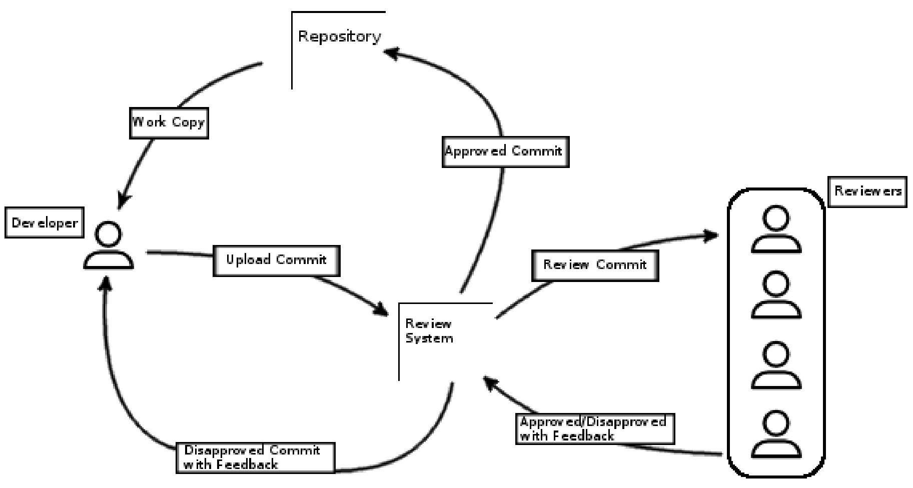


Figure 1.1: This is the process when a development team is using a reviewing system to correct commits before it is finally pushed to a software repository. The review system lets others review a change that was made by a developer. Until it is approved, the developer will push commits to the reviewing system.

- An open source implementation of splitting software repository data into training and test sets based on Tan et al.'s Online Change Classification [30]
- An evaluation of training a machine learning model on data generated from the Jenkins core repository.

We hope that by releasing our code as open source we will make the future work of researchers easier and more reliable. Most current research on the field has either been using private owned implementations of the SZZ or has manually performed the steps of the algorithm. We have also found some open source implementations of the SZZ algorithm, see section 3.5. But they are not complete implementations and do not cover the algorithm completely. Therefore, there is a need for an implementation of the SZZ algorithm for researchers to keep datasets up to date. The code used in research would also be more robust if different people contributed to it.

This study was initially designed to lay the foundation for an integrated bug prediction tool to be used by developers at Axis. As we did not get that far, the task of building a complete tool using our implementations is left as future work.

The paper is structured as follows: In the Background chapter we describe relevant theoretical concepts and give a technical background of our choice of infrastructure. In the Related Work chapter we describe a few studies that either are similar to ours or touch on concepts we considered including. Next in the Approach chapter, we describe our approach to solve our initial problem. This is followed by an Evaluation chapter in which results of our findings and contributions can be found. Lastly, we sum up the report in a conclusion chapter were we summarize our results and lay out suggestions for future work.

Chapter 2
Background

In this chapter we will go through relevant technical concepts for understanding this paper. We will also briefly explain our initial aims in relation to our final contributions.

Section 2.1 describes the process that led us to our final contributions, and how our scope changed during the course of our work. Section 2.2 describes the basics of how the SZZ algorithm works and its history in research. Section 2.3 describes the basics of machine learning. Section 2.4 describes the various features that are mined for each commit. Section 2.7 describes sampling which is important when applying machine learning on imbalanced datasets. Section 2.6 describes the validation techniques k-fold cross validation and Online Change Classification. Section 2.7 covers the basics of the metrics used to score the performance of machine learning models.

2.1 Initial Aims and Final Contributions

This work initially aimed to produce a tool which could support Axis reviewers in the reviewing process, and to try different methods of making reviewers likely to adapt the tool into their workflow. However, to produce this tool several key components were required. Since the tool was going to be based on machine learning we needed a dataset, specifically a dataset of labeled commits, for training purposes. Such a dataset did not exist and we had to produce it ourselves, which turned out to take a lot of effort; Enough so that it changed the course of our work.

In the end what we have managed to produce is code infrastructure for: generating relevant datasets, training a machine learning model on the datasets, and validating the trained models. The task of integrating the model recommendations with the review software, in the case of Axis this is Gerrit, is left as future work. We have written about a few considerations that ought to be taken into account when eventually undertaking this venture, see the latter half of Section 5.2.

2.2 The SZZ algorithm

The purpose of the SZZ algorithm is to identify bug introducing commits in a software repository. It was introduced by Śliwerski et al. [28], and was later given its name after the initials of the authors. This algorithm can be used to generate a ground truth dataset for a machine learning model. This is done in two parts.

For the first part, issues in the bug tracking system, or BTS, are linked to bug fixing commits. This is done by using regular expressions to search commit messages for references to issues in the BTS. If there is no BTS or if it is poorly maintained, then commit messages that contain the word ”fix” or similar terms are considered to be bug fixes. From these bug fixing commits, the lines that were changed are extracted.

For the second part, which can be seen in Figure 2.1, SZZ uses the lines of code that were changed by a bug fix to trace down all commits which previously made changes to the same lines as the bug fix. It does this by using the blame functionality provided by Git. The blame functionality in Git is a way to track down which commit that last changed a specific line of code. It is the same as asking the question ”Who is to blame for this line of code?”. By blaming each line or each changed line in a file for a bug fixing commit, it is possible to find commits that could potentially be responsible for a bug.

The resulting commits of that blame are compared to the date that the bug was reported. If the commit was committed before this date, then it is labeled a bug introducing commit. But if a commit was committed after the reported date, when the bug was already known to exist in the code, the commit is labeled bug introducing only if it is a partial fix or considered to be responsible for another bug. A partial fix in this context is a fix that did not resolve the bug it intended to resolve. The assumption that it did not fix the bug is drawn from the fact that there exists a later fix for the same issue.

A commit can be responsible for a bug other than the one that blamed it. This means that another bug fixing commit can have its bug origin in this commit because they have both made changes to the same file. In Figure 4.2, it is possible to study a scenario like this. The image shows a number of versions of a single file, where each version is created by a commit. If version 5 was made by a bug fixing commit, it will blame the commit responsible for version 3 and version 1 as its bug origin. Let us say version 3 was created after the bug was reported and the responsible commit is not a bug fix commit. Then the first and second criteria are not fulfilled. However, that commit can still be responsible for another bug. If the commit creating version 23 is a bug fix commit as well, then the commit that created version 3 can be a potential candidate for this bug. It has made changes to the same lines of code as the commit responsible for version 23 has made. This means that the third criteria is fulfilled and the commit responsible for version 3 is to be considered a bug introducing commit.

To improve upon the first version of the SZZ algorithm, annotation graphs and ways of filtering out cosmetic changes were proposed by Kim et al. [15]. The annotation graph, which is illustrated in Figure 2.2, is a more sophisticated method of tracing bug introducing commits from bug fixes.

The annotation graph maps one revision of the software to another by blaming a commit using the git blame functionality. A revision is a version of the software, and a new revision is obtained each time a commit is made. In Figure 2.2, the first graph indicates how a file has been modified between three revisions. In the second graph, the first four lines are unmodified between the three revisions. However, the fifth line in revision 2 is not present in revision 3 which indicates that it is removed. Therefore, the fifth line in revision 3 is

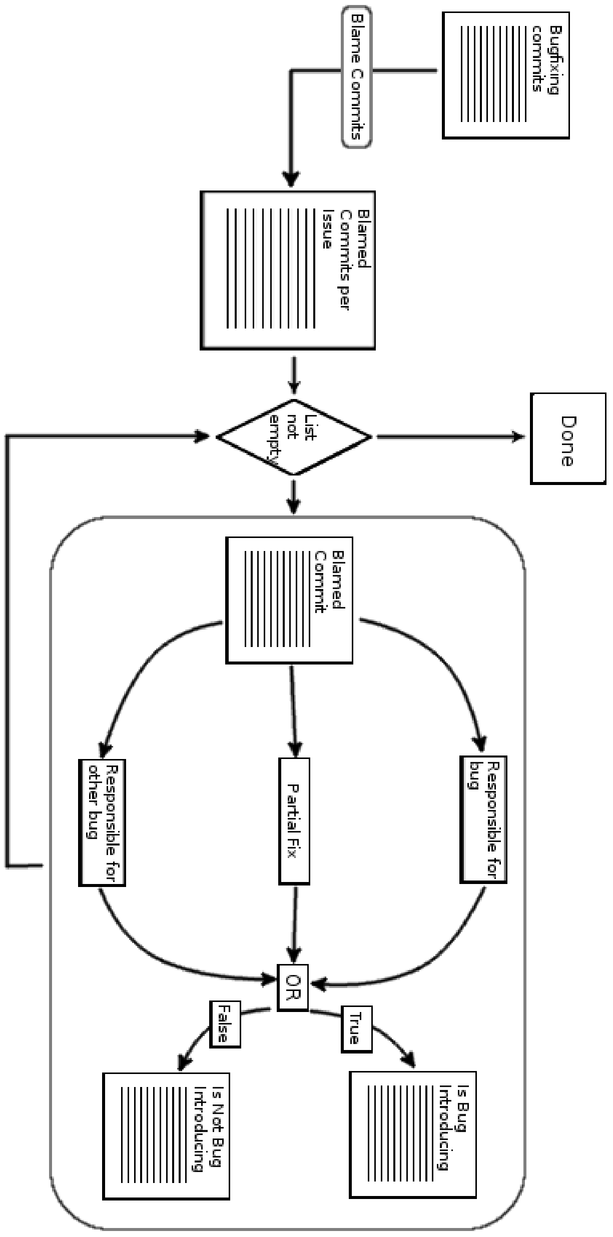


Figure 2.1: The flow shows how the SZZ algorithm generates a list of bug introducing commits from a list of bug fixing commits. It begins with blaming each changed line of code in each bug fixing commit. The result is a list for each bug fixing commits containing their respective blamed commits. These lists are then iterated and each blamed commit is examined with regard to the originating issue against three criteria. It is first checked if it is responsible for that issue by checking the time of when it was committed. If its not, the commit message is checked if it is a fix. Would it not be a fix either, then it is checked if it can be responsible for another bug. That means checking whether a later bug fixing commit has made changes to the same lines of code.

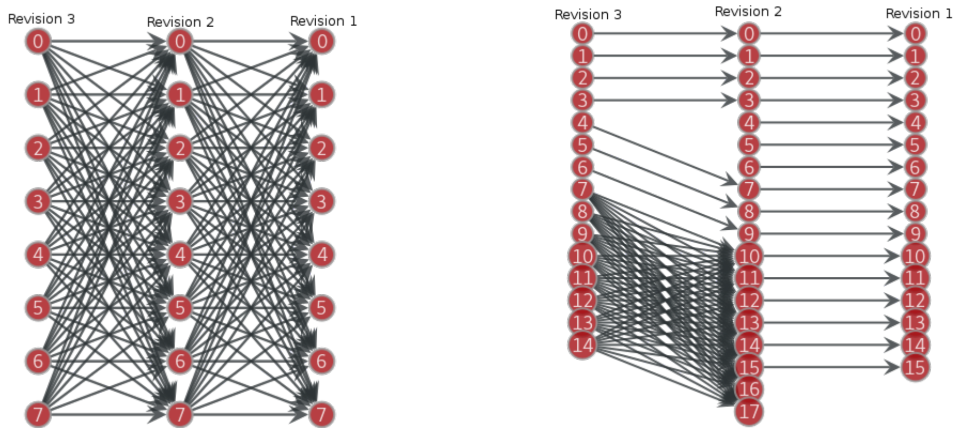


Figure 2.2: Visualization of the annotation graphs. A node represents a line of code and the edge leads to where it is represented in another revision. If multiple adjacent lines are changed, then the annotation graph technique fails to map those lines between revisions. This can be seen in the second graph at node 7 to 14 from revision 3 to revision 2. This issue is solved by line number mapping.

instead mapped to the eighth in revision 2. As for line 16 and 17 in revision 2, they are added since they are not mapped to any line in revision 1. The rest in revision 3 is modified and therefore mapped to the rest of lines in revision 2. These changes are revealed by comparing the revisions and then extracting the changes. By mapping these revisions, it becomes easier to actually distinguish changed lines and to filter cosmetic changes from logical changes. In the previous setup such filtration are harder to distinguish because of the lack of comparing several changes over time.

Williams and Spacco [33] further improved the algorithm by replacing the annotation graphs with line number mappings. In the annotation graph, each modified line in a group of adjacent lines can represent any of those line in the next corresponding revision. With a line number graph, each line is mapped to exactly one line in the next corresponding revision. This enables the algorithm to track a line to its origin. It also, in addition to sorting out cosmetic changes, enables sorting out changes that are semantic but have no impact on the outcome of the program. Semantic changes are language specific and need tools that are specialized for one language.

2.3 Machine Learning

Classification, a subcategory of machine learning, is a concept where a statistical model is fed with data and used to predict a future data point. In this report, the data point will be a commit and the model will predict if it is bug introducing or not. To be able to make an abstraction of a commit and thus make it possible for the model to predict on, a set of features is extracted. These features are measurable properties of the data point. Specific features will be discussed in Section 2.4.

To know whether a prediction is correct or not, it is evaluated against a ground truth. The ground truth is a dataset containing data points which are labeled either positively or negatively, which in our case represents whether or not they are bug introducing. This dataset can either be generated or manually constructed.

A common model which can be used for machine learning is decision trees. Decision trees can be used to classify data using multiple features. This technique uses a tree structure to decide whether or not a data point belongs to a certain class. Each split is the result of a decision that a data point either is something or something else. It could be the decision that a commit has changed above 10 files or not. With a deep tree, such decisions could be many and the model could learn many patterns to make the decision more fine grained. However, with a tree that is too deep the risk of overfitting is high. A solution to this problem is the Random Forest model. It uses multiple trees to predict and calculates the common output from them. This report will use the Random Forest model as it its classification model.

2.4 Features

Several different sets of machine learning features have historically been used for defect prediction. A feature indicates an individual property of a data point, which in our case is a commit. Features that have historically been explored are for example code-churns, file metrics, process metrics, code complexity metrics, Bag-of-words, Characteristic Vectors, file coupling and more. In this report we have decided to use a selection of these features. The features used in this report can be found in Table 4.2.

Nagappan and Ball [18] studied the use of absolute and relative code churn features for predicting software defect density at a component level. They concluded that absolute code churn measures are poor predictors of defect density and that relative code churn measures are superior. Absolute code churns denotes classic code churns where variables are checked as an absolute measurement, such as the number of added lines in a commit. The corresponding relative code churn is then the number of added lines over the total number of files in a commit. So overall, the relative code churns are measurements of how much of a file or files have changed in a commit. Nagappan and Ball's study only focuses on one snapshot of a software project and does not take the project history into consideration. However, the code churns can be extracted over history. It means that for each commit all code churns are extracted individually and can be used as features in a machine learning model. The approach in this study are using the relative code churns as features.

Logical coupling is a measure of the degree to which two files are related [10]. If the two files are frequently changed together, then they are highly coupled and vice versa. The important thing to note is that two files can be highly (or lowly) coupled independently of any semantic or dependency based connection between the two.

D'Ambros et al. [7] studied the relationship between logical coupling and software defects. They studied the hypothesis that bugs are likely to be introduced when developers neglect to change logically coupled files and, for example, only change one of them. Logically coupled files are files which are often changed together but do not have a direct coupling, like an import in a source code file. D'Ambros et al. noted that the correlation between defects and logical coupling was stronger than for the complexity metrics they examined. However, they found the correlation to be weaker than that of total number of changes to defects.

An additional source of potential defects in software is the experience, or inexperience, of the involved developers. Kamei et al. [13] tried using developer experience for defect prediction with three features. The first one was the overall experience of a developer, the second was how recent the developer experience was and the third was how experienced the developer was with the system. An important discovery they made was that more experience did not necessarily go hand in hand with a decreased rate of bug introductions. On the contrary, if a developer had more experience with a system they were more likely to work with more complex parts which in turn often lead to introducing bugs.

2.5 Sampling

Defect prediction is made harder by the fact that relevant datasets are, as a rule, imbalanced. This is a problem shared with other fields of applied machine learning such as fraud detection and breast cancer screening. For many machine learning algorithms, this creates a bias towards the majority class, even though it is the minority class that one is normally interested in [22]. A model can achieve high accuracy by simply always predicting instances to belong to the majority class, which makes for a useless model.

Sampling can be used to alleviate this problem either by oversampling the minority class or by undersampling the majority class, to make the dataset more balanced. Oversampling means generating new instances of the majority class, while undersampling means cutting down the size of the majority class.

One popular oversampling technique is SMOTE, which stands for Synthetic Minority Oversampling Technique, and was introduced by Chawla et al. [3]. The technique has been successfully applied to bug prediction in previous studies [21, 30].

2.6 Machine Learning Model Validation

When evaluating the performance of a machine learning model, a common method is to use k-fold stratified cross validation. This method splits a dataset into k different partitions where, for each iteration, one of these partitions is used as test data and the rest is used for training. The scores of the model for each iteration are averaged together to produce the final result. The 'stratified' part of k-fold stratified cross validation means that the proportion of positive to negative samples are preserved in each split. One of the benefits of k-fold stratified cross validation, compared to splitting the data just once, is that all of the data can be utilized for training.

Tan et al. [30] studied bug prediction and had a different take on the subject compared to previous work. Specifically they criticized the use of k-fold cross validation. It was also critized by Jonsson et al. in the related field of bug assignment [12]. Tan et al. argued that it produces false high precision scores due to using future data predict on past data, as well as labeling samples incorrectly with respect to the time of prediction. To tackle this issue they used a time sensitive approach, which they dubbed Online Change Classification, for splitting data into training and test sets. Time sensitive in this context means that all the data points in the training set are chronologically older than those in the test set.

The way Online Change Classification works, which is illustrated in Figure 2.3, is that it allows several parameters to be set in terms of time units. An example would be determining the size/length of a test set in terms of days instead of a specific number of data points. These parameters include: start and end gaps, which exclude commits at the chronological beginning and end of a repository; a gap between the training and test sets; and update duration. The idea behind the start gap is that "the characteristics of a project

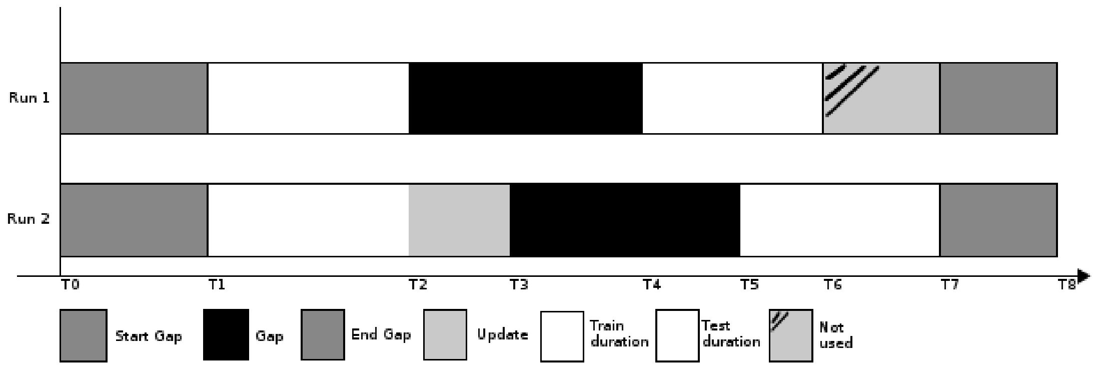


Figure 2.3: An illustration of how the Online Change Classification works and updates between iterations.

may be unstable at the beginning of its history" according to Tan et al. [30], and therefore they should be excluded. The end gap is meant to exclude changes where insufficient time has passed for bugs to be discovered. The gap between training and test sets is supposed to emulate the end gap that would have been in place when predicting at a given point in time, in order to give a realistic estimate of how the model would have performed without the benefit of hindsight. Update time specifies the time used to update the training set of a previous run, and is typically set to be equivalent to the test set duration.

In this report, we have implemented Online Change Classification and compared it to the k-fold stratified cross validation approach. This comparison can be found in section 5.1.

2.7 Prediction Performance Metrics

When making binary classifications there are four possible outcomes: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). True positives are positive samples that are correctly classified as positive. False positives are negative samples that are incorrectly classified as positive. For true and false negatives the same is true but for negative samples.

Accuracy, precision, recall and F1 are ways of measuring model performance, where a higher quotient is better. They are defined as follows:

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

$$F1 = \frac{2*Precision*Recall}{Precision+Recall}$$

The first three metrics can be seen as a way of answering the following questions:

- Precision - If the prediction model predicts a sample to be positive what is the chance that it actually is?
- Recall - Out of all positive samples how many are correctly classified as such?

- Accuracy - How many of the samples are correctly classified as either positive or negative?

Finally, the F1 score is the harmonic mean of precision and recall; it summarizes these two measures into one number with a bias towards the lower of the two values.

When weighing scores over multiple runs of classifications together there are two methods, macro- and micro-averaging. When macro-averaging, the scores are calculated for each metric and each fold, and averaged together afterwards. When micro-averaging, each individual classification for each run is summed together and calculated at once. Below is an example of how precision would be calculated for two runs with each method.

$$Macro-average = \frac{(\frac{TP_1}{TP_1+FP_1} + \frac{TP_2}{TP_2+FP_2})}{2}$$

$$Micro-average = \frac{TP_1+TP_2}{TP_1+FP_1+TP_2+FP_2}$$

Given that each fold is the same size, and that the balance of positive and negative instances in each fold is the same, the two averages will be the same for both precision and recall. However, when applying Online Change Classification neither of these conditions hold. Therefore we have provided micro-averaged results for this validation technique, see Table 5.4.

Chapter 3
Related Work

In this chapter we describe work that is similar to ours or work that we consider interesting as potential future improvements. In section 3.1, we describe a bug prediction study which is closely related to ours in terms of methodology. Section 3.2 describes the so called dependency approach, which is an alternative to the second part of SZZ which links bug fixes to bug introducing commits. In section 3.3 we describe attempts to verify data produced by the SZZ algorithm. In section 3.4 we outline previous attempts to introduce bug prediction in a real life setting and lessons learned therefrom. Finally we describe currently existing SZZ implementations in section 3.5

3.1 Just-in-time Quality Assurance

Several studies have been conducted in the area of automated code reviewing systems. Kamei et al. [13] performed one of the more recent studies which approaches the subject in a similar manner to this paper, called Just-In-Time Quality Assurance. Many of our features are taken from this study and they also predict on the same level as we do, which is to say the commit level.

Kamei et al. used a number of different features for identifying bug introducing changes. These features are based on five categories which are diffusion, size, purpose, history and experience. The diffusion group denotes features that describes how many systems that have been modified in a commit. The size category is made up of features related to how many lines of code a commit has changed. The purpose group contains only a single feature and that is whether or not the commit is stated to be a fix in the commit message. Features in the history group are features that describe properties of earlier commits to the same file(s). That could be the number of authors for a file, the time between when commits made by an involved author and the number of unique changes in files. Lastly, the experience group contains features that measures an author's experience in the analyzed software repository.

Kamei et al. also propose and evaluate a simpler version of the SZZ algorithm called ASZZ which stands for Approximate SZZ. This approach adapts SZZ for use with repositories where mining BTS data is not possible or feasible. Instead of using BTS data, ASZZ simply parses commit messages to identify fixes.

Yang et al. [34] improved the results of this study by using a deep belief network. Using the same datasets and features as Kamei et al. [13] they achieved marginally better predictions with regards to precision, recall and F1-score. They also achieved significant improvements in cost effectiveness, however we have not evaluated this metric in our work. Cost effectiveness in this study was measured by how many of the bugs in a repository were included in the top 20% of changes ranked by risk of bug introduction by the prediction model. In our own work we chose not to use a deep model due to them being hard to interpret in general, see Section 5.2.

3.2 Dependency Graph Analysis

An alternative to the SZZ algorithm, suggested by Sinha et al. [27], uses dependency graphs to link bug fixes with bug introducing commits. In this paper SZZ is referred to as the text based approach since SZZ additionally refers to the process of linking BTS issues to bug fixes. Whereas the SZZ approach compares lines and how they change between iterations, the dependency approach compares dependency graphs to identify bug introducing commits. Importantly, the dependency approach can find bug introducing commits even when the bug fix only adds lines, which SZZ can not.

Davies et al. [8] compared SZZ performance to performance of the dependency graph approach. They provided suggestions regarding usage for both solutions as well as how to combine them. They found that the text based solution tends to find the correct bug origin more frequently, at the cost of more false positives. This is because the SZZ approach will try to find all possible origins while the dependency graph solution can only, by design, point to a single origin. Thus there is a better chance that at least one of the origins identified by the SZZ approach will be the true one.

3.3 Verification of SZZ results

Verifying SZZ results can be difficult and laborious. Verifying the origin of a bug manually takes a lot of time and requires intimate knowledge of the source code. To make evaluating the quality of SZZ results easier, da Costa et al. [6] have suggested a framework for evaluating SZZ results as an alternative to manual verification. The report includes three evaluation criteria which were applied to five different versions of SZZ. Although we have not evaluated our data using this framework we consider it a valuable direction for future work, see chapter 6.

It should also be noted that the previously mentioned Davies et al. [8] study included a manual verification of SZZ results. They manually simulated the SZZ approach for two different repositories and reported a precision score of $29 - 70\%$ and a recall score of $48-70\%$, which suggests that performance is highly variable depending on the examined repository.

3.4 Previous Case Studies

Given our initial aims of producing a working tool for developers at Axis, we have looked into previous case studies to see what lessons can be learned. Overall the main concerns seem to be improving accuracy of predictions as well as providing helpful explanations to accompany said predictions.

Prechelt and Pepper [23] have outlined the efforts to introduce bug prediction at the company Infopark and layed out several potential reasons for why this was hard to do. The most important reasons were that the reliability of the produced dataset was hard to verify, and that prediction results were lackluster in terms of precision and recall.

A paper by Lewis et al. [17] concerns the efforts of Google to introduce bug prediction to their development tools. They used the FixCache and Rahman algorithms, described by Rahman et al. [24], to predict bug-proneness. The Rahman algorithm in this case refers to the naive prediction model that FixCache is compared to by Rahman et al. Lewis et al. stressed the desirability of producing actionable messages related to a buggy classification. Specifically, they say that there ought to be a clear set of steps that a developer could take to make sure the code is no longer flagged as buggy. They concluded that their tool was not good enough to be helpful to developers, but considered the potential for this to change once the technology is improved.

The previously mentioned paper by Tan et al. [30], includes a case study done at Cisco with a deployed tool which failed to be useful to developers. This was blamed on insufficiently accurate predictions as well as insufficiently helpful explanations. Tan et al. recommended looking into better feature engineering as a tool for making predictions more accurate. They also suggested using features derived from the BTS as well as mixing semantical and structure-based features as directions for future work.

3.5 Existing SZZ implementations

There exist a few incomplete implementations of the SZZ algorithm. The first one is the CAS_RepoAnalyzer repository made by Rosen et al. [26]. They use the SZZ algorithm to find bug introducing commits. However, the implementation lacks most of the heuristics that are described by the original authors, Śliwerski et al. [28]. They do check if the potential bug introducing commit has been committed before the issue report date. However, they do not check if the commit is a fix, or if it could be responsible for another bug fix.

The second implementation we have found uses the same heuristics as the CAS_RepoAnalyzer implementation, and is made by João Correia [5]. It does not use a BTS to find bug fixing commits. Since it does not use a BTS, neither does it examine the timestamp of an issue report to see if the commit actually is a potential bug introducing commit.

Chapter 4
Approach

In this chapter we describe the method that was used to produce our results, both regarding the generation of our ground truth dataset as well as machine learning model performance. This includes motivations for various decisions that were made in designing our solution.

Section 4.1 describes how our SZZ implementation works and the motivations for our design choices. It also includes our motivations for picking the Jenkins repository as our target for prediction. Subsection 4.1.1 concerns the process of identifying relevant issues in the BTS and linking these to specific commits in the repository, which is to say bug fixes. Subsection 4.1.2 concerns the process of taking these bug fixes and using them to identify bug introducing commits, including what specific considerations and decisions were made for our implementation.

Section 4.2 describes the steps involved in training a machine learning model on the resulting data. Subsection 4.2.1 details what features were used and why, along with appropriate references to previous work. Subsection 4.2.2 explains how k-fold stratified cross validation and Online Change Classification was used to obtain training and test set splits. Subsection 4.2.3 concerns how sampling was used to improve prediction results. Finally, subsection 4.2.4 details what considerations were made when applying a machine learning model.

4.1 SZZ implementation

Figure 4.1 illustrates the structure of our code. It produces a dataset of labeled commits for the Jenkins core repository, based on issues from its BTS. The flow of information can roughly be summarized as follows:

1. Issues that match our query for resolved bugs are fetched from Jira, which is Jenkins' BTS.
2. The corresponding issue keys are used to link issues to their respective bug fixing commits.
3. The bug fixing commits are used to find their corresponding bug introducing commits. These bug introducing commits are the positively labeled examples in our dataset from

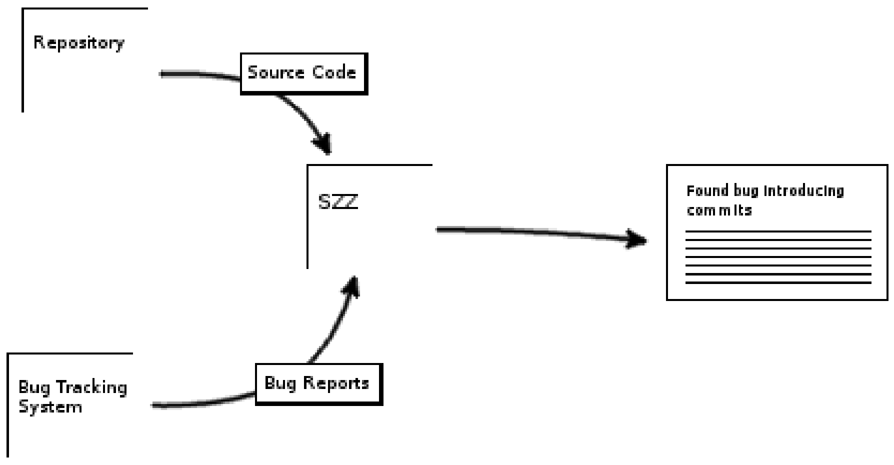


Figure 4.1: The automatic labeling process. SZZ is fed with the bug reports and the repository source code which makes it able to distinguish the bug introducing commits.

which training and test sets are chosen.

The Jenkins core repository was chosen for our research for several reasons. First off we wanted to pick a repository that was relevant to Axis. While Axis developers are not contributing specifically to the core repository, they are contributing to plugins for Jenkins. The second reason was that it had a well maintained BTS. First, bug-related issues were explicitly labeled and are not confused with other issues. Second, there was a standard for bug-fixing commit messages to reference the issue id (for example JENKINS-1234) in the BTS. Third, all available information on issues were easily attainable using the Jira REST API and associated Jira Query Language. The Gerrit repository was also considered but rejected due to there not being a publicly accessible API for fetching issues from its BTS, Monorail.

It should also be noted that while the original SZZ algorithm was designed for use with the CVS version control system, our implementation is adapted for use with Git.

4.1.1 Linking issues to bug fixes

The first step of the process is to identify bug-fixing commits. This is done by linking issues in the BTS of a repository to specific commits. This part of our solution is specific to Jenkins and would have to be adapted to be applicable to a different repository. The JQL (Jira Query Language) query used for fetching relevant issues was formulated as follows:

```
project = JENKINS
    AND issuetype = Bug
    AND status in (Resolved, Closed)
    AND resolution = Fixed
    AND component = core
    AND created <= "2018-02-20 10:34"
    ORDER BY created DESC
```

Breaking down the components: *issuetype* eliminates other types of issues such as features, *status* eliminates issues that are still open, *resolution* eliminates things like duplicate issues, *component* excludes issues concerning other repositories, created enforces the time range restrictions (see Table 4.1) and finally the last line orders issues in reverse chronological order.

The resulting issue ids are used to find bug fixing commits by doing regex searches on the git log. Through manual inspection of the git log, three different formats for referencing issues were found, namely JENKINS-XXX, HUDSON-XXX and #XXX where XXX is the number associated with the issue. The Python code below specifies the regex pattern used to identify bug fixing commits where *key* is a Jira issue id on the form JENKINS-XXX and *nbr* is the associated number XXX. If the #XXX pattern was matched an extra regex search is done to make sure that the commit message also contains the word ”fix”, otherwise it is not considered a bug fix.

```
pattern = key + '\D|' + '#' + nbr + \
                '\D|HUDSON-' + nbr + '\D'
```

The above pattern can generate multiple matches for an issue. We use regex to sort out the real bug fix from these matches and eliminate irrelevant matches, such as merge commits. The Python code below illustrates how this is done:

```
def commit_selector_heuristic(commits):
    for commit in commits:
        if(re.search('[Mm]erge|[Cc]herry|[Nn]oting',
          ↪  commit)):
            continue
        return commit
    return commits[0]
```

This Python code picks the most recent commit that does not match either pattern. The merge and cherry patterns sort out merge and cherry pick commits respectively. The last pattern was identified through manual inspection of the Jenkins repository and eliminates what seems to be a documentation related category of commits.

4.1.2 Linking Bug Fixes to Bug Introducing Commits

The SZZ algorithm is the de facto standard for linking bug fixing commits to bug introducing commits. The original paper from 2005 [28] has been cited more than 600 times. The only other approach that we were able to find is dependence analysis [27] which seemed more difficult to implement since it required many language specific tools and had less supporting research. For comparison the original SZZ paper had been cited 611 times according to Google Scholar compared to 14 times for the dependence analysis paper as of May 21 2018. For these reasons we went with the SZZ algorithm.

Our implementation is based on the 2008 study by Williams and Spacco [33] where line number mappings were used to backtrack through the change history. Since our algorithm is designed to work with multiple languages, cosmetic changes are not filtered. Each changed line, either cosmetic or logical, is tracked to its first initialization or a newer

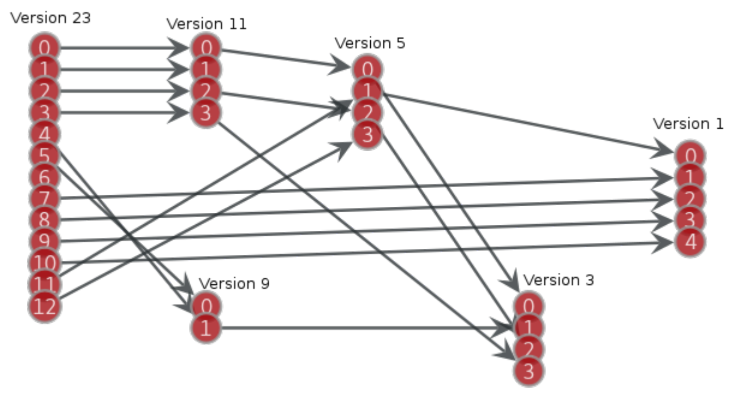


Figure 4.2: A simple line numbering graph. By blaming each line in version 23, versions 1, 5, 9 and 11 will be found. With a greater depth, more versions will be found. For example, if the first four lines in version 23 is blamed and the depth is set to 2, then version 3 will be found since it is blamed for changing line number 4 in version 11.

version at a specified depth. In Figure 4.2, if the algorithm blames each line in a file at version 23 with a depth of one it would find all versions except for version 3. However, with a depth of two or higher the algorithm would find all versions. Using line number mappings it is also possible to trace the exact origin of a row. If one wants to check how a row evolves over each version, like the zeroth row in version 23 in Figure 4.2, it can be traced back to version three as the origin.

We used the following regex to decide if a commit was a partial fix:

```
String pattern = "jenkins[-]?\\d|hudson[-]?"+
                  "\\d|fix(?:es|ed)*|solve(?:d)*";
```

As described in Section 2.2, a partial fix denotes a commit that solves an issue but later on gets solved again by another commit.

For the SZZ algorithm to be able to work directly on a git repository, we have used the JGit library [29] provided by the Eclipse community.[1] Using this library, we reduced the use of text parsing and could work directly on the git revision structure.

To get statistics for the results that the SZZ algorithm produces, we used a script that tries to extract the purpose of the commit. The categories of purposes are: added, updated, fixed, contains an issue number and none of the above. To extract them, we used the following regex:

```
def has_added(message):
    if (re.search(r"add(?:ed)*|implement(?:ed)*|introdu⌋
     ↪  ce(?:d)*|improve(?:ment|ments)*", message.lower())):
        return True
    return False
```

[1]The version used is 4.10.0.201712302008-r.

```
def has_updated(message):
    if (re.search(r"update[d]*|mov(?:ing|e|ed)|refactor|mo⌋
     ↪  difying|switching|deprecate(?:d)*|clean(?:up|ed)*",
     ↪  message.lower())):
        return True
    return False

def has_bugfix(message):
    if (re.search(r"jenkins[-]?\d|hudson[⌋
     ↪  -]?\d|fix(?:es|ed)*|solve(?:d)*", message.lower())):
        return True
    return False

def has_issue(message):
    if (re.search(r"issue\snumber", message.lower())):
        return True
    return False
```

4.2 Machine Learning

These were the steps involved in our process of applying machine learning:

1. Relevant features were extracted for all commits.
2. The data was split into training and test sets by two methods: cross-validation and Online Change Classification (see section 2.6).
3. Over- and undersampling or both were applied to the training set
4. A random forest classifier was trained on the data.

We have used version 0.19.1 of the Python library scikit-learn [20] for machine learning purposes along with its various implementations of machine learning algorithms. Our implementation of Online Change Classification is adapted for use with scikit-learn as well.

Our SZZ implementation and machine learning approach is based on producing predictions for commits, similar to Kamei et al. [13]. A commit in the Git sense is a collection of changes to files. It should be noted that bug prediction can also be done at multiple other levels, for instance file [36] or component [18, 36] level. The advantages of predicting at a commit level are that developers can act immediately on the results of the prediction, with the code and its context fresh in their memory, and that there is less code to examine given the finer granularity. The disadvantages of predicting at a commit level is that finer granularity also makes it harder to make accurate predictions. Predictions can also be made at the level of individual file changes [14, 35] and the same concerns of granularity are applicable here.

When training our machine learning model we used a fixed initial random seed of 675 for each run. This number was obtained by generating a random number in the range of 0 to 999 inclusive. The training and test set splitting, sampling and classification were all based on this random number. The range of dates that were considered is shown in Table 4.1. This was also the range that we used for fetching issues from the BTS, which is reflected in the JQL query used, see Section 4.1.1.

	Hash value	Commit date
First commit	8a0dc230f44e84e5a7f7920cf9a31f09a54999ac	Sun Nov 5 21:16:01 2006 +0000
Last commit	02d6908ada70fcf8012833ddef628bc09c6f8389	Tue Feb 20 10:33:53 2018 +0100

Table 4.1: The first and last commit used in our dataset. The date range also applies to the issues that were fetched from the BTS.

4.2.1 Features

In table 4.2 all tested features are listed. Most of them have been used previously by Kamei et al. [13] and are improvements on previous studies like Nagappan et al. [18]. The first thirteen features was extracted by parsing the Jenkins repository either through the git log or by blaming each file in each change.

As for the last two features, a special tool was used to first mine the couplings between each file. The tool we used to mine the coupling features was code-maat, version 1.1 [31] which is described more thoroughly in the book *Your Code as a Crime Scene* [32]. This is a tool primarily made for analyzing software repositories. Using the ”logical coupling analyze” feature provided by the tool, it is possible to extract the logical coupling for each pair of files in a software revision. The output is then a dataset with each possible pair of files' grade of coupling. With this dataset, we could look at what files were changed in a commit and compare these with the files in the dataset.

The feature for the number of highly coupled files is calculated for each changed file, and measures to what degree they are coupled to other files. Highly coupled files are files that are changed very frequently together in previous commits. In the dataset produced by code-maat, highly coupled files will have a grade of coupling close to 100%. The highly coupled feature is then how many files have a coupling degree of 100% in a commit. Next we also have the number of non modified coupled files feature. If there are files that are highly coupled to the changed files, but have not been changed in the same commit, there might be a higher risk of introducing a bug.

4.2.2 Training and Test Set Splits

To allow for comparison of validation techniques, training and test sets were chosen both by k-fold stratified cross validation and Online Change Classification [30]. However, when we applied Tan et al.'s approach [30] we did not use an updatable algorithm, i.e. ADTree, but instead batch trained with new model instances on each split with the Random Forest algorithm. This decision was taken for two reasons. First, scikit-learn at the time of writing this paper had no available implementation of ADTree. Second, scikit-learn had a limited number of other options for online learners which were unsophisticated in comparison to Random Forest, such as perceptrons.

In table 4.3, our parameter setup for the Online Change Classification is presented. The SGAP is the gap, in terms of days counted from the first commit in the repository, where commits are discarded. This is due to them potentially being unrepresentative of later commits. EGAP is the opposite and in terms of days before the last change was made. The reason for discarding these is that bugs in these commits have potentially not yet been discovered. The train duration determines initial span of the training set; the span increases for consecutive iterations of training. The test duration determines the time span for the test set and is constant for each iteration. In addition to these parameters, the hash value of the last

Definition	**Hypothesis**	**Related work**
$\frac{\text{Line of code added}}{\text{Total lines of code}}$	The overall number of lines of code added is a good measurement if a change introduces bugs. If a commit adds a lot of code, then it is more likely to introduce bugs.	Nagappan et al. [18], Kamei et al. [13]
$\frac{\text{Lines of code deleted}}{\text{Total lines of code}}$	The same as with added lines of code.	Nagappan et al. [18], Kamei et al. [13]
$\frac{\text{Files churned}}{\text{Number of files}}$	The same as with added lines of code.	Nagappan et al. [18], Kamei et al. [13]
Previous version lines of code	If the previously touched files are big, then the risk is higher that a change introduces a bug.	Kamei et al. [13]
Number of modified subsystems	If a change touches many different subsystems, it is more likely to introduce bugs.	Kamei et al. [13]
Number of modified subdirectories	The same if a change involves many subdirectories as for subsystems.	Kamei et al. [13]
Entropy	Entropy gives a measurement in how big the spread of the change is for all involved files. If the entropy is high, the risk is higher that a change introduces bugs.	Kamei et al. [13]
The purpose of a change	If a change is set as a fix it could be part of a longer chain of bug fixes. That means it could be prone to further bugs.	Kamei et al. [13]
The number of authors	With a higher number of authors for a file, the risk for introducing bugs is higher.	Kamei et al. [13]
The time between an authors contributions	If the last change of a file is close to the current one in time, the risk of introducing a defect is higher.	Kamei et al. [13]. See section 4.2.1 for further explanation.
The number of unique changes	The number of unique changes, changes that link back to another change uniquely, give a measurement of how many previous changes a developer needs to keep track of.	Kamei et al. [13]
Overall experience	A developer with experience in the code repository introduces fewer bugs compared to an unexperienced.	Kamei et al. [13]
Recent experience	With recent experience with code, a developer has less risk of introducing bugs.	Kamei et al. [13]
Number of highly coupled files	Intuitively, a change that makes changes to many coupled files is more prone to introduce bugs.	D'Ambros et al. [7]. See section 4.2.1 for further explanation.
Number of coupled files for all degrees	This measures the total number of coupled files, regardless the coupling grade.	D'Ambros et al. [7]. But with a slight modification.
Number of non modified coupled files	A change that does not make changes to all coupled files but only a few of them is prone to introduce bugs.	D'Ambros et al. [7]. See section 4.2.1 for further explanation.

Table 4.2: The features that are extracted and tested on the Jenkins repository.

included commit needs to be specified as well in our implementation. As previously mentioned in Table 4.1 the hash we used is 02d6908ada70fcf8012833ddef628bc09c6f8389.

We tried our best to choose values for these parameters in line with recommendations by previous work. For SGAP, we follow the recommendation of Jiang et al. [11] of omitting the first three years of commits. However, for the Jenkins repository this equates to a start gap of only 331 days. While the first commit of the repository is dated to 5 November 2006, the project is known to have existed well before that. The best guess we have for when the project actually started is 3 October 2004 [2], which leads to the SGAP value in question.

For the values of EGAP, GAP and test duration Tan et al. [30] recommend values based on the average bug fix time. This is the average duration between a bug introducing commit and its corresponding bug fix. The EGAP, and the sum of GAP and test duration, should each be equal to the average bug fix time. However, we calculated this value to be 3.5 years for our data. This is too long to be a feasible guideline for our parameter values; we would not be able to run very many rounds of training and testing, and it makes the chronological gap between training and test data points very large. Therefore we used other reasoning for setting these parameter values. For the EGAP value we examined the spread of bug introducing commits per year, see Figure 5.1, and observed that for year 2016 and onwards there were very few bug introducing commits. On the assumption that this is due to latent bugs that have not yet been discovered, we set the EGAP such that only commits made before 2016 are considered. We decided on a test duration of 400 days to make sure that there are a reasonable number of positive samples in each test set. A rough estimate of multiplying 400 by the average number of commits per day (about 6.5) and the percentage of bug introducing commits (about 4%) would then put the number of positive samples at 104 per test set. If a test set is very large, there is a risk that the training set is to old in comparison to the test set to allow for reliable predictions, but the alternative means that prediction score metrics become volatile and unstable. Since Tan et al. offer no other guidelines on setting the GAP parameter we use 0.2 years, or 73 days, as the value since they themselves use this value for two of their examined datasets.

The remaining values for initial training set length and update length are simply set so that 5 rounds of training and testing are performed.

SGAP	GAP	EGAP	Update	Train duration	Test duration
331	73	781	200	1700	400

Table 4.3: Online Change Classification setup. The unit of all values is days.

As for the ordinary cross validation, we use the default scikit-learn stratified K-fold function to perform a 10 fold cross validation. When reporting results for each validation technique, the results of each run are averaged together. Notice that this allows for the F1-score to be lower than both precision and recall scores.

4.2.3 Sampling

For the Jenkins core repository which we have examined, we identified about 3.6% of commits as bug introducing. This imbalance between the classes negatively impacts many

[2]https://web.archive.org/web/20140701020639/https://www.java.net//blog/kohsuke/archive/20070514/Hudson%20J1.pdf, fetched 17 May 2018

machine learning algorithms. We used a few different sampling techniques to balance the datasets out, so that there was an equal 50-50 balance of positive and negative samples, in order to improve prediction results. We also tried training prediction models without sampling for comparison. The sampling techniques that we used were SMOTE for oversampling, Cluster Centroids for undersampling, and finally SMOTE + Tomek links which combines over- and undersampling.

We used version 0.3.3 of the Python imbalanced-learn library [16] which adds sampling functionality to the scikit-learn library. We also implemented a wrapper for scikit-learn classifiers which allows the imbalanced-learn sampling techniques to be used in conjunction with the scikit-learn functions $cross\_validate$ and $cross\_val\_score$. This allows us to utilize the existing scikit-learn functionality in terms of iterating through each training-test split and returning scores. This was not possible without the wrapper since there was no way to do the sampling properly. This means sampling only on the training set, after the data has been split.

4.2.4 Machine Learning Model

In a previous study made by Yang et al. [35], they trained multiple Random Forest classifiers on several datasets with good results. In our setup we have chosen to use a number of 200 trees to predict with. From trial and error we concluded that performance did not significantly improve for higher number of trees, and that this number still allowed fast training.

To support the validity of using Random Forest as our model we apply it to the datasets given by Kamei et al. [13] and compare their results against ours, see tables 5.5 and 5.6.

The score of our model is measured by precision, recall and f1-score. Accuracy is a poor indicator of performance since the dataset is unbalanced. For example, given a dataset that has 10% positive examples, a model that always predicts negative will have an accuracy of 90%.

Given that our goal is to ultimately make predictions that are useful to developers, we reason that it is desirable to optimize our tool for precision. If developers feel like the model does not provide reliable predictions they will quickly learn to ignore the predictions. False positives in particular are damaging in this regard.

Chapter 5
Evaluation

In this chapter we present our evaluation and results from the steps in our chain, from data mining to prediction. It includes a discussion regarding the results as well as threats to its validity.

In section 5.1 our results are presented, which includes metrics scores as well as repository statistics. It also includes tables comparing our machine learning model applied both to Jenkins but also to the datasets presented in Kamei et al's study [13]. In section 5.2 we discuss our results, the availability of open source implementations, adapting our implementation for a different repository and the challenges involved in eventually developing a usable tool for developers. Section 5.3 deals with threats to validity, and especially the various pitfalls associated with using the SZZ algorithm in general.

5.1 Results

When analyzing the Jenkins repository we used a timespan of 12 years, see Table 4.1. With this, the distribution of bugs between these two commits can be extracted, see Figure 5.1.

In table 5.1, the statistics for SZZ on the Jenkins repository is shown. Each bug data row is a result of using the regex described in 4.1.2 onto commit messages. The general commits indicates commits which do not contain any of the keywords for the other fields. These could be messages like ”creating an RC branch” and ”i18n” which do not carry valuable information.

Table 5.2 shows how many bug introducing commits and bug fixes that have been identified in each respective dataset. It also shows how many of the bugs that were also fixes. A fix indicates a change that has been labeled as fix by the purpose feature criteria. An example of a commit that has been labeled as a bug introducing commit can be found in appendix B.

The prediction results of our classifier can be found in table 5.3, using three different sampling techniques and two different validation techniques. Out of the sampling techniques SMOTE oversamples, Cluster Centroids undersamples, and SMOTE + TOMEK combines over- and undersampling. In addition to averaging over individual runs we also

General data	**Quantity**
Total number of commits	26,378
Number of identified issues	2408
Number of identified bug fixes	1607
Percent of issues matched to a bug fix	66.7%
Bug data	**Quantity**
Number of bug introducing commits that added something	350
Number of bug introducing commits that updated something	100
Number of bug introducing commits that fixed a bug	168
Number of bug introducing commits with a issue number	3
Number of general commits	336
Total number of identified bug introducing commits	957
Average time between a fix and a bug introducing commit in days	1,247
Median time between a fix and a bug introducing commit in days	1,101
Average time between resolution date for an issue and its creation date in days	155
Longest time between resolution date for an issue and its creation date in days	3146

Table 5.1: Statistics for Jenkins after applying the SZZ algorithm.

Dataset	**Bugs**	**Fixes**	**Fixes ∩ Bugs**	**Total number of commits/transactions**
Bugzilla	1,696	3,973	1,586	4,620
Columba	1,361	1,463	439	4,455
JDT	5,089	10,799	2,218	35,386
Mozilla	5,149	62,888	3,943	98,275
Postgres	5,119	8,933	2,043	20,431
Jenkins (Ours)	954	2,979	808	26,378

Table 5.2: A comparison with the JIT datasets studied in previous work [13]. The number of bugs is the number of bug introducing commits that has been found.

Jenkins			
	K-fold Stratified Cross Validation		
Sampling method	Precision	Recall	F1
No Sampling	0.156±0.246	0.026±0.042	0.029±0.034
SMOTE	0.123±0.076	0.212±0.136	**0.154±0.096**
SMOTE + TOMEK	0.117±0.071	0.206±0.130	0.148±0.091
Cluster Centroids	0.037±0.001	0.945±0.037	0.072±0.002
	Online Change Classification		
Sampling method	Precision	Recall	F1
No Sampling	0.210±0.177	0.017±0.014	0.031±0.026
SMOTE	0.147±0.041	0.104±0.034	0.116±0.031
SMOTE + TOMEK	0.163±0.018	0.126±0.043	**0.137±0.030**
Cluster Centroids	0.028±0.004	0.917±0.037	0.054±0.008

Table 5.3: Averaged prediction results and standard deviations

Sampling method	Precision	Recall	F1
No Sampling	0.486	0.038	0.071
SMOTE	0.143	0.091	0.112
SMOTE + TOMEK	0.165	0.104	**0.127**
Cluster Centroids	0.028	0.923	0.055

Table 5.4: Micro averaged results for Online Change Classification

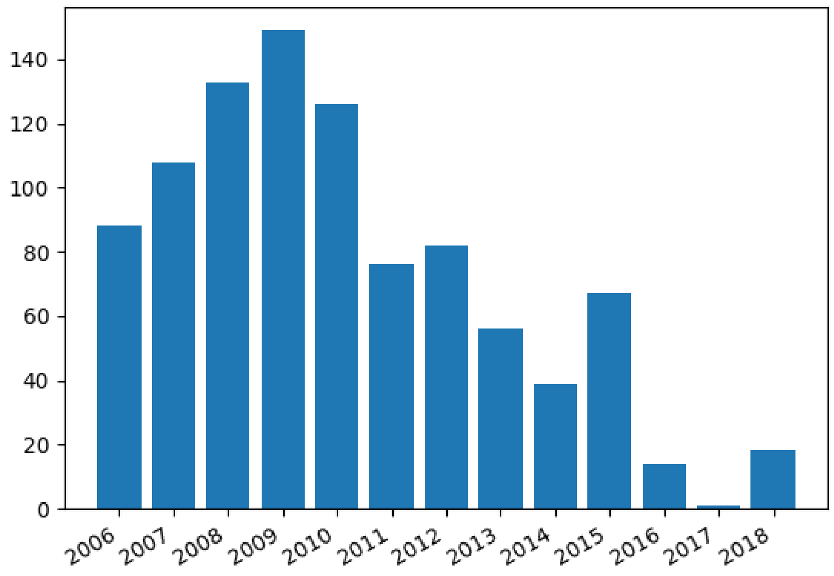


Figure 5.1: The distribution of found bug introducing commits. Each bar represents the number of commits that have been marked as bug introducing for a certain year.

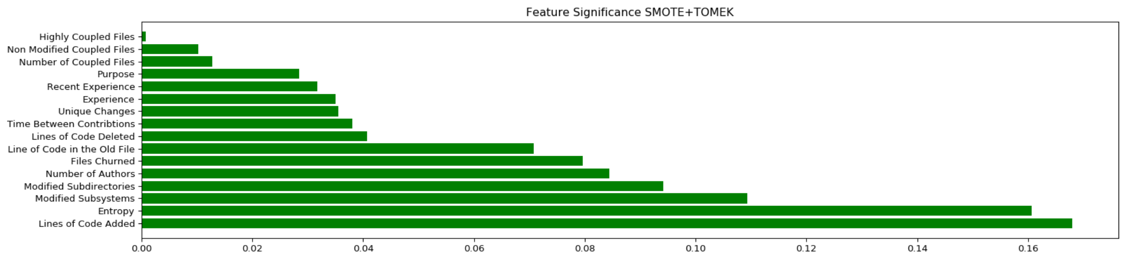


Figure 5.2: Feature significances using SMOTE+TOMEK. The features are sorted with the least significant feature at the top of the y axis. The significance is then shown at the x axis where each significance is a number between one and zero.

show the micro-averaged results for Online Change Classification in Table 5.4, since partition size and class balance varies for each run with this validation method.

The results in 5.5 are the results from running our classifier on the JIT datasets, with the same parameters as on Jenkins. The validation has not been done using the Online Change Classification technique.

In Figure 5.6 the original results from Kamei et al.'s research are presented. Kamei et al. stated that they have resampled the data to achieve these results. Compared to our sampling methods, they have only used a single sampling method which was random undersampling.

Finally we have extracted the feature significances when training a Random Forest Classifier on the entire dataset, the result of which are presented in Figure 5.2.

	Bugzilla			**Columba**		
Sampling method	Precision	Recall	F1	Precision	Recall	F1
No sampling	0.71	0.54	0.62	0.67	0.46	0.54
SMOTE	0.71	0.56	0.63	0.64	0.51	0.57
SMOTE + TOMEK	0.73	0.53	0.61	0.65	0.47	0.55
Cluster Centroids	0.52	0.79	0.63	0.41	0.80	0.54
	JDT			**Mozilla**		
Sampling method	Precision	Recall	F1	Precision	Recall	F1
No sampling	0.41	0.06	0.11	0.52	0.07	0.11
SMOTE	0.45	0.08	0.13	0.55	0.07	0.12
SMOTE + TOMEK	0.45	0.05	0.08	0.61	0.04	0.08
Cluster Centroids	0.15	0.86	0.26	0.06	0.96	0.11
	Postgres					
Sampling method	Precision	Recall	F1			
No sampling	0.67	0.41	0.48			
SMOTE	0.65	0.42	0.50			
SMOTE + TOMEK	0.68	0.37	0.47			
Cluster Centroids	0.35	0.84	0.49			

Table 5.5: K-fold Stratified Cross validation comparison with the JIT datasets

	Precision	Recall	F1
Bugzilla	0.54	0.69	0.60
Columba	0.51	0.67	0.58
JDT	0.26	0.65	0.37
Mozilla	0.13	0.63	0.22
Postgres	0.49	0.65	0.56

Table 5.6: Original results from Kamei et al. [13].

5.2 Discussion

Our model achieved an F1-score of 15.4% using k-fold stratified cross validation, and a somewhat worse result using Online Change Classification of 12.7%. The difference in performance was expected and is in line with previous work [30, 12]. The best results were achieved when oversampling, where performance was similar between SMOTE and SMOTE+TOMEK. The performance is likely not good enough to make the model useful in a real life setting; it is notable however that the precision performance of 16.5% using Online Change Classification does represent a 458% improvement compared to random chance, which would be 3.6%. Additionally, as the F1-score performance is comparable to Kamei et al.'s results for the JDT and Mozilla repositories it seems reasonable to think that our implementation works as intended.

Undersampling worked poorly, likely due to the small amount of data points left after sampling, and led to the classifier predicting almost all samples to be positive. As for using no sampling at all, while precision score was high it had abysmal recall. Of course, precision is the most important metric with considerations to making the predictions useful which means we should not ignore this result.

It should be noted as well that we could likely improve precision scores for the other methods at the cost of recall if we set the decision threshold higher than the default of 0.5. Where to draw the line on what is an acceptable level of trade-off between precision and recall is an important question for future work.

To be able to evaluate which features are worth using and which features that has to be improved, we extracted the feature significances from the Random Forest. In Figure 5.2, each features significance are displayed as a value between one and zero. They are also sorted so that the least significant are the top most and the most significant on the lower part of the graph. The feature significances do not vary depending on which sampler we have used so we are only showing the results from the best one, SMOTE+TOMEK. As can be seen in the Figure5.2, the most significant features are the code churns and the diffusion features. Since many of the commits that has been classified as bug introducing have changed in many files but also in many subsystems, it is no surprise that these features are the most significant. As for the coupling features, they need to be studied further. According to D'Ambros et al. [7], these features should be more useful than they are in our model. Currently, all coupling features are extracted as absolute values. As an alternative, the number of highly coupled files could be divided over the total number of coupled files in total. Another idea would be to measure the number of coupled files with regard to the number of subsystems. If many files change together, but belong to different subsystems, such feature could give a measurement in how much the subsystems actually depends on each other.

Overall we concur with the criticism made by Rodríguez-Pérez et al. [25] of the fact that very few studies make their source code available. Our study shifted focus as a direct consequence of this, since the implementational work took a lot of time and effort. Rodríguez-Pérez et al. observed that many studies do not use the best versions of the SZZ algorithm, and offered the explanation that this is due to lack of readily available implementations. Some reports do not clearly state which version was used, or they use their own mixture of ideas. This hurts not only reproducibility, but also quality of results, since a common implementation would have more people working on it. For this reason we have released our code on Github [19] so that future research can be made easier.

When considering future directions for our project, making our SZZ implementation compatible with other repositories is essential. Although most of the code is applicable to any repository, the part that identifies bug fixing commits is specific to the Jenkins repository. If one wishes to include new repositories, this part would need to be adapted or rewritten to different extents, depending on what project or BTS is to be included.

An alternative would be to implement a repository-independent version that sacrifices accuracy by not considering any BTS and instead relying solely on the version history logs, i.e. ASZZ which was mentioned in Section 3.1. However, this approach would not guarantee acceptable results. For instance, it would not work at all for the Jenkins repository. In Jenkins, bug fixing commits do not necessarily include the words "bug" nor "fix"; they reference an issue id (e.g. "JENKINS-1234") whose format is specific to Jenkins.

A second alternative would be to use a classifier trained on one project to predict on samples from a different project, so called cross project prediction. This was tried by Zimmermann et al. [36], however with poor results: *"Out of the 622 non-trivial cross-project combinations, only 21 had precision, recall, and accuracy values which satisfied our criteria; an alarmingly low success rate of 3.4%"*.

When eventually building a tool for developers there are a few things that should be kept in mind. Previous attempts at introducing bug prediction in real life settings have demonstrated difficulties in providing value to developers. The ones we have looked at are case studies done at Google [17], Infopark [23] and Cisco [30]. These case studies have pointed to two main obstacles in making bug prediction useful to developers. First, unreliable or unknown quality of predictions, and second a lack of actionable interpretations of the predictions.

The first part concerns the fact that it is hard to achieve satisfying results on precision and F1 measures. The quality of the model can be difficult to even assess reliably since this would require laborious manual inspection of identified bug introducing commits. This was also the case for us when we attempted to build a model based on the Jenkins core repository.

The purpose of an eventual tool would be to find bugs that are not typically caught either by testing or reviews. This means that even if the model correctly predicts a commit to be bug introducing, the inspecting developer might still not find the bug even with this information. This means that even if high precision is achieved by the model, the perception of precision could still be low.

As for the second obstacle, a finding that has been consistent for all three of these attempts is that actionable recommendations are needed for developers to trust the model and to be able to act on predictions. By actionable recommendations we mean recommendations that give developers concrete suggestions for actions to be taken. For example, TLOC or Total Lines of Churn is a common process metric used for bug prediction. This feature does not however make for actionable interpretations. If a developer is told that a commit was predicted buggy due to it being long, the explanation is not helpful. In contrast, if a developer is told that a commit was predicted buggy due to introducing a modulo operator; then they can review that the modulo operator has been applied correctly.

Taking actionable recommendations into consideration constrains the choice of machine learning algorithms to those which are interpretable, i.e. white box models. These white box models, such as decision trees, can be contrasted with black box models, such as deep neural networks. The disadvantage of using such models is that you typically trade-off performance when choosing a white box model over a black box one.

To verify that our implementation of the SZZ algorithm is somewhat correct, we have manually taken a few samples of the commits that were labeled bug introducing and inspected them. As seen in Table 5.1, most of them either added or fixed a bug. By picking a random commit from these bug introducing commits, like the one in Appendix B, we could confirm that it actually fixed a bug and that it then should be labeled as a partial fix.

5.3 Threats to Validity

The SZZ algorithm has a number of acknowledged limitations, both regarding the first part linking bug tracker issues to bug fixing commits as well as the second part linking bug fixes to bug introductions. These limitations are summarized in Table 5.7

The efficacy of the first part of SZZ is largely dependent on developer discipline. Consistency of commit messages across bug fixes and a well maintained bug tracker system are essential. As mentioned previously, an important reason for picking the Jenkins repository as our object of study was that it had a well maintained BTS. With this in mind we consider inaccurate mapping and systematic bias to be of lesser concern. It should be noted as well that we have manually inspected commit messages in order to verify that our results were reasonable. It is however true that we were only able to find bug fixes for 2 out of 3 issues in the BTS, which means we have an incomplete mapping with a significant percentage of issues for which we could not identify a bug fix.

As for the limitations associated with the second part we can not tell to what extent they impact our resulting data. Verifying the veracity of data that this part of the SZZ algorithm

Part	Type	Description
First part	Incomplete mapping [1]	The fixing commit cannot be linked to the bug
	Inaccurate mapping [2]	The fixing commit has been linked to a wrong bug report, they don't correspond to each other
	Systematic bias [1]	Linking fixing commit with no real bug report
Second part	Cosmetic changes, comments, blank lines [15]	Variable renaming, indentation, split lines, etc.
	Added lines in fixing commits [6]	The new lines can not be tracked back
	Long fixing commits [6]	The larger the fix, the more false positives
	Semantic level is weak [33]	Changes with the same behavior are being blamed
	Correct changes at the time of being committed [6]	Changes in other parts of the source code base trigger a bug issue in another part
	Commit Squashing [9]	Might hide the bug introducing commit, loosing authorship information

Table 5.7: Limitations of the SZZ algorithm. Table is copied from Rodríguez-Pérez et al's paper [25].

provides is both time-consuming and requires knowledge of the underlying source code. In our work, since we have neither the time nor the knowledge to manually verify results, we instead assume the data to be correct. For those who are interested, the best evaluation we could find regarding SZZ performance was done manually in 2014 by Davies et al. [8], see section 3.3.

What also could threaten the validity of the result is the fact that only the two of us have worked with the code base of our implementations. Due to this, the code could potentially contain a lot of bugs. However, our hope is that our implementation will be improved by a larger community which would in time raise the robustness of the code.

Chapter 6
Conclusions

In this study we have implemented infrastructure for predicting bug introducing commits with the help of machine learning. We have developed the tool chain, from mining a dataset to making the actual predictions, that is needed to make further investigations in this field.

We have implemented the SZZ algorithm with regards to the Jenkins core repository and then used this implementation to produce a dataset of labeled commits. Furthermore, we have implemented feature mining scripts to extract features for the labeled commits and trained a Random Forest classifier on the dataset. Lastly, we have implemented Online Change Classification in order to accurately validate our prediction model. The best F1-score results of the trained classifier with k-fold cross validation was 15.4% and with Online Change Classification 12.7%.

With the help of our implementations, future researchers will be able to reproduce our results but will also be able to generate new datasets. When considering avenues for future work, especially as it pertains to making more open source implementations available, we consider the following ideas to have good potential.

- Implementing the dependency approach [27].
- Combining SZZ with the dependency approach as suggested by Davies et al. [8].
- Implementing a system for providing actionable recommendations from the machine learning model [30].
- Implementing da Costa et al's [6] method of evaluating plausibility of SZZ results without manually reviewing code.

Utilizing a combination between the SZZ algorithm and the dependency graph approach, Davies et al. [8] showed that there is potential to improve dataset generation. The combination can be done in different ways depending on what metric is prioritized such as precision, recall, F1 score or even computational effort. Davies et al. also showed that if only one approach is to be used, then the dependency approach wins out when it comes to metric scores. This is done at the cost of computational effort. This shows the importance of implementing the dependency approach independently of whether it is combined with SZZ or not.

For a future tool to be truly useful for reviews, the output must be helpful and interpretable for the reviewer so that a clear course of action can be taken. Otherwise the tool might not convince the reviewers that it is helpful and lead to it becoming obsolete and abandoned. Actionable recommendations is argued both by Tan et al. [30] and Lewis et al. [17] as being crucial in terms of industry adoption of bug prediction tools.

A method for automatically evaluating the result of SZZ would be very valuable. As long as the reliability of underlying bug prediction datasets are uncertain, we can not truly judge the efficacy of models built on top of these datasets. As mentioned previously, manually verifying SZZ-generated data is often unfeasible which makes da Costa et al's [6] framework valuable as an alternative.

Our work will be available on Github [19] where the complete chain can be executed.

Bibliography

[1] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[2] Tegawende F Bissyande, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, and Laurent Reveillere. Empirical evaluation of bug linking. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 89–98. IEEE, 2013.

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[4] The Jenkins Community. Jenkins. `https://github.com/jenkinsci/jenkins`, 2018. Version 2.107.2.

[5] João Correia. old-szz. `https://github.com/intelligentagents/old-szz`, 2017. Version 0.0.1.

[6] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017.

[7] M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *2009 16th Working Conference on Reverse Engineering*, pages 135–144, Oct 2009.

[8] Steven Davies, Marc Roper, and Murray Wood. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, 26(1):107–139, 2014.

[9] Georgios Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories*, pages 233–236. IEEE Press, 2013.

[10] K. Hajek H. Gall and M. Jazayeri. Detection of logical coupling based on product release history. page 190, 1998.

[11] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 279–289. IEEE Press, 2013.

[12] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, 2016.

[13] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. volume 39, pages 757–773, June 2013.

[14] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[15] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[16] Guillaume Lemaître, Fernando Nogueira, and Christos K Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[17] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 372–381. IEEE Press, 2013.

[18] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292. ACM, 2005.

[19] Svensson O. and Berg K. SZZ unleashed. `https://github.com/wogscpar/SZZUnleashed`, 2018. Version 0.1.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Lourdes Pelayo and Scott Dick. Applying novel resampling strategies to software defect prediction. In *Fuzzy Information Processing Society, 2007. NAFIPS'07. Annual Meeting of the North American*, pages 69–72. IEEE, 2007.

[22] Ronaldo C Prati, Gustavo EAPA Batista, and Maria Carolina Monard. Data mining with imbalanced class distributions: concepts and methods. In *IICAI*, pages 359–376, 2009.

[23] Lutz Prechelt and Alexander Pepper. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. *Information and Software Technology*, 56(10):1377–1389, 2014.

[24] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[25] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 2018.

[26] 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, pages 966–969, New York, NY, USA, 2015. ACM.

[27] Vibha Singhal Sinha, Saurabh Sinha, and Swathi Rao. Buginnings: identifying the origins of a bug. In *Proceedings of the 3rd India software engineering conference*, pages 3–12. ACM, 2010.

[28] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[29] M. Sohn, S. Pearce, A. Loskutov, C. Aniszczyk, C. Halstrick, C. Ranger, D. Borowitz, D. Pursehouse, G. Wagenknecht, J. Nieder, K. Sawicki, M. Kinzler, R. Rosenberg, R. Stocker, S. Zivkov, S. Lay, T. Parker, and T. Wolf. Eclipse jgit. `https://github.com/eclipse/jgit`, December 2017. Version 4.10.0.201712302008-r.

[30] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 99–108. IEEE Press, 2015.

[31] A. Tornhill. code-maat. `https://github.com/adamtornhill/code-maat`, 2017. Version 1.1.

[32] Adam Tornhill. *Your code as a Crime Scene*. The Pragmatic Programmers, LLC, 2015. ISBN: 978-1-68050-038-7.

[33] Chadd Williams and Jaime Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.

[34] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, Aug 2015.

[35] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220, 2017.

[36] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

Appendix A

Code structure

1. **fetch.py** - A script that gets the issues from the JIRA and saves them to a JSON object.
2. **git_log_to_array.py** - A script to convert the git log to a JSON object.
3. **find_bug_fixes.py** - The script that tracks down bug fixes in the BTS system. The output is a JSON with the format:

```
{
    "Bug-ID": {
        "creationdate": "YY-MM-DD HH:mm:ss <time_zone>",
        "commitdate": "YY-MM-DD HH:mm:ss <time_zone>",
        "resolutiondate": "YY-MM-DD HH:mm:ss <time_zone>",
        "hash": "<hash>"
    },
    "Bug-ID": {
        "creationdate": "YY-MM-DD HH:mm:ss <time_zone>",
        "commitdate": "YY-MM-DD HH:mm:ss <time_zone>",
        "resolutiondate": "YY-MM-DD HH:mm:ss <time_zone>",
        "hash": "<hash>"
    }
}
```

4. **szz-0.1.jar** - This is the part of the SZZ algorithm which tracks bug introducing commits using the issues given by the **issue_list.json** script. It is executed by either running the gradle build script with the runJar command or by generating the jar. The runJar option uses the default input arguments and do not produce anything other than the resulting find_and_introducers_pairs.json file. This is useful when further developing the algorithm. As for the second alternative, the gradle script should be executed with the fatJar option. With this file, the user can specify the following parameters:
   - **-i** The path to the issue JSON file.
   - **-r** The path to the local git repository.
   - **-d** The depth that that the line numbering graph should use. See Section 4.1.2 for further explanation.

- **-c** How many threads that should be used by the algorithm. The default is the number of cores on the computer.

The output will be the fix_and_introducers_pairs.json which includes possible pairs of bug fix commits and the possibly responsible bug introducing commits.

```
[
    [
        "<bug_fix_hash>",
        "<bug_introducing_hash>"
    ],
    [
        "<bug_fix_hash>",
        "<bug_introducing_hash>"
    ]
]
```

5. **assemble_labels.py** - This script parses the fix_and_introducers_pairs.json file and produces a csv file containing the labels that can be used by a machine learning model. It also has the possibility to draw the distribution of bug introducing commits as a bar diagram.
6. **assemble_code_churns.py** - This script analyses a local git repository and extracts the relative code churns, which are described in Section 2.4. The output is a csv file with each change listed in order with its features in its columns.
7. **assemble_diffusion_features.py** - This is the diffusion features extraction script. The diffusion features is described in Section 2.4. It analyses a local git repository and saves them to a csv file.
8. **assemble_purpose_features.py** - This is the purpose feature extraction script. This feature is described in Section 2.4. It analyses a git repository and saves them to a csv file.
9. **assemble_history_features.py** - The history features extraction script. It extracts the history features from a local git repository to a csv file. The features are described in Section 2.4.
10. **assemble_experience_features.py** - The experience features extraction script. It extracts the experience features from a local git repository. The features are described in Section 2.4.
11. **analyze_commit** - A simple bash script which is intended to be executed in a docker container. This script runs the code-maat tool [31] on a git repository. It analyses the git log and produces a file that describes the coupling between all files in the repository.
12. **assemble_features.py** - This scripts takes advantage of docker to run scripts on git repositories. Using scripts like the **analyze_commit**, it is possible to run tools and scripts that have dependencies that normally is not installed on the running system.
13. **assemble_coupling_features.py** - This script extracts the coupling features. It uses the resulting files from the code-maat tool and extracts the features to a csv file. The features are described in Section 2.4.
14. **general_data.py** - This script extracts information such as how many actual fixes that exists, how many bugs there is and how many bugs that are fixes.
15. **model.py** - This is the prediction script. It can evaluate models, train models and classify individual examples. Using the combined features, which are extracted and merged with the assemble scripts before, this script predicts if a change is buggy or not.
16. **random_forest_wrapper.py** - Contains an implementation of a wrapper which applies

sampling before doing the cross validation.

17. **time_sensitive_split.py** - Contains an implementation of the Online Change Classification used to create train and test data.

Appendix B
A Bug Introducing Commit example

An example of a commit that is a bugfix but which is not parsed as one by the SZZ algorithm, is the one below. The commit message indicates that a filereader is not properly closed thus a memory leak occurs. The commit message also indicates that a bug has been found through a FindBugs finding. Inspecting the code further gives an indication that only a cosmetic change in placement of the imports has been made.

```
commit 0cd964074a723a08c3bec23cfc73f0c5721e6868
Author: C..... K..... <....@..x.de>
Date:   Sun Jul 17 23:26:24 2011 +0200

    Close FileReader (FindBugs finding)

diff --git
 ↪  a/core/src/main/java/hudson/ClassicPluginStrategy.java
 ↪  b/core/src/main/java/hudson/ClassicPluginStrategy.java
index 853ff1c8ef..b1e84bebc5 100644
--- a/core/src/main/java/hudson/ClassicPluginStrategy.java
+++ b/core/src/main/java/hudson/ClassicPluginStrategy.java
@@ -23,20 +23,20 @@
   */
 package hudson;

+import hudson.Plugin.DummyImpl;
 import hudson.PluginWrapper.Dependency;
 import hudson.model.Hudson;
 import hudson.util.IOException2;
 import hudson.util.MaskingClassLoader;
 import hudson.util.VersionNumber;
-import hudson.Plugin.DummyImpl;
```

```
 import java.io.BufferedReader;
+import java.io.Closeable;
 import java.io.File;
 import java.io.FileInputStream;
 import java.io.FileReader;
 import java.io.FilenameFilter;
 import java.io.IOException;
-import java.io.Closeable;
 import java.net.URL;
 import java.net.URLClassLoader;
 import java.util.ArrayList;
@@ -46,14 +46,14 @@ import java.util.Collections;
 import java.util.Enumeration;
 import java.util.HashSet;
 import java.util.List;
-import java.util.jar.Manifest;
 import java.util.jar.Attributes;
+import java.util.jar.Manifest;
 import java.util.logging.Level;
 import java.util.logging.Logger;

+import org.apache.tools.ant.AntClassLoader;
 import org.apache.tools.ant.BuildException;
 import org.apache.tools.ant.Project;
-import org.apache.tools.ant.AntClassLoader;
 import org.apache.tools.ant.taskdefs.Expand;
 import org.apache.tools.ant.types.FileSet;

@@ -86,8 +86,13 @@ public
 ↪  class ClassicPluginStrategy implements PluginStrategy {
         boolean
           ↪  isLinked = archive.getName().endsWith(".hpl");
         if (isLinked) {
             // resolve the
               ↪  .hpl file to the location of the manifest file
-            String
 ↪  firstLine = new BufferedReader(new FileReader(archive))
-                    .readLine();
+            final String firstLine;
+            BufferedReader
 ↪  reader = new BufferedReader(new FileReader(archive));
+            try {
+                firstLine = reader.readLine();
+            } finally {
+                reader.close();
+            }
             if (firstLine.startsWith("Manifest-Version:")) {
                 // this is the manifest already
             } else {
```

```
@@ -302,7 +307,7 @@ public
 ↪  class ClassicPluginStrategy implements PluginStrategy {
                 wrapper.setPlugin(new DummyImpl());
             } else {
                 try {
-
 ↪   Class clazz = wrapper.classLoader.loadClass(className);
+                    Class<?>
 ↪  clazz = wrapper.classLoader.loadClass(className);
                     Object o = clazz.newInstance();
                     if(!(o instanceof Plugin)) {
                         throw new IOException(className+" d⌋
                          ↪  oesn't extend from hudson.Plugin");
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2018-06-07

EXAMENSARBETE SZZ Unleashed: Bug Prediction on the Jenkins Core Repository
STUDENT Oscar Svensson, Kristian Berg
HANDLEDARE Markus Borg (LTH), Sven Selberg (Axis Communications AB)
EXAMINATOR Emelie Engström (LTH)

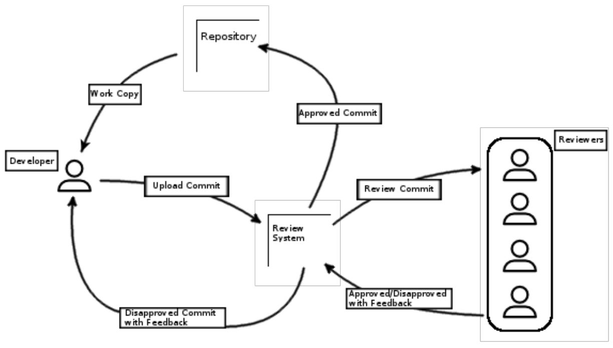
Open Source Implementations of Bug Prediction Tools on Commit Level

POPULÄRVETENSKAPLIG SAMMANFATTNING **Oscar Svensson, Kristian Berg**

Code reviews are ubiquitous in large scale software development. We have implemented an approach using machine learning to help reviewers prioritize their efforts on high risk code.

Code reviewing is a common tool in software development for analyzing commits. A commit is a collection of changes to a program made up of added and deleted lines of code. When a developer puts up a commit for review, their peers look at the committed code and make sure the change is sound, that the code follows established conventions and that it does not introduce any bugs. The last part can be tricky, as well as time consuming, and some bugs are bound to slip through. However, it is of great interest to catch these bugs as early on as possible. Fixing a bug once a program has been released is much more expensive than fixing it during the review process. If more bugs could be caught at this early stage, it could save development teams a lot of time and money. One way to do this would be to direct code reviewers so that they focus on commits that are more likely to introduce bugs.

Machine learning could possibly be of help here. If a machine learning model is presented with many examples of bug introducing commits, as well as clean commits, it could learn to recognize them. Unfortunately, which commits introduced what bugs is not usually kept track of. That means a big part of our work consisted of taking a large number of commits and labeling them, so that we could use them to train our machine learning model.



A Code Reviewing Process.

The method we used to go about labeling these commits is called the SZZ algorithm. It takes bug reports as input and from these reports it tries to deduce what commits were responsible for introducing each bug. However, even though the SZZ algorithm has been around for 13 years, there were no suitable implementations available for us. Therefore we undertook the effort of implementing this algorithm.

The SZZ algorithm works in two steps: first it tries find a bug fixing commit for each bug report, and second it tries to pair each bug fix with one

EXAMENSARBETE SZZ Unleashed: Bug Prediction on the Jenkins Core Repository
STUDENT Oscar Svensson, Kristian Berg
HANDLEDARE Markus Borg (LTH), Sven Selberg (Axis Communications AB)
EXAMINATOR Emelie Engström (LTH)

or more bug introducing commits.

In our study we implemented and applied the SZZ algorithm to a large open source software repository, Jenkins. Out of 26,000 commits in the project, we identified 1,000 commits as bug introducing, or slightly less than 4 percent.

The metric by which we measured the performance of our machine learning model is called the F1-score. It is a combination of how often the model is correct when it says a commit is bug introducing, and how many of the bug introducing commits it correctly identifies. The best realistic score we achieved was 13.7%. Although this is probably not good enough for use in a real life setting yet, it is a significant improvement when compared to random chance which would give an F1-score of only about 4%.

In addition to our machine learning model results, one of our main contributions is that we have released the code we used for our implementation as open source software. Our hope is that future researchers will use our implementations to improve upon our results. The software repository can be found at `https://github.com/wogscpar/SZZUnleashed`.