MASTER'S THESIS 2021

Predicting Bugs to Reduce Debugging Time

Oscar Werneman

LU-CS-EX 2021-11

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-11

Predicting Bugs to Reduce Debugging Time

Oscar Werneman

Predicting Bugs to Reduce Debugging Time

Oscar Werneman
oscar.werneman@verifyter.com

May 27, 2021

Master's thesis work carried out at Verifyter.

Supervisors: Markus Borg, markus.borg@cs.lth.se
Daniel Hansson, daniel.hansson@verifyter.com

Examiner: Per Runeson, per.runeson@cs.lth.se

Abstract

Finding and fixing bugs constitutes a significant portion of software development costs. Regression test suites of a varying size run as often as possible to capture bugs. With the aid of logs, revision history, test results, and planning, this thesis aims to utilise machine learning to speed up the debugging process. Instead of treating all revisions equally, creating a bug prediction model can rank revisions according to risk, thereby reducing the number in focus. Using the CRISP-DM methodology while building the models, evaluation of the results lead us to develop a concise and practically applicable method for debugging ML implementations. Results show that although risk-based verification is currently beyond reach, a bug prediction model can reduce the time spent debugging.

Keywords: Bug Prediction, Automatic Debug, Machine Learning, Automatic Program Repair, Mining Software Repositories, Minority Class

Acknowledgements

Time flies like an arrow, Fruit flies like bananas…
Many thanks to Markus Borg, Daniel Hansson, Patrik Granath, Per Runeson & Skilla. Markus has inspired a love for research and opened more doors and topics than I could have imagined. Daniel grounded me to differentiate between cost and benefit.

Contents

Chapter 1
Introduction

This chapter serves as an introduction to the problem and the motivation behind conducting this research. We will also examine previous contributions and recount what this report hopes to achieve.

1.1 Introduction

In today's competitive global market, a company's success is often dictated by how fast they can deliver their products. The reader may react to the previous statement, noting that although the speed of delivery is essential, there are other aspects to keep in mind. For example, a premature release might highlight another such aspect: the absence of bugs. "An *error* is a mistake, misconception, or misunderstanding on the part of a software developer. A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification" as phrased by Ilene Burnstein [7]. A fault (defect) is colloquially called a software bug, i.e. the manifestation of an error. Depending on the bug type, they can be caught by different techniques.

There are static and dynamic testing techniques aimed at different types of bugs. Static testing means that the code does not need to be executed and includes manual document inspection and checking compilation issues. On the other hand, dynamic testing means executing the software and includes everything from *assertion* and *coverage testing* - to *performance testing*. *Assertion testing* means evaluating a logical expression. *Coverage testing* is a measure of which lines a test executed. *Performance testing* includes testing the software under different types of stress, e.g. high workload, erratic workload and sustained load.

Testing techniques, as mentioned, are aimed at different types of bugs, but can be categorised as *functional*, *performance* and *usability* defects. *Functional* defects have a unique property. Often understanding the bug can lead the developer to the specific function or line at

fault, whereas *performance* and *usability* defects are harder to grasp.

It is good practice to create tests alongside functionality (test-driven development or TDD) to catch bugs. However, with all software development, legacy code and software with unknown provenance tend to amass more significant portions of the codebase. In addition, the developers that implemented the tests and code might have moved on to different projects or companies. The solution seemingly comprised of: following strict coding guidelines, continuous refactoring and deleting unused code. Following these guidelines can be time-consuming and is, of course, an issue of prioritisation, meeting deadlines and producing code now on credit borrowed from the future. Eventually, most projects will reach a point where developers spend more time debugging than creating bugs [10, 21]. As the cost of debugging has risen, alongside the complexity, the need for intelligent debugging tools has increased [27, 42, 4].

Verifyter offers such a tool, PinDown [39]. PinDown is an automatic debugger of regression failures. PinDown uses binary search to locate a bug and validates the bug by automatic program repair (ARP) [24]. The customer can choose granularity of the bug report, ranging from exact line of the bug to only reporting which revision contains the bug.

Verifyters customers mainly use PinDown for Application-specific integrated circuits (ASIC) verification, i.e. regression testing of hardware description languages. ASIC verification (comparable to software testing) requires substantial resources and extensive test farms to run regression testing. Due to the long test executions, continuous integration is not common practice. Instead, the number of commits between executing the test suite can be in the magnitude of 100s.

When a test has failed, the developers start the debugging process. Debugging would generally involve running the necessary tools and manually debug the anticipated faults, reviewing which revisions are under test and which tests failed. When a test has failed, *Verifyter* proposes using machine learning (ML) models, similarly to how humans debug. The ML model, just like humans, rank the revisions under test by risk instead of assuming no knowledge of the codebase. By reverting commits, PinDown can validate whether the removal of the proposed fault can make the tests pass again. The term *revert* means that a specific set of recorded changes are removed from the code. "A specific set of recorded changes" is also called a commit or revision. The ranking of commits not only works as a guide for the automatic debugging process, but it also helps manual debugging as reviewing five high-risk commits costs less than treating all 100 as equals.

An outcome of this thesis is the creation of a bug prediction model. *Bug prediction*, in this thesis, means identifying a revision that produces a test failure. PinDown is especially fit for this ML implementation as they already have collected large amounts of data from ASIC verification and debugging. Verifyter has also explored many *features*. A *feature* is a numeric value aimed to characterise a phenomenon, in our case, to distinguish a commit containing a bug from one that is "clean". These features are the cornerstone of ML-based bug prediction and often derived from papers published at the conference *Mining Software Repositories* (MSR [25]).

An ML implementation is often largely entangled, which means the practice of abstractions crumbles. In ML "Changing Anything Changes Everything" [32] applies. Traditional software, being Boolean-based logic, stands in stark contrast to the black box probabilistic nature of ML regarding testing and debugging. Where once true or false tests could be used to verify functionality, an ML model cannot be deemed right or wrong from a single output.

In that sense, an ML bug has a performance bug's characteristics. Issues in the ML implementation will inevitably require alternative methods to debug the root cause and catch bugs. In tandem with developing the bug prediction model, we will also create a method for debugging it.

1.2 Related Work

In this section, we describe some studies relevant to this thesis research. This includes prior work on bug prediction models, as well as algorithm choices. We will also describe some of the complications reported in the literature when developing a bug prediction model and debugging ML implementations.

1.2.1 Bug Prediction

Many studies assess ML-based bug prediction models. Borg et al. [6] worked with predicting bugs on a revision level, similar to this thesis. They implemented and utilised the SZZ algorithm (explained in Section 3.1) to gather the required training data. Their suggestions for future research included combining the SZZ algorithm with an alternative method. Although this thesis does not use the proposed method, combining data gathering techniques will be evaluated.

Another important aspect of bug prediction is that the training data is imbalanced, i.e. the number of faulty revisions is few compared to correct revisions. This is highlighted by Pandey et al. [26]. Class imbalance can adversely affect the feature selection process and the stability of the produced models. Pandey et al. utilise SMOTE (see Section 3.4) to handle imbalance and cross-validation when comparing their approach to previous bug prediction techniques.

1.2.2 Obtaining a Dataset

At the core of ML is big data. Zeller et al. note that a significant challenge is obtaining correctly labelled datasets [17]. When analysing bug databases, a notable portion of issue reports pertains to enhancements rather than defect correction [2]. Zeller et al. also found that 33% of bugs in open source projects were mislabelled as bugs, and close to 40% of files labelled as defective never contained a bug [17] – so called false positives. Methods for mining software repositories are required to obtain the datasets needed to train a bug prediction model. One such technique, the SZZ algorithm [34], has potential, and there is also an open implementation available [6]. The SZZ algorithm mines the revision history by identifying keywords in commit messages, such as permutations of the word "*fix*". When the algorithm finds a keyword in a commit message, it assumes that the revision previously altering those lines introduced a bug (generating a labelled dataset from the revision history). In our work, we found this implementation to be helpful in combination with PinDown's own labelling

technique.

1.2.3 Data Difficulties

Learning from *imbalanced data* can prove a challenge [16, 20]. Class imbalance refers to one outcome being more likely than the other, and the resulting models often classify all samples as belonging to the majority class. Stefanowski [35] raises important questions regarding the advances made with specific classifiers and sampling techniques when he notes that these experiments have been conducted in research settings using generated rather than real-world data. Stefanowski details *data difficulty factors* that complicate learning from imbalanced data: decomposing the minority class into rare sub-concepts, overlapping, and presence of outliers, rare instances and noise. Our main concerns are overlapping and noise, as the discerning factors for a bug are expected to be complexly coupled with potential implementation errors regarding the features used. Standard approaches to handling noise in imbalanced data can be catastrophic [38] and one must use them with care.

1.2.4 Debugging ML Implementations

A problem we expect to see when developing the ML models is how to debug potential flaws in the ML pipeline. Borg et al. [5] note that safety engineering is lagging behind and stands in stark contrast to the pioneering spirit found in developing novel ML approaches. That being said, there is work being done on explainable AI [23, 29] and the surrounding infrastructure [32, 3]. Islam et al. investigate where, when and why bugs are introduced when deep learning algorithms are added to existing software [19].

Polyzotis et al. [28] focus on validating the input data fed into ML pipelines. Polyzotis et al. note that, although data validation is neither new nor unique to ML, there are differences that require novel solutions. Their proposed solutions include well-placed alerts and codified expectations of correct data. They also suggest using a quantifiable measure for detecting changes in distributions.

1.3 Goal

This thesis aims to create a bug prediction model and concretise a workflow for detecting where a bug in an ML implementation pipeline is located. The research questions (**RQ**) explored in this thesis are:

> **RQ 1:** *How can we use ML-based bug prediction to speed up the fault localisation provided by PinDown?*
>
> **RQ 2:** *In the case of inaccurate ML-based bug prediction, how can we support the root cause analysis?*

We can expand **RQ1** to different levels of success. If the model is exact, we could run the bug prediction model on each commit to conclude if it contains a bug or not. If we manage to build a model that, instead of being very good at predicting bugs, is very good at predicting when no bugs are present, this too has its usefulness. However, with a less precise model, perhaps we cannot be sure that a commit contains a bug or not, but we can be sure there is a bug within the five highest-ranked commits. To summarise, here are some

of the ways we can use an ML model to speed up fault localisation:: We can expand **RQ1** to different levels of success. If the model is exact, we could run the bug prediction model on each commit to conclude if it contains a bug or not. If we manage to build a model that, instead of being very good at predicting bugs, is very good at predicting when no bugs are present, this too has its usefulness. However, with a less precise model, perhaps we cannot be sure that a commit contains a bug or not, but we can be sure there is a bug within the five highest-ranked commits. To summarise, here are some of the ways we can use an ML model to speed up fault localisation:

1. Can the model be certain a revision contains a bug?
    (a) This would mean that no further verification is needed; hand out the bug report
2. Can the model be certain a revision does not contain a bug?
    (a) This would mean risk-based verification, i.e. running short test suites where no bugs are expected and longer if there is uncertainty
3. In the case of multiple revisions (a revision window/time frame):
    (a) Can the model classify the window as containing a bug or not?
        i. This would allow risk-based verification as listed above.
    (b) How many PinDown slots are needed on the "test farm" to identify the bug in the first iteration?
        i. This could support test efficiency by minimising the number of required slots on the test farm.

Risk-based verification mentioned above is the act of using the model's output to guide the size of a test suite. If the model predicts a revision window to be high-risk, a more extensive test suite is executed; conversely, a shorter test suite can be used if the revision window is low risk.

The term *slot* refers to the number of jobs launched in parallel by PinDown. The number of *iterations* required to find the bug is determined by the number of slots available and which rank the revision containing the bug received by the prediction model. If the bug is ranked third and three slots are available, PinDown will find the bug in the first iteration. If, however, only one slot is available, the same bug will require three iterations before PinDown validates the reverted commit.

1.4 Scientific Contribution

The scientific contributions of this thesis include the following :

- A proof-of-concept ML-based bug prediction model in an industrial debugging context
- A detailed workflow/method for how to debug an ML implementation

Verification is an expensive part of all software development. The fact that speedup using a bug prediction model is a viable solution, more research in industry is easier to motivate.

Chapter 2
Case Description

2.1 PinDown

PinDown is a tool that automatically debugs regression failures. PinDown does this by reverting the faulty revision and validates it by rerunning the test/s that failed to ensure that they pass again. The bugs that PinDown finds are stored in a database.

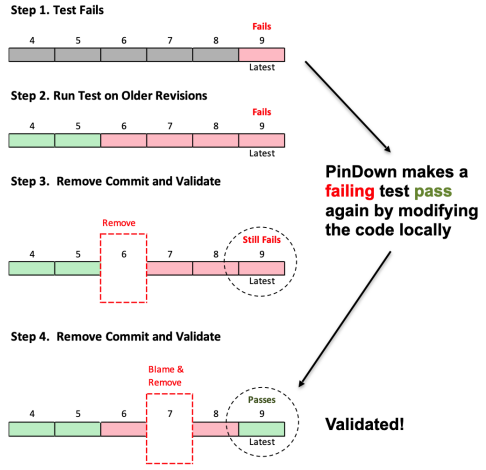
PinDown finds the faulty revision by two competing methods, a search algorithm and a trained ML algorithm. The search algorithm finds a pass/fail transition (Tipping Point), i.e. test *X* fails on revision R_6 onward, but passes on R_5 and earlier, see fig. 2.1a.

Contrary to the search algorithm, the ML algorithm does not require any test execution. The ML algorithm has a clear speed advantage since there is no test execution. However, tools such as PinDown rely on producing correct output. If the tool wrongfully blames a developer or developers must spend time questioning issued bug reports, the tool's purpose as an aide would be lost. Hence, bug predictions must always be validated by PinDown's validation mechanism. The act of validating predicted revisions (see fig 2.1b) will be referred to as *speculative validation* in this thesis.

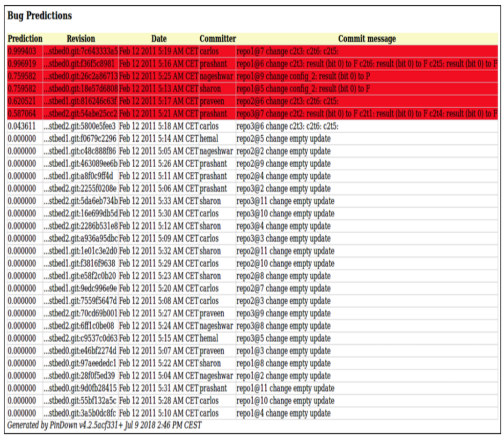
As PinDown runs multiple jobs in parallel several slots can be reserved for speculative validation. Ideally, if the prediction model is certain, more resources can be aimed at speculative validation. Conversely, if it is uncertain resources will be allocated to the search algorithm.

2.2 PinDown's Labelled dataset

PinDown's own labelling technique is based on the validated bugs. Bugs are, as mentioned, validated by locally reverting a faulty revision, i.e. automatic program repair (ARP) [11]. Some aspects of PinDown's ARP implementation and reasoning are outlined in Hansson's paper "Automatic Bug Fixing" [13]. Hansson notes that one should use ARP with caution. If the tool were to push commits to the central repository, certain unwanted scenarios might



(a) Searching for a tipping point



(b) Ranking revisions by risk

Figure 2.1

occur. These include:

- *Human - tool race conditions* – Engineers and the tool try to fix the same problem. Two fixes can be committed on top of each other.
- *Fault oscillation* – Removal of one commit fixes one test failure but makes another test fail. The tool may re-introduce that revision to make the new failing test pass, only to re-introduce the previous failure.
- *Removal of partial implementations* – A partial implementation may introduce test failures. The automatic program repair functionality could potentially back these out.

We can avoid the scenarios outlined above if the tool only makes local and limited changes.

All bugs are validated to make a failing test pass again (by its removal). No other bug prediction models in previous research were found to have been trained on validated bugs of this sort.

2.3 ASIC Verification

PinDown is used for ASIC verification, not traditional software development projects. Although the development processes are similar, some key differences are detailed below:

- Software releases – Software often go through multiple releases; bugs are found in production and patched in later releases
- ASIC Tape out – If a bug persists after tape out, the cost is of a different order of magnitude to software development. This means that ASIC developers are very keen to find all errors before tape out
- Test suite runtime – To solve the previous difference, long test suites are used.

2.4 Long Test Suites

Projects that require long test suites cannot run the entire suite on each revision. Instead, they run a short test suite before each merge and the more extended test suite at night. For the longer test suite, the general case is that multiple revisions are under test, and potential errors are dealt with the following day.

PinDown aims to run the full test suite over night and provide a validated bug report the following day so that the customers can fix bug/s first thing in the morning. For PinDown to produce a bug report early in the morning, speed is required. If an individual test takes more than four hours and PinDown requires five iterations to find the faulty revision, PinDown cannot finish debugging overnight. Hence either the number of iterations or individual test times need to be reduced. A bug prediction model can hopefully solve the first case.

```
Test branch_test4 mismatch, expected 0x42 got 0x32
Test cache_test3 timed out
Test connect_test1 axi request not valid
Test usb_test2 timed out
Test usb_test mismatch, expected 0x42 got 0x32
Test connect_test2 axi request not valid
Test dma_test1 timed out
Test connect_test3 axi request not valid
Test alu_test1 mismatch, expected 0x42 got 0x32
```

(a) Test failures and their corresponding failure signature

```
Test alu_test1 mismatch, expected 0x42 got 0x32
Test branch_test4 mismatch, expected 0x42 got 0x32
Test usb_test mismatch, expected 0x42 got 0x32

Test cache_test3 timed out
Test dma_test1 timed out
Test usb_test2 timed out

Test connect_test2 axi request not valid
Test connect_test3 axi request not valid
Test connect_test1 axi request not valid
```

(b) Same tests sorted by run time, in buckets of failure signature

Figure 2.2: Bucketization of test failures

When running a long test suite, an error is likely to cause multiple tests to fail, see fig. 2.2a. Multiple test failures do not necessarily mean that each test failure has uncovered a unique fault. To reduce time spent searching for the faulty revision, PinDown groups each test by their respective error signature (see fig 2.2b). If PinDown only selects the fastest test from each bucket when narrowing down the faulty revision, time searching minimised. When PinDown finds the faulty revision, PinDown launches the remaining tests to verify that the selected revision was the culprit. In case test failures persist, PinDown will back out a second revision; the fastest test in the remaining error message buckets is used.

PinDown users often rely on *randomised test seeds*. Randomised test seeds as used in random testing in ASIC verification is a way of reducing manual effort while increasing coverage by randomised input. Randomised test seeds open up for two scenarios, one of which is relevant for this thesis:

1. If PinDown cannot find previous test results for that specific seed, the seed has found a new coverage point.

2. If PinDown can find a historical pass for that seed, the number of revisions between a previous pass and a current fail could be considerable.

When the number of revisions under test grows, so does the time spent searching, especially when multiple revisions combined create the error. Scenario 2 mentioned above is where the bug prediction model can help the most by ranking the revisions by likelihood to cause a failure.

2.5 Revision Control Systems

A revision control system is a group of programs responsible for managing changes and hence, versions of files. PinDown uses different revision control systems in similar ways. However, Git stands out as more complicated as its history is better expressed as a graph than a linear timeline, see fig. 2.3. This is due to all the mergers and adds to the time taken by the search algorithm. Projects using Git would benefit more from the intelligent ranking by a bug prediction model.

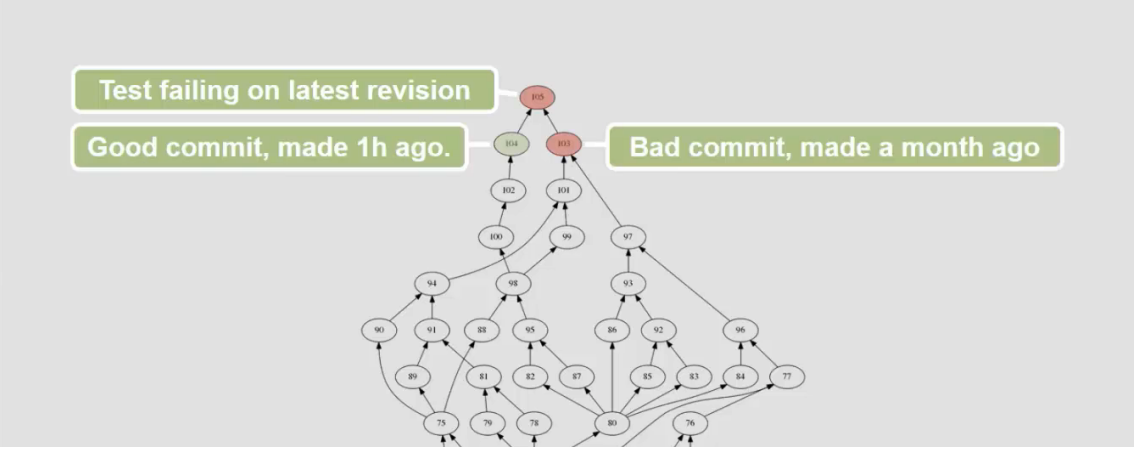


Figure 2.3: Due to all the mergers in Git the history takes shape of a graph. In this example, looking only at recent revisions would have excluded a possible fault

2.6 Why Bug Prediction?

The aspects highlighted in this section motivate why we believe a bug prediction model would be a good approach to speed up debugging with PinDown. Our unique access to a large training set is expected to be beneficial for the model's performance. Since ASIC verification requires large test suites, resulting in long test execution times to find a faulty revision, we believe a predictive ML model could outperform a conventional binary search method.

Chapter 3
Theory

This chapter covers the basic theory of the methods employed and some of the difficulties in generating the bug prediction model.

3.1 The SZZ Algorithm

The SZZ algorithm, mentioned in section 1.2, has multiple purposes in this thesis. We now take a closer look at how the implementation works.

The SZZ algorithm can either extract bug introducing revisions from a *Bug tracking system* (BTS) or by analysing the revisions history. Finding a bug introducing revisions can be difficult. Table 3.1 presents an example that may leave the reader confused as to which revision the SZZ algorithm registers as a bug. We can establish that revision 365 has fixed a problem, but was the bug introduced in rev 363 or 362? If an engineer reported an issue after revision 362 but before revision 363, the algorithm assumes that revision 363 introduced the bug. If no issue report exists, either of the revisions may have introduced the bug. Hence none are registered as the bug.

Revision	File	Lines	Commit message
Rev 365	main.java	L245	Fixed error in method
Rev 364	main.java	L120-145	Refactored graph generation
Rev 363	main.java	L240-255	Added ability to read from db
Rev 362	main.java	L240-260	Add new function to create table

Table 3.1: SZZ algorithm: Locating which revision introduced a bug

3.2 Choice of ML Algorithm

An ML algorithm is a method of building a statistical model meant to classify the input into different output groups. The input data consisting of features and labels (in the case of supervised ML, unsupervised ML is outside this thesis's scope). The ML algorithm chosen for the thesis is XGBoost[9]. XGBoost has a Java implementation available, making it easy to fit into PinDown's implementation (also developed in Java). XGBoost is a gradient boosted decision trees and has won several competitions hosted by the Kaggle data science community.

XGBoost belongs to the gradient boosting family of ML algorithms, which means that it is an ensemble of weak learners. Each learner builds upon minimising the previous learners error. The error is defined by a differentiable loss function that measures the difference between the learners output and the target output.

3.3 Feature Selection

As explained in the introduction, a feature is a numeric aspect describing a phenomenon, e.g. day and night could be described by the feature sunlight/m^2 or by time and location. The key to a well-performing ML model is not necessarily the amount of data or the fanciest of algorithms but rather the data's quality. Hence, if some features have a high correlation with each other (redundant) or are dependant on each other, are all questions that need answers. By reducing learning complexity, the hope is to minimise training time and improve performance [1, 37].

XGBoost relates the importance of each feature during training. The "feature importance" list that the XGBoost model produces is calculated for a single decision tree in the ensemble by how valuable a given feature is for a nodes split and later averaged over the ensemble [15].

We can use the feature importance list to discard the least helpful feature until an optimal set is left. Reducing features one by one may be a time-consuming technique but would yield valuable data on combinations of features.

Another feature selection technique to test is the Boruta [22] method. Boruta is a method aimed to find an *all relevant* feature set. Boruta creates shadow features by copying one features data but jumbled across rows, see table. 3.2. If the shadow feature is as useful as the original, the original feature has no predicting power. Whereas if the original feature has some predicting power, we concluded it to be relevant.

Features		Shadow Features	
F1	F2	S1	S2
a	1	b	3
b	2	a	2
c	3	c	1

Table 3.2: Boruta shadow features are copies of the real features but in randomised order

Both feature selection techniques scrutinise the usefulness of features. Given a perfect

implementation, the results can be trusted. However, keeping in mind that there might be implementation issues the features discarded can also be used as a list to take a closer look at when debugging the implementation.

3.4 Sampling Techniques

Due to the nature of the dataset, faulty commits are vastly underrepresented. Different sampling strategies must be employed to tackle this problem. An algorithm optimising accuracy tends to disregard the smaller class favouring high accuracy. E.g. if 99% of the commits contain no bugs, a classifier predicting no bugs to be present will reach an accuracy of 99%. A very high score, but entirely useless for the task at hand. One way of dealing with this problem is to employ different sampling techniques. Below we describe the techniques used in this thesis project.

SMOTE [8] is short for Synthetic Minority Oversampling TEchnique. Inspired by the perturbation of handwriting by Ha & Bunke [12], Chawla et al. devised a technique to oversample the minority class. Instead of duplicating the original sample, SMOTE creates synthetic samples. This technique may be helpful, but if noise is present among the features, SMOTE may enhance these effects and reduce the ability to predict bugs accurately.

Edited Nearest Neighbour (ENN) [41] is another sampling technique. Instead of oversampling the minority class, this is an undersampling technique. The algorithm removes instances of the majority class on the border of the minority class. Stefanowski [35] notes that undersampling techniques such as ENN help reduce noise. However, this technique may not be applicable if the number of minority class samples are too low.

3.5 Evaluating an ML Model

Standard metrics to evaluate an ML model include *accuracy*, *precision* and *recall*. In order to explain the metrics for our specific task we need to categorise the ML models' output:

	Predicted Clean	Predicted Bug
Is Clean	TN	FP
Is Bug	FN	TP

Table 3.3: Confusion Matrix definition

Accuracy is defined as $Acc = \frac{TP+TN}{TP+TN+FP+FN}$, ranging from 0.0 to 1.0. However, it is important to note that clean commits outweigh bugs by a ratio of about $1:37$ [14], meaning that predicting all commits as clean will reach an accuracy of 97%.

Recall, defined as $Recall = \frac{TP}{TP+FN}$, focuses on how many of the bugs the algorithm finds. The problem with this metric is that it disregards **FP**, i.e. clean revisions predicted as bugs. Focusing on recall as a primary metric of success could lead to a model being oversensitive, predicting too many commits as containing bugs even though they are not.

Precision, defined as $Prec = \frac{TP}{TP+FP}$, is maximised by reducing number of **FP**. Reducing false positives means that the model is more likely to be correct when predicting a bug. Con-

servative predictions come at the cost of predicting bugs as clean revisions, i.e. increasing **FN**s.

3.6 Debugging ML

A significant challenge with debugging ML implementations is the lack of deterministic output. In traditional debugging, we are (most of the time) dealing with deterministic values that we can identify as wrong with ease. ML, however, is a statistical model meant to generalise from training data and apply it to new input. One observation cannot identify the model's output as right or wrong. A single incorrect prediction may be an extreme case that we can ignore, an outlier with no discernible effect on daily performance. However, it could also indicate a fatal bug that we need to fix.

Assuming we know the prediction is wrong, the next challenge is to understand where the problem lies. Thung et al. performed an empirical study in machine learning systems [36], finding that the majority were algorithm/method related. This type of bug is beyond the scope of this thesis. The ML algorithm is assumed to function as intended. We considered it more likely that a fault exists in our implementation. Humbatova et al. analysed a large number of Deep Learning (DL) frameworks and projects, aiming to define a taxonomy of bugs [18]. Arriving at five categories:

1. **Model** - Structures and properties of a DL model
2. **Tensors & Input** - Shape of tensors and datatype issues
3. **Training** - Quality and preprocessing of training data, tuning of algorithm, choice of loss function
4. **GPU Usage** - wrong reference to GPU device, failed parallelism, etc.
5. **API** - Wrong use of APIs

Some being DL specific, our main concerns pertain to the training category. However, incorrect API calls may occur and need to be investigated where external libraries are used.

Chapter 4
Designing the Bug Prediction Model

This chapter explains the methodology chosen when addressing **RQ1** . ML development is experimental in nature and thus must be conducted in a highly iterative fashion. We follow the established methodology CRISP-DM when designing a tailored bug prediction model for PinDown.

Section 4.1 presents an overview of the CRISP-DM methodology. Section 4.2 details the feature implementation and verification steps. Section 4.3 describes how we created the dataset used for training. The next two steps are feature selection, found in section 4.4.1, and algorithm tuning, presented in section 4.5. Section 4.6 describes the model training and section 4.7 introduces our approach to evaluation and deployment. Finally, section 4.8 presents the main threats to validity.

4.1 The CRISP-DM Methodology

We created the bug prediction model by following the CRISP-DM methodology. CRISP-DM, cross-industry standard process for data mining, is a methodology devised under ESPRIT (European Strategic Programme on Research in Information Technology) funding.

The CRISP-DM methodology consist of six steps, see fig. 4.1. Next we will briefly explain the steps of the methodology.

1. Business Understanding

Step 1 aims at getting a broad understanding of the problem at hand. Understanding is gained by formulating the success criteria and setting up the business and data mining objectives. Step 1 concluded by formulating **RQ1** in Section 1.3.

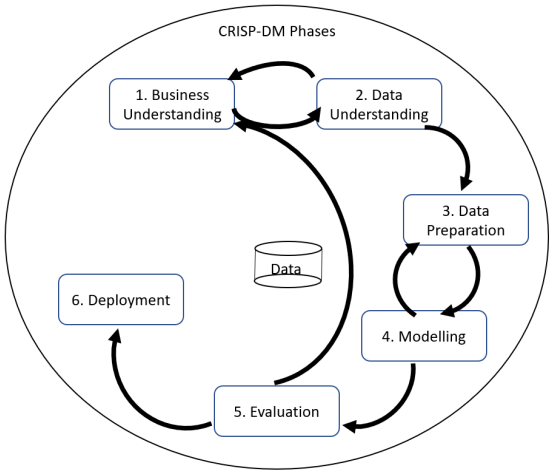


Figure 4.1: CRISP-DM methodology, adopted from a comparative study on data mining methodologies [33]

2. Data Understanding

Step 2 aims to collect knowledge regarding the data, i.e. analysing quality of the data, and forming hypothesis regarding extracting hidden concepts. This CRISP-DM step includes verifying the quality of the candidate features, which is found in subsection 4.2.5.

3. Data Preparation

Step 3 includes various tasks to prepare the dataset used when modelling. This includes cleaning the dataset, possible transformation of features, and feature selection. Step 3 includes labelling the dataset (section 4.3), cleaning the data (section 4.4) and selecting features for training (subsection 4.4.1).

4. Modelling

Step 4 involves creating the model. Creating the model entails testing various modelling techniques and different parameters. This step is described in sections 4.5 and 4.6.

5. Evaluation

Step 5 is an evaluation of the previously made models. Reviewing whether the objectives can be met and how well they conform to the set goals. Step 5, evaluation of the models, is done two-fold as described in subsection 4.7. Once during training and again when deployed in customer projects. The results from the two evaluation phases are reported in sections 5.1 and 5.2, respectively.

6. Deployment

Step 6 concerns itself with gathering information, reporting and presenting knowledge gained through the processes. How we deploy models in shadow mode in customer projects is briefly

explained in section 4.7.

4.2 Implementing and Verifying Features

The second step of CRISP-DM, Data Understanding, had already been started at Verifyter before this thesis project. Inspired by research papers from the MSR conference [25], Verifyter had prototyped extraction of features from source code repositories reported in previous work on bug prediction.

The features implemented can be divided into four categories. Author features, revision features, code features and project/historical features. Each feature aims to capture any subtle sign of risk. Many of the implemented features are related, containing different ways of presenting similar information.

The set of features used in the end are initially unknown. They are chosen by how useful they are for the ML algorithm ability to classify the commits. Building on Verifyter's preliminary work, we continued by implementing as many additional features as possible to explore which of them provided the highest predictive power. Below we present an overview of the four feature categories. See appendix B for a selection of the features implemented.

4.2.1 Author Features

This set of features pertain to information derived from the committers' history and patterns. In total, there are around 15 author feature implemented (samples found in appendix B). Instead of giving a complete list of features for the reader to interpret, these are some of the questions these features aim to illuminate:

- Are the files in the revision often edited by the committer?
- How active has the committer been recently?
- Are the files largely "owned" by the committer?
- What is the authors' relative experience with these files?
- How long has the author been involved with these files?

4.2.2 Revision Features

These features aim to capture information from the revision control system, relationships that cannot be inferred from individual source code files. *Folder level* mentioned below is defined as folder level 1 being the checkout root. The questions intended to be answered by these features include:

- How central to the structure are the files in the revision?
- What folder level and how often is that level revised?
- Are files added, deleted or changed?

- What weekday/time/timezone (timezone can capture multi-site projects)?
- What is the commit message length and does it contain risky words?
- Is the revision a standard/merge/merge-resolve revision (git specific)?

4.2.3 Code Features

These features are concerned with the code contained in the commit. Close to 40 features fall under this category.

- How many lines of code are added, changed and deleted?
- What is the complexity of the code?
- Does it contain comments?
- What language was used?

4.2.4 Project/Historical Features

These features are rather specific to a project and may not generalise across companies or even different projects. There are about 30 features of this category.

- How many bugs have been found in each folder level?
- How many files are contained in each folder level?

4.2.5 Verifying Feature Implementation

Verifyter has a test suite running regression tests on PinDown to verify that it can solve tricky scenarios that might happen when running live on customers' test farms. Although these tests were designed for the search algorithm, they contain all the information needed to generate training data (done over the entire test suite) and the live feature sets (treating the current tests as the revisions to be predicted on, and older tests as the history leading up to this point). We can verify that features have been implemented correctly by manually calculating the results for each test. This is only possible when we have full access to the revision control system. See section 6 on how to handle implementation issues when only the ML models generated features are at hand without full access to the source (revision control system in this case).

4.3 Creating the Datasets

The third phase in CRISP-DM, Data Preparation, is concerned with creating the dataset. This includes cleaning and possibly transforming features. The dataset is initially created by the customer running PinDown's generate training data command for a specified time

period, e.g. *set_ml − option*"*trainingdata*" − *value*"**5M**"; (5M = 5 Months). If PinDown has been running on the project for a while, PinDown will have saved all the validated bugs in a database to be extracted. For projects that haven't been running PinDown, another method to extract bugs is utilised, namely the SZZ algorithm [34]. Verifyter has already explored the SZZ algorithm in previous work [6].

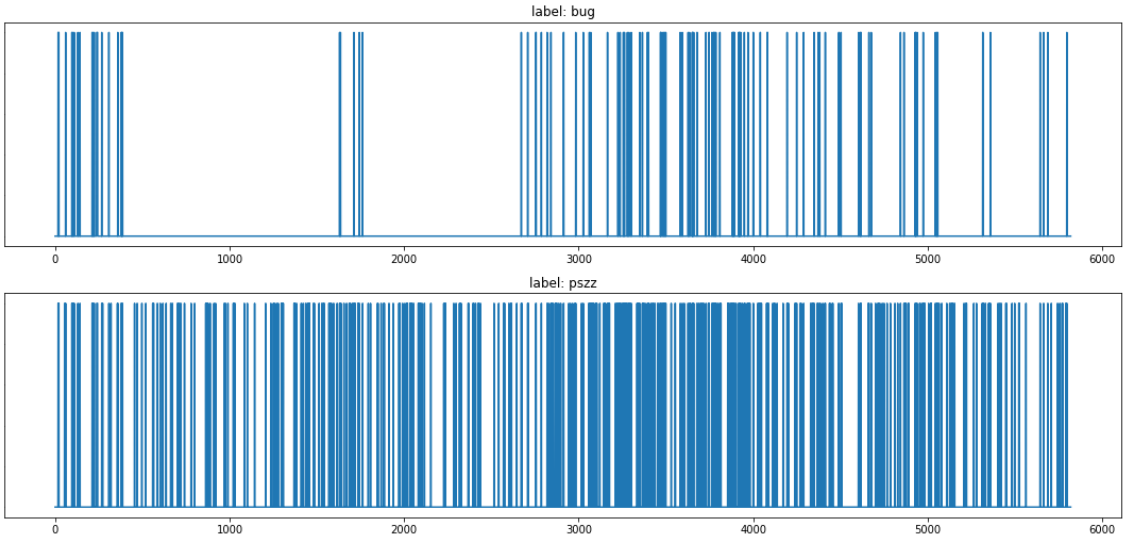


Figure 4.2: Two approaches to labelling the dataset for the same time period. Each number on the x-axis constitutes a revision and each vertical line being a revision labelled as bug. The graph above showing a lower frequency and noticeable gaps (missing labels).

As seen in fig. 4.2, there is a discrepancy between the commits labelled bugs by the two different methods. PinDown bugs (above in fig 4.2) are validated; we are confident those labelled as bugs are correct. However, during periods when PinDown has not been running, there are gaps. If Pindown has not been active, it poses a problem because bugs might be mislabelled as clean in these gaps. To reiterate, PinDown marks the revisions containing bugs in a database as True (containing bugs). PinDown does not label the remaining revisions as False (not having bugs). These revisions are not labelled at all. There is no way of discerning whether a revision without a label is due to it not containing a bug or if PinDown has not investigated that revision. A better approach would have been to mark bugs as True and revisions that have been tested and found not to contain bugs as False. If any are left, leave those without a label. This approach was implemented, but we did not have enough calendar time to collect sufficient amounts of results to report the outcome in this thesis.

When looking at the labels generated by the SZZ algorithm (the lower subplot in fig. 4.2), we see a much higher density and smaller gaps. The SZZ algorithm has been shown to mislabel bugs [17] but has the upside of generating bug labels for projects where PinDown has not been used (PinDown bugs are created over time, whereas SZZ bugs can be mined after-the-fact).

Based on the above reasoning, we identify four approaches to prepare the training data before step 4 in the CRISP-DM methodology. The first approach means relying on the PinDown labels, whereas the other three approaches uses the output from the SZZ algorithm in different ways to analyze the training data.

4.3.1 Approach 1: Leave Dataset Untreated

Leave untreated means not augmenting the PinDown bug labels at all. All positive examples are correct, but the negative examples might have the wrong label. There is a risk of having chunks of commits incorrectly labelled as clean, i.e. false negatives (potentially present in the big gaps in the upper part of fig. 4.2). There could be several explanations for this phenomenon. PinDown might have been disabled, the management may have decided to move development to a different project, there might have been vacation period or the project could be far from any deadlines (the importance of debugging is low). Training bug prediction models with false negatives would likely be detrimental to the ML algorithms ability to generalise.

4.3.2 Approach 2: Bug Frequency Analysis

The large gaps in the upper part of fig. 4.2 looks suspicious. Is it possible that hundreds of sequential revisions were all clean? We argue that the SZZ algortihm can be used to analyse these areas. The lower part of fig. 4.2 shows that the SZZ algorithm reports roughly the same bug frequency during the entire time window. There appears to be nothing special during the large gaps.

As explained in Approach 1, false negatives in the training data would reduces the ML models ability to generalise. A simple approach to clean the dataset is to remove the segments containing the large gaps. We achieve this by identifying periods without gaps. For these periods, we can calculate the average bug frequency. We then use this bug frequency to identify segments that are likely to contain false positives.

Compounded probability arrives at a 95% chance that an unlabelled set of 110 revisions contains a bug if the frequency is one bug every 37 revisions. In this instance, we can use the SZZ algorithm to identify if this region is dissimilar from its immediate surrounding revisions. If the bug frequency is similar to its surroundings, then the likelihood that PinDown has missed labelling bugs is considered high. Note that we do not expect the bug frequency to be static over a project life cycle. That is why the alternate labelling technique is used to corroborate or discredit the bug frequency in these regions (see fig. 4.3).

The following five steps summarise Approach 2:

1. Identify regions without obvious gaps
2. Calculate bug frequency for these regions
3. Cross-reference empty regions with SZZ algorithms bug frequency
4. Step backwards in time
5. If a bug is followed by unlabelled revisions from a period longer than 110 revisions, remove until next bug

4.3.3 Approach 3: Misclassified Samples – Strict

Approach 3 utilises ML algorithm predictions. As explained in Approach 1, false negatives in the dataset could end up in both the training and validation sets during ML development.

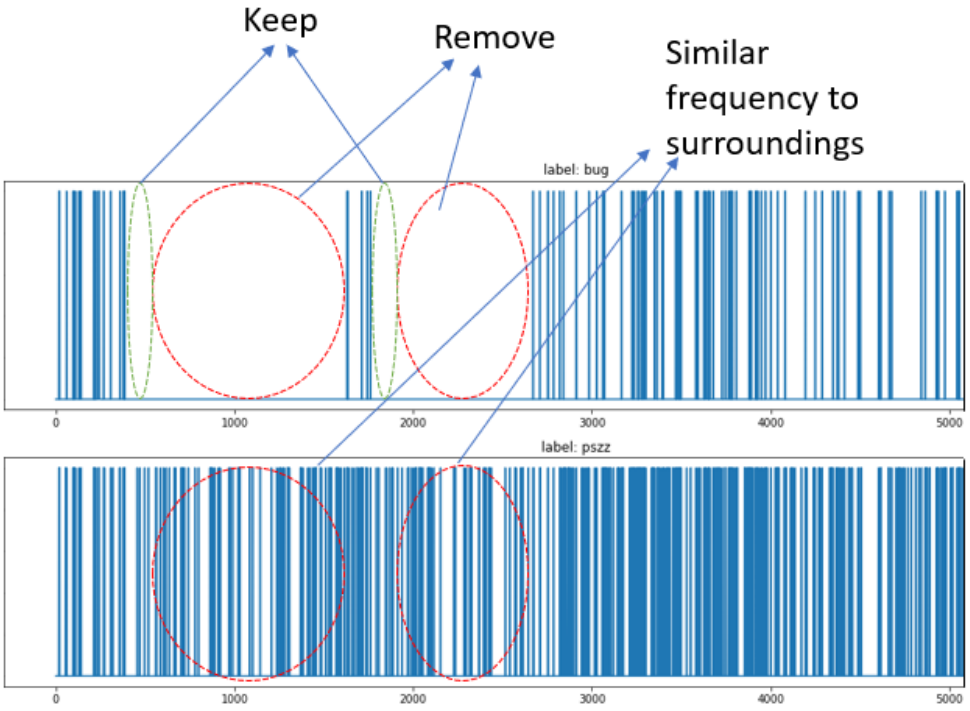


Figure 4.3: Removal of gaps in labelled dataset

Suppose specific revisions are more frequently misclassified, by the ML model, as bugs than others and also happen to be within an unlabelled gap. In that case, these revisions might contain false negatives. To further strengthen this hypothesis, if the SZZ algorithm labelled these revisions as bugs, it might help to remove them from the dataset altogether.

The following four steps summarise Approach 3:

1. Train a classifier using Approach 1 with multiple split seeds
2. Save the number of times each revision has been misclassified
3. If a revision is misclassified more often than the average and coincides with an unlabelled region, cross-reference with the label obtained by the SZZ algorithm
4. If, as opposed to PinDown's label, the SZZ algorithm has labelled this revision as a bug, remove this revision from the dataset

4.3.4 Approach 4: Misclassified Samples – Relaxed

Approach 4 is a more relaxed version of Approach 3. Instead of cross-referencing with the SZZ algorithm, we propose removing all samples within unlabelled regions that are misclassified more often than average.

4.3.5 Which Method to Use?

We regard Approaches 1 and 2 as the safest and will use them in the remainder of the thesis. For a project that has been monitored over a long time, the bug frequency can be established within reasonable limits. Fig. 4.4 shows the dataset labelled using Approach 1 (upper part) and cleaned using Approach 2 (lower part). Approach 2 reduces the size of the dataset with roughly 40%, but the dataset is less likely to contain false positives.

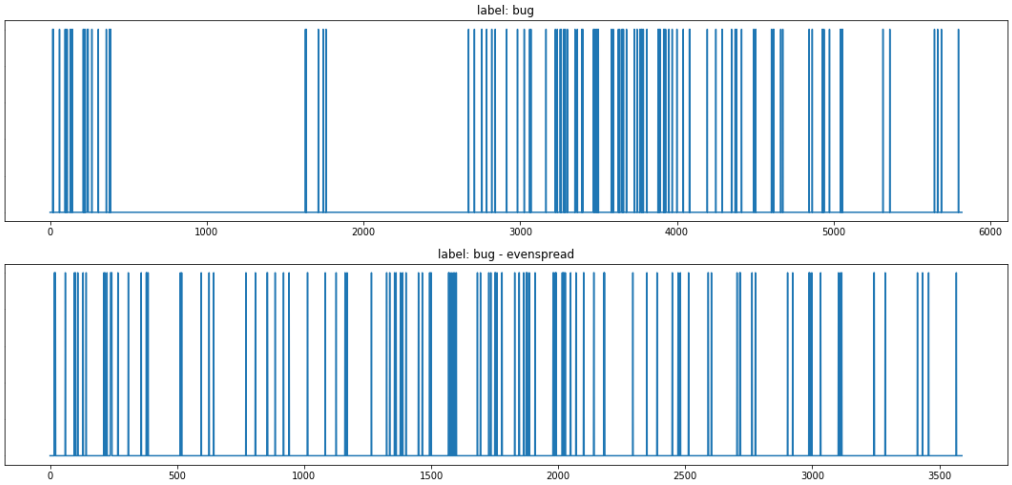


Figure 4.4: Treated version of the original (upper) dataset show below: 2,230 of almost 6,000 commits have been removed.

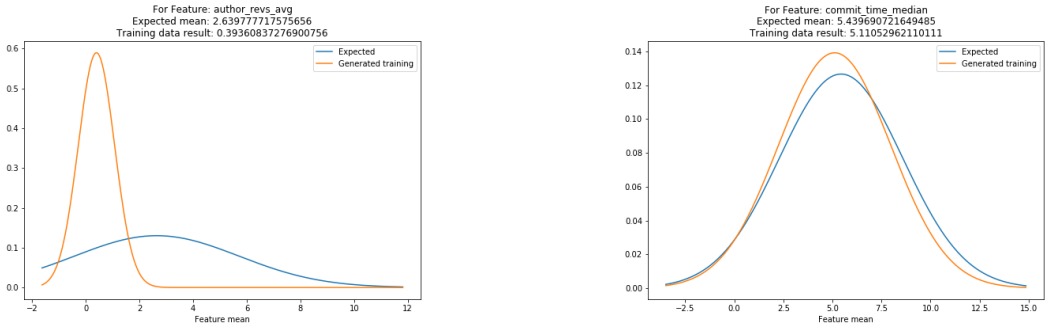
4.4 The Features

We have mainly focused on the labels in the dataset until now. Although an essential part of the dataset, the remaining features on which the ML model is trained also require analysis and cleaning, as per step 3 in CRISP-DM. Due to the fact that some features are time-dependent, we have had to implement scaling when gathering the training data (which is generally extracted from historical data 6-12 months back in time). We compare a given feature on live customer data vs. historical training data distributions for these features to verify that they behave as expected during live inference. If they are too different, see fig 4.5a, there may be an implementation error, or the feature needs to be redesigned.

In order to verify feature implementation, the results from PinDown's own testbed is used. We manually calculated the features for each test and then ran the tests to verify that the generated data is identical.

4.4.1 Feature Selection

Feature selection, still a part of step 3 in CRISP-DM, can be tackled using different techniques. This thesis focused on utilising XGBoosts feature importance and the Boruta algorithm. The Boruta algorithm, as explained in the theory section 3.3, creates shadow features



(a) Unacceptable difference in distribution

(b) acceptable distributions

Figure 4.5: While creating the dataset, some features need to be scaled in order to correctly simulate live conditions. Here is an example of incorrect implementation for the feature author_revs_avg

by copying feature data and randomising the order. If the feature does not exhibit significantly better predicting power than its shadow feature, it can be deemed irrelevant. Boruta aims to find an "all-relevant" feature subset.

Feature selection by brute force is an iterative process of removing the least essential feature until none are left. In fig. 4.6, we can see the x-axis showing the number of features used during training and the y-axis showing the validation set's precision. The precision appears to converge with roughly five selected features.

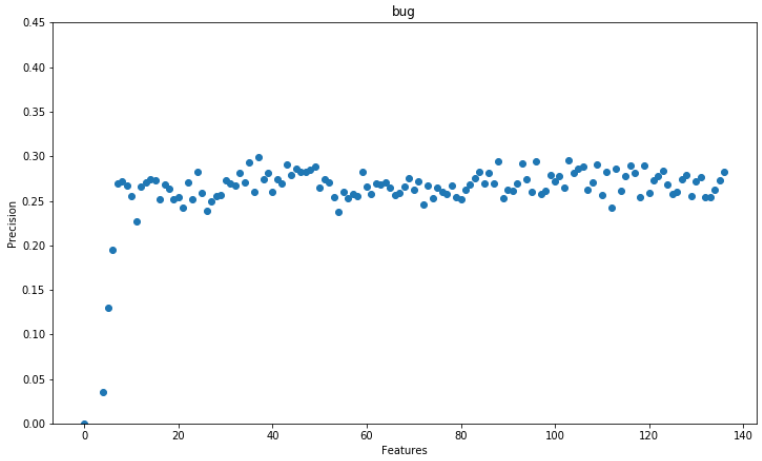


Figure 4.6: Precision vs. number of features

In fig. 4.7, the rightmost scale shows the colour coding for displaying correlation to the training labels (bug & bug_pszz), light colours indicate high correlation and darker indicates low correlation. When analysing the correlation between features and the labels used in training we see that none correlate strongly, hence they are only weak indicators of bugs. This, in turn, means that a subset cannot be reduced to just a few features. Hence we need to analyse the inter-feature correlations to, instead of identifying the good features, identify those that correlate with each other. The optimal subset of features is minimal and contains

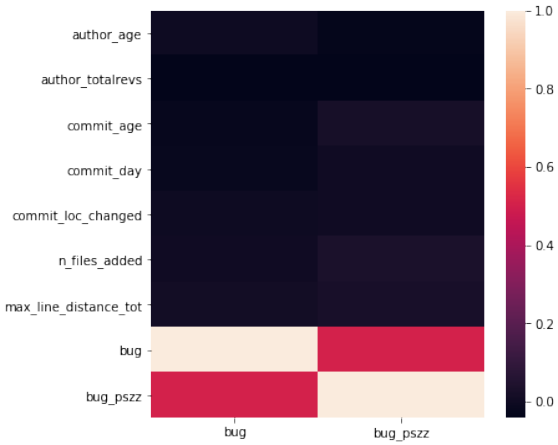


Figure 4.7: Feature correlation to labels in the dataset

those that have little correlation with each other.

4.5 Algorithm Tuning

The tunable hyperparameters and description [43] for XGBoost are found in appendix.A. The method used for tuning hyperparameters is GridSearchCV [31] as an exhaustive search given a set of inputs for each parameter, as seen in table 4.1.

Hyperparameter	Range
Learning Rate	[0.01, 0.05, 0.1, 1]
Max Depth	[3, 4, 5, 6, 8, 10, 15, 20]
Subsample	[0.2, 0.4, 0.6, 0.7, 0.8, 0.9]
Alpha	[0, 0.2, 0.4, 0.8]

Table 4.1: A selection of the tunable hyperparameters found in XGBoost and the ranges used with GridsearchCV. Remaining hyperparameters and their meaning are found in appendix A

In accordance with the feature selection process proposed above, using the feature importance list from XGBoost, the hyperparameters are tuned before the selection starts. Tuning is done again once a feature subset is found to ensure that the correct hyperparameters are chosen or changed to yield better results.

4.6 Model Training

Training the bug prediction model, in reality, means splitting the customer data into three groups: *training set*, *test set* and *validation set*. Instead of using the same split each time, randomised seeds can be used to ensure different combinations. During training, the ML algorithm is exposed to the training and test dataset. The training set has been subject to a sampling method explained in section 3.4. Each generation is evaluated against the *test*

set. When the training phase is complete, we validate the model with the *validation set*. The predictions produced on the validation set are compared to the ground truth, and the result is displayed in a *Confusion Matrix* as seen in table 3.3.

Due to the bug prediction's imbalanced nature, the training, test and validation split leave very few bugs in the validation set. We noted that rerunning the training step yielded varying results on different split seeds. The varying results can be due to several reasons, one hypothesis being that bugs are not always of the same type. The varying validation scores have led us to, for each model, retrain 1,000 times with different split seeds so that we can evaluate the mean precision and a standard deviation of the precision. A setup generating models with good mean precision coupled with low standard deviation deemed more beneficial than erratic models with higher precision and more significant standard deviation.

PinDown compiles all features regardless of whether the model uses them or not. Since there is no noticeable extra resource cost if more than one model produces predictions, one model can act as a primary model. An additional set of models can be deployed alongside the primary, producing their predictions without consequence on the debugging process. These additional models act in a so-called shadow mode. The results evaluate different techniques, subsets of features, hyperparameters, progress and regression bugs.

4.7 Evaluation and Deployment

First, we evaluate the bug prediction models both during training. Second, we evaluate the models, as per step 5 in CRISP-DM, when the models are running live at Verifyter's customer projects. To allow quantitative evaluation, we need a method for retrieving statistics. Our main measurement instrument is bug reports. These bug reports contain a section of information about the predictions. See the fig. 4.8 below. As we evaluate the models using live data, by running them in shadow mode in customer projects, we consider the models to be deployed.

4.8 Threats to Validity

This section reports what we believe are the biggest threats to the validity of our conclusions and how we mitigate them.

4.8.1 Generalisation – Number of Projects Studied

During the period of deployment, the project used for training was not active, which means that prediction data for that project could not be obtained. This limits analysis regarding how general the bug prediction models are. All data presented in results (section 5.2) is from a single company. If a comparison could be made between the project used for training and other live projects or other companies projects, a stronger case for generalisation could be made.

As **RQ1** is concerned with the speed of fault localisation, one could argue that time spent developing the ML-models should be factored in. If each project requires its own analysis and

```
<!--
Performance Stats.  1   2  3   4    5
pred-main bb.smo.avg.nn 1/10 LR1/1  LP1/1, 0.5765/0.5765;
pred-extra bb.smo.awe.nn 1/10 LR1/1  LP1/1, 0.6594/0.6594;
pred-extra bpbp.smo.avg.nn 5/10 LR1/1  LP1/1, 0.0792/0.7757;
pred-extra bb.enn.avg.wofl 7/10 LR1/1  LP1/1, 0.0533/0.0746;
pred-extra bb.enn.awe.wofl 3/10 LR1/1  LP1/1, 0.0686/0.0831;
pred-extra bb.enn.avg.woflrem 3/10 LR1/1  LP1/1, 0.1395/0.1768;
pred-extra bb.enn.avg.redFeats 3/10 LR1/1  LP1/1, 0.2326/0.4279;
pred-extra bb.enn.awe.redFeats 6/10 LR1/1  LP1/1, 0.1153/0.1655;
-->
Validated in iteration 2, pindownID1   6
diagnosisCause = Validating that pilot by undoing the faulty
                 commit on head as a patch
    With this patch. 23418441 -> 23412800
validatedPilotByUnrollingBadCommit = true
validatingHead = true
speculative = false   7
findMinPatch = false
lastMergeWithPass = false
historicalCause = false
```

Figure 4.8: 1) Name of the model. 2) x/y, x = after validation, what rank the bug got, y = number of revisions predicted on. 3) x/y, x = Same as previous, y = revisions since Last Run. 4) x/y, x = Same as previous, y = revisions since Last Pass. 5) x/y, x= prediction score for bug, y = highest prediction of the revisions predicted on. 6) number of iterations until bug was found. 7) if the problem was solved speculatively, i.e. by prediction

preparation for ML models, a required data point would be the above-mentioned comparison between project/company interchangeability.

4.8.2 Project Similarity

As mentioned above, different projects were predicted on in the data collection phase than the one trained on. An attentive reader might note that although differences are mentioned, just how different are they? What are the discerning factors between projects through the eyes of PinDown, and what would generally be regarded as differences in ASIC design projects lends itself to discussion. The lack of an established measure for project similarity comparisons could threaten the validity of the results. More research is needed to study the potential impact.

4.8.3 SZZ Algorithm – Differences in Implementations

The SZZ algorithm is used throughout the thesis. However, there are different implementations available. An evaluation was recently published [30], that also set out to create a framework for comparison. If another implementation were used, would the results have been different? Exploring this would however require more implementation than time and resources allow for during the thesis.

Chapter 5
Evaluating the Bug Prediction Model

The possibilities of a precise bug prediction model are endless. The model's prediction value can, not only, be used to predict where the bug is, but also classify the risk of the revisions under test. Lets revise what possibilities and consequences answering **RQ1** has:

1. Can the model be certain a revision contains a bug?
    (a) This would mean that no further verification is needed; hand out the bug report
2. Can the model be certain a revision does not contain a bug?
    (a) This would mean risk-based verification, i.e. running short test suites where no bugs are expected and longer if there is uncertainty
3. In the case of multiple revisions (a revision window/time frame):
    (a) Can the model classify the window as containing a bug or not?
        i. This would allow risk-based verification as listed above.
    (b) How many PinDown slots are needed on the "test farm" to identify the bug in the first iteration?
        i. This could support test efficiency by minimising the number of required slots on the test farm.

We are certain that 1., mentioned above, will not be fulfilled, at least one iteration of PinDown validation will be required. That is to say, a bug report will not be sent before PinDown has removed the proposed revision and rerun the tests to validate that they pass. In this chapter, we report the results that motivate our conclusion. First, we present an evaluation of the models during training. Second, we present results from live inference in customer projects.

5.1 Model Training Results

Table 5.1 shows the models' results after 1,000 different randomised split seeds. The table presents eight different models and their corresponding precision and recall. Standard deviations (std) are also reported.

Model Name	Precision	Precision std	Recall	Recall std
bb.smo.avg.nn	0.303	0.082	0.076	0.044
bb.smo.awe.nn	0.303	0.082	0.076	0.044
bpbp.smo.avg.nn	0.393	0.052	0.068	0.069
bb.enn.avg.wofl	0.309	0.094	0.078	0.043
bb.enn.awe.wofl	0.309	0.094	0.078	0.043
bb.enn.avg.woflrem	0.345	0.087	0.100	0.069
bb.enn.avg.redFeats	0.305	0.097	0.074	0.041
bb.enn.awe.redFeats	0.305	0.097	0.074	0.041

Table 5.1: Model training summary

Table 5.1 needs some further explanation. The names of each model are split in 4 parts separated by a period.

- *bb* or *bpbp* – training & validation label, bb means trained and validated on PinDown bugs, whereas bpbp is trained and validated on SZZ-generated bugs.
- *smo* or *enn* – sampling technique used, either SMOTE or ENN
- *avg* or awe – As running 1,000 seeds generates 1,000 models, how do you select one of those? avg means the selected model is close to the average precision, whereas awe is a model that preformed better than average on the validation set.
- *nn*, *wofl*, *woflrem* or *redFeats* – any directed feature experimentation. nn denotes no manipulation, wofl signifies without folderlevel features, woflrem without folderlevel features and removed gaps. redFeats is a reduced feature set, i.e. less than 20 features.

We packaged all models together to gather data on their performance. The model we selected as primary was chosen from the bb.smo.avg.nn training round. For the primary model, we wanted minimum manipulation to the feature set chosen, which narrowed it down to the "nn" models. Although the "bpbp" model performed well during validation, "bb" was chosen as these are the true bugs we expect to find when running live. The model's validation results can be seen in table 5.2 below.

	Predicted Clean	Predicted Bug
Is Clean	844	14
Is Bug	10	5

Table 5.2: Confusion Matrix: Validation set for main model

If the validation results would have had zero **FP**s we could be confident that when the model predicts a revision to be a bug, the revision contains a bug. Conversely, if the validation

results had zero **FN**s, we would expect a predicted clean revision to be clean. As neither is the case during validation, we cannot expect the model to perform with better certainty when deployed in the real operational environment.

Although the prediction model and the search algorithm work in tandem, we do not want the model to be too sensitive. Reducing sensitivity means that we prioritise precision over recall, i.e. minimising **FP** at the cost of missing bugs. This trade-off stems from the reasoning that if the model is unsure, hopefully, all revisions will be given low prediction scores, prompting resource allocation to favour the search algorithm.

5.2 Live Inference Results

This section recounts the prediction results for the models deployed in customers ASIC development projects. We also discuss what knowledge we can gain from the results.

Important note regarding the data gathering: The project used when gathering training data was not active when we deployed the models. This limits analysis regarding how project-dependent the models are, which was the objective with the *wofl* models.

After deployment, the models produced predictions as soon as a test failed, and PinDown started its debugging process. In this section, we will show several plots organised as fig. 5.1 reporting prediction results on 14 real customer bugs. Along the leftmost y-axis, we see, in red text, "Occurrences". The bars in this graph represent the number of bugs or occurrences for each given rank displayed along the x-axis. In blue on the rightmost y-axis, we see the "prediction " or prediction value also displayed along with the rank that prediction got.

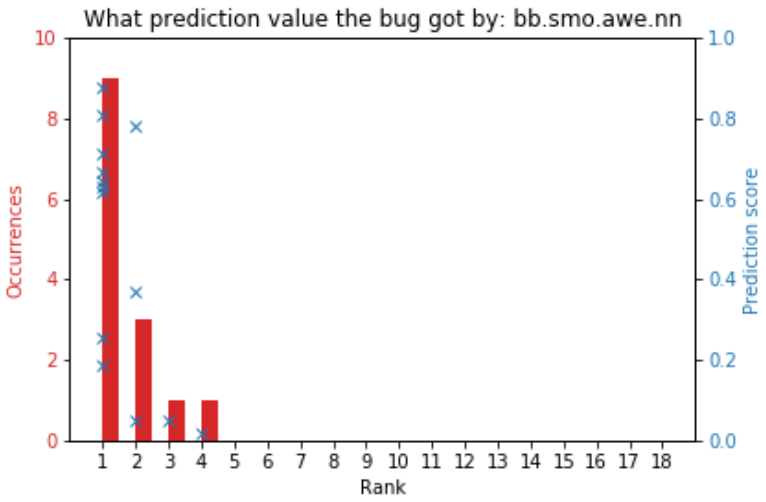


Figure 5.1: Ranks for each bug as well as their prediction value

Fig. 5.1 details the prediction results of model bb.smo.avg.nn. The majority of the bugs (9/14) were ranked first. PinDown would hence have found them in the first iteration. If we look at the prediction scores, we also note that rank 1 and 2 have much higher predictions than rank 3 and 4.

Fig. 5.2 shows the prediction results for model bb.enn.awe.redFeats. A majority of the bugs are ranked third or lower. There is another peculiar finding in this graph. The bugs

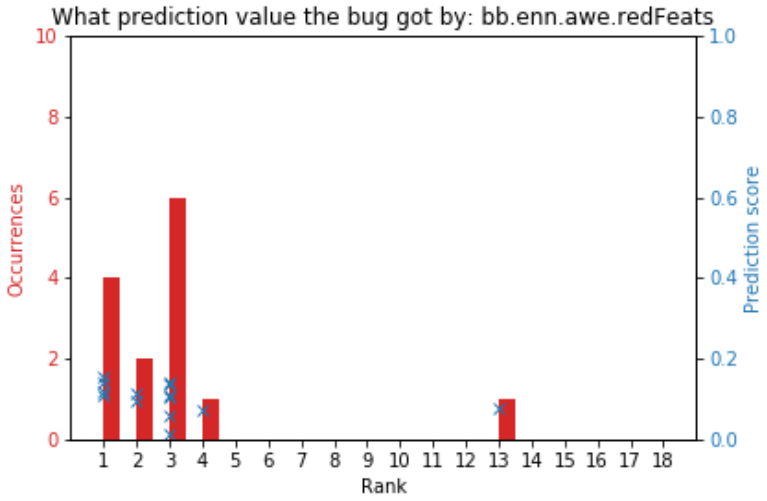


Figure 5.2: Ranks for each bug as well as their prediction value

ranked first seem to have very low prediction scores, almost on par with the other ranks. The bugs ranked fourth and 13th have lower prediction scores than the other revisions in that revision window. That is no surprise, but what do we expect to see if we compare the top prediction in each window against the bugs' prediction values?

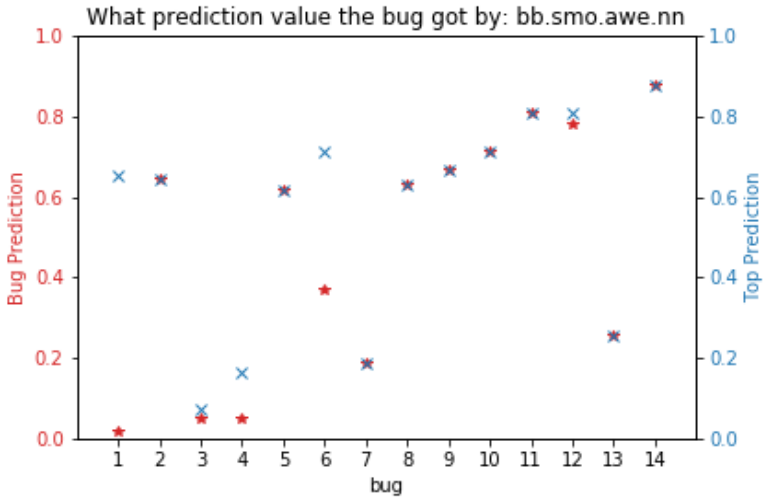


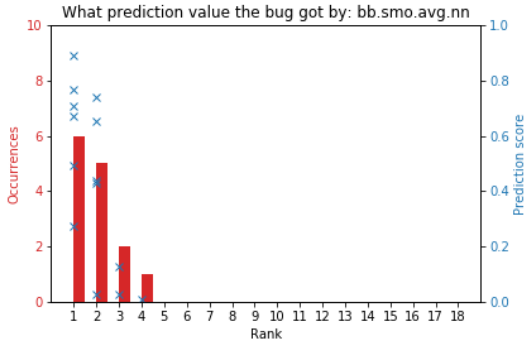
Figure 5.3: Top prediction (blue) in debug window as well as the bugs' actual prediction value (red)

Fig. 5.3 details the highest prediction value in each bug's revision window (blue "x") as well as the bug's prediction value (red "*"). There were two types of results we were hoping to see in this graph:

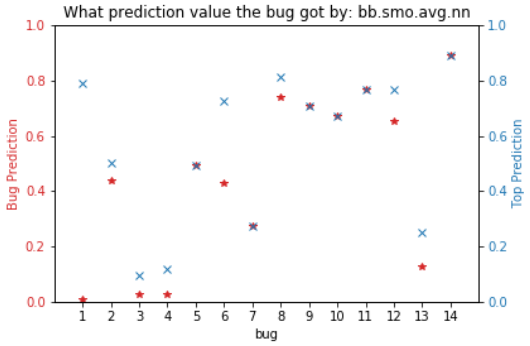
- All high predictions – If all bug windows received high predictions, we could cross reference the results with debug windows that do not contain bugs. This would result in a model that can classify a risky revision window. Bugs number four and 13 contradict this hope.

- Top prediction and bug prediction closely related – this would allow for dynamic resource allocation between speculative validation and the search algorithm. Bugs number one and six contradict this hope.

With the graphs' layouts explained, we can see the results from all eight models in table 5.1 side by side in fig. 5.4 - 5.11.

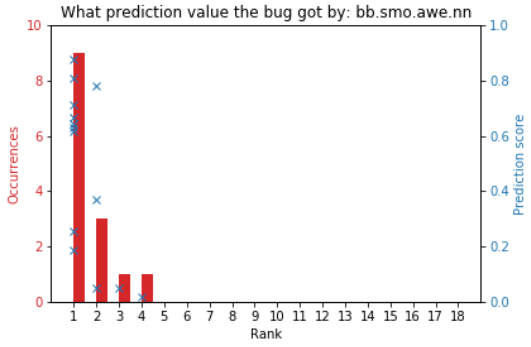


(a) Ranks for each bug as well as their prediction values

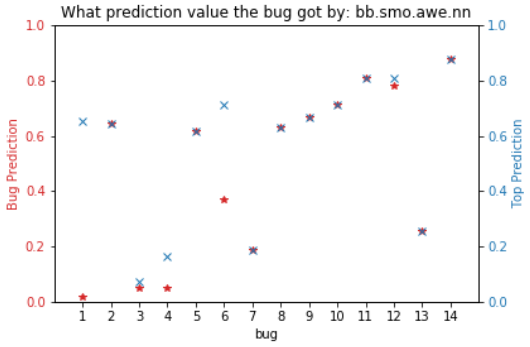


(b) Top prediction (blue) in debug window as well as the bugs' actual prediction value (red)

Figure 5.4: The model deployed as primary model

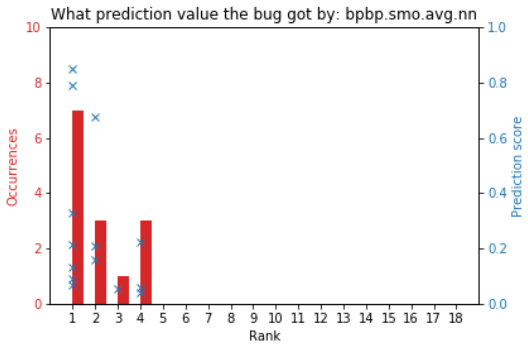


(a) Ranks for each bug as well as their prediction values

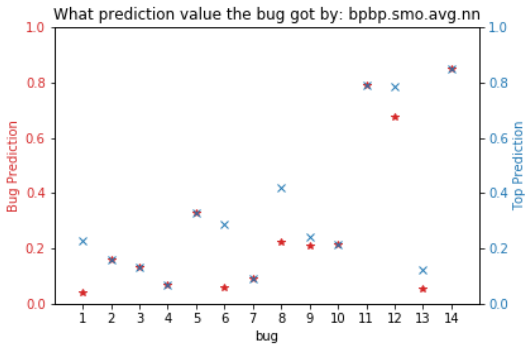


(b) Top prediction (blue) in debug window as well as the bugs' actual prediction value (red)

Figure 5.5: Shadow model A

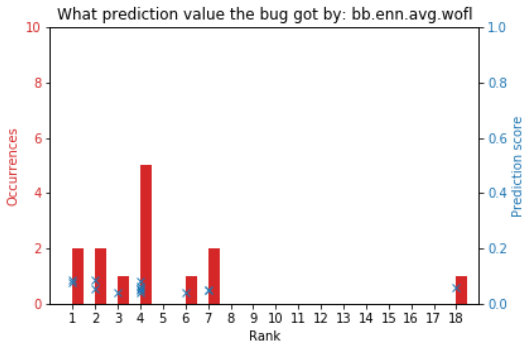


(a) Ranks for each bug as well as their prediction values

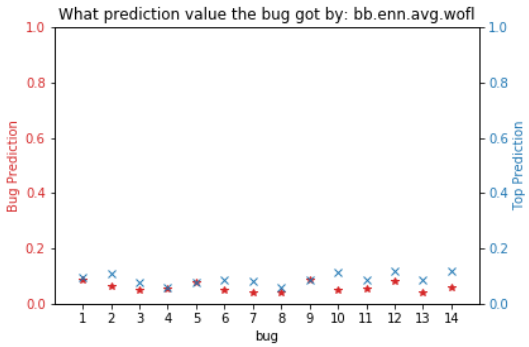


(b) Top prediction (blue) in debug window as well as the bugs actual prediction value (red)

Figure 5.6: Shadow model B

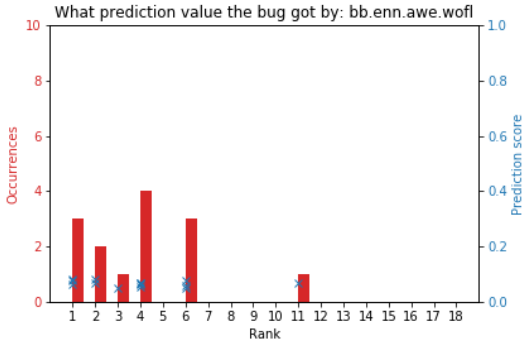


(a) Ranks for each bug as well as their prediction values

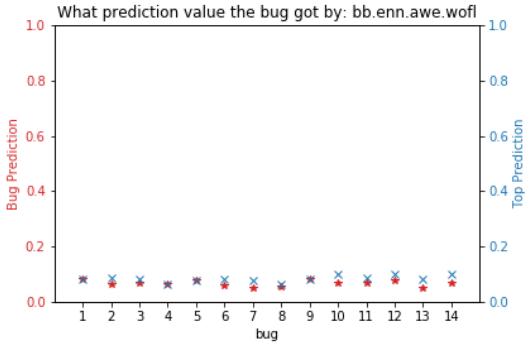


(b) Top prediction (blue) in debug window as well as the bugs actual prediction value (red)

Figure 5.7: Shadow model C

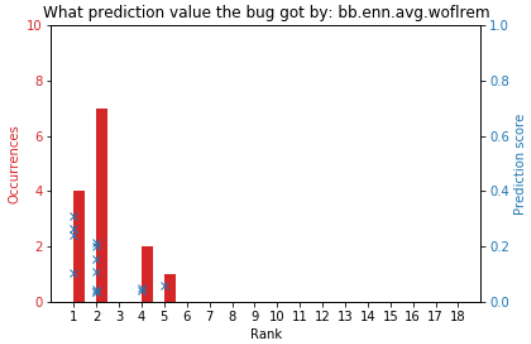


(a) Ranks for each bug as well as their prediction values

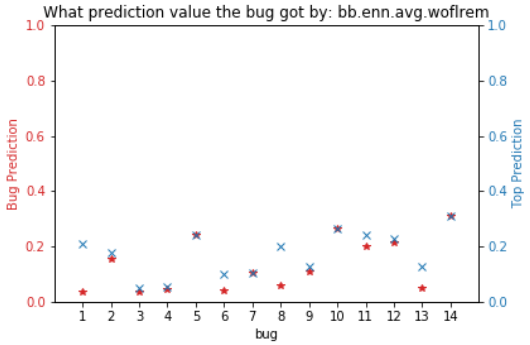


(b) Top prediction (blue) in debug window as well as the bugs actual prediction value (red)

Figure 5.8: Shadow model D

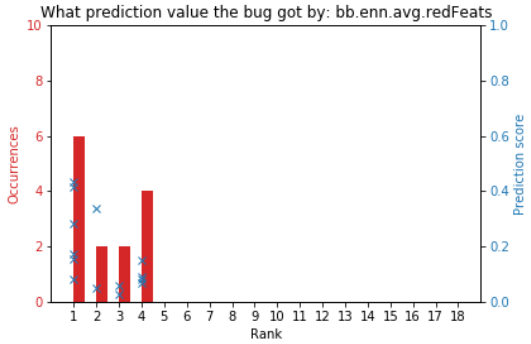


(a) Ranks for each bug as well as their prediction values

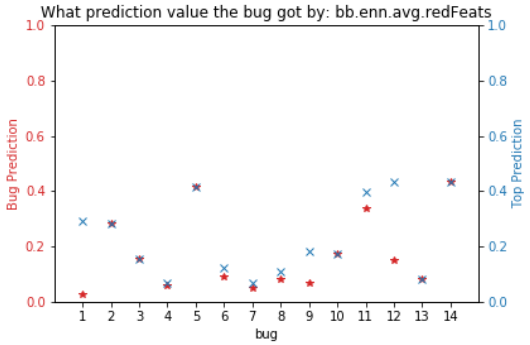


(b) Top prediction (blue) in debug window as well as the bugs actual prediction value (red)

Figure 5.9: Shadow model E

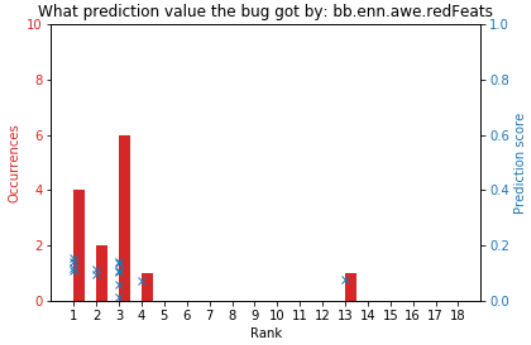


(a) Ranks for each bug as well as their prediction values

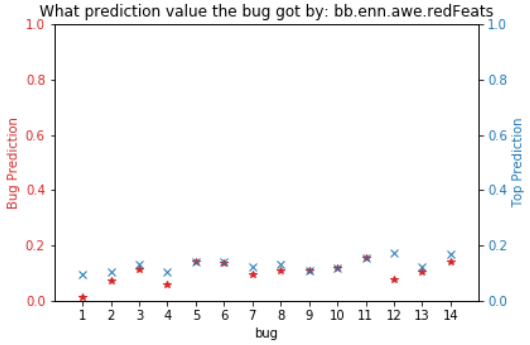


(b) Top prediction (blue) in debug window as well as the bugs actual prediction value (red)

Figure 5.10: Shadow model F



(a) Ranks for each bug as well as their prediction values



(b) Top prediction (blue) in debug window as well as the bugs actual prediction value (red)

Figure 5.11: Shadow model G

For the first three models, fig. 5.4, 5.5 & 5.6, we see that the majority of bugs are ranked first and second. The remaining models (fig. 5.8 - fig. 5.11) were trained using the sampling technique ENN. They showed good results during training. However, the hope that, as Stefanowski noted [35], ENN would reduce noise during training did not seem to help generalise the model in this case. The general trend for these models was too conservative, never predicting a bug in the debug windows. Without ML-based bug prediction, the mean number of iterations to find a bug is 2.5. From the results in fig. 5.4 - 5.11 we can see that the best performing model ranked the bug first 9/14 times, meaning that one slot for speculative validation could reduce this number to 1.57.

The results seen in figures 5.4 - 5.11 seemed to show a pattern. Does a model with high top prediction values also rank bugs correctly more often? Plotting mean top prediction against median rank, as seen in fig. 5.12, shows an indication that higher prediction values seem to correlate with better ranking.

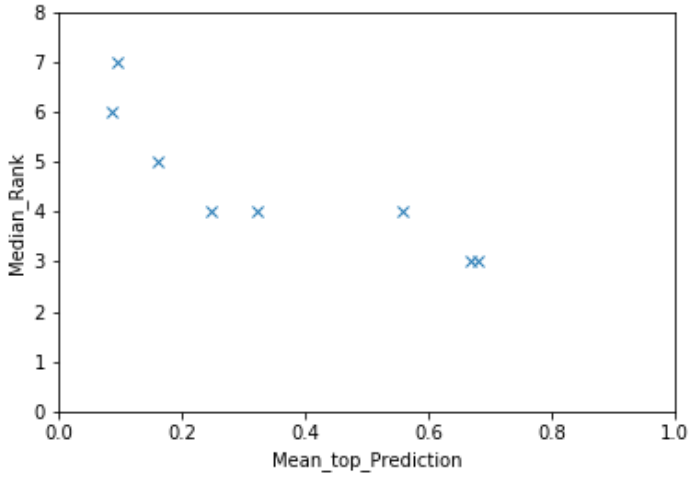


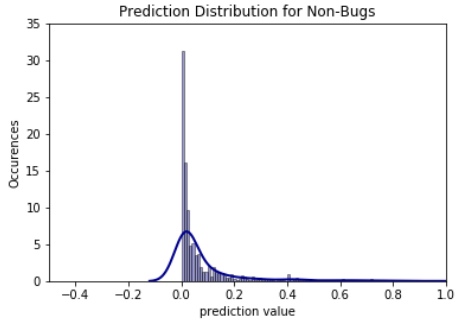
Figure 5.12: Rank vs. top prediction

The results in fig. 5.12 are helpful in the sense that a model can be evaluated as good or bad earlier if we can establish a link between high predictions and good ranking. During the training phase, good precision was the main goal, but with these results, high prediction values can be prioritised as well.

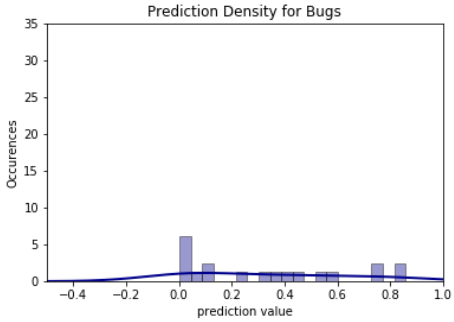
To establish whether we can use the prediction values to direct the test suite, i.e. running long test suites on revisions classified as risky and shorter suites on low-risk revisions, we need to see a clear split between prediction values for bugs vs. clean revisions. Fig. 5.13 shows the distribution of prediction values. Here we can see that, although the distribution for bugs has higher values in general, there are still some predicted as clean.

The mean prediction value for clean revisions was 0.09 with 0.15std (fig. 5.13a) whereas bugs had a mean prediction value of 0.34 with 0.3std (fig. 5.13b). This means a prediction value of close to 0.0 does not, with certainty, mean there are no bugs. Does this mean that the model is not trustworthy or can we classify risk some other way?

Fig. 5.14 shows distributions of prediction scores when split into two groups. The blue line shows a subset with prediction scores 0.23 or higher. The orange line shows the subset with lower prediction scores. We observed that the group with lower prediction values (or-



(a) Kernel density plot of prediction value for non-bug commits



(b) Kernel density plot of prediction value for commits containing bugs

Figure 5.13: Prediction distributions for clean revisions to the left and bugs to the right

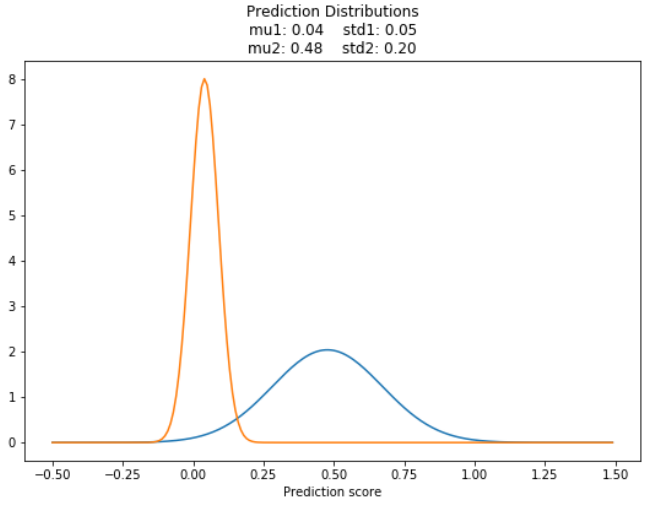


Figure 5.14: The prediction values split into two groups, those with a value below 0.23 and those above. This split creates two almost distinct groups where the bug density above 0.23 is 10x that of the commits with predictions below 0.23

ange line) had a bug frequency of 1 every 117 revisions whereas the higher predictions (blue line) had a bug once every 11 revisions. This is a great improvement, but nevertheless not certain enough to be used for risk based verification. This could however be used internally by PinDown. As a means to decide whether to focus resources on validating from the ranked list or to spend more resources on the search algorithm.

5.3 Results Summary

The training results seen in table 5.1 show that the models' perform about the same, whereas the figures 5.4 - 5.11 tell a different story. See table 5.3 for the summarised iterations needed per model presented in figures 5.4 - 5.11 to find the bugs. The number of iterations required are varying, but compared to the mean number of iterations required for PinDown without bug prediction (2.5 iterations), only three out of eight performed worse.

Model name	Mean iterations
bb.smo.avg.nn	1.86
bb.smo.awe.nn	1.57
bpbp.smo.avg.nn	2.0
bb.enn.avg.wofl	4.78
bb.enn.awe.wofl	3.93
bb.enn.avg.woflrem	2.21
bb.enn.avg.redFeats	2.29
bb.enn.awe.redFeats	3.07

Table 5.3: Number of iterations needed to find to bug with one slot. This compares to a mean number of 2.5 iterations for PinDown without bug prediction

The models that performed the best did not rely on any directed feature experimentation. The use of *wofl* models was to verify if these features were as project-dependent as expected. However, as the project we trained the models on was not live when deployed, we cannot draw any conclusions from these results.

We can also see that the *bpbp* model, although trained on the SZZ bugs, performed better than PinDown standalone. The SZZ algorithm's generated bugs performed well means that an out of the box model would be possible to create, as PinDown can mine SZZ bugs immediately when installed.

In two out of three cases, the *awe* models performed better than PinDown without bug prediction. However, we would need more data to draw any firm conclusions.

We noted a correlation between high prediction values during deployment and good predictions. We can possibly use this correlation to evaluate if a model is inadequate earlier than waiting for it to miss several bugs.

Lastly, we separated the prediction values by a cutoff limit of 0.23 to see whether risk-based verification would be possible. The revisions with a prediction value of 0.23 or higher were ten times more likely to contain a bug than those below 0.23.

Chapter 6
Experience from ML Debugging

A question raised in this thesis, **RQ2** , is how to debug an ML implementation. As mentioned in section 3.6, debugging the ML algorithm itself is out of scope and assumed to function as intended. However, viewing the ML algorithm as a function f(x) = y, transforming the feature vector x into a prediction value y, how can the prediction value aid in tracing where and if there is a bug in the implementation? As debugging our bug prediction model was a continuous activity during our work, we capture our experience in a tailored ML debugging method [40].

6.1 Workflow

Our starting point for debugging the implementation is the prediction values produced from the model. When we train the model and its performance evaluated against the validation set, the ground truth of each revision known at this time, we can calculate the average prediction value for bugs and clean commits. We can use these distributions to compare against live inference, using conventional statistical significance testing (student's t-test for normally distributed data, otherwise the Mann-Whitney U test) to detect differences. Worth mentioning is that an experience report by Google engineers report that statistical testing is too sensitive at their scale [28]. Even minute differences are statistically significant when working with Internet-scale data. However, in the debugging context, statistical difference testing has proven helpful. Furthermore, rule-based illegal value detection is beneficial.

Our debugging method, as seen in fig. 6.1, describes the four steps taken to identify where and if there is a problem.

The four steps shown in fig. 6.1 utilise similar techniques explained in the list below.

1. Comparing the mean values – When the mean and median values differ a lot, this could be enough indication that there is an implementation error. If bugs' mean prediction

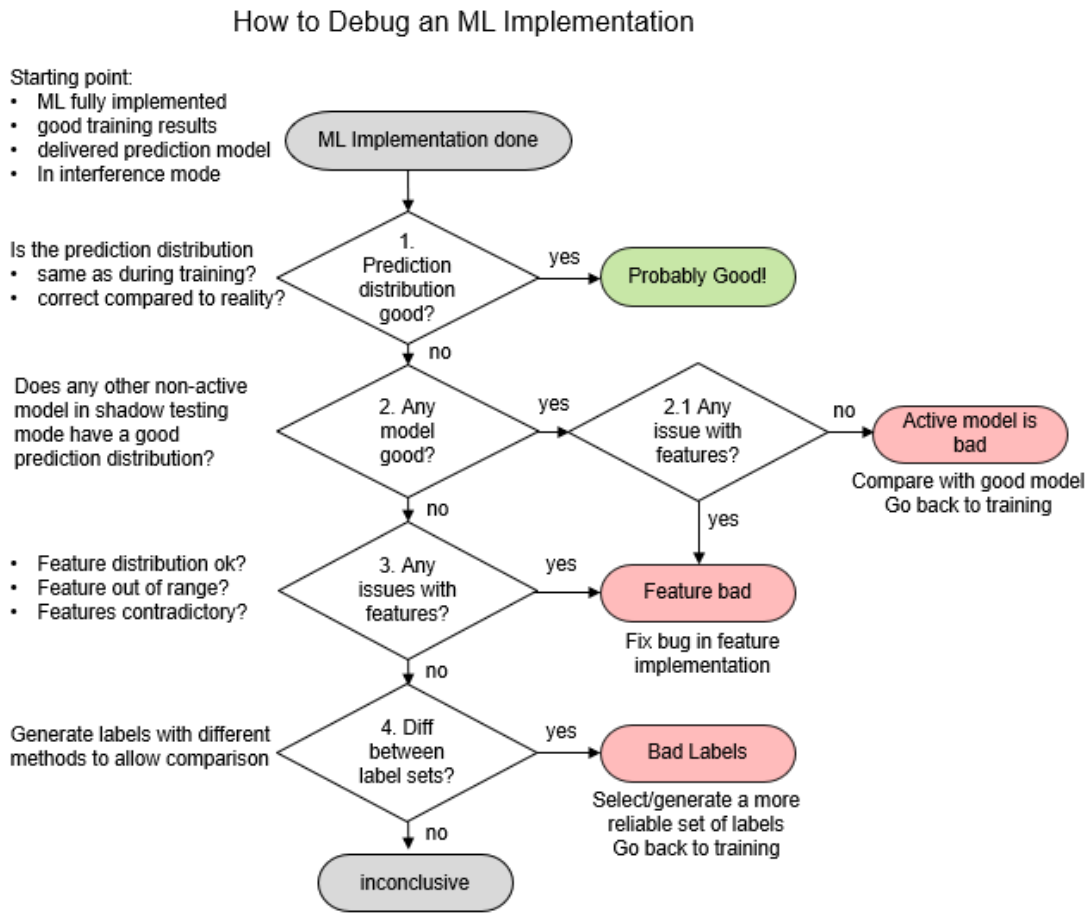


Figure 6.1: Debugging an ML-implementation methodology

value during validation is 0.65 and 0.08 during live inference, no further significance testing is required.

2. Student's t-test – using Student's t-test, we can identify distributions that are significantly different. The Student's t-test assumes normally distributed data.

3. Mann-Whitney U test – for many features, the distribution of data is not normally distributed. Some features appear to have a heavy-tail distribution, i.e., a portion of the distribution has many occurrences far from the central part. An alternative approach is to use a more robust statistical test, i.e., a nonparametric test such as the Mann-Whitney U test.

4. Detecting illegal values – some values are illegal and can be easily detected. Illegal values do not happen often, but there is a clear deterministic way of testing that a fault exists. To determine whether a value is legal or not requires domain knowledge.

A closer look at each step in fig. 6.1 will be recounted in the coming sections.

6.1.1 Step 1. Prediction Distribution Good?

The condensed workflow described in fig. 6.1 starts with a close look at the model's output. Any ML model is subject to numerous biases and anomalies and sometimes fails to generalise. Can the output of the models be used to trace if and where there is a problem?

The prediction distribution of the model, step 1 in fig. 6.1, is hence used to indicate if the ML implementation and surrounding infrastructure is on the right path. The indicators identified are:

Ind1 *Number of faulty commits predicted (**TP** + **FP**)* – is the model's performance similar to the precision/recall produced during training?

Ind2 *Prediction value distributions* – is the mean prediction value for the groups clean/bugs as low/high as during training?

If the precision/recall is similar to that observed during training, there is likely no error in the implementation. However, if the model has failed to predict faulty commits consistently, there is cause for concern.

The next step is to verify that the prediction distribution is as expected. Each project may be different, but in general, a bug frequency of once every 12 – 45 commits [14] is expected. If the model does not predict bugs as often or more often, a closer look at the prediction values is warranted.

In the example in fig. 6.2, prediction values are consistently low, never predicting that a bug is present. We expect the prediction distribution to be in close range of the distribution during the ML model's validation stage. This is generated when predicting never-seen-before data, similar to the live inference stage.

Given the prediction distribution seen in fig. 6.2 both indicators are in play; the bug is ranked 5th and below 0.5 (i.e. not predicted as a bug). In fact, no bugs are predicted, and all prediction values are lower than that of the validation set. This brings us to step 2 in fig. 6.1, which aims to look at if any model produces a reasonable result and whether that could be traced to certain features.

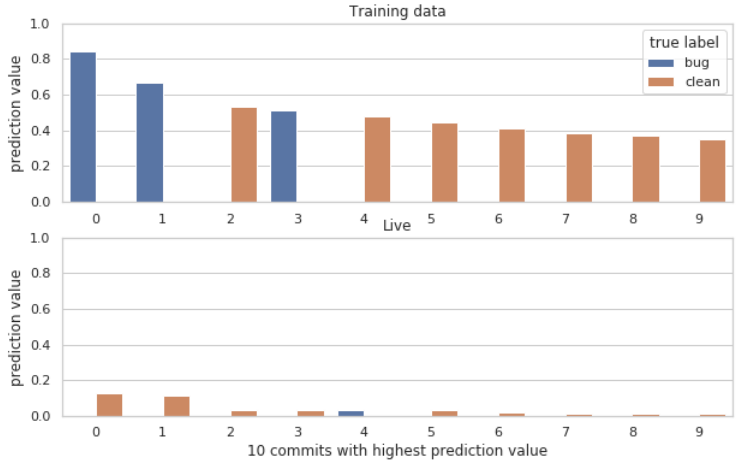


Figure 6.2: Prediction values as seen during training/validation vs live inference. With the same prediction model, the expected prediction values should be of the same order; this is a strong indication that something is wrong. Note: prediction values range from 0.0-1.0, 1.0 meaning the model is confident that a commit is a bug

6.1.2 Step 2. Is Any Model Good?

As we are running multiple models, we can include the previously used models to verify progress or track regression bugs and models trained on different subsets of features.

The procedure in step 2 is the same as the previous step. The difference lies in that we are now looking at the models running in shadow mode. Comparing each model against its results during the validation stage, we can discern whether the model is performing as expected or not.

The results from analysing all models can have different outcomes. If some models perform as expected, the problem may lie within which features the other models used. If no models perform as expected, there might be a more systemic issue, leading us to the next step in fig. 6.1 is a closer look at the features.

6.1.3 Step 3. Any Issue with the Features?

When training a neural network to distinguish images of cars from camels, the developer can manually assure that all information the classifier needs to identify the car is in the images; an absence or presence is easy to identify for humans. What if the developer cannot see a pattern in the input data? How can we make sure it contains relevant information and presented correctly?

To find out if a fault lies within feature implementation (step 2.1 and 3 in fig. 6.1), we have identified three assertion methods that can be used:

Assert1 *Statistical distribution assertions* – are the feature value distributions within a reasonable range of the feature value distributions during training?

Assert2 *Value-range assertions* – do the features' values fall in the same range as during training

Assert3 *Logical inter-feature assertions* – if features are logically linked, set up tests to verify that they comply to the expected rules.

As seen in fig. 6.3, the distribution to the left has the majority of samples below **1.0**, whereas the distribution to the right has a minimum value is 1.0. For this feature, values below 1.0 are illegal, yet during training data extraction, an implementation error caused pre-processing to scale the feature down to below 1.0. We caught this error by Assert 1 & 2, median value discrepancy between live inference and training, and an illegal value range.

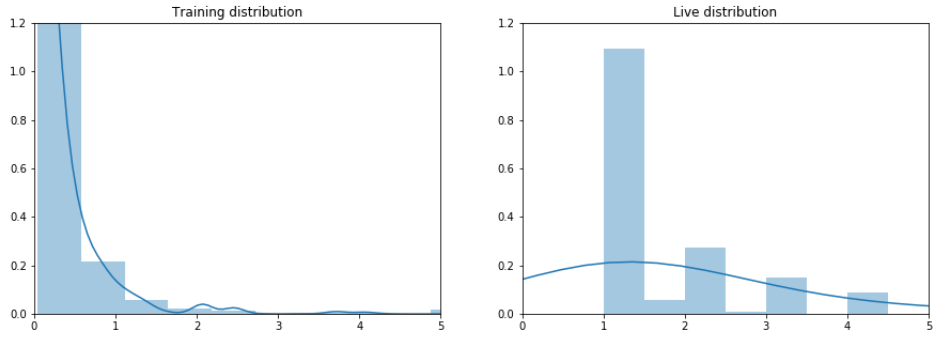


Figure 6.3: Identified by discrepancies in median value. The distribution to the left was scaled to illegal values.

Inter-feature assertions (Assert3) require feature knowledge. In our case some feature values are dependent, such as: *changed files* cannot be more than *changed* + *added* + *deleted lines*. Implementing Assert3, wherever possible, can locate possible implementation errors during feature extraction and save much time trying to debug the ML-based solution.

6.1.4 Step 4. Diff between Label Sets?

Gathering real-world data can be a challenging endeavour. Supervised ML leverages on large annotated datasets. The datasets can be extracted and labelled by scripts or annotated by hand. Manual annotation is a tedious activity.

The Problem

Script-annotated datasets have to be handled with care if there are large numbers of false positives (**FP**s) or false negatives (**FN**s), the possibility to train an accurate model might be lost entirely.

Whether PinDown's search algorithm or prediction model found a bug, the results are always safety-caged by PinDown's validation mechanism, see fig. 2.1. PinDown removes all potential bugs (one at a time), and the tests that previously failed are executed again to verify that they pass. It is correct to assume that the bugs validated by PinDown cannot be **FP**s. What about **FN**s? If we have not found any other flaws, multiple labelling techniques are a key piece of debugging advice.

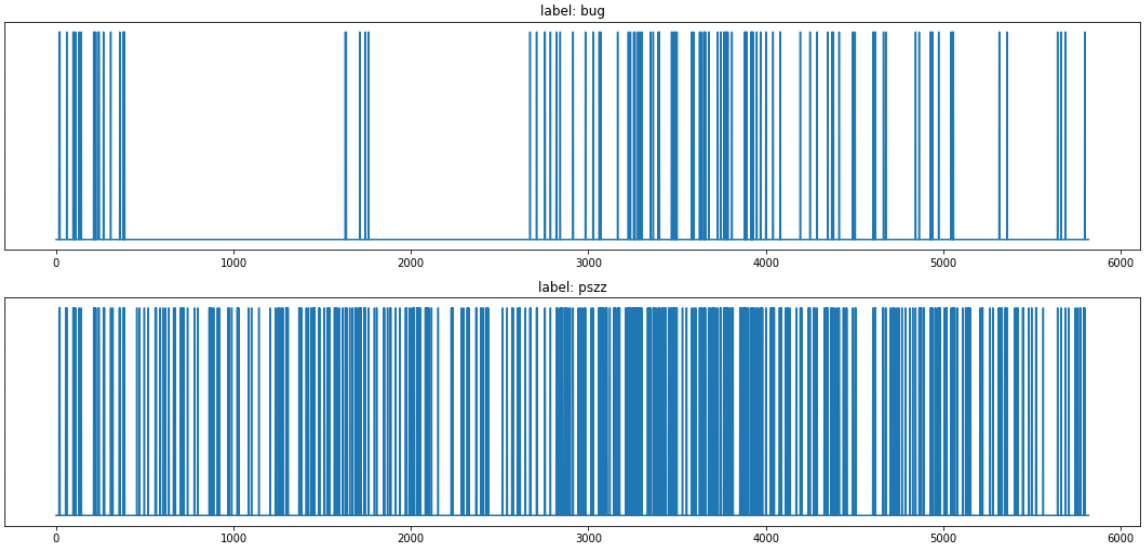


Figure 6.4: Segments in training data without positive label, a telltale sign of false negatives.

The Lesson

When relying on script-annotated data labelling, we provide two recommendations. First, analyse the resulting class distribution. Second, if possible, use more than one labelling technique. We found that during periods when PinDown was not used, no bugs were validated. This means that the data may contain segments where no bugs are stored (**FN**s as discussed in section 4.3), upper plot in fig. 6.4. In order to avoid **FN**s in the data set, these unlabelled segments can be removed from the dataset.

6.2 Workflow Summary

In this chapter we used statistical tools to inspect whether there is an issue in the ML infrastructure and how to find it. With a model deployed, access may be limited. We designed the workflow to accommodate minimal insight, working from the least to most intrusive steps to find where the issue may be.

Chapter 7
Future work

In this chapter, we outline four directions for future work that we perceive as particularly promising.

7.1 Classifying Bugs

The thesis covers just a small fraction of what could be studied in this area of research. One improvement would be to analyse and classify different types of bugs. Certain bugs might be easier to predict that others. This could be used to train bug specific models.

7.2 Revised Training Phase

The training data generated was indiscriminate of bugs in the revision history, whereas PinDown is only launched when a test has failed. There is a discrepancy between the type of data that the models are trained on and the live debug frames they are applied on. That is to say, only around 3% of the revisions in the training data contained bugs, but when PinDown is predicting there is almost a 100% chance that there is a bug. This knowledge could be used to improve the training technique.

7.3 Company Independent Features

Further research on which features are company independent would be greatly beneficial to resolve how an out of the box solution could be obtained. As seen in the results, the models trained on PinDown generated bugs were better than the model generated with the SZZ algorithm. The problem with PinDown bugs is that they require PinDown to be running

over a period of time. If company independent features could be isolated, a bug prediction model could be delivered on day one.

7.4 Always Failing Bugs

This topic was briefly mentioned in the theory section. When tests use randomised seeds, there is a chance that a test fails not because of an introduced bug, but because a new coverage point has been found. If the prediction model were to classify all recent commits as low risk, could this mean that a new coverage point has been found?

Chapter 8
Conclusion

8.1 RQ 1 – Is There a Speedup?

As software/hardware is introduced to new markets and industries at an ever-increasing pace, so too is the need for finding and fixing bugs. **RQ1** aims to investigate how ML can be used to speed up and alleviate the often manual task of debugging regression failures by, in a sense, mimicking the way an engineer would rank incoming revisions. This was done by training and deploying an ML model integrated into the workflow of Verifyter's tool PinDown. The training of any ML model is dependent on the quality of the input data trained on. As PinDown validates the bugs found by automatic code repair, the tool is especially fit for this type of integration. There were many potential use cases depending on how accurate the resulting models are.

As we saw in the results, bug prediction is not precise enough to send out a bug report without further verification. Rerunning the tests to verify that they pass again is a necessary step. If the tool were to send out bug reports without removing the faulty code and validating that the test passes again, there is a chance that the predicted revisions do not result in the test failure it proposes. If false bug reports are sent, the credibility of the tool may be in question. The aim of the tool is to assist developers and reduce time spent debugging; if credibility is in question, time may be spent discussing the results of the tool rather than fixing bugs.

The prediction model is not precise enough to be certain that a revision does not contain a bug either. There are revisions predicted to be clean, but in fact, contain bugs. There was a conscious decision to skew the model towards false negatives since if the prediction model did not predict any bugs, fewer resources would be consumed for speculative validation. This, however, means that the prediction values cannot be used to direct test suites to run shorter runs on low-risk revisions and longer on high-risk revisions.

The models were, however, shown to be of help in reducing the number of iterations required for PinDown to validate a bug. The number of iterations required was reported to be 2.5 without a bug prediction model, whereas 1.57 if one slot was reserved for speculative

validation. Depending on the time it takes the rerun the test that failed, this speed-up could mean the difference between getting results first thing in the morning and fixing the bug or starting new tasks, only to context switch later to fixing said bug. The result also lends itself to further investigation on how to optimise the resource allocation between speculative validation and the search algorithm.

8.2 RQ2 – Debugging ML

We conclude our lessons learned on how to debug ML implementations below.

When training is completed, and the live inference works as intended, there is no immediate need to search for potential errors. Monitor the model to verify that the performance does not degrade over time.

If the accuracy during live inference is not as high as during training, check the prediction distributions. If the distribution appears valid, then give it time. However, if the distribution deviates from what is expected, there might be an implementation error.

If any of the models used in shadow mode have shown accurate predictions, the problem is likely within the feature subset used in the main model. If all of the models in shadow mode are inaccurate, the problem might be with multiple features or during labelling of the dataset.

To find out if and/or which feature is problematic, compare mean/median values and check for illegal values for each feature in the live inference vs training dataset to see if anything sticks out.

If the features seem valid, but all models are performing inaccurately, the root cause might lie in the training data. If the model is trained on a dataset with corrupted labels, the chance of producing accurate predictions are slim. The fastest way to check for faulty labelling is to use different labelling techniques. If there are areas in the dataset that seem too suspicious, try training an algorithm without those chunks.

The final remedy that we can prescribe is labour intensive. Manual verification of features and labels. If nothing could be found in the previous steps, calculate and verify as many features as possible. Revise the labelling to be certain that no junk, i.e., invalid or noisy data is present when training the algorithm.

References

[1] Noura Al Nuaimi, Mehedy Masud, Mohamed Serhani, and Nazar Zaki. Streaming feature selection algorithms for big data: A survey. *Applied Computing and Informatics*, ISSN:2634-1964, 01 2019.

[2] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 304–318, New York, NY, USA, 2008. Association for Computing Machinery.

[3] Rob Ashmore, R. Calinescu, and Colin Paterson. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *ArXiv*, 1905.04223, 2019.

[4] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 117–128, 2017.

[5] Markus Borg, Cristofer Englund, Krzysztof Wnuk, Boris Duran, Christoffer Levandowski, Shenjian Gao, Yanwen Tan, Henrik Kaijser, Henrik Lönn, and Jonas Törnqvist. Safely entering the deep: A review of verification and validation for machine learning and a challenge elicitation in the automotive industry. *Journal of Automotive Software Engineering*, 1:1–19, 2020.

[6] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. Szz unleashed: an open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pages 7–12, 2019.

[7] Ilene Burnstein. *Practical Software Testing*. Springer, New York, NY, 2002.

[8] Nitesh Chawla, Kevin Bowyer, Lawrence Hall, and W. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Intell. Res. (JAIR)*, 16:321–357, 01 2002.

[9] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016.

[10] Harry Foster. 2018 wilson research group functional verification study. `https://blogs.sw.siemens.com/verificationhorizons/blog/2019/01/29/part-8-the-2018-wilson-research-group-functional-verification-study/`.

[11] Claire Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21:421–443, 09 2013.

[12] T. M. Ha and H. Bunke. Off-line, handwritten numeral recognition by perturbation method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):535–539, 1997.

[13] Daniel Hansson. Automatic bug fixing, 2015. `https://verifyter.com/images/whitepapers/An-Automatic-Flow-for-Bug-Fixing-v4.pdf`.

[14] Daniel Hansson and Christian Graber. One bug every 37th commit - planning for bugs, 2014. `https://verifyter.com/images/whitepapers/Planning-for-Bugs-Paper-v3.pdf`.

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[16] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[17] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 392–401, 2013.

[18] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1110–1121, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 510–520, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Bartosz Krawczyk, Michał Woźniak, and Gerald Schaefer. Cost-sensitive decision tree ensembles for effective imbalanced classification. *Applied Soft Computing*, 14:554 – 562, 2014.

[21] Grace Lewis, Daniel Plakosh, and Robert Seacord. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley Professional, 2003.

[22] Aleksander Jankowski Miron Kursa and Witold Rudnicki. Boruta – a system for feature selection. *Fundamenta Informaticae, vol. 101, no. 4, pp. 271-285*, 2010.

[23] Christoph Molnar. *Interpretable machine learning*. Lulu.com, 2020. ISBN9780244768522.

[24] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[25] msrconf.org. Working conference on mining software repositories, the main software engineering conference in the area.

[26] Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Systems with Applications*, 144:113085, 2020.

[27] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[28] Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. Data validation for machine learning. *Proceedings of Machine Learning and Systems*, 1:334–347, 2019.

[29] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[30] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. Evaluating szz implementations through a developer-informed oracle. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 436–447, 2021.

[31] scikit-learn developers. Tuning hyper-parameters of an estimator. Available at `https://scikit-learn.org/stable/modules/grid_search.htm`, version 0.22.1.

[32] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, page 2503–2511, Cambridge, MA, USA, 2015. MIT Press.

[33] Umair Shafique and Haseeb Qaiser. A comparative study of data mining process models (kdd, crisp-dm and semma). *International Journal of Innovation and Scientific Research*, 12:2351–8014, 11 2014.

[34] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[35] Jerzy Stefanowski. Dealing with data difficulty factors while learning from imbalanced data. In *Challenges in computational statistics and data mining*, pages 333–363. Springer, 2016.

[36] F. Thung, S. Wang, D. Lo, and L. Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280, 2012.

[37] Muhammad Fahim Uddin, Jeongkyu Lee, Syed Rizvi, and Samir Hamada. Proposing enhanced feature engineering and a selection model for machine learning processes. *Applied Sciences*, 8(4), 2018.

[38] Jason Van Hulse and Taghi Khoshgoftaar. Knowledge discovery from imbalanced and noisy data. *Data & Knowledge Engineering*, 68(12):1513 – 1542, 2009. Including Special Section: 21st IEEE International Symposium on Computer-Based Medical Systems (IEEE CBMS 2008) – Seven selected and extended papers on Biomedical Data Mining.

[39] Verifyter. Pindown. Available at: `http://verifyter.com/support/documentation`.

[40] Oscar Werneman, Markus Borg, and Daniel Hansson. Supporting root cause analysis of inaccurate bug prediction based on machine learning - lessons learned when interweaving training data and source code. In *Proc. of Design and Verification Conference & Exhibition (DVCon)*, 2021.

[41] D. L. Wilson. Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-2(3):408–421, 1972.

[42] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[43] XGBoost Developers. Xgboost documentation. Available at `https://xgboost.readthedocs.io/en/latest/`, version 1.2.1.

Appendices

Appendix A
XGBoost Hyperparameters

- learning_rate
  - Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
- Gamma
  - Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be.
- max_depth
  - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 is only accepted in lossguided growing policy when tree_method is set as hist and it indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree.
- min_child_weight
  - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger min_child_weight is, the more conservative the algorithm will be.
- max_delta_step
  - Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making

the update step more conservative. Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced. Set it to value of 1-10 might help control the update.

- subsample
  - Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.
- colsample_bytree, colsample_bylevel & colsample_bynode
  - All colsample_by parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled.
  - colsample_bytree is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.
  - colsample_bylevel is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree.
  - colsample_bynode is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.
  - All colsample_by parameters work cumulatively. For instance, the combination 'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5 with 64 features will leave 8 features to choose from at each split.
- lambda
  - L2 regularization term on weights. Increasing this value will make model more conservative.
- alpha
  - L1 regularization term on weights. Increasing this value will make model more conservative.
- scale_pos_weight
  - Control the balance of positive and negative weights, useful for unbalanced classes. A typical value to consider: sum(negative instances) / sum(positive instances)
- objective
  - binary:logistic: logistic regression for binary classification, output probability
- base_score
  - The initial prediction score of all instances, global bias
  - For sufficient number of iterations, changing this value will not have too much effect.

- eval_metric
  - Evaluation metrics for validation data, a default metric will be assigned according to objective (rmse for regression, and logloss for classification, mean average precision for ranking)

Appendix B
Features

Below is a shortened list of features generated by PinDown.

F1: **author_revs_avg** - average author revision since earliest set revision on files included in commit

F2: **author_revs_top** - file included in commit with most nbr of revisions done by author, since earliest set revision

F3: **author_revs_tot** - total author revision since earliest set revision for files included in commit

F4: **n_authors_avg** - average nbr of authors editing files included in commit since earliest set revision

F5: **n_authors_top** - top nbr of authors editing files included in commit since earliest set revision

F6: **n_authors_tot** - sum of authors editing files included in commit since earliest set revision

F7: **n_revs_avg** - average nbr or revisions done to files included in commit since earliest set revision

F8: **n_revs_top** - file with most number of revisions included in commit since earliest set revision

F9: **n_revs_tot** - total nbr of revisions to files included in commit since earliest set revision

F10: **author_share_of_revs_avg** - average share of revision done since earliest set revision included in commit

F11: **author_share_of_revs_top** - file where author has heighest nbr of revisions since earliest set revision

F12: **author_totalrevs** - total nbr of revisions since earliest set revision

F13: **author_age** - how long the author has worked in this project

F14: **soc_avg** - how coupled are the files on average

F15: **commit_loc** - sum of lines of code commited

F16: **commit_loc_added** - nbr of lines added in commit

F17: **commit_loc_deleted** - nbr of lines deleted in commit

F18: **commit_loc_changed** - nbr of lines changed

F19: **n_files** - nbr of files in commit

F20: **n_files_added** - nbr of files added (new files)

F21: **n_files_changed** - nbr of files changed

F22: **n_files_deleted** - nbr of files deleted

F23: **n_focus_files** - user definable feature

F24: **max_folder_distance** - distance between files in FL hierarchy

F25: **max_line_distance_avg** - average distance between commited lines in their respective files

F26: **max_line_distance_top** - maximum distance in single commited file

F27: **max_line_distance_tot** - total distance summed over all files in commit

F28: **commit_day** - day of the week the commit occured

F29: **commit_time** - local time of the commit

F30: **commit_time_median** - CET of commit

F31: **commit_age** - age of commit at head

F32: **commit_msg_fix** - commit message contains a variation of "fix"

F33: **commit_msg_words** - commit message contains a word deemed risky

F34: **commit_msg_length** - length of commit message

F35: **commit_mrg_resolve** - commit is a merge, resolve or regular

F36: **commit_mrg_flag** - commit is a merge or not

EXAMENSARBETE Predicting bugs to reduce debugging time

STUDENT Oscar Werneman
HANDLEDARE Markus Borg (LTH), Daniel Hansson (Verifyter)
EXAMINATOR Per Runeson (LTH)

Kan man förutspå buggar?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Oscar Werneman**

I takt med att kodbaser växer blir det allt mer omfattande arbete att hitta och fixa buggar. Som lösning till detta växande problem krävs allt mer sofistikerade verktyg. Detta arbete utvärderar potentialen för en ML-baserad prediktionsmodell för att snabbare hitta buggar.

PinDown (utvecklat av Verifyter i Lund) är ett verktyg specialiserat på automatisk debug av regressions buggar. Regressionsbuggar är nya buggar som förstört tidigare fungerande funktionalitet, dvs. kvalitén har regresserat. Kunder är utvecklare av integrerade kretser, vilka ofta kräver stora testbänkar där antalet revisioner, dvs. förändringar, mellan varje testbänkskörning kan variera från ett tiotal till ett hundratal. När antalet revisioner är stort leder det ofta till omfattande arbete att finna vilken som får ett test att fallera. Detta faktum tillsammans med att ett flertal olika typer av fel kan uppstå ur en testbänkskörning utgör en stor del av utvecklingskostnaden. För att reducera utvecklingskostnaderna automatiserar PinDown sökningen efter vilka revisioner som får testen att fallera. För att finna buggarna kör PinDown om testen som fallerat. Om PinDown ska leverera en färdigställd buggrapport inför nästa dag krävs det att kombination *testtid* och *antal revisioner* att testa inte tar för lång tid.

I mitt examensarbete har jag utvärderat möjligheten att träna en ML-baserad prediktionsmodell med syftet att korta ned sökningstiden för buggar. Istället för att behandla alla revisioner likvärdigt producerar ML-modellen en rankad lista (se figur 1) utifrån vilken PinDown kan testa om de skapat testfelet. Prediktionsmodellen minskar antalet revisioner som PinDown behöver testa.

För att skapa en prediktionsmodell krävs en stor mängd träningsdata. Verifyter har, genom åren, implementerat features och ansamlat bugg data, vilket utgör goda förutsättningar för denna typen av utvärdering.

Bug Predictions

Prediction	Revision	Date	Committer
0.999403	...stbed0.git:7c643333a5	Feb 12 2011 5:19 AM CET	carlos
0.996919	...stbed0.git:f36f5c8981	Feb 12 2011 5:16 AM CET	prashant
0.759582	...stbed0.git:26c2a86713	Feb 12 2011 5:25 AM CET	nageshwar
0.759582	...stbed0.git:18e57d6808	Feb 12 2011 5:13 AM CET	sharon
0.620521	...stbed1.git:816246c63f	Feb 12 2011 5:17 AM CET	praveen
0.587064	...stbed2.git:54abe25cc2	Feb 12 2011 5:21 AM CET	prashant
0.043611	...stbed2.git:5800e5fee3	Feb 12 2011 5:18 AM CET	carlos
0.000000	...stbed1.git:f0679c2296	Feb 12 2011 5:14 AM CET	hemal
0.000000	...stbed1.git:c48c888f86	Feb 12 2011 5:05 AM CET	nageshwar

Figure 1: Revisioner rankade efter risk. Rött indikerar bugg

Resultaten visar att tiden minskar som PinDown kräver för att utröna vilken revision som får testen att fallera. Denna förbättring kan innebära skillnaden mellan att få buggrapporten innan arbetsdagen börjat och att få den senare på dagen när annat arbete påbörjats, vilket är en väsentlig skillnad för utvecklarna.