A human-centric approach for adopting bug inducing commit detection using machine learning models

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Naz Zarreen Oishie

Abstract

When developing new software, testing can take up half of the resources. Although a considerable amount of work has been done to automate software testing, fixing bugs after adding them to the source repository is still a costly task from both management and financial perspectives. In recent times, the research community has proposed various methodologies to detect bugs just-in-time at the commit level. Unfortunately, this work, including state-of-the-art techniques, do not provide real-time solutions for the problem. Such a limitation restricts developers from utilizing them in their day-to-day programming tasks. Our study focuses on providing solutions that deliver real-time support to the developers by warning them about potential bug-inducing commits. Such support can help developers by preventing them from adding a bug-inducing commit to the source repository. Keeping this goal in mind, we conducted a developer survey to understand the expectations of developers for bug-inducing commit detection tools. Motivated by their responses, we built a GUI-based plug-in that warns the developers when they attempt to perform a potential buggy commit. We accomplished this by training machine learning models on relevant features. We also built a command-line tool for the developers who prefer to use a command-line interface. Our proposed solution has been designed to work with various machine learning models (e.g. random forest, decision tree, and logistic regression) and IDEs (e.g. Visual Studio, PyCharm, and WebStorm). It enables developers to work with a familiar interface without leaving the IDE. As a proof of concept, we implemented a VSCode plug-in and an accompanying command-line tool. Developers can customize these tools by choosing among various machine learning models and features. Such customizability empowers the developers to understand the toolchain better and lets them fit it into their specific use cases. Our user study shows that the toolchain offers satisfactory performance in detecting bug-inducing commits and provides a sound user experience. The decision tree model achieved the best performance with a 79% accuracy and an f1-score of 0.70 among the tested models. In addition, we performed a user study with developers working in the software industries to validate the usability of our toolchain. We found that the users can detect whether a commit is bug-inducing or not within a short period of time. Furthermore, they prefer our tool over the state-of-the-art to detect potential bugs before the commit operation. Alongside contributing a new multi-UI toolchain, our work enriches the research community's knowledge regarding developer usability of real-time bug detection tools.

Acknowledgements

First, I would like to express my heartfelt gratitude to my respected supervisor Dr. Banani Roy for her continuous guidance, suggestions, motivation and patience during my thesis work. I am truly grateful to my co-supervisor Dr. Kevin Schneider for his invaluable feedback on this thesis.

I would like to thank Dr. Gord McCalla, Dr. Aryan Saadat Mehr and Dr. Zadia Codabux for their willingness to take part in the advisement and evaluation of my thesis work. In addition, I am grateful to them for their valuable feedback, and suggestions.

I would like to express my special appreciation and thanks to the anonymous reviewers for their valuable comments and suggestions which helped to improve this thesis work.

Special thanks goes to the Software Research Lab (SRLab) and Interactive Software Engineering (iSE) lab members with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Dr. Chanchal Roy, Dr. Manishankar Mondal, Dr. Masudur Rahman, Avijit Bhattacharjee, Amit Kumar Mondal, Shamse Tasnim Cynthia, CM Khaled Saifullah, Daniel Abediny, Muhammad Mainul Hossain, Zonayed Ahmed, Shamima Yeasmin, Farouq Al-omari, Sristy Sumana Nath, Golam Mostaeen, Kawser Wazed Nafi, Debasish Chakroborti, Saikat Mondal, Md Nadim, Md Shamimur Rahman, Judith Islam, Tonny Kar, Md. Abdul Awal, Hamid Khodabandehloo, Rayhan Islam, Saumendu Roy Saumo.

I am grateful to the Department of Computer Science of the University of Saskatchewan for their generous financial support through scholarships, awards and bursaries that helped me concentrate more deeply on my thesis work. Moreover, I would like to thank all the staffs of the Department for their constant support. In particular, I would like to thank Sophie Findlay, Heather Webb, Greg Oster, Shakiba Jalal, James Ko, Jeff Long, Maurine Powell, and Cary Bernath.

I would like to thank my friends and family who were available for me when I needed them instead of being a thousand miles apart. In Particular, I would like to thank Parvez Mahbub, Sumaya Islam, Sadia Mahjabin and Mahfuzul Alam for their enormous support. In addition, I would like to thank Afsana Sultana Ruma and Ushasi Srija Chakroborti for their time, love and mental support during my stay in Saskatoon. Last but not least, I would like to thank Aniruddha Prithul for all his support, love and patience throughout my M.Sc. journey.

This thesis is dedicated to my mother – Nasreen Alam.

For her endless love, faith, support and encouragement.

She inspires me every day to become a better version of myself.

Contents

List of Tables

List of Figures

List of Abbreviations

BIC	Bug Inducing Commit
DL	Deep Learning
DT	Decision Tree
IDE	Integrated Development Environment
JIT	Just-In-Time
LR	Logistic Regression
ML	Machine Learning
RF	Random Forests
TLX	Task Load Index
VCS	Version Control System

1 Introduction

In this chapter, we provide an overview of the thesis. Section 1.1 discusses the motivation. Section 1.2 describes the problem we addressed. Section 1.3 introduces the research questions and Section 1.4 provides our contributions. Section 1.5 outlines the thesis.

1.1 Motivation

Bug resolution is a crucial task during software development and maintenance. According to several reports, bug resolution takes ≈50% of the development time [6, 8], consumes up to 80% of the total budget [17, 20], and costs billions of dollars each year [37, 56]. As an obvious consequence, bug prediction is one of the most active and important research areas in software engineering.

Ideally, any software application that is already on the market should not contain bugs; otherwise, resolving a bug may be quite costly depending on its type and severity. Software defects and bugs are generally fixed based on their severity and priority. Lucia et al. reported that 84-93% of bugs are found in just one or two of a system's source files [36]. However, finding these buggy files from thousands of candidate source files may be quite difficult; like finding a needle in a haystack.

Software evolves by means of code changes to fix bugs, introduce new features, and refactor the existing codebase. Such code changes, however, can also introduce new bugs [28, 33]. In the software development lifecycle, there is a dedicated test phase to identify bugs. However, during the test phase, the buggy code is already part of the *in-development* codebase and other developers may have already started working with the buggy codebase. Furthermore, often there are quite a few back-and-forths between the development and testing phases to eradicate a bug completely. Such situations obstruct the progress of development and delay new software releases. Therefore, it is more convenient to identify bugs before sending the code changes to the testing phase. To address this problem, in parallel to test automation, software bug prediction is an active and growing research field. Bug prediction is the identification of a code-chunk or file which has the chance of containing bugs, before formal testing. Such a prediction can significantly save the time and resources allotted for formal testing. As software companies seek different ways to deliver high-quality software without spending too much on software maintenance, quality assurance and testing, reliable prediction of bugs helps the software industry [4]. Bug prediction also plays a vital role in the improvement of software architectures, identification of refactoring candidates and selection of best design approaches [4]. As bug prediction can help in quantitative planning and management of projects, it also benefits project managers on software development teams [16].

Bug prediction can be done either at the file level or at the code change level. Developers use different version control systems (VCS) to keep track of code changes. Git is the most popular version control system where these code changes are known as a commit. A commit is a set of code changes that spans across one or more files. Commits that introduce software bugs are known as bug-inducing commits, which contain important information about when and how bugs were introduced. Due to the importance of bug-inducing commits, it has been extensively studied by researchers [54].

Shrikanth et al. [43] conducted research on defect prediction early in the software life cycle by analyzing 84 popular GitHub projects with a life cycle of a minimum of 84 months. They found that most of the defects for these projects occurred in commits from the first four months and in the first 150 commits. Therefore, it is important to detect bug-inducing commits in earlier versions of a project.

Although there are numerous studies on how to detect bug-inducing commits just as a developer makes the commits, very few techniques are adopted practically to help a developer in real time. In this thesis, we focus on different approaches to detect bug-inducing commits, identify what the developer community thinks about these approaches, and determine how to provide real-time support for developers to detect bug-inducing commits.

1.2 Problem Statement

1.2.1 Sub-Problem #1: Inadequate understanding from a developer's perspective of having a JIT bug prevention system

In the literature, there are several studies on how to detect bug-inducing commits, which metrics are useful and which algorithms should be used to predict those commits just-in-time. However, we have not identified any research that considers the opinion of the developer community regarding the bug prevention support they expect. Although extensive testing is done to limit the number of bugs in a system, developers still uncover bugs later in the development lifecycle. Understanding a developer's viewpoint is vital if a real-time bug prevention system is to be adopted by developers.

1.2.2 Sub-Problem #2: No real-time support system to prevent a bug inducing commit from occurring

There are several works in the literature where developers can upload a link of a GitHub repository and analyse the repository. Through the analysis, they detect bug-inducing commits [51] or detect bug-inducing reports from pull requests [30]. However there is no study to prevent the bug-inducing commit before the bug is introduced in the system.

1.2.3 Sub-Problem #3: Lack of understanding of the development impact by a real-time bug prevention system

Undoubtedly, extensive and thorough research precedes developing any kind of system. Nonetheless, evaluating the system's usability and effectiveness shares the same importance, if not more. How a system interacts with the target users and whether the users are comfortable enough to use the system are hard but critical things to find out.

1.3 Research Questions

In this thesis, we aim to understand the developers' need for bug prediction support, provide them that support and ensure this support addresses their need. In order to accomplish this goal, we answer the following research questions.

- **Research Question 1: Does the developer community need a real-time bug prediction technique which detects bugs before it is introduced to the central repository?**

  To date, there are numerous studies on detecting bug-inducing commits. However, there is no study that finds out what kind of support the developers need to identify bug-inducing commits. In this thesis, we conduct a survey to find out the developers' preference about bug prediction support. We find that bug prediction is still a significant problem that developers suffer from. Furthermore, they need better tool support to address this problem.

- **Research Question 2: What are the ways to provide real-time support to the developers to detect bug-inducing commits?**

  To develop a complete toolchain to support developers in bug prediction, we found that the most preferred support type is an IDE plug-in along with command line support. We also found that developers would like to customize these tools to further utilize for their particular use-case.

- **Research Question 3: How to evaluate the effectiveness of the provided real-time support provided to the developers?**

  After providing tool support the developers need, we further ensured that these tools are usable in regular development. Therefore, we measure the mental load, physical load, time, effort level, and frustration level of developers while using the tool. We further ensured that developers can easily interact with the toolchain. They find our tools are better compared to the *state-of-the-art* in most cases.

1.4 Our Contribution

1.4.1 Understanding the perspective of developer community

Generally, developers use an IDE for developing software and prefer to perform the commit operation using the IDE. However, most UNIX-based developers prefer to use a command-line interface in their regular workflow [13]. To understand what kind of support is preferred by the developer community, we performed a survey with 20 developers from popular GitHub projects. We collected information from their responses and analyzed them to determine the results.

1.4.2 Providing real-time support to the developer community

Analyzing the survey results, we understand that most developers prefer plug-in support for IDEs to prevent them from performing bug-inducing commits. However, many developers use command lines regularly and the developers appreciate having a command-line tool as well. Given this understanding, we build a plug-in and a command-line tool to detect the bug-inducing commits using several machine learning algorithms. To further enhance the developer experience, we provide a configuration option where developers can upload their dataset, choose their preferred features and algorithms, customize their machine learning models and use those customized models in the plug-in.

1.4.3 Evaluation of the real-time tools

To evaluate our provided support, we measure the time to install, potential bug detection time, and custom model training time. Developers often abandon a tool if it is too slow to work on a regular basis. As opposed to some of the prior works that want the whole repository to be uploaded at once and run the analysis for a long time, our tool detects bugs exactly when they attempt to commit. Therefore, developers need to interact with these tools often. We measured the mental load, physical load, effort level, and frustration level of developers while using the tool. The result showed that the toolchain imposes a lower load on developers for these aspects, which makes these tools effectively usable in regular development. While evaluating the performance of our tool, the participants found that it is identifying buggy commits with satisfactory accuracy. They also preferred our tool in most cases when compared to the *state-of-the-art*.

1.5 Thesis Outline

In Chapter 2, we discuss some background on integrated development environments, version control systems, bug-inducing commits and the machine learning algorithms used in this study. Chapter 3 focuses on the related literature of our research. In Chapter 4, we discuss the perspective of the developers' community on real time support for bug-inducing commits. Chapter 5 discusses the techniques and tools that support real-time bug-inducing commit detection. Finally, Chapter 6 discusses the evaluation process we follow to validate our study. Finally, in Chapter 7, we conclude the overall summary of the thesis and discuss potential future directions.

2 Background

In this chapter, we briefly discuss the terms, topics and techniques related to this thesis. Section 2.1 introduces integrated development environments (IDE) and their significance in software development. In Section 2.2, the basics of a version control system is explained. Section 2.3 discusses bug inducing commits and how bugs are detected throughout the software life cycle. Finally, Section 2.4 discusses the machine learning algorithms we used in this thesis.

2.1 Integrated Development Environment

An Integrated Development Environment (IDE) is a software application that helps programmers and developers increase their productivity by providing different facilities. It enables developers to combine all their activities while writing software for an application. Generally, an IDE consists of three main elements which includes a source code editor, build automation tools for executables and a debugger. Writing code is an essential part of software development. The developer opens a blank file and starts to write code. The IDE also highlights the syntax of the code for different programming languages to make the code more readable. Modern IDEs also help to auto-complete code chunks by anticipating what a developer is going to write next. Intelligent code completion helps developers to concentrate on their logic and speed up programming by saving keystrokes. Some advanced IDEs also provide support to remove code smells and write clean code by auto-refactoring. Some programming languages need support from compilers. Other than that, the programs that are already written need to be built and run by the IDE. Once the program is running, almost every time developers find bugs in their code. Debugging is an essential feature of an IDE because writing bug-free or error-free code on the first try is rare. The more debugging support an IDE has, the more popular it is among the developer community. The main goal of an IDE is to reduce the necessary configuration and setup time to work in multiple development tools. It can also extend its capability by integrating different types of plug-ins. Some examples are integrating support for docker, version control systems, bash scripts, grep console, etc. Figure 2.1 illustrates the functionalities and supports provided by a modern IDE.

IDEs increase developers' overall productivity by combining all individual tools to provide the same set of capabilities in a single software tool. For example, an IDE can highlight the syntax, auto-complete the code at the same time when a developer is editing is code. It helps the developers to write, refactor and debug their code way faster than it takes to do them manually with individual applications.

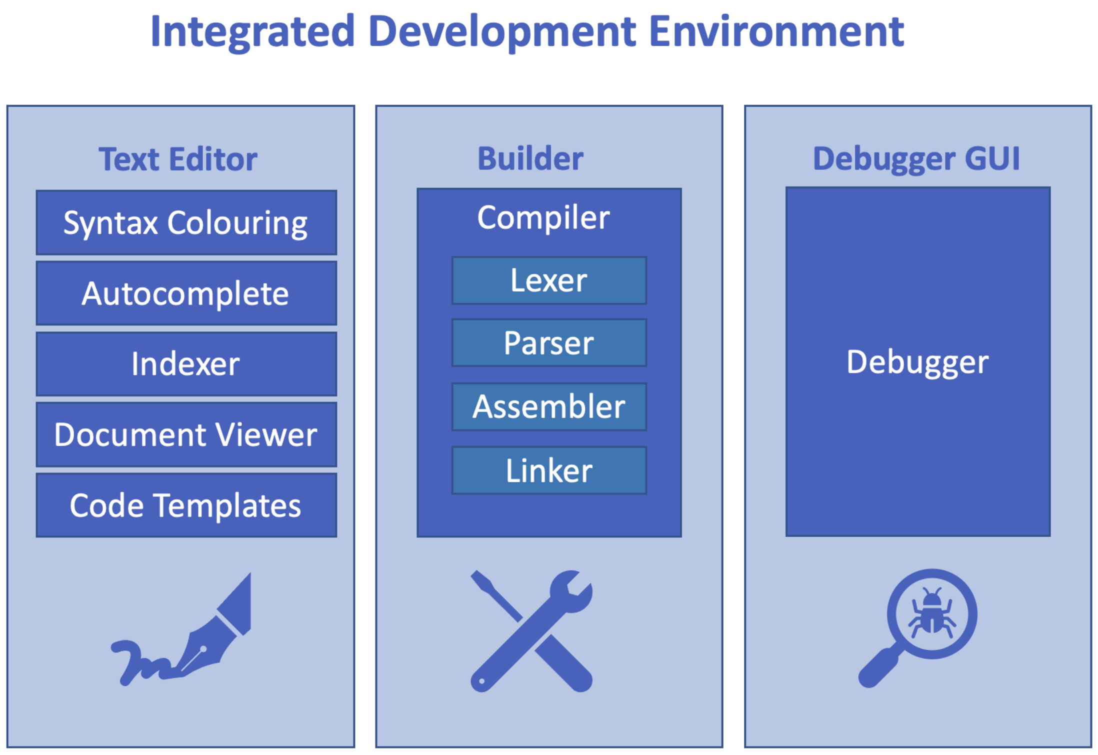


Figure 2.1: Functionalities of an IDE

2.1.1 Visual Studio Code

Visual Studio Code (VS Code) is a popular lightweight IDE by Microsoft which is built for multiple operating systems like Windows, Linux and Mac OS. The features of this IDE have all the supports discussed in Figure 2.1 and also include embedded git. The IDE is also very user-friendly by letting them choose their theme, preferences, keyboard shortcuts and third-party extensions for added functionality.

In a Stack Overflow Development Survey in 2021, it is found that VS Code IDE is the most popular developer environment tool among the developer community. 70% of 82,000 respondents of the survey reported it as their IDE of choice [25]. The IDE was first announced in April, 2015 and released and made available on GitHub as an open-source project in November, 2015. This means not only is the IDE free to use, but also the developers can make contributions in order to improve it and get engaged in the community. The IDE is straightforward, cross-platform and follows minimal design. One of the most important things about VS Code is it is open for extension support. Developers can build and customize their own extension along with downloading useful extensions from the marketplace. Moreover, VS Code is a language-agnostic IDE which has support for most of all the major and popular languages (e.g. Python, JavaScript, HTML, CSS, TypeScript, , Java, PHP, Go, C++, PHP, SQL, Ruby, Objective-C and much more).

2.2 Version Control System

A Version Control System (VCS) is a software tool which helps developers to store and manage different versions or changes of specific files or a set of files of a software system. It is also known as source control or revision control. A software product is developed by a group of collaborators workings in different sectors of a system (e.g. design, front-end, back-end, database, testing and so on). Moreover, sometimes developers in the same area work on different features and functionalities individually. A version control system helps the developer team to manage all the versions of their works, merge them to the main code base, keeps a record of who made the changes and what change are made. It also helps to revert all the changes and go back into a previous state, discard all the new changes, compare different versions of the files so that if a developer loses files or make mistakes that introduces bugs to the system, he or she can go back to the previous version without breaking the system. Figure 2.2 gives us a better idea of how the version control system works for different collaborators.

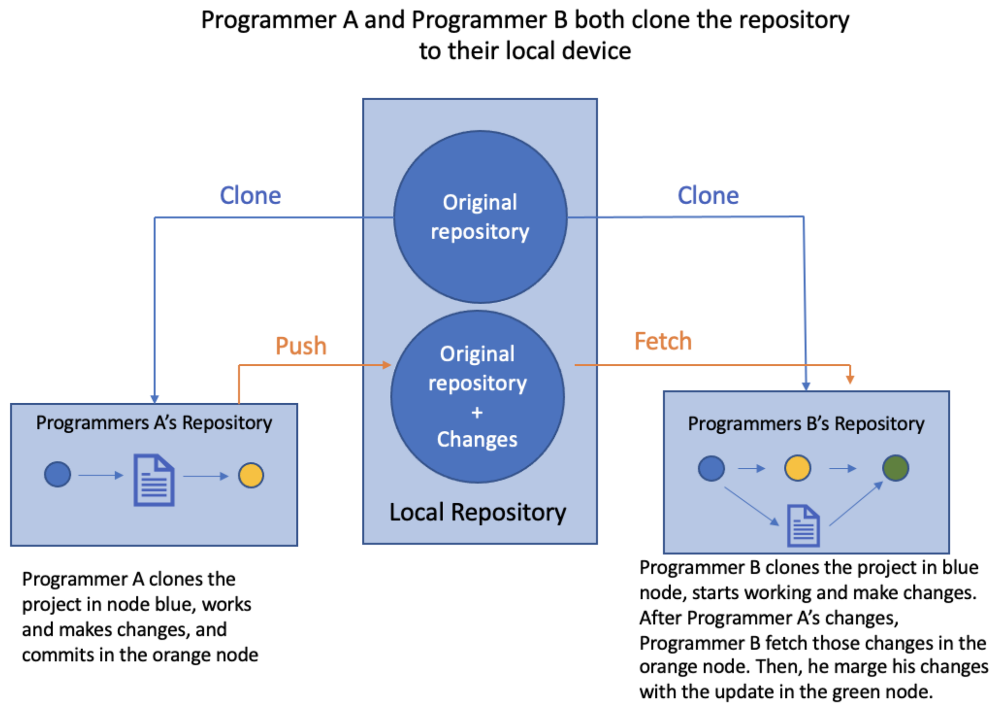

Programmer A and Programmer B both clone the repository to their local device
Clone
Original repository
Clone
Push
Original repository + Changes
Fetch
Programmers A's Repository
Programmers B's Repository
Local Repository
Programmer A clones the project in node blue, works and makes changes, and commits in the orange node
Programmer B clones the project in blue node, starts working and make changes. After Programmer A's changes, Programmer B fetch those changes in the orange node. Then, he marge his changes with the update in the green node.

Figure 2.2: Version Control System

There are two kinds of Version Control Systems - Centralized VCS and Distributed VCS. In a central VCS system, there is only one single server where all the version files are stored. Developer who needs to collaborate with others in the software system can check out files from that server. For many years, this centralized VCS has served the community. However, the main pitfall of this system is that whenever the whole history of a system is stored in one single place, and there is always a chance to lose everything. Moreover, if the server goes down for some time, nobody will be able to collaborate or save their changes in the server for that period of time.

To solve this, distributed VCS is introduced where the developers can mirror their whole repository with full into the server instead of checking out snapshots of individual files. So if anytime the server dies, any of the collaborators has the full backup data. That backup can be copied back to the server at any convenient time. There are many popular version control systems (e.g. Git, SVN and so on).

2.2.1 GitHub

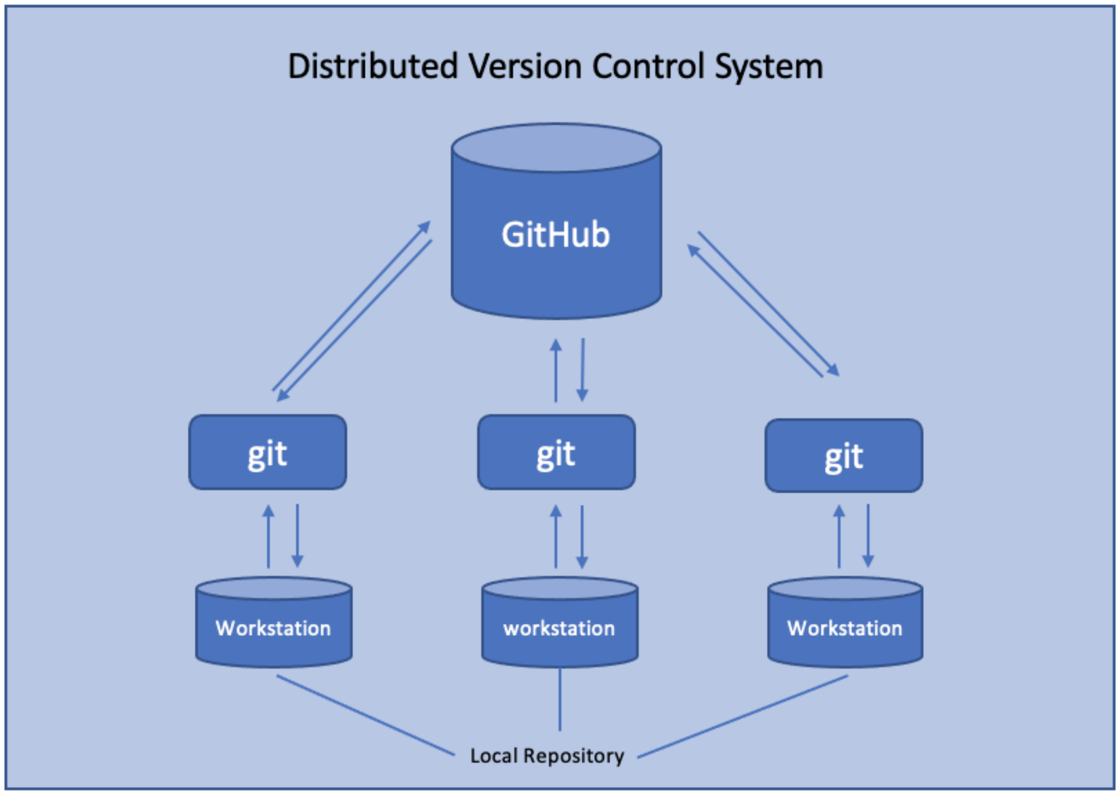


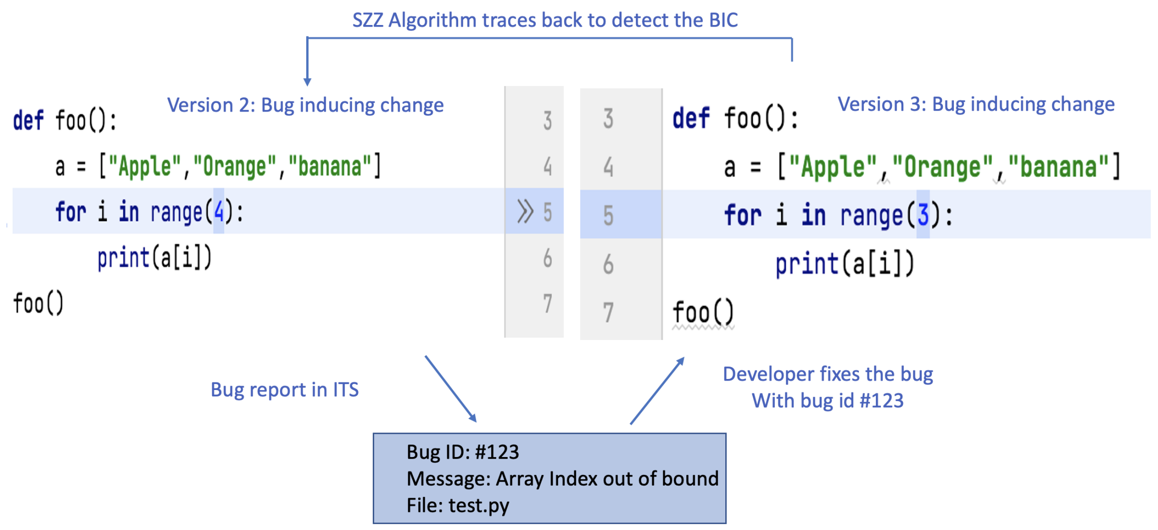
Figure 2.3: How GitHub works

GitHub is a popular distributed version control system. The way Git thinks about the data is a significant difference between Git and other VCS systems. Most of the information is stored by the other VCS system as file-based changes. These systems are known as delta-based version control as they think of the data as a set of files and store the changes over a set of files. However, Git stores its data as a series of snapshots of a miniature file system. Every time a developer or collaborator makes changes to their files and saves the changes, Git takes a snapshot at that moment of the whole state of his or her project, and uses that snapshot to save a reference. The state that was saved in the Git server, including the changes made by the collaborator, is known as commit. Conceptually, the collaborators commit every time they make some changes in the project to same them in the local repository and then push them in the central distributed server (Figure 2.3).

2.3 Bug Inducing Commits

A software bug is a problem or error in the software systems that causes the system to crash, produce invalid output or behave in unexpected ways. When developers code, they often unintentionally produce bugs which costs them in the long run. Sometimes while integrating a new feature to the system or making changes in code for refactoring or updating the system, new bugs are created. Sometimes all the testing cannot detect the bug before it goes to the production.

While using the version control system, developers integrate their changes into the main code base using the commit operation. Commits that induces bugs in the software system are known as bug-inducing commits. These commits contain important information about when and how the bugs were induced[54]. A plethora of research has been done to understand the characteristics of bug inducing-commits, how to locate bug-inducing commits or predict this bug-inducing commit in the software system life cycle. Researchers observed that whenever developers try to fix a bug or patch, they try to find the bug-inducing commit in order to find the source of the bug [54].


SZZ Algorithm traces back to detect the BIC
Version 2: Bug inducing change
Version 3: Bug inducing change
Bug report in ITS
Developer fixes the bug
With bug id #123

```
def foo():
    a = ["Apple","Orange","banana"]
    for i in range(4):
        print(a[i])
foo()
```

```
def foo():
    a = ["Apple","Orange","banana"]
    for i in range(3):
        print(a[i])
foo()
```

Bug ID: #123
Message: Array Index out of bound
File: test.py

Figure 2.4: How SZZ algorithm detects bug inducing commits

SZZ algorithm is a widely used algorithm to detect and find this bug-inducing commits. Once a bug is introduced in the software system, that bug needs to be fixed. Whenever the developers realize that a bug has been reported in the issue tracking system, they immediately try to resolve the bug and commit the fixed version of the code. Commits that are performed to fix a bug or issue are known as bug fixing commits. To proceed with detecting bug-inducing commits, the SZZ algorithm starts with analyzing the bug-fixing commit [45]. We can see an illustrative example of how the SZZ algorithm works in Figure 2.4. A simple function of printing the elements of an array has been committed in V2. Here we can see that an array index out of range bug has occurred, because the number of iteration in the loop is greater than the size of the array defined before. This commit is a bug-inducing commit. This bug is reported later in the Issue Tracker System with a Bug ID of #123. The developers later fixed the bug by updating the loop condition in V3, which is a bug fix commit for Bug ID #123. With the logs of the commits of V3, SZZ algorithm traces back to the commit in which the bug was initially induced and detect it as a bug-inducing commit. It can also find who introduced the bug and when from the code history using git blame command.

2.4 Machine Learning

Machine learning is a field of artificial intelligence that deals with statistical techniques that give computers the ability to learn from data without being explicitly programmed [52]. Tom Mitchel gave a famous and more formal description of machine learning - "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [40].

In general, a machine learning algorithm is used to map some input features to an output using a non-linear transformation. For example, we can consider a rice-variant identifier problem. Say, the input features for this problem are the length of the leaf, the width of the leaf and colour of the leaf. The output is to which variant of rice the given leaf belongs to. A machine learning algorithm can generate a function that can map the input features to the corresponding output labels. This function is known as a model or trained model. After learning the mapping, the model can even map a label without previous experience of that particular feature set.

Figure 2.5 shows the flow chart of the training phase of a machine learning algorithm. Given the input and output, a machine learning system will extract the features and feed them to the machine learning algorithm along with corresponding labels. The algorithm will output a trained model.

Figure 2.6 shows the block diagram of the testing phase of the performance of a machine learning algorithm. In this phase, a separate set of input features are fed to the trained model. The trained model predicts the label for each feature set and the predicted label are matched with the actual label to calculate the accuracy of the model.

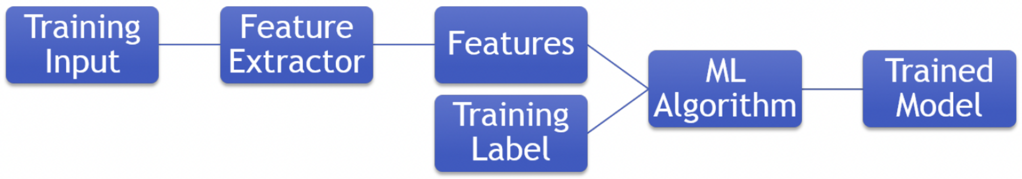


Figure 2.5: Training phase of a machine learning algorithm

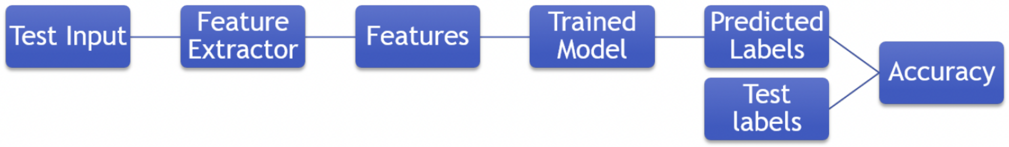


Figure 2.6: Testing phase of a machine learning algorithm

2.4.1 Feature Extraction

Feature extraction is the first and one of the most critical tasks in machine learning. For any input, it is required to extract the useful features that are directly responsible for the corresponding output label. Let's reconsider the rice-variant identification problem. If the input is images of the rice-plants, then firstly, it is required to detect the position of the leaves from the images. Later, using the position of the leaves, the length, width and colour of the leaves can be extracted. Finally, the extracted features will be fed to the machine learning algorithm. A note to remember that for a particular input, a large number of features can be extracted. However, only a very few of them are responsible for the corresponding output label. These characteristics made feature extraction an iterative process that requires several iterations of the whole training phase to find the best feature set.

2.4.2 Decision Tree

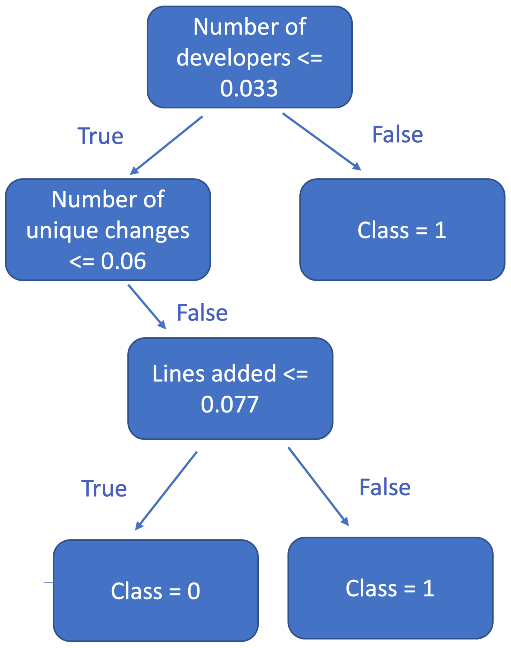


Figure 2.7: Overview of decision tree

Decision Tree is a popular predictive modelling algorithm in the field of statistics and machine learning. It is a classification algorithm that takes decisions based on a series of questions. The root node represent the primary decision you want to make. Branches of the nodes represents the available options or a course of actions to make the particular decision and are indicated with an arrow line. The nodes that are attached at the end of the last branches are known as leaf nodes. Leaf nodes represent the decision made based on the parent nodes and represent a class. For example, Figure 2.7 shows us a sub-tree of detecting bug-inducing commit. Here, we can observe how the decision tree is working to detect if a commit is bug-inducing or not based on three features: number of developers, number of unique changes and number of lines added in a particular commit. Here, the bug inducing commits are class 1 and non bug-inducing commits are class 0. Here, we can see that depending on a value of a particular feature, the decision tree takes a decision in each node.

The splitting of a decision tree is based on a set of rules or algorithms based on the features of the data. Information Gain and Gini index are the two best-known metrics used by the algorithms to determine the features which separate them into different classes and usually construct the tree top-down.

The decision tree is non-linear and flexible in the way they learn, which is the reason why they have low bias. However, this low bias causes a high variance in the algorithms because they learn from noisy or unrepresentative training data. This leads the modes to overfitting and making overconfident predictions.

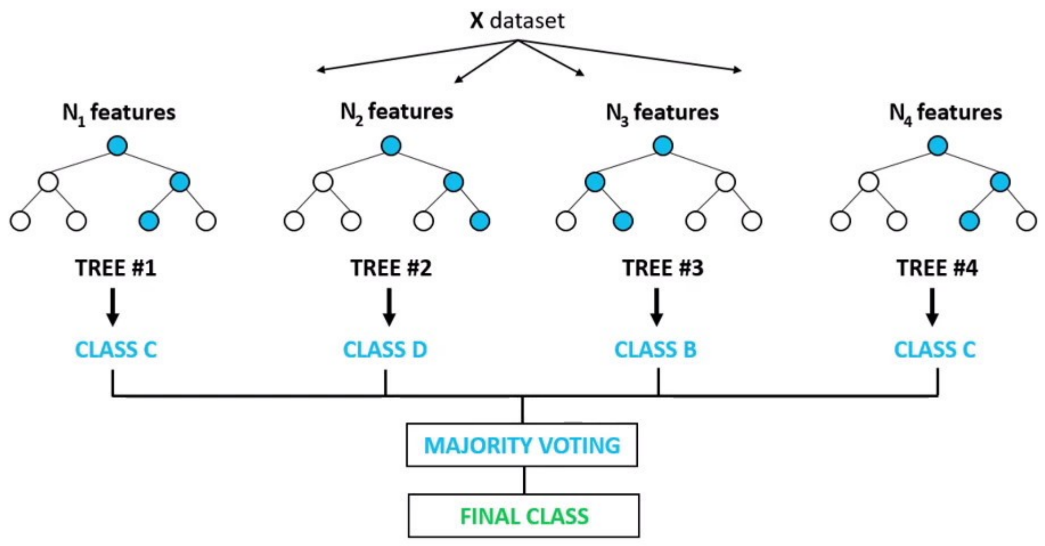


Figure 2.8: Overview of random forests

2.4.3 Random Forests

Decision trees are used widely as classifiers because of their high execution speed. Nevertheless, the trees are designed to perfectly fit all samples in the training set that causes overfitting. Tim [24] introduced a method of constructing a tree-based classifier whose capacity can be extended in order to increase the accuracy of both training set and test set.

Random forest is an ensemble learning method where each classifier in the ensemble is a decision tree classifier. This collection of classifiers is called a forest. During classification, each of the decision trees gives their votes and the result is based on the majority of votes. Figure 2.8 gives an overview of the random forest ensemble method.

The set of the attributes for a particular tree in the forest is randomly selected. The size of the attributes subset is determined by log2 N + 1 where N is the size of attribute set. To construct the decision trees, Classification and Regression Tree(CART) [7] and C4.5 [26] algorithms are used. Caruana et el. [10] showed that in general, for classification problems, random forest performs most consistently in all dimensions.

2.4.4 Logistic Regression

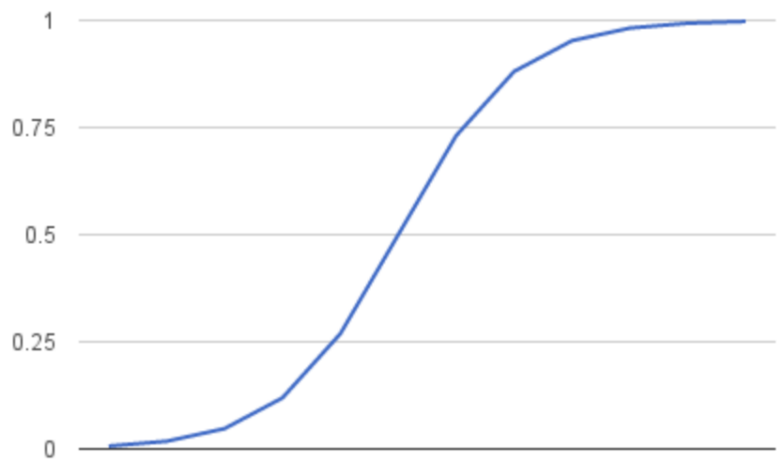


Figure 2.9: Logistic Function

Logistic regression is used to model the probability in binary classification problems. It uses a logistic function to classify a dependent variable which normally has the probability of having two outcomes by analyzing the relationship between one or more independent variable. These independent variables can be numerical or categorical, but the dependent c=variable always has to be categorical. The logistic function is also known as the sigmoid function that is an S-shaped curve that can take any real value as input and map it in a value between 0 and 1. The logistic function is defined as,

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

Where x is the input value which is going to be transformed into a value between 0 to 1, but not exactly those values. Figure 2.9 shows a plot where the curve represents the logistic function. Here, the value from -5 to 5 are mapped between the range of 0 to 1 using the function.

2.5 NASA-TLX

Table 2.1: An overview of NASA-TLX (adapted from [39])

Item	**Endpoints**	**Description**
Mental demand	1 - 10 Low / High	How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?
Physical Demand	1 - 10 Low / High	How much physical activity was required (e.g.. pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?
Temporal Demand	1 - 10 Low / High	How much time pressure did you feel due to the rate or pace at which the tasks occurred? Was the pace slow and leisurely or rapid and frantic?
Performance	1 - 10 Low / High	How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?
Effort	1 - 10 Low / High	How hard did you have to work (mentally and physically) to accomplish your level of performance?
Frustration level	1 - 10 Low / High	How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?

The NASA Task Load Index (NASA-TLX) is a widely used subjective workload assessment tool. It is used to assess a task, system, or team's effectiveness and other aspects of performance [39]. It was developed by the Human Performance Group at NASA's Ames Research Center. It took them over a three-year time frame and more than 40 laboratory simulations to complete the development cycle [23]. Here, the total workload of an individual is divided into six subjective sub scales Mental Demand, Physical Demand, Temporal Demand, Performance, Effort and Frustration. All these scales are measured in a range between 1 to 10 and then analyzed to understand the workload. Figure 2.1 gives us a better understanding of how the workload is measured using this system.

3 Related Studies

In this chapter, we discuss the related literature of our thesis. Section 3.1 discusses the bridge between Human-Computer Interaction and Software Engineering. Then, Section 3.2 discusses how applying machine learning approaches in software development effectively leads us towards a successful future in the software industry. Bug Inducing Commits are discussed in Section 3.3. Finally, Section 3.4 discusses the existing JIT supports for bug-inducing commits in the literature briefly.

3.1 User Interaction and Software Engineering

User interaction and usability patterns are significantly connected with software engineering. Folmer et al. [18] worked in the gap between software engineering (SE) and human-computer interaction (HCI). They proposed a bridging pattern which consists of a user interface part and an architecture/implementation part. This pattern extends interaction design patterns by presenting the generic implementation and architectural considerations. They presented four cases: selective Undo, multi-channel access, Wizard, and single sign-on. Bridging patterns can be used for architectural analysis. When the generic implementation is known, software architects can assess what it means in their context. They can decide whether they need to change the software architecture to support these patterns.

Another usability problem of the developers has been worked on by Murphy-Hill et al. [42]. They tried to provide support to the developers in the area of code smells. Code smells can warn against deterioration and encourage the redesign and refactoring necessary to keep the code clean, but only when programmers are aware of code smells. The paper described a smell detector that uses an interactive ambient visualization to help make programmers aware of smells and make informed as well as confident refactoring judgements. They built the tool as a plug-in for the Eclipse IDE. In order to design the plug-in, they proposed a set of guidelines that can aid the programmers' JIT support in their work. They also performed a user study in order to evaluate their work considering four hypotheses related to the quantity of detected code smells, the subjectivity of code smells, the confidence of the programmer while making refactoring judgements, and the validity of the guidelines they followed in order to build the tool.

3.2 Machine Learning and Software Engineering

Integrating Artificial Intelligence (AI), with software engineering (SE), is a new research trend. Foutse et al. [31] discuss challenges, new insights, and practical ideas regarding the engineering of machine learning-based systems. Meinke et al. [38] focused on the questions which should be answered before applying any machine learning algorithm to existing software engineering problems. These questions include what class of learning model is appropriate to solve the problem, is there any machine learning model that exists to adopt the required model and work for typical instances and sizes of the problem and if anyone else has solved this similar kind of problem using machine learning.

Amershi et al. [2] recently worked on a case study on how Microsoft integrates these customer-focused machine learning-based approaches in their software applications in order to introduce new AI features. They interviewed Microsoft employees on how they handle the challenges that arise in the development of these AI features, including the end-to-end pipeline support, data availability, collection, cleaning and management and most importantly, education and training to adopt the integration of these new technologies. In addition, they discussed the application of machine learning workflow in software engineering, best practices of machine learning, a custom machine learning process maturity model which can assess the success of software developers in the integration of AI in software development and the underlying differences between the machine-learning-centric SE approaches and previous SE approaches.

3.3 Predicting Bug Inducing Commits

There has been a plethora of research done on bug-inducing commits. It was not possible to study the origin of bugs in large-scale scenarios until the introduction of the SZZ algorithm. This algorithm is primarily introduced by Sliwerski, Zimmermann and Zeller [53] where they attempted to identify those changes that caused bugs or errors. They first indicate the bug from a bug report of the bug database, and then they indicate the association from the version archive to get the location of a probable fix. Then they find the earlier change at that location which was applied before the bug was reported. Later, the algorithm was improved by Kim et al. [34]. Researchers have found that there are many limitations of this algorithm and improved the algorithm by proposing different variations [47], [14], [15], [45].

Wen et al.[54] collected bug-inducing and bug-fixing commits from seven large open source projects to explore the correlations between them in terms of code elements and modifications. First, they explored how these significant correlations make the SZZ algorithm, the most widely adopted algorithm to detect bug inducing commits, imprecise. Then they observed that supporting the information of bug-inducing commits can significantly boost the performance of existing automated fault localization and program repair techniques.

Many researchers applied supervised and unsupervised classification algorithms to detect defect-inducing changes in software systems. Yang et al. [32] proposed a technique called change classification to predict hidden software bugs. Their technique used a Machine Learning classifier to determine whether a new software change will produce bugs or not. Features extracted from the revision history of software systems stored in its software configuration management repository used to train the classifier. Then the classifier can predict new software changes, whether it will produce bugs or not, with 78% accuracy and 60% recall on average. Yang et al. applied unsupervised approaches to build prediction models in effort-aware "Just-in-Time (JIT)" defect prediction. They showed that under cross-validation, time-wise-cross-validation and cross-project prediction settings, simple unsupervised models outperformed many state-of-the-art supervised models. Yang et al. [55] proposed a DL-based approach *Deeper* to predict defect-prone changes to improve the performance of JIT defect prediction. *Deeper* is divided into two phases: one is the feature selection phase and another is the machine learning phase. Their approach extracts a set of expressive features from an initial set of basic features by leveraging the Deep Belief Network in the first phase. In the second phase, they build a classifier based on the selected features.

3.4 Just-In-Time Support

An explainable JIT defect prediction framework is proposed by Khanan et al. [30], which automatically generates feedback for developers. This app is named JITBot which takes pull requests as well as related commit information and detects the riskiness of a commit, explains the reasons for the risk and suggests improvements in order to mitigate the risks. The tool in integrated with GitHub CI/CD pipeline for continuous monitoring and analysis. In this way, it helps the code reviewers prioritize the pull requests by understanding which commits need to be fixed and saves a significant amount of time and effort in code reviews.

Rosen et al. [51] worked CommitGuru, a language-agnostic analytics and prediction tool, which performs analytics and predicts risky software commits. It is a web app where one can upload any Git SCM repository and identify recent commits that are more likely to contain bugs and better understand the overall quality of a software project. This can be useful to find those changes to prioritize verification activities such as code inspections, as well as seeing which quality change metrics might be good indicators for bugs. Finally, to facilitate future research in the area, users of Commit Guru can download the data for any project that is processed by Commit Guru with a single click. Several large open source projects have been successfully processed using Commit Guru.

The research conducted by Catolino et al. [11, 12] showed that different metrics used for traditional bug-inducing commit detection are not applicable in the context of mobile applications. They conducted an empirical study on which metrics are applicable to detect bug-inducing commit in mobile applications and cross-project bug prediction models. They also showed the impact of different classifiers and ensemble techniques on the performance of prediction models. InfoGain filtering technique used to filter out unnecessary features to avoid multicollinearity. An experiment conducted on 14 mobile applications and 43,543 commits, showed that Naive Bayes achieved the best performance compared to other classifiers. Kamei et al. [27] also conducted an empirical study on the application of cross-project models in the context of "Just-In-Time" prediction of defect-inducing changes. At the beginning of the development phase of a software, it is difficult to acquire a large amount of training data. The authors showed how the prediction models learned from other software systems with a high volume of historical data can be applied in "Just-In-Time" detection of defect-inducing changes of other software systems.

Kamei et al. [28] conducted a large-scale empirical study on JIT quality assurance of large software systems. They proposed a prediction model that is capable of detecting defect-prone ("risky") software changes in a software system. While most of the quality assurance techniques work on critical files or packages level, the prediction model proposed in this paper works on change-level when developers commit code to their private or to the team's workspace. In order to build the prediction model, the authors used different factors based on the characteristics of software change, such as the number of added lines and developer experience. Using 11 large software systems (six open sources and five commercial projects), the authors evaluate the performance of their proposed model. Based on the accuracy and recall scores, which are 68% and 64%, respectively, the authors conclude that "Just-In-Time Quality Assurance" might help the developers to focus on the riskiest changes of software at the change-level and thus, early detection of these risky changes may reduce the overall cost of developing high-quality software systems.

3.5 Summary

From the above discussion, we see that there are some studies to mitigate the gap between human interaction and software engineering. Different machine learning algorithms are used to resolve software engineering problems including bug-inducing commit detection. Nonetheless, the real-life implementation of those bug-inducing commit detection algorithms is limited. For example, CommitGuru [51] detects the bug-inducing commits after they are merged into the central repository. JITBot [30] uses pull requests to detect whether there is any bug in the code or not. This might help the team leader or senior developers during code-review. However, none of these approaches prevent the developers from writing buggy codes and introducing them to the central repository. In contrast to the former studies, our proposed approach warns the developers before performing bug-inducing commits in real-time. It will help the developers to write better code in a shorter time. Furthermore, the software industry will be able to significantly reduce the bug fixing cost as well as save time and resources.

4 Analysing the opinion of developers community on real-time support for bug inducing commit

The existence of software bugs dramatically affects software reliability, quality and maintenance cost. Achieving bug-free software is known as rare and hard work, even when the software is tested carefully [22]. Most of the time, software bugs are not detectable until it is merged with the whole software system. Although there has been much work on automating the testing phase, fixing a bug after its presence has been discovered is still a duty of the programmers [3]. Figure 4.1 shows us how the cost of fixing bugs gets more expensive with the time. The earlier it can be detected, the less time and resources are needed to fix the problem. In our study, we want to prevent the bug from being introduced in the software system at the very beginning. We work on a solution that will detect whether there is a probability of having a bug or not while performing the commit operation. If the developer knows their code might contain bugs, there is a significant possibility that they would not commit those changes and the bug would not be introduced in the software system.

To understand what kind of support we can provide to the developers to prevent bug-inducing commits, we need to understand their perspective on this topic. User research is the only way to understand the people and their preferences who will use our work. Researchers can use users to inspire their design, to evaluate their solutions, and to measure their impact [41]. Therefore, we conducted a survey to understand the users to make our work more fruitful.

4.1 Study Design

While going through related studies, we found many studies on how to detect bug-inducing commits. However, we could not find many real-time uses. Therefore, the goal of this study is to identify the severity of the problem, what kind of support the developers need to detect bug-inducing commits and at what time this tool should act. Based on this goal we conduct a user study to answer the RQ1 stated in Section 1.3. We divide RQ1 into the following fine-grained research questions.

- RQ1: How often do the developers introduce bug-inducing commits?
- RQ2: During the software development lifecycle, when do the developers need support to identify bug-inducing commits?

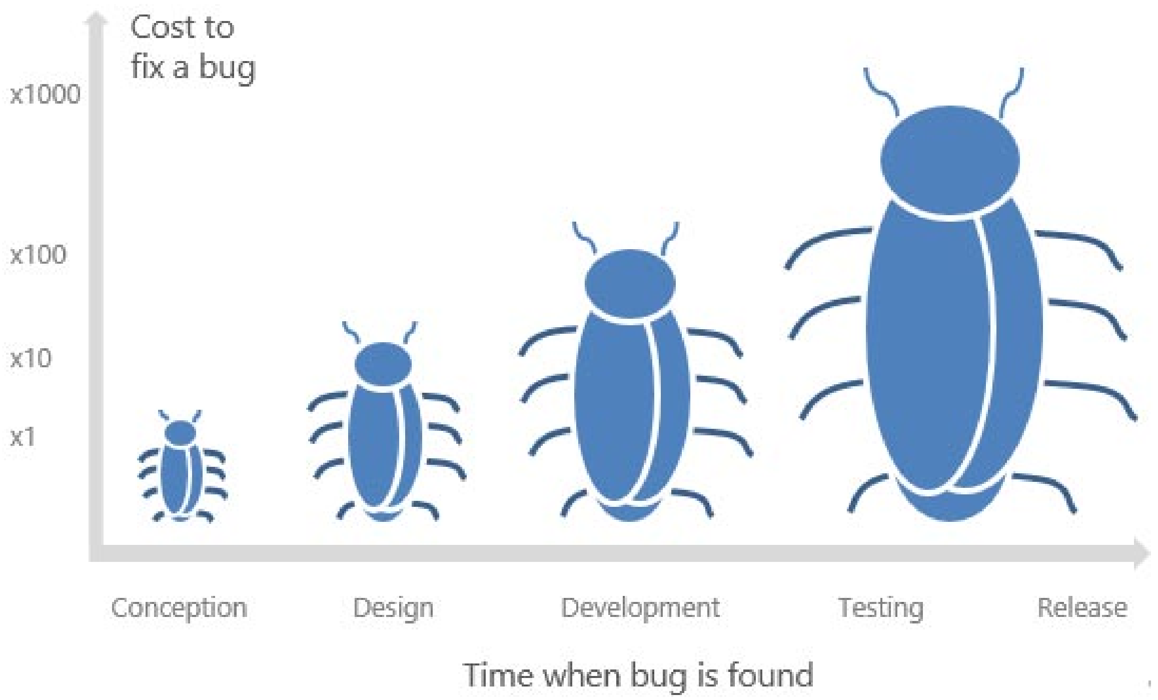


Figure 4.1: Cost of fixing bug in different phases of software life-cycle [48]

- RQ3: What kind of tools do the developers prefer to identify bug inducing commits?

4.2 Research Methodology

We performed a survey that included 20 developers to identify the need for this kind of technique. Our participants share different levels of experience working on different platforms to ensure the generalizability of the survey result. Based on the survey results, our study focused on how to make existing approaches real-time and detect bug-inducing commits before the developer performs a commit in order to prevent the bug earlier before it is introduced to the central repository (e.g., Git). First, we kept the survey open for one month. However, we did not receive enough responses. For this reason, we extended the timeline for another month and sent a reminder email to the participants to collect the maximum number of responses.

4.2.1 Choosing the evaluation method

There are many different evaluations method like observation, interview, questionnaire, think-aloud, co-discovery, formal experiments, which we can perform during the life-cycle of a system. But which evaluation technique one should choose for their system is a big challenge. We have studied almost all the methods above and have gone through the pros and cons of each method. As our goal is dependent on the opinions of the developers' community, it is really important for the participants to understand the importance of the problem we are trying to solve. Again, it is a matter of concern that if they are able to express their opinion in a convenient way. Therefore, we decided to go for the questionnaire method so it could involve more participants. The more participants we can involve, the better feedback we will be able to get. If we go for only the interview or observation method, getting the accurate data is challenging because there might be a lack of insight of participant's decisions, and our presence can affect their thinking. However in questionnaire method, we can select the question we actually wanted to ask. Again, different types of questions can be asked and an option can be given to give their overall opinion. A wide subject group can also be reached, and it does not need the presence of the evaluator, so the chance of bias is less. Again, we can easily quantify the results. Because of these reasons, we chose the questionnaire method. Nonetheless, the problem is sometimes, understanding why the participants are choosing their options is a bit tricky using a questionnaire. Although we can do the questions and get feedback through a questionnaire, to understand the real situation, we decided to kept some open-ended questions in the questionnaire. We let the participants answer the questions and analyze to understand what they are thinking to understand more about how we should design the real-time support.

4.2.2 Survey Design

We used Google forms for the questionnaire. The questionnaire contains both open-ended and closed-ended questions. Open-ended questions to collect participants' opinions were added to find out why participants chose certain options and made certain decisions, if they have anything better to suggest, and if they would like to share an opinion or thought. Most of the other questions were multiple-choice questions. The time duration was 15-20 minutes to complete the questionnaire. The questions of this survey can be found in Appendix A. Since the answers to the open-ended questions were concrete, so we directly present them in the result Section 4.3.

4.2.3 Survey Validation

We invited two professionals from our contact list who are highly experienced in the field of software engineering to validate the questionnaire of this study. The pilot study was done to validate our survey by observing the wording, sequence and consistency of our questions, and the time to complete the questionnaire.

4.2.4 Prerequisites For Recruiting The Participants

To identify and contact survey candidates, we make a set of prerequisites to identify prospective participants. We describe our prerequisites below.

Developers From Popular GitHub Projects

The number of stars of a GitHub project is an excellent indicator of its popularity and quality. Ranking the GitHub projects by its number of stars is known as Gitstar ranking [35]. We selected our participants from 20 popular GitHub projects with the highest number of stars. It is an indication of the quality of their contribution to the software development community.

Developers With Less Industry Experience

We selected a large number of developers who are comparatively new to the industry. As senior developers are mostly involved with the design and maintenance of a software project, most of the code is initially written by junior developers. For this reason, we considered their opinion valuable for our study.

Developers With Issue Solving Experience

As we are asking questions about how the developers want to detect a bug-inducing commit, it is obvious that our goal is to create an opportunity for the developer to fix the bugs before committing. Therefore, alongside recruiting developers with less industry experience, we also selected developers that have experience in fixing bugs. We chose those developers who have solved at least ten bug-fixing issues in those popular GitHub repositories.

4.2.5 Recruitment of Survey Participants

We chose developers from 20 popular GitHub projects by Gitstar ranking [35]. We filtered the developers of these projects according to their industry experience and issues fixing experience. We measured their *experience* by counting the number of days from their first commit in GitHub. Then, we measured their *issue fixing experience* by counting the number of solved bug report issues. We collected their email from the corresponding GitHub account. Finally, we sent them an email describing the goal of the survey and asked them to participate.

We did not include any incentive to participate in the survey. The developers participated in the survey for the sole reason of enhancing the knowledge of bug-inducing commit prediction. About 200 developers were contacted. We received responses from 20 of them. Given the prerequisites (Section 4.2.4) and diversity (Section 4.3.1) of the participants, 20 developers were enough to reach a meaningful result.

4.3 Results and Discussion

4.3.1 Participants Information

The developers for our survey were chosen from 20 popular GitHub projects which have the highest number of stars. Three of the developers have seven years of software development experience, five have five years of experience, seven of them have experience of two to three years and the other five participants have been working as a software developer for six months to one year. Four of the developers identified themselves as female and the other participants as male.

4.3.2 RQ1: How often do the developers introduce bug-inducing commits?

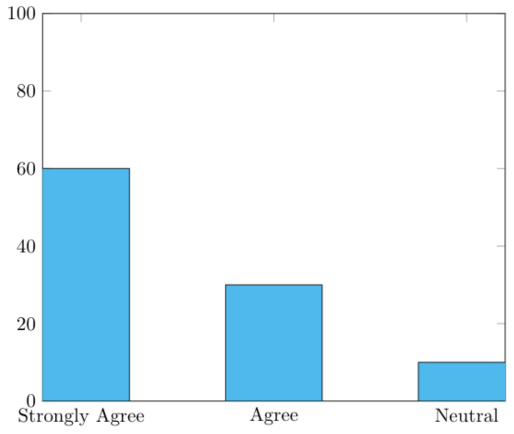


Figure 4.2: Do you often find bugs in your code after you commit your version of code?

The first question to the participant was whether they often find bugs in their code after they commit their versions. 60% of them strongly agreed, 30% of them agreed and 10% of them, which means one was neutral which is neither agreed, nor disagreed (Figure 4.2). The second question was, "does it often introduce new problems in future?" This question received a similar type of answer. 15 strongly agreed, 4 agreed and one was neutral (Figure 4.3). Then to determine whether the problem we decided to solve is significant enough or not, we asked them if they would like to find bugs before they perform a commit operation and 19 out of 20 participants agreed that they would like to (Figure 4.4).

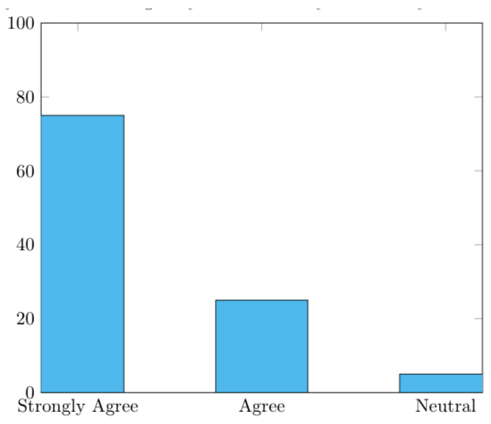


Figure 4.3: Does it often introduce new problems in the future?

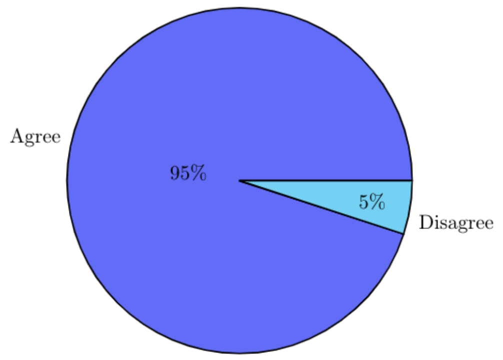


Figure 4.4: Would you like to detect bugs in your work before you perform the commit operation?

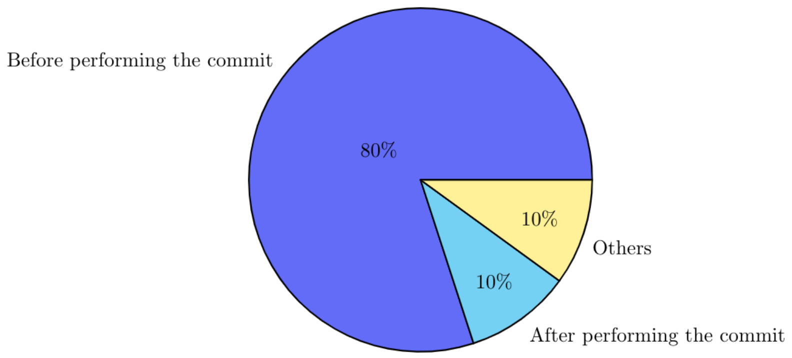


Figure 4.5: When do you think detecting bug-inducing commits are helpful?

4.3.3 RQ2: During the software development lifecycle, when do developers need support to identify bug-inducing commits?

The participants were asked when they prefer to detect bug-inducing commits and 80% of them answered that they would like to detect them before the commit operation, 10% of them answered that they would like to detect it after the commit has been performed and 10% of them chose others (Figure 4.5). There was an open-ended question for them who chose others to write about when they wanted to detect them. Some said they wanted to detect in the moment of commit, some said before the pull request and most of them did not answer the question as that was not a mandatory question.

4.3.4 RQ3: What kind of tools do the developers prefer to identify bug inducing commits?

The participants were given two options: an IDE-based plug-in or a command-line-based tool and asked which one they would prefer. 90% of them agreed that an IDE-based plug-in would be good support (Figure

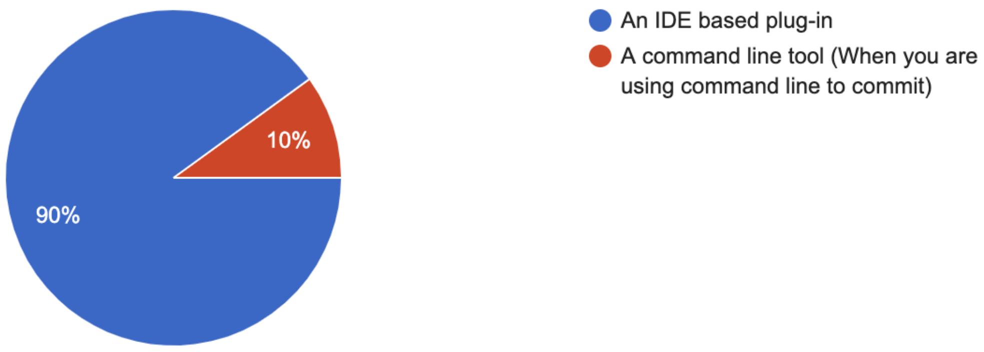


Figure 4.6: What kind of support do you think might help you to detect bug inducing commits?

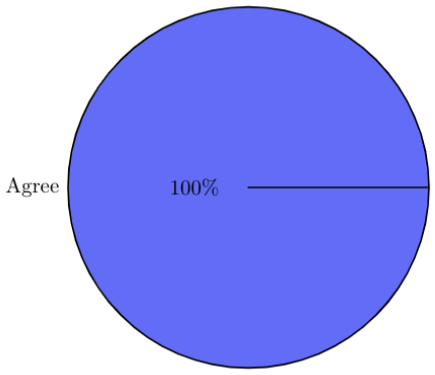


Figure 4.7: Would you like both kind of support?

4.6). Additionally, there was a question that if there is an option which can give better support according to their opinion, and most of them answered that they would like to have both kinds of support. Therefore, either they commit using an IDE or a command line, it would not be a problem (Figure 4.7).

The last question was if they have a bug-inducing algorithm of their own, would they like to build a new plug-in from scratch on their own or support where they can select features and add their algorithm so the plug-in would use their customized algorithms to detect bugs. 80% of them answered that they would like to add that kind of support and 20% of them said that they would like to build their own plug-in from scratch (Figure 4.8). Upon further investigation, we found that all of the developers who opposed the customization options are newcomers to the industry by checking their years of experience.

In summary, analysing the survey results, it is found that the developers suggest having a real-time bug detection tool that will find a bug inducing commit before committing, considering it will help them in the long run.

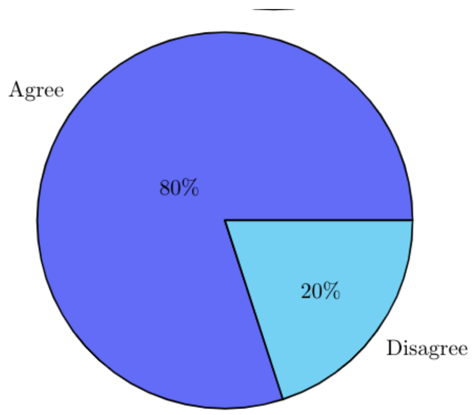


Figure 4.8: If you have your own bug-inducing commit algorithm, would you like support where you can add your bug inducing algorithms in a plug-in to customize it?

4.4 Threats To Validity

4.4.1 Internal Validity

Threats to internal validity relate to the design of this study. In our case, it questions whether this survey is designed in a proper way and is able to collect the missing knowledge for making bug prediction support that the developers need. To mitigate this threat, as mentioned in Section 4.2.3, two highly experienced professionals in the field of software engineering validated our survey prior to sending it to the developers.

4.4.2 External Validity

Threats to external validity relate to the generalizability and reliability of the survey response. To mitigate this threat, we carefully choose our prospective participants. We created a set of criteria, as mentioned in 4.2.4, to filter out the developers who perfectly portray our targeted participant-base. The number of participants in this survey is 20, which is not a large number. However, the response from the participants agrees roughly 80% in the quantitative questions. Furthermore, in the qualitative questions, our knowledge from the result is saturated. Therefore, the number of responses was sufficient.

4.5 Conclusion

User research is an essential part of any kind of study. It is important to periodically assess our activities to ensure they are as effective as they can be. User research and evaluation can help researchers to identify areas for improvement and ultimately help them realize their goals more efficiently. Software bug prediction at the initial stages of software development improves significant aspects such as software quality, reliability, and efficiency. However, in the majority of software projects which are becoming increasingly large and complex programs, bugs are a severe challenge for system consistency and efficiency. Since all these challenges have to be faced by the developers, knowing how an external system can provide efficient support to the developers is very much needed.

In this study, we answered the RQ1 mentioned in Chapter 1. We understand that the developers who are comparatively new to the industry are more likely to want to have real-time bug detection support. Also, they prefer properly working algorithms instead of doing experiments. On the other hand, the participants who have spent a good time in the industry are eager to have more configurable options so that they can use their own datasets, features and algorithms which will make them more comfortable in their field. Surprisingly, some of the developers do not prefer to detect bug-inducing commits before performing the commit operation. Nevertheless, it is understandable that as any kind of tool does not provide 100% accuracy, the developers do not want to spend their time without becoming sure about the occurrence of a bug.

5 Commit-Checker: A toolchain for bug inducing commit detection using machine learning models

Chapter 4 shows that developers need appropriate tools and supports to detect bug-inducing changes immediately after making changes to the software. In software quality assurance, we have the concept of "Just-in-Time (JIT)" quality assurance. However, JIT only works after a change has been committed to the central repository. To prevent bug-inducing commits in the first place, we introduce the idea of "real-time" quality assurance. In this paradigm, the developers are warned of the presence of potential bugs in their changes prior to committing. It is quite common to change a software system to enhance its features or fix existing bugs. Sometimes these changes might induce new bugs. It can be especially challenging to detect bugs in files or packages if the software system is large and the defects are identified after it has passed a long time. If it is possible to identify whether a change is bug-inducing or not immediately, the developers can take urgent actions to fix it and reduce the occurrence of bugs in their software.

During the last couple of years, several research papers proposed JIT bug prediction approaches that are capable of detecting bug-inducing changes in a software system [28, 27, 19, 55, 11, 12]. In the context of detecting bug-inducing changes, JIT bug prediction is more pragmatic than identifying bug-prone modules at a regular interval. This is because JIT bug prediction assists the developers in checking and fixing the bugs as they occur with promptness to ensure software quality in the development process. It is also more practical in the sense that developers can review and test the changes on the fly as the changes are still fresh in their minds [28]. JIT bug prediction has several advantages [55]. First of all, it tends to check and test a small portion of code fragments, i.e. it focuses only on individual changes compared to changes in the entire files or packages. Secondly, assigning developers to fix the bugs takes less time because we can easily identify the developers of the changes that induce bugs in the software.

As mentioned earlier, the main limitation of JIT support is that it can only identify a bug-inducing commit after it has been added to the version history. Intuitively, it is more helpful if the bug-inducing change is not added to the version history at all, i.e. detecting a bug-inducing change before it is committed. The aforementioned real-time support can help accomplish this.

Along with the lack of real-time support, current research studies do not provide any IDE-based plug-in nor any tool support to detect bug-inducing changes on the fly during commit operations. However, our survey result in Chapter 4 shows that the developers need both IDE-based plug-in and command-line tool support to benefit from the quality assurance researches.

In this thesis, to address these problems, we develop a machine learning-based plug-in support for VS Code to assist developers in ensuring real-time detection of bug-inducing changes. We consider random forests (RF), decision trees (DT) and logistic regression (LR) as the ML models. Decision trees (DT) achieved the highest accuracy among the candidate models. To train the model, we used the dataset provided by Gemma et al. [11, 12]. The dataset consists of 14 open-source Android applications containing 30,341 commit operations extracted from the Commit Guru platform [27]. The dataset is unbalanced as the number of bug-inducing changes is comparatively lower than non bug-inducing changes. To balance the dataset, we apply undersampling and oversampling techniques as suggested in the literature [12]. We use 80% of the data for training and the other 20% for testing the model.

After providing bug-inducing change detection support, we focused on providing customizability to the developers so that they can use our tools in a way that better suits their use cases. To this end, our toolchain provides configuration options regarding the choice of machine learning algorithms, datasets and features.

5.1 Research Questions

In RQ2 stated in Section 1.3, we aim to find ways to support the developers to detect bug-inducing commits. To answer this research question, in this study, we developed a toolchain to provide real-time bug-inducing change detection support to the developers. To identify the means of providing bug-inducing change detection support and evaluate the performance and usability of our provided support, we divide RQ2 of this thesis into three fine-grained research questions in this study.

RQ1: What are the ways to interface the probability of having bugs for developers?
The developers use different environments to perform different version control operations. In our study, we explored different options of interactivity to provide support for the developers so they can easily detect the bug-inducing commits in their preferred environment.

RQ2: How much time and effort is needed to use interactivity among the developers in order to detect bug inducing commits?
We perform a user study exploring how the visualization of the probability of having bugs impacts the developers in terms of time and effort. We conducted a survey to understand the time requirements and used NASA-TLX to assess if the tools are too demanding. To verify and understand the reasons for some answers, we also interviewed some participants. This research question gives us an idea of the effectiveness of the interactivity tools.

RQ3: How can we define usability patterns to solve usability problems of detecting bug-inducing changes in software?
In this research question, we try to investigate usability problems of existing research methodologies. Finally, our main focus will be finding usability patterns that how we can relate the developers to their IDEs in order to improve their maintenance quality.

5.2 Research Methodology

Our previous study finds that the developer community supports a JIT tool to prevent bug-inducing commit very much. Therefore, we decided to build both an IDE-based plug-in and a command-line tool and use the same usability pattern in other environments.

5.2.1 Dataset Collection

We collected the dataset that was used in the study of Catolino et al. [12]. Based on the features of Kamei et al. [28], the dataset consists of 14-open source Android applications of different sizes and domains where the total number of commit operations is 30,341 (5.1). The dataset is unbalanced because the number of bug-inducing changes is relatively lower than the number of non bug-inducing changes. To make the dataset more general, the authors selected mobile applications having different domains, including commercial apps, frameworks and toolkits. All of these projects are uploaded to the play store and used by android users. Table 5.1 shows us the details of the projects with their play store URLs and GitHub URLs. The number of developers in these projects is roughly 60 on average, with a high standard deviation of roughly 85. Therefore, they can provide a complete overview of software maintenance in different sizes of teams. The dataset can be found in Appendix B.

5.2.2 Dataset Balancing

Predicting bug-inducing changes before commit operation is an imbalanced classification problem [28, 12]. It refers that in the training dataset the instances of majority class (e.g. number of non bug-inducing changes) dominate the instances of minority class (e.g. number of bug-inducing changes). From this collected dataset, we can also see that the skewness of the dataset is also high. There are different techniques available to handle this issue [28]. To resolve the imbalanced problem, we apply the undersampling and oversampling techniques. For undersampling, we randomly deleted instances of majority class until the majority class became equal to the minority class. For oversampling we apply Synthetic Minority Oversampling Technique (SMOTE) which is also used in former studies [12] to make the minority class equal to the majority class.

5.2.3 Feature Selection

Kamei et al. [28] proposed 14 features which are shown in table 5.2 to detect bug-inducing changes. Later Catolino et al. [12] applied the information gain (InfoGain) technique to figure out which features from Kamei's work are the most effective to train Machine Learning (ML) models for detecting bug inducing changes. For all the features, InfoGain technique quantifies the gain obtained by adding them to the predictive model and thus helps us to find out which ones have more impact on bug-inducing changes prediction. We can formally say that, for a predictive model M and features set $F = f_1, f_2, ..., f_n$, InfoGain calculates

Table 5.1: Dataset of 14 open source project

Projects	**Play Store URL**	**Repository URL**	**Number of Developers**	**Number of Commits**	**% of Buggy Commits**
Afwall	https://tinyurl.com/opd8628	https://tinyurl.com/m722ouo	20	1,127	37%
Alfresco	https://tinyurl.com/kfv93ez	https://tinyurl.com/ya533yya	5	1,449	2%
Android Sync	https://tinyurl.com/yafbk6f2	https://tinyurl.com/y9vcudjt	27	280	51%
Android Walpaper	https://tinyurl.com/hpl65mr	https://tinyurl.com/y7f5bjpt	1	605	22%
AnySoft Keyboard	https://tinyurl.com/k9s97zl	https://tinyurl.com/lqzxc3v	38	3,250	26%
Apg	https://tinyurl.com/cxwqp5n	https://tinyurl.com/y6vxu57x	56	4.363	30%
Atmosphere	https://tinyurl.com/ybdofq5h	https://tinyurl.com/y9ovae5z	101	5,757	38%
Chat Secure Android	https://tinyurl.com/lero26e	https://tinyurl.com/pxcupkk	35	2,869	30%
Facebook Android SDK	https://tinyurl.com/6ueeu7y	https://tinyurl.com/yctphsxw	64	636	28%
Flutter	https://tinyurl.com/y9q57wa8	https://tinyurl.com/yd25noy3	352	13,067	1%
Kiwix	https://tinyurl.com/ognzugp	https://tinyurl.com/ycvufxwn	60	1,571	22%
Own Cloud Android	https://tinyurl.com/8vfuemy	https://tinyurl.com/ptflhfe	70	7,144	22%
Page Turner	https://tinyurl.com/y9qgcffz	https://tinyurl.com/yc26yklh	16	193	22%
Notify Reddit	https://tinyurl.com/nd835ec	https://tinyurl.com/ydce46ge	2	231	26%

Table 5.2: Features proposed by Kamei et al. [28]

Scope	Name	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Number of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a bug fix
History	NDEV	Number of developers working on the files
	AGE	Average number of days since the last change
	NUC	Number of unique change to modified files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

the difference in entropy before and after adding features in the predictive model employing the following equation:

$$InfoGain(M, f_i) = E(M) - E(M|f_i) \tag{5.1}$$

Where the function $E(M)$ represents the calculated entropy when the predictive model incorporates a feature f_i and $E(M|f_i)$ represents the calculated entropy without considering the feature f_i. Entropy of a model M is calculated as follows:

$$E(M) = -\sum_{i=1}^{n} prob(f_i) log_2 prob(f_i) \tag{5.2}$$

The output of InfoGain technique is represented in a list in decreasing order where the most useful features have more information gain (maximum reduction in entropy) are placed in the top position. After that, Catolino et al. [12] used a threshold value of 0.1 to filter out irrelevant features and thus, they kept only six features out of fourteen features proposed by Kamei et al., which is described below and finally, we use these six features to train the predictive model (Table 5.3).

Table 5.3: Selection of features using InfoGain

Features	**Info Gain**
nuc	**0.25**
ld	**0.25**
la	**0.22**
nf	**0.19**
nd	**0.17**
ndev	**0.15**
ns	0.08
entropy	0.06
sexp	0.06
rexp	0.06
exp	0.05
lt	0.05
age	0.01
fix	<0.01

5.2.4 Predictive Model

Bug-inducing changes prediction is a binary classification problem. While choosing the machine learning algorithm to train the classifier model, we followed three criteria.

1. **Computational Cost**: While developing a user-facing tool, computational cost plays one of the major roles. We cannot guarantee that our users will have high configuration machines to run complex machine learning models. Therefore, we only choose the algorithms that produce light-weight and fast executable models.

2. **Complexity and Popularity**: As described in Section 5.3.4, we let our users to train their own models. In order to do so, the users need to understand or learn the models. Therefore, we only choose the models that not only have lower complexity but also are used for decades and plenty of resources available online to learn the models.

3. **Explainability**: In the past few years, explainability has become a critical concern in adopting the machine learning model. While certain algorithms can produce better results, if the user cannot understand how the model is taking the decision, it is unlikely that the software will gain the trust of the user.

Based on these criteria, we selected random forest, decision tree and logistic regression (LR) algorithms for model training. These models are lightweight, fast, easy to train, and easy to explain. The accuracy of the algorithms is shown in Table 5.4. From the table, we can see the logistic regression performed the best for our problem. The LR model gives an output value between 0 and 1 for each change during a commit operation. The threshold value of 0.5 is used to classify the change as bug-inducing or not. If the output value is greater than or equal to 0.5 then the change will induce bugs in the future. Otherwise, the change will be treated as non bug-inducing change [12].

Table 5.4: Accuracy of Machine Learning Models

Algorithm	Accuracy	f1-score	Precision	Recall
Random Forest	0.7815	0.4386	0.3907	0.5
Decision Tree	0.7889	0.7010	0.6952	0.7082
Logistic Regression	0.7790	0.4701	0.6082	0.5111

We split the dataset into training and testing set to train and test the model. 80% data is used for training and the rest of them is used to test the model performance. To evaluate the performance of the predictive model we employ commonly used metrics: accuracy, precision, recall and f-1 score. Here, we briefly discuss these metrics.

- **True Positive (TP)**: It defines the correctly predicted instances by the classifier. That is the actual class was defect inducing and the classifier also predicted as defect inducing.
- **False Positive (FP)**: It defines the incorrectly predicted instances by the classifier. That is the actual class was non-defect inducing and the classifier predicted as defect inducing.
- **False Negative (FN)**: It defines the incorrectly predicted instances by the classifier. That is the actual class was defect inducing and the classifier predicted as non-defect inducing.
- **True Negative (TN)**: It defines the correctly predicted instances by the classifier. That is the actual class was non-defect inducing and the classifier also predicted as non-defect inducing.

Accuracy: Accuracy is defined as the ratio of total number of examples correctly classified by the classifier to the total number of examples in testing dataset.

$$Accuracy = \frac{T_P + T_N}{T_P + F_P + T_N + F_N}$$

Precision: It is defined as the ratio of total number of T_P correctly classified by the classifier to the total number of predicted T_P. It can answer the question, "What proportion of positive identifications was correct?"

$$Precision = \frac{T_P}{T_P + F_P}$$

Recall It is defined as the ratio of total number of T_P correctly classified by the classifier to the total number of T_P in testing dataset. Recall attempts to answer the following question, "What proportion of actual positives was identified correctly?”

$$Recall = \frac{T_P}{T_P + F_N}$$

f-1 score: It is useful for imbalanced dataset, that means when the class distribution is uneven. It is the harmonic mean of precision and recall and can be defined as follows:

$$f\text{-}1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Finally, to quantify the model’s performance we also use another metric, Area Under the Curve (AUROC). The value of AUROC falls between 0 and 1 and with the higher number means better performance of a classification model.

5.2.5 Feature Extraction

After training our machine learning model, we needed to extract features from the current version for which the developer is going to commit. All the previous studies, including the one we followed to build our model, have collected the features after the commit has been done. But for our work, we needed those same features before the commit in order to test if the new commit is going to be bug-inducing or not, which is way more challenging than the former studies.

Number of Lines Added (LA)

This feature was basically how many lines would be added in the next commit including all the files. We have used the git *diff* command in order to extract those added lines. Git *diff* command basically shows differences one developer made in the working tree relative to the index. Here, index means the staging area for the next commit. We used “–compact-summary” as the parameter to get a summary of the modified files and extracted the number of lines added from the output.

Number of Lines Deleted (LD)

Number of deleted lines are extracted in the same way as the number of added lines using git *diff* command. The output was the total number of lines deleted in the current working tree.

Number of Modified Files (NF)

Number of modified files is how many files have been modified in the current working tree since the last commit. We used *git status* comman, which shows the files that have differences between the working tree and the index file as well as the files which have differences but are not tracked by Git. We filtered out the untracked files and extracted the modified files and counted the total number of them as the feature.

Number of Modified directories

After getting the modified files in the current working tree, we extracted the number of directories that were modified and used it as a feature for testing.

Number of Developers (ND)

The feature number of developers means the number of developers who previously changed the touched or modified files. For example, if a change has three modified files X, Y and Z, and file X is modified by developer A and files Y and Z are modified by developer B, then the number of developers will be 2 who are A and B. In order to extract this feature, we used git *log* command to get the commit history. Then we used "–pretty=short" parameter so we can get each commit grouped by author and title. Then we extracted the number of authors of the modified files till the previous commit. Then, we checked if the current author is present in the author list of previous commits. If not, then we added the developer to the list and counted the total number of developers as the feature.

Number of Unique Changes (NUC)

NUC is the number of unique last changes of the modified files. For example, if file A was previously modified in change alpha and files B and C were modified in change beta, then NUC is 2. We used the *git log* command as before to get the commit history and from there, we extracted the number of commits which are involved with the modified files.

5.2.6 An IDE-based Plug-in

We want to provide the developers information about having bugs in their code while they are committing their new or updated portion of code in their project repositories. In order to do so, we decided to use Visual Studio Code (VSCode) as our IDE. The reason behind this is this IDE combines better in the case of native, web and language-specific technologies architecturally. It is free, light-weight and an open-source editor whose source code is made available on GitHub. In addition, it is a cross-platform editor which was developed for Windows, Linux and Mac OS and it supports numerous programming languages like PHP, Python, HTML5, JavaScript and many more.

To build the plug-in for our IDE, first we train a logistic regression model with our dataset containing six features and saved the model. We use random forest, decision tree and logistic regression as our machine learning model. The performance of Decision three was better than the other algorithms. After that, we work on three operations - commit, push and pull so that the developers can perform the operations while using our plug-in. When the developer is selecting the option of commit in our plug-in, we extract the six features from the current version of the project. We extract them using Typescript. Then we pass the value of those features as parameters to a file and test them that the version is bug-inducing or not using the saved

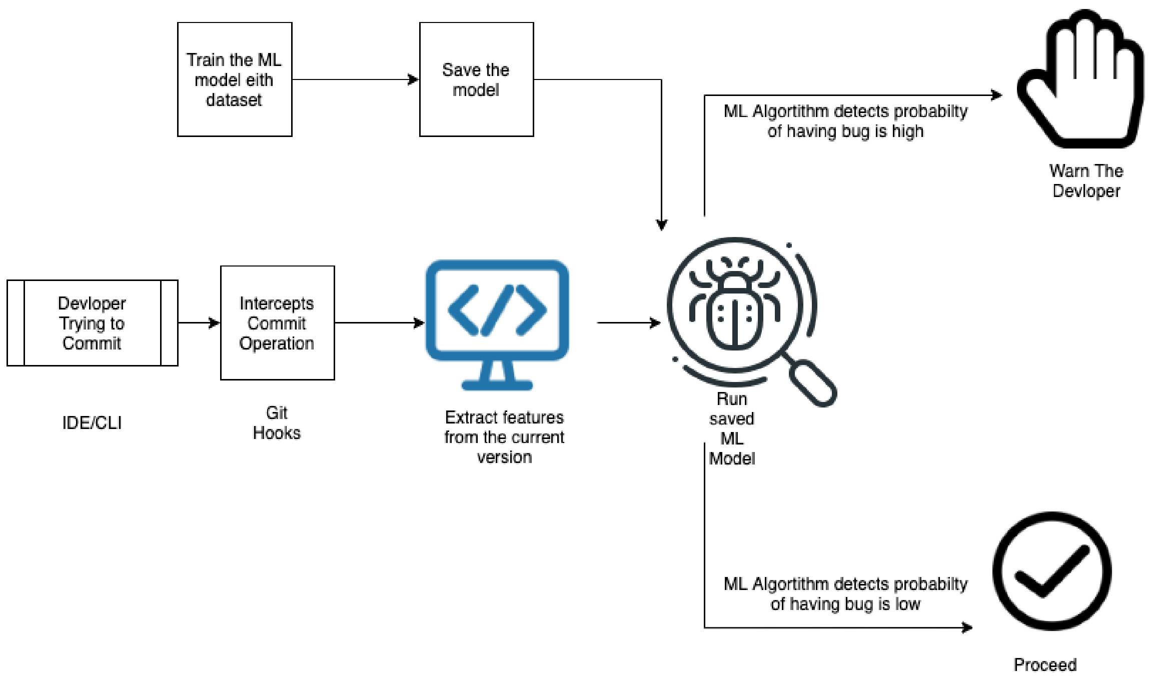

Train the ML model eith dataset
Save the model
ML Algortithm detects probabilty of having bug is high
Warn The Devloper
Devloper Trying to Commit
IDE/CLI
Intercepts Commit Operation
Git Hooks
Extract features from the current version
Run saved ML Model
ML Algortithm detects probabilty of having bug is low
Proceed

Figure 5.1: Methodology of our study

machine learning model. Then we pass the result to the VSCode plug-in. If the result is not bug-inducing, we will let the developers proceed. Otherwise, the system is going to warn the developer that their commit might induce a bug.

To build the Machine Learning model, we have used the TensorFlow [1] API named TensorFlow.js which is a library for Machine Learning in JavaScript. It can be used to convert Machine Learning models implemented in Python to TensorFlow models to run under Node.js environment. This is important in the context of git hooks because git hooks are just executed as a shell script.

5.2.7 Command-line based tool

For the command-line tool, we followed the same design but we used git-hooks in order to extract the features from the current version and pass them to the saved model. Git-hooks are scripts that run automatically when an event occurs in a git repository. Using git-hooks, developers can customize git's internal behaviour and trigger customizable behaviours. These behaviours can be triggered at any key point in the software development life cycle. Client-side hooks and server-side hooks are basically two kinds of git hooks. We are using a client-side hook named "pre-commit" hook. The pre-commit hook runs even before the commit message is typed and any kind of scripting language can be used in the hooks in order to execute them. And if the hook returns a non-zero code, the commit is aborted. So, if the machine learning model returns that the commit is buggy then the git hook aborts the commit. Otherwise, it lets the developer proceed.

5.3 Experimental Result and Analysis

5.3.1 Random Forests

From table 5.4, we see that Random Forest has given the lowest accuracy in our experiment. The score of precision and recall is also relatively low than the others. Imbalanced data set is the main reason for showing this kind of behaviour. We also performed oversampling and undersampling to balance our imbalanced data set, but we have not found any promising result for the random forests.

5.3.2 Decision Tree

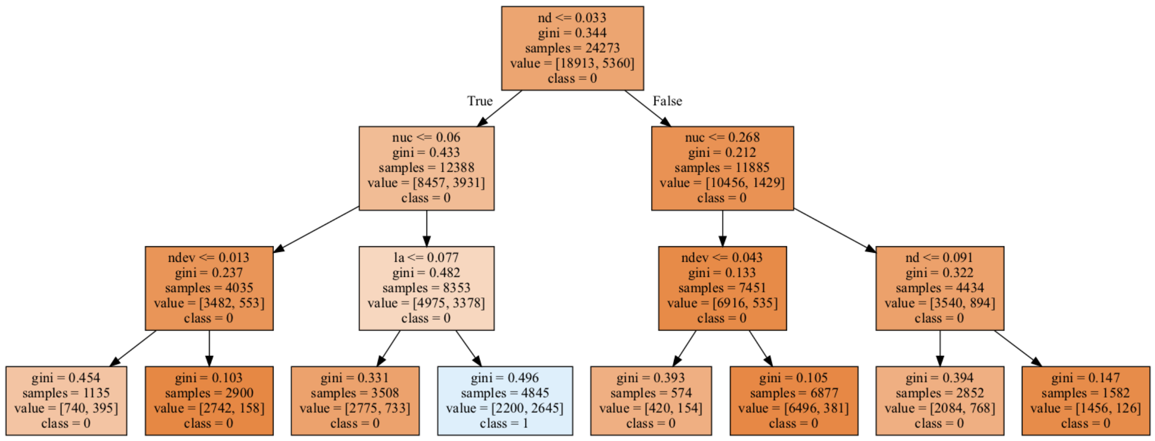


Figure 5.2: Visualization of Decision Tree Algorithm

Decision Tree has performed relatively better than the Random Forests model. The accuracy is a bit higher. But it showed a great amount of improvement in case of precision and recall which prevents the chance of false-positives and false-negatives. Figure 5.2 shows us a visualized version of how the decision tree is making a decision based on our classes. We can see that, the decision tree chose the number of developers as the root condition of the algorithm. So, the first feature that can distinguish the bug-inducing and non bug-inducing commit is the number of developers in a commit. The second feature that is use as the condition is the number of unique changes. We used 3 as the parameter maximum_depth of the tree, because after that, any value of the parameter gives us the similar results.

5.3.3 Logistic Regression

The experimental results of LR model are shown in Table 5.5.

From this table, we can observe that LR model without no sampling of dataset has the highest accuracy score of almost 80% although the recall score is comparatively low compared to dataset with undersampling and oversampling. This happens because of an imbalanced dataset and LR model has an inclination to the majority class (non-bug inducing changes). For undersampling and oversampling the accuracy, precision, recall and f-1 scores are almost similar.

Table 5.5: Accuracy of Logistic Regression model

Evaluation Metrics	No Sampling	Undersampling	Oversampling
Accuracy	0.7986	0.6460	0.6503
Precision	0.7039	0.6305	0.6378
Recall	0.5726	0.6943	0.6966
f-1 score	0.5787	0.6070	0.6164

Receiver Operating Characteristics (ROC) curve determines the capability of classification algorithms to separate data among classes and Area Under ROC (AUROC) narrates the measurement of that separability. Higher AUROC means high number of positive classes are labelled as positive and negative classes are negative and hence justifies the performance of the classification algorithms. In Figure 5.3, we show ROC curve of LR model with calculated AUROC value. From the figure, it is obvious that, LR model for dataset with no sampling has the highest AUROC value because of biasness to the majority class. From the figure we can conclude that our LR model is capable of detecting bug-inducing changes during commit operation with an expected level of accuracy.

5.3.4 Configuration tool for customized models

After providing support for the IDE, we decided to work on how a developer can use our plug-in to customize it according to their preference. In order to do that, Firstly, we built a server, where the developers can select the features they want to use. Currently, there are six available features which are the ones we used before. After choosing the features, they can also choose their machine learning algorithm and train a model. Each user's model will be saved under a unique identification number. The database of the server contains the unique identification number of the user and the features which were selected by the user. After training, the model was saved and it can be used later in the plug-in while performing a commit to detect bug-inducing commits. Figure 5.4 shows us the interface of the system we have designed.

We test the configuration with two more datasets. One is the dataset of Kamei et al. with all 14 features, another one is the Jenkins data set. Borg et al. [5] did a study on an open-source software system (Jenkins) where they labelled the bug-inducing and clean commits using SZZ algorithm. They used the Random Forest Algorithm to detect those bug-inducing commits. The features that are used by this are mostly different from the dataset of Kamei et al. However, these features can also be used to train our model by the authors. This usage of different datasets with different machine learning algorithms shows us the customizability of

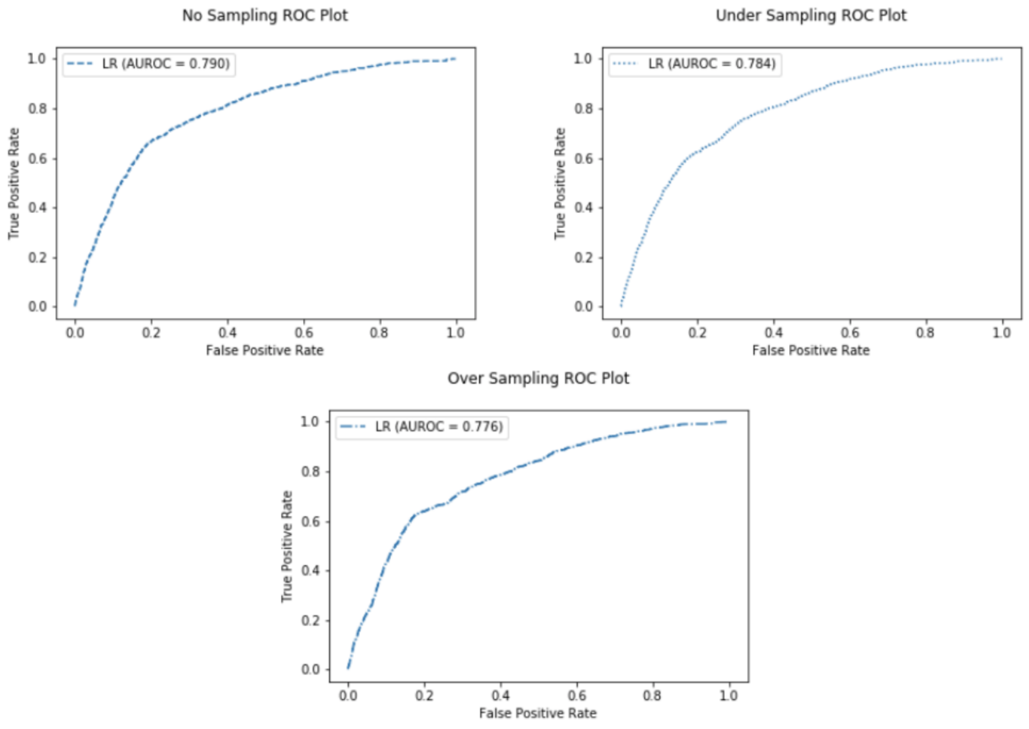


Figure 5.3: Area under ROC curve for dataset without sampling, undersampling and oversampling

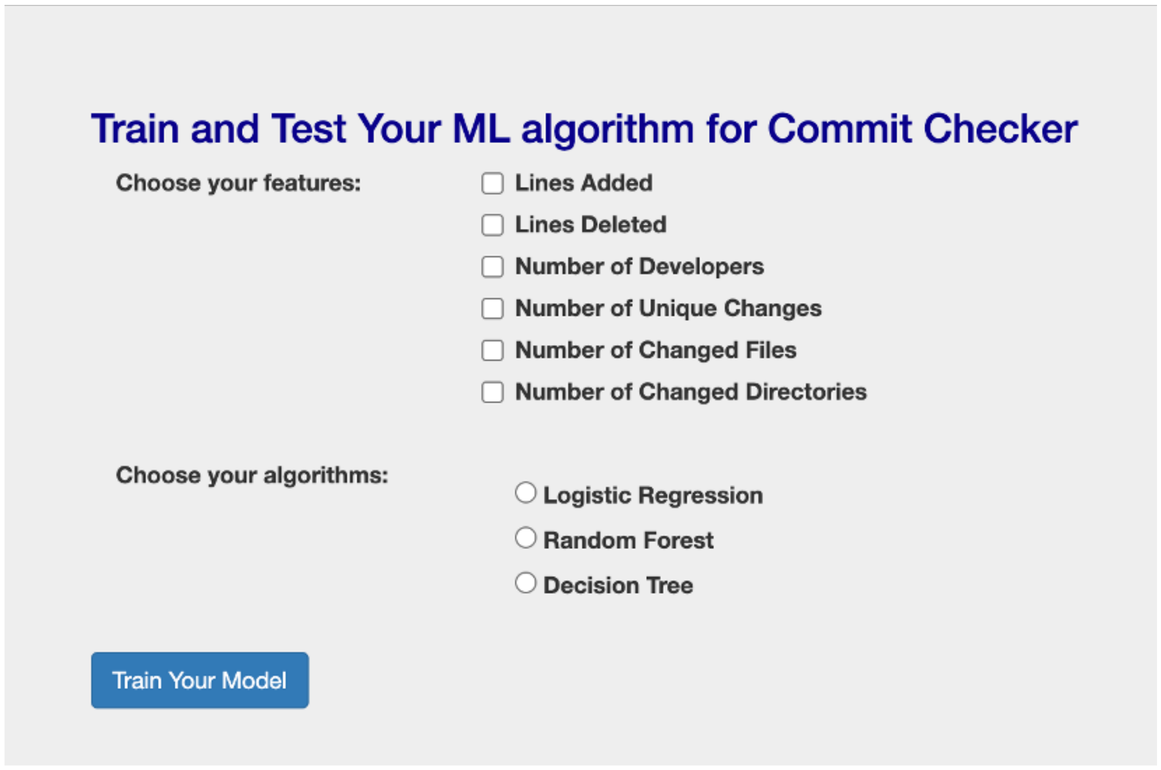
our tool.

Then we thought about giving more customization and not to limit the selection of features into our given six features. Therefore, we give the users the option to upload their own data set. If they have their personal collection of data which are more related to their works, they can use those data to train their model. When the user will upload the data set, our system will show which features are available in the data set. Then the users can select from those available features, select their algorithm and train their models. Finally, they can compare the accuracy, f1-score, precision and recall of their different trained model and choose the one they want to use in there customized plug-in.

5.4 Discussion

In this chapter, we attempt to answer our RQ2 (presented in Chapter 1.3) which aim to find the ways to provide support to the developers to detect bug-inducing commits. To address different aspects of our survey, we divide our original RQ2 into three fine grained sub-research questions (presented in section 5.1). Analyzing the results of our survey (presented in Chapter 4), we built a plug-in and a command line tool along with support for configuring the tools. Then we performed a user-study to understand the effectiveness of the tools. They are proved to be useful to the developers.

RQ1: Does the developer community need a JIT bug prediction technique which detects


Train and Test Your ML algorithm for Commit Checker

Choose your features:

- ☐ Lines Added
- ☐ Lines Deleted
- ☐ Number of Developers
- ☐ Number of Unique Changes
- ☐ Number of Changed Files
- ☐ Number of Changed Directories

Choose your algorithms:

- ○ Logistic Regression
- ○ Random Forest
- ○ Decision Tree

Figure 5.4: Interface of configuration tools for customized models

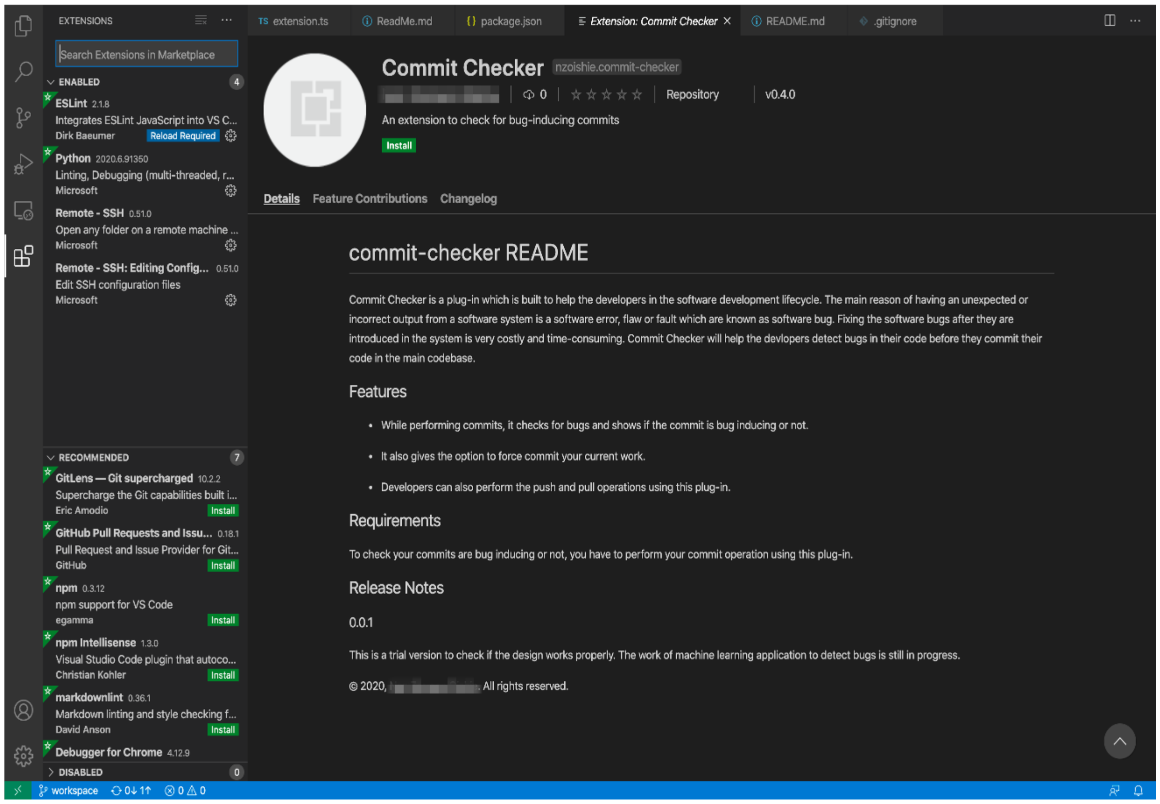
bugs before it is introduced to the central repository?

In section II, we discussed details on how developers from different platforms supported the idea of having a JIT tool to detect bug inducing commits. We have seen a lot of previous work which was done for detecting bug inducing commits, but all of them are done after the commit is performed and eventually causes different bugs in the software life-cycle. As our main focus of the study is providing developer support and for the machine-learning part, we have followed the previous work. Getting all those features form the previous work to feed the machine learning model before the commit operation was performed was a challenge. We overcame this challenge using different git commands. In the feature extraction section, we have discussed briefly about it.

RQ2: What are the ways to visualize the probability of having bugs to the developers?

An IDE-Based plug-in: While doing our research, how we are going to provide support to the developers so that it will make the detection of bug easier was a big challenge. Modern developers are often known as polyglots as they explore in different operating systems, programming languages, frameworks and tools in order to build the applications for future. Additionally, cross-platform applications are widely desired in recent days as modern applications tend to run on different devices. As a result, command line and CLI (Command Line Interface) tools have been often appreciated among the developers.

Another thing is, every developer uses an IDE to code and almost every IDE provides a plug-in using which the developers can perform several git operations like pull, push, commit and so on. For these reasons,



EXTENSIONS
Search Extensions in Marketplace
ENABLED
ESLint 2.1.8
Integrates ESLint JavaScript into VS C...
Dirk Baeumer
Reload Required
Python 2020.6.91350
Linting, Debugging (multi-threaded, r...
Microsoft
Remote - SSH 0.51.0
Open any folder on a remote machine ...
Microsoft
Remote - SSH: Editing Config... 0.51.0
Edit SSH configuration files
Microsoft
RECOMMENDED
GitLens — Git supercharged 10.2.2
Supercharge the Git capabilities built i...
Eric Amodio
Install
GitHub Pull Requests and Issu... 0.18.1
Pull Request and Issue Provider for Git...
GitHub
Install
npm 0.3.12
npm support for VS Code
egamma
Install
npm Intellisense 1.3.0
Visual Studio Code plugin that autoco...
Christian Kohler
Install
markdownlint 0.36.1
Markdown linting and style checking f...
David Anson
Install
Debugger for Chrome 4.12.9
DISABLED
workspace
extension.ts
ReadMe.md
package.json
Extension: Commit Checker
README.md
.gitignore
Repository
v0.4.0
Install
Details
Feature Contributions
Changelog

Commit Checker nzoishie.commit-checker

[redacted] | 0 | Repository | v0.4.0

An extension to check for bug-inducing commits

commit-checker README

Commit Checker is a plug-in which is built to help the developers in the software development lifecycle. The main reason of having an unexpected or incorrect output from a software system is a software error, flaw or fault which are known as software bug. Fixing the software bugs after they are introduced in the system is very costly and time-consuming. Commit Checker will help the devlopers detect bugs in their code before they commit their code in the main codebase.

Features

- While performing commits, it checks for bugs and shows if the commit is bug inducing or not.
- It also gives the option to force commit your current work.
- Developers can also perform the push and pull operations using this plug-in.

Requirements

To check your commits are bug inducing or not, you have to perform your commit operation using this plug-in.

Release Notes

0.0.1

This is a trial version to check if the design works properly. The work of machine learning application to detect bugs is still in progress.

© 2020, [redacted]. All rights reserved.

Figure 5.5: Published extension in VS-code marketplace

we decided to put support for both IDE and CLI. For IDE, we built a plug-in and for command line, we built an executable file which can warn a developer while performing commit operations.

As discussed in the research methodology, we made a plug-in for the Visual Studio Code IDE. Our plug-in has basically three action buttons - push, pull and commit. We have published our plug-in in VSCode market place which can be used by any user. The plug-in is named as “Commit-Checker”. One just have to search it and click the install button in order to use the plug-in.

After installation, the user can see an icon of the plug-in in the activity bar of the IDE. Figure 5.6 shows that after clicking on the button the user will be able to see the commands and they can perform those actions by simply clicking on it. The users can also write those commands on the VSCode command palette, which is a well-known use for the plug-ins of VSCode. Figure 5.7 shows how they can write the command.

Whenever the developer is going to commit using our plug-in, it will automatically extract all the features from the current work tree and send them to test using the trained Logistic Regression model. To extract the features we have used Typescript and to test them using the trained model, we have used a python script. In the python script, we have taken all the six features as parameters and send them to the trained model in order to predict that the commit is going to induce bug or not. The python scripts test the features and return the result that was predicted by the model. If the result returns that the commit operation is buggy, then the IDE shows a warning to the developers and also give them an option to force-commit their work

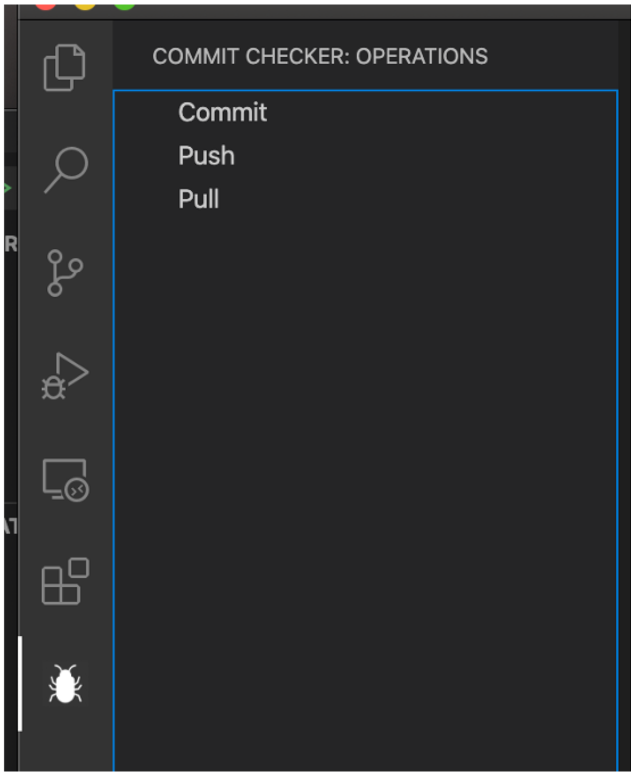


Figure 5.6: Icon and commands in VSCode activity bar

if they are confident enough and doesn't want to check for bugs in their work. Figure 5.8 shows how our plug-in warns the developer.

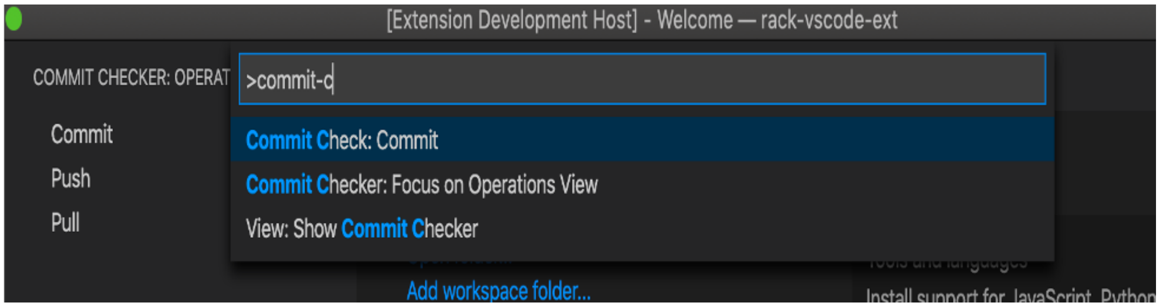
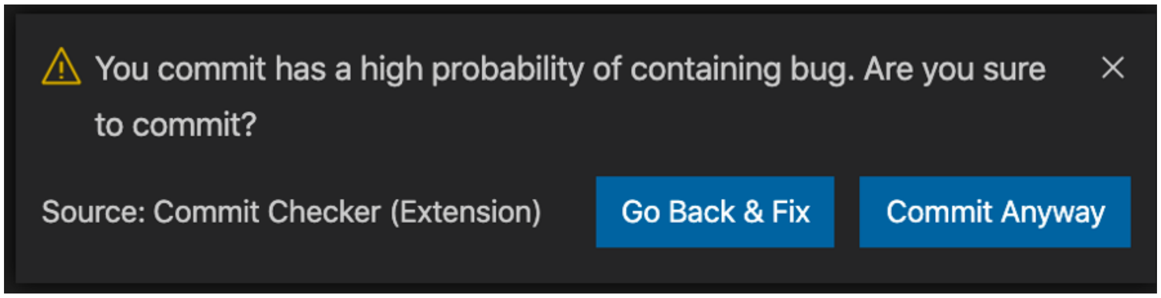

[Extension Development Host] - Welcome — rack-vscode-ext
COMMIT CHECKER: OPERAT
Commit
Push
Pull
>commit-c
Commit Check: Commit
Commit Checker: Focus on Operations View
View: Show Commit Checker
Add workspace folder...

Figure 5.7: Command line tool

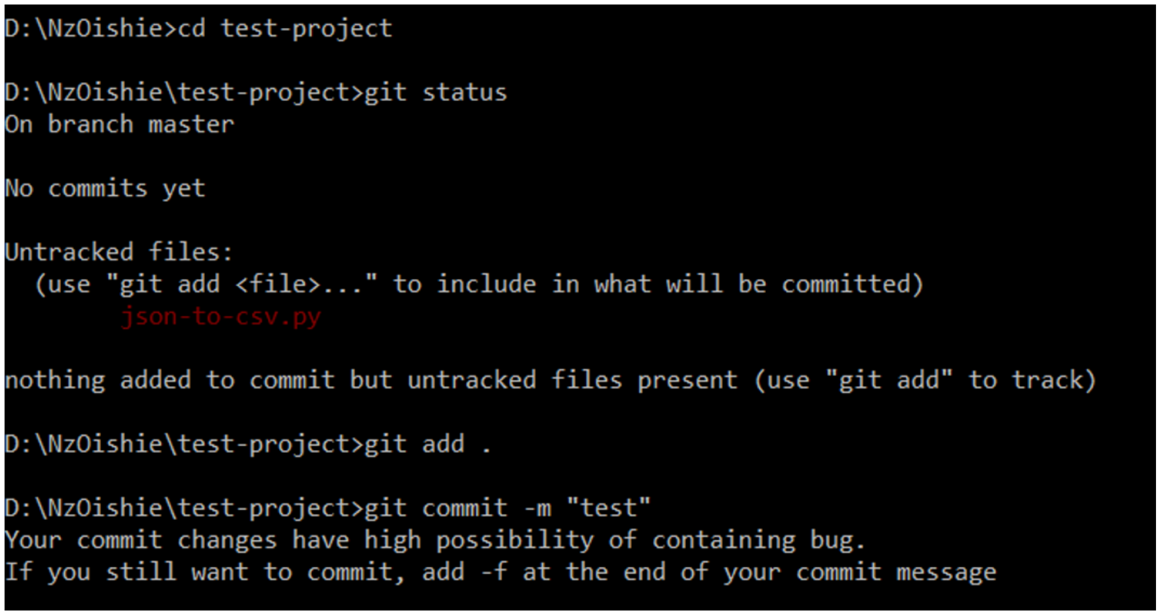
CLI based tool: As command-line interface is very popular among the developers, we build an executable file so that developers can download the executable file and run it. After that, they have to give the root-path of the project they want to use the tool for as a parameter of the executable file. The tool first checks if it is a git repository or not. If it is not, then the tool throws an error that it must be a git repository otherwise, the commit operation cannot be performed. In addition, when the developers are going to type the git commit command to commit operation, a warning will be shown. Figure 5.9 shows the command line interface where the warning is given to the developers. This is done using the pre-commit hook, which aborts the commit



You commit has a high probability of containing bug. Are you sure to commit?

Source: Commit Checker (Extension)

Figure 5.8: Extension giving warning while performing commit



```
D:\NzOishie>cd test-project

D:\NzOishie\test-project>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        json-to-csv.py

nothing added to commit but untracked files present (use "git add" to track)

D:\NzOishie\test-project>git add .

D:\NzOishie\test-project>git commit -m "test"
Your commit changes have high possibility of containing bug.
If you still want to commit, add -f at the end of your commit message
```

Figure 5.9: Warning the developer in command-line interface

by showing the warning. In the figure, we can also see that, an option has been provided to do force commit their work. If they include "-f" in the end of their commit message, then the pre-commit hook will ignore the chance of having bugs and let the developers commit their work.

The main challenge we faced is to extract features from the current version of project. The data set we used contains features, which are collected from previously performed commits. Extracting the number of developers working on currently updated file was challenging using git commands. In the case of detecting number of unique changes, we faced similar kinds of challenges.

RQ3: How our usability pattern can be used by other researchers working on the algorithms for bug-inducing commits?

Researchers who want to change features and try different algorithms can use our server that was built to train models. After completing the training, they can use that trained model in our plug-in. Along with VSCode, we have also tested the server with JetBrains IDEs. So, this same usability pattern can be used in different IDEs and different situations.

5.5 Threats to Validity

In this section, we briefly discuss the factors that might be related to the internal and external threats of our study and how we mitigate them.

5.5.1 Internal Validity

As for threats to internal validity relates to errors in our implementation of ML algorithm and plug-in development. For the ML part, we use Tensorflow library and also double-check our implementation. Another threat might be the generalization of our collected dataset to train the predictive model. In this study, we collect the dataset that was used in a previous study by Catolino et al. [11, 12]. The dataset is already preprocessed and unnecessary features were excluded using the information gain technique. Again, the selection of ML algorithm might be another threat to our study. As bug-inducing commit detection was a binary classification problem. Hence, we select Random Forests, Decision Trees and Logistic Regression as our ML algorithm. They were also used in the previous state-of-the-art proposed approaches [28, 27, 12]. Thus, we mitigate the internal threats that might have affected our study. In the case of the configuration option of the plug-in, to select features, we just give them options between the six existing features. In future, we will provide them all the features extracted in the study of Kamei et el. [28]. However, it is a great challenge to solve as all the features are needed to be extracted before the commit operation.

5.5.2 External Validity

The first threat related to external validity is the evaluation method when testing our predictive model. To evaluate the performance of our predictive model, we apply different metrics: accuracy, precision, recall and f1 score. All these metrics were applied in previous studies [27, 12, 32, 50, 44, 49, 9]. Another external validity might be the generalizability of our experimental results. We train and test our predictive model on the dataset collected from the study of Catolino et al. [12]. The dataset consists of 14-open source Android applications of different sizes and domains. In our future study, we plan to add desktop-based software systems and open-source commercial iOS applications to reduce this threat. Finally, the accuracy of the model is 79% which is pretty good in terms of general machine learning applications. However, this also implies 1 out of 5 warnings by our toolchain will be false. This will cause the developer to do extra-check and have a negative effect on developers' trust. Nonetheless, in terms of software development, doing some extra-check is always better than working with some potential buggy code. Furthermore, as explained in Section 5.2.4, explainability and lower complexity were our primary criteria behind choosing these models. However, in future, we will also incorporate more complex models like neural networks so that developers with such experience can make use of them.

5.6 Conclusion

Organizations spend a large amount of money and effort on fixing usability problems during late-stage development. Some of these problems could have been detected and fixed much earlier. This avoidable rework leads to high costs and systems with less than optimal usability, because during the development, different trade-offs have to be made, for example, between cost and quality. Detection of bug-inducing changes in software is an ongoing research topic in software engineering. Real-time quality assurance does matter if we want to reduce the cost of software maintenance.

In this paper, we provided an IDE-based Plug-in support to detect bug-inducing changes before the commit operation. Then we also provided a server where researchers can select their own features and algorithms. It performs some analytics and predicts whether a commit operation is going to induce bugs in software systems or not. To build the predictive model, we apply the dataset that was used in previous study by Catolino et al. To handle the imbalanced problem of dataset, undersampling and oversampling techniques are also applied on the collected dataset. Information gain technique is used to filter out irrelevant features from the initial set of 14 features and finally, 6 features are used to train the predictive model. We evaluated our predictive model using different metrices such as accuracy, precision, recall and f-1 score. The accuracy is almost 79%.

6 Empirical Evaluation of Commit-Checker Tool

In order to evaluate the effectiveness of our tool, we performed a user study with 12 developers. We have performed three kinds of experiments to understand the time frame and workload the developers need to install and use it if the developers can detect the bug-inducing and non-bug-inducing commit properly and finally, how our tool is acceptable to the users compared to the state-of-the-art.

In this study, we split the RQ3 stated in Section 1.3 into the following fine-grained research questions.

- RQ1: How much time and effort is needed to use interactivity among the developers to detect bug-inducing commits?
- RQ2: How effectively can the users interact with our tool to detect bug-inducing and non-bug-inducing commits?
- RQ3: How much do the users prefer our tools compared to the state-of-the-art?

6.1 Methodology

In our study, we have performed three kinds of experiments described as follows.

- **Experiment 1:** We performed a user study with five developers. Three of them were working with Java frameworks, one with JavaScript and another one with Python. Two of them had experiences of five to seven years, and others were working as developers for two to three years. One of the five developers identifies herself as female. In this user study, we have divided the tasks into three parts: (1) Installing the VSCode Plug-in and using it while performing a commit, (2) Downloading and running the command line tool and performing a commit, and (3) Going to the configuration option and training the model according to their choice of a dataset, features, and algorithms.
- **Experiment 2:** We extracted 10 commits consisting of both bug-inducing and non bug-inducing commits from a popular Github project. We gave the participants proper instructions to clone the specific repository in their system. Then, we asked the participants to commit 10 chunks of codes in different files of that repository. We had already extracted those code chunks from the projects and classified which were buggy and which were not by manually analyzing the commit messages. We first detected the bug fixing commits, and then from there, we went back to the version where the bug was first introduced. Nevertheless, that classification is unknown to the participants. They performed

those commit operations using the plug-in and reported which commits are detected buggy and which commits are detected non-buggy using our tools. Then we analyzed how accurately our tool worked.

- **Experiment 3:** We provided the link of the repository they had asked to clone in the previous experiment and asked them to upload it in CommitGuru, the state-of-the-art tool. Then, we prepared a questionnaire to compare their experience using both tools. We find out in which cases they prefer our provided tool and when they prefer CommitGuru.

6.2 Procedures, Results and Analysis

6.2.1 Experiment 1

Survey design

We used Google forms for the questionnaire. It contained both open-ended and close-ended questions. Some open questions for participants' opinions were added to know why they chose the options and made the decisions, if they have anything better to suggest and if they would like to share an opinion or thought. Most of the other questions were closed or multiple-choice questions. The question to measure the NASA-TLX workloads uses the Likert scale. The scale ranges from one to ten. The time duration was 15-20 minutes to complete the questionnaire. The data were collected in Google Sheet. This lets us view the data in rows and columns with timestamps in spreadsheet format.

6.2.2 Recruitment Of Survey Participants

We surveyed five software developers working in the industry to understand what kinds of support will help software developers. We used emails as it is the easiest way to get to the subjects and the subjects had the flexibility to answer in their own time.

The five developers use three different operating systems and coding environments, which helped us understand the time frame of installing and using our toolchain in different environments.

Although five is a small number to reach a decision, the goal of this study was to ensure whether our toolchain works appropriately in different environments or not and to understand the timing of installing them. Another important goal was to know how much time is needed if a user trains a model. We successfully get those answers and understand how comfortable the users are while completing these tasks performing this user study.

Participation Information

Among our participants, three of the developers had five years of software development experience. The other two had one to two years of experience. Four of the developers identified themselves as male and one as female.

RQ1: How much time and effort is needed to use interactivity among the developers to detect bug-inducing commits?

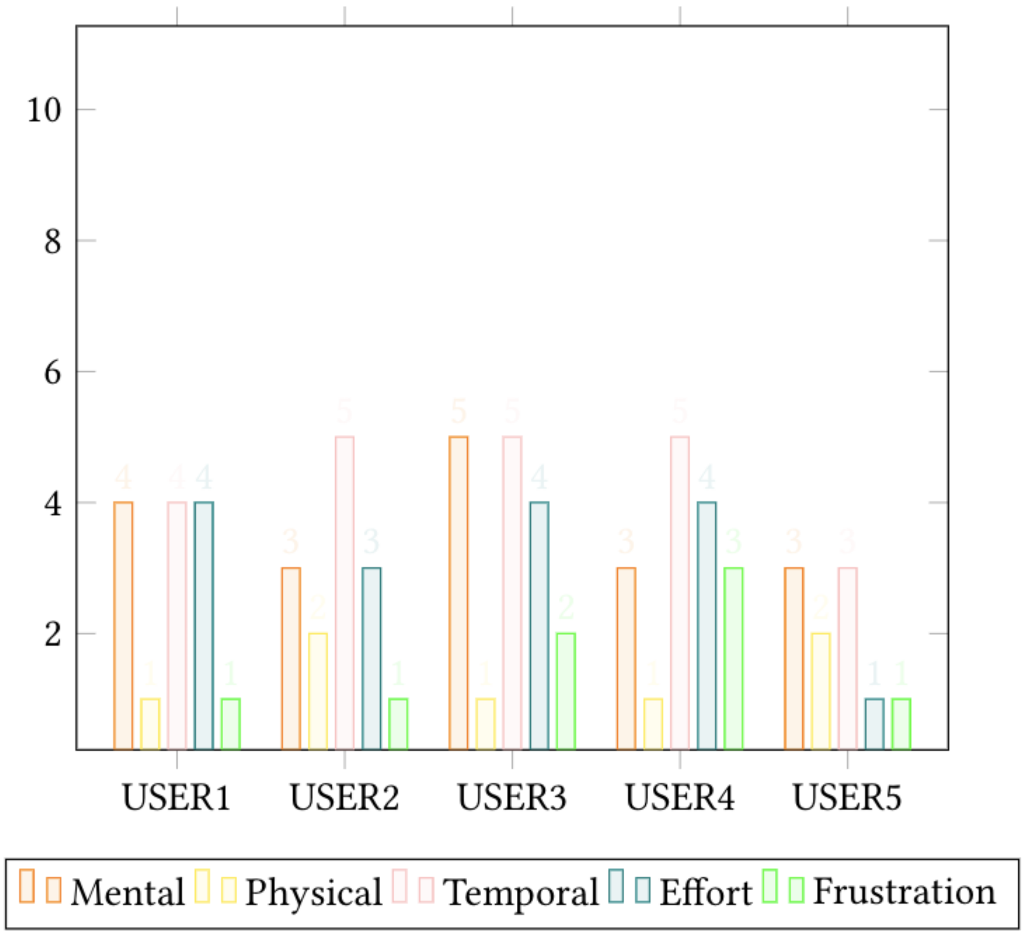


Figure 6.1: Evaluation of the Tools using NASA-TLX for IDE based Plug-in

Our first intention from the user study was to know how much time it takes to install the plug-in and how much time it takes to detect whether the commit is bug-inducing or not in real time. To install the plug-in, everyone reported they needed about half to one minute. For the command-line tool, the response was similar except for one developer who responded with 1 to 2 minutes. To detect bug-inducing commits, using both the plug-in and command-line tool, the developers needed 1 to 2 minutes except the same person needing more time. For this reason, we took an interview with him so that we could understand the reasons for our tool taking more time to operate. We found out the system he was using was weaker than the system used by the others.

Our second intention was to determine the subjective mental workload performing the task. We used NASA Task Load Index (TLX) assessment system to evaluate the workloads of our participants. NASA TLX is a well-known and widely used system to measure a system's effectiveness and different aspects of

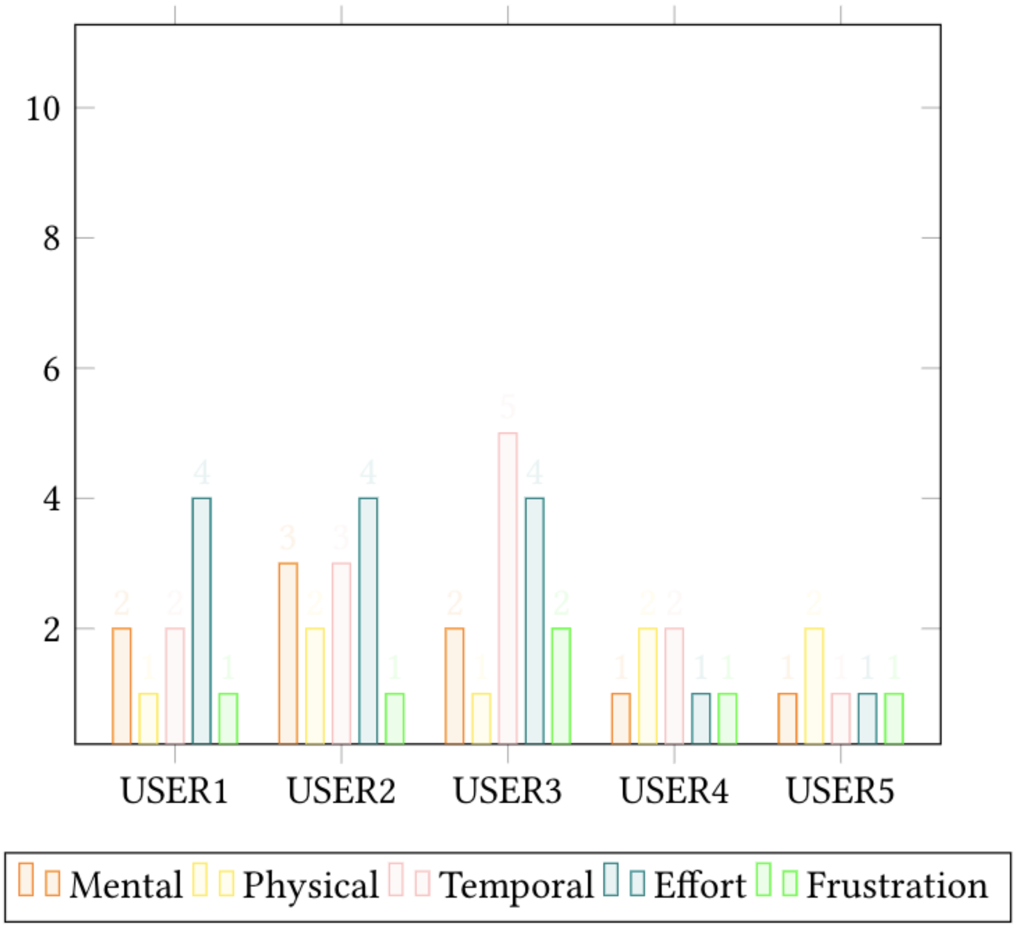


Figure 6.2: Evaluation of the Tools using NASA-TLX For Command Based Tool

performance. These assessments are based on six subscales: physical demand, mental demand, temporal demand, effort performance, and frustration level. We used the Likert scale to collect the data from the users on a scale of one to ten. As we saw, all of the participants could successfully perform the tasks. In addition, we recorded the amount of time they needed for these tasks. We will be discussing the other subscales in our study. The results of this user study are shown in Figures 6.1, 6.2 and 6.3.

The Figures show the mental demand, physical demand, temporal demand, effort, performance, and frustration level of the participants. The results show that on a scale from one to ten, most of the results are below fifty, which implies that none of the tasks are much demanding.

6.2.3 Experiment 2 and Experiment 3

Survey Design

- We used Google forms for the questionnaire. It contained both open-ended and closed-ended questions.

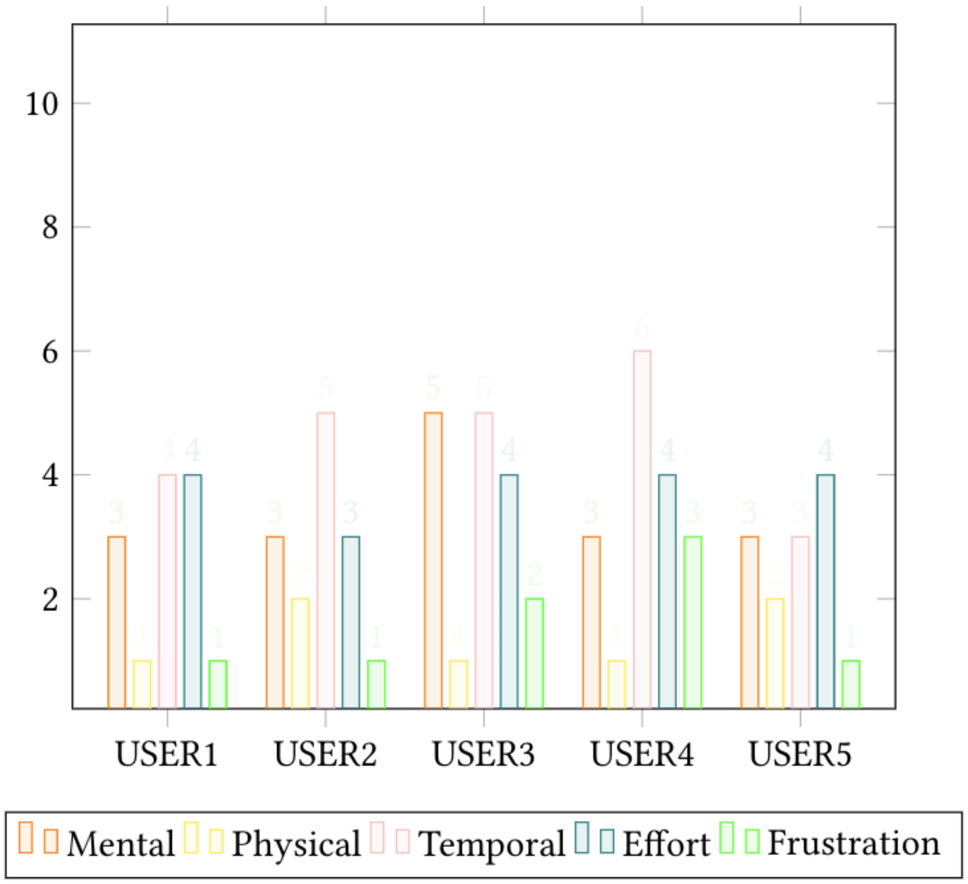


Figure 6.3: Evaluation of the Tools using NASA-TLX for server

- Some open questions for participants' opinions were added to know why they chose the options and made the decisions, if they have anything better to suggest and if they would like to share an opinion or thought. Most of the other questions were closed or multiple-choice questions.
- Fifteen developers were contacted through email with a google form for the second and third experiments.
- First, we asked them if they had Visual Studio and Git installed on their computers. They answered the questions by running a couple of commands in a terminal.
- If they were not installed, then the participants were guided through the process of installing them on their computers.
- Then they are asked to clone a repository from GitHub. The instructions on how to clone the repository were provided to them.
- Then they were asked to checkout to a specific version of commits to get back to the exact situation of the project when the commit operation was performed.
- We selected five chunks of bug-inducing and non-bug-inducing commits from the project. We filtered the issues that were already merged into the project and had "bug-fixing" in their label. Then we manually analyzed the commits associated with them.
- We selected those issues which only had one commit associated with them. Then we filtered out those commits in which the number of changes was less than five lines and had only one file with changes.
- Then we ask the participants to remove all contents of the changed file and replace it with the content of commit of that specific version.
- The contents of the specific version of that commit were stored in a google doc and the link of that document was provided in the questionnaire. The participant copied the lines of codes and replaced the file with them.
- Then they were asked if they could find the bug-inducing or non-bug-inducing commits.
- The participants were then asked to upload that repository into CommitGuru.
- Finally, the participants were asked about their experience with our tools and with CommitGuru so that a comparison could be done.
- The time duration was 35-40 minutes to complete the questionnaire.
- The data were collected in Google Sheet. This lets us view the data in rows and columns with timestamps in spreadsheet format.

6.2.4 Recruitment of Survey Participants

We contacted fifteen developers to participate in our user study through emails. They have been contributing actively to the industry, and we wanted to understand their opinion. The goal of our study was to know if they could identify the bug-inducing and non-bug-inducing commits successfully using our tools. We were also eager to get their feedback on our research after comparing them with the state-of-the-art.

Participation Information

Thirteen of the contacted developers across six countries had agreed to participate in our study. The developers were from Bangladesh, USA, Canada, Germany, and Thailand. We provided some help through virtual meetings. Five of the developers had experience in the industry of over three years, four of them were novice in the industry, and four others had experience of 2-3 years. Two of the developers were female, and the rest of the eleven were males.

RQ2: How effectively can the users interact with our tool to detect bug-inducing and non-bug-inducing commits?

Eleven of the thirteen developers detected all the five bug-inducing and non bug-inducing commits correctly. Two of the developers identified one particular non bug-inducing commit as bug-inducing. As only we know which commits are bug-inducing and which are not, we observed that 80% of the developers were able to detect these commits correctly and successfully.

RQ3: How much do the users prefer our tools compared to the state-of-the-art?

To compare their experience with both our tool and CommitGuru, we ask them four questions. We also request them to add any comment they think might help to improve our work.

The first question was regarding which of the two tools gave them a good overview of the commits of the project. Twelve of the thirteen developers chose CommitGuru, the state-of-the-art, as their answer. As we have not provided many options to analyze the repository, we can understand why most of the participants chose CommitGuru. The second question was regarding which of the two tools helped them to analyze their current commit realtime. Nine of the thirteen developers chose our tool, Commit-Checker, as their answer.

The third question we asked was regarding which of the two tools helped prevent bug-inducing commits before introducing it to the central repository. All of the developers chose our tool as their answer as it prevents bug-inducing commits before performing the commit. The last question was about which tool they preferred the most when it came to exploring configurable options. Twelve of the thirteen developers choose our tool as the answer.

From the user study, we can summarize that, although CommitGuru serves best to analyze all the commits of a whole repository, most developers chose our tool to detect the bug-inducing commit before performing commit operation in real time or to customize their model choosing their dataset, features, and models.

6.3 Threats To Validity

6.3.1 Internal Validity

Threats to internal validity relates to the design of this study. To ensure this user study is appropriately designed and is able to understand the interaction between the users and our tools, two highly experienced professionals in the field of software engineering validated our survey prior to sending it to the developers. We have updated our survey according to their suggestions, revised the questionnaire and then performed the user study.

6.3.2 External Validity

For experiment-1, the number of participants in our survey was only five. Our primary target of this survey was to ensure that our toolchain works well in different operating systems and machine configurations. Therefore, even though five is a small number of participants, it was sufficient for this experiment. In addition, all participants from this study worked with the same GitHub repository. This might question the generalizability of the toolchain. However, the generalizability of this toolchain are ensured in study 2.

For experiment-2, the number of participants was 13. However, they are from five countries in three continents which ensures the diversity of the participants and, in turn, the generalizability of their response. Furthermore, they are actively working in the software industry with diverse environments and languages. Hence, 13 developers are sufficient for this experiment.

6.4 Conclusion

Evaluation is a process that critically examines a subject or a program. It involves collecting and analyzing information about a program's activities, characteristics, and outcomes. The purpose of such evaluation in our context is to make judgments about a program, to improve its effectiveness, and/or to inform programming decisions [46]. In our research, after successfully developing the tool, the feedback of the users has been very important in understanding the effectiveness. To answer the research question RQ3 mentioned in Chapter 1, we performed three experiments to understand every aspect of our tool's performance. We find that the time needed to install and use our tools is very standard. The workload is below 50% for almost all participants. We also find that the buggy and non-buggy code chunks are detected mostly accurately by the participants. And in the case of bug-inducing commit detection in real time, the participants prefer our tools compared to the state-of-the-art tool.

7 Conclusion and Future Work

In this thesis, we conducted three studies to minimize one of the significant issues in the field of software engineering. Software bugs are costly for both developers and end-users. In a world where things in the software industry are getting more reliant, lessening the time and maintenance cost of fixing unexpected bugs is more important than ever before. Most of the developers write their code in different IDEs. An excellent IDE makes developers more efficient and thus speeds up their productivity. Therefore, the goal of our research is to provide support for the developers in the IDE so that they will be able to review their risky code while those chunks of codes are still fresh in their minds. This might reduce the cost of developing high-quality software by preventing the bugs even before they are introduced into the system.

7.1 Understanding The Perspective of Developer Community

Up to this point, researchers have developed several approaches to identify prior bug introducing changes. However, we find very few works on how these techniques of detecting bug-inducing commits can be applicable in real life. Moreover, to promote the adoption of these techniques, one significant concern is what are the thoughts of the developer community. For this reason, we have performed a survey with 20 developers to understand their perspective on real-time support for bug-inducing commits. We contacted them through email and collected their emails from GitHub as all of them are contributors to popular GitHub projects.

7.2 Providing Real-time Support To The Developer Community

Analyzing the survey results, we have built a plug-in for the VScode IDE and a command-line-based tool to detect bug-inducing changes before the commit operation. The developers can install and run it on their computers to prevent bug-inducing commits. These tools are built on machine learning models. Moreover, we add a configuration option where researchers can select their data set, features, and algorithms to build their own customized models. To build the predictive models, we studied the literature and found out Random Forests, Decision Trees, and Logistic Regressions to be the best algorithms to train our model. The information gain technique is used to filter out irrelevant features from the initial set of 14 features and finally, 6 features are used to train the predictive model. We have experimented with all three algorithms, calculated their accuracy,f1-score, precision, and f1-score. We have tried to understand why machine learning is behaving while getting trained on our data set. To balance our imbalanced data set, we also perform oversampling and undersampling on our data set. We have found that decision trees achieves the highest accuracy of 79% with a moderate f1-score, precision, and recall.

7.3 Evaluation Of The Real-time Tools

To evaluate the tools based on our research, we conducted three experiments. First, we measured the time the developers needed to install and run the tools. We used NASA-TLX to measure the mental demand, physical demand, temporal demand, performance, effort, and frustration of the participants. We found that all of the measurements were below 50% for all the participants. In fact, for most of them, the range was between 10% to 30%. We performed another study to see how effectively the participants were able to detect the bug-inducing and non bug-inducing codes. We found our tools to be working effectively in most of the cases. Finally, we request the participants to compare the experiences with the state-of-the-art tool. We found that in the case of real-time detection of bug-inducing commits, most participants preferred using our tools in their development environment.

7.4 Future Work

In this thesis, we investigated tools and techniques to detect bug-inducing commits in real time. However, our research does not indicate the source of the bug. In our future plan, we propose to investigate developers' perspectives on automated program repairing [21] so that we can integrate that into our tools. That way, the developer would be able to find the bugs in less time after detecting their risky commits. In this study, we have used data sets from both computer and mobile applications. A comparison of these two kinds of datasets can be done to understand their differences and the impact of different features and machine learning algorithms on them. These will help us provide specific support for IDEs as developers often use different IDEs for computer and mobile software development.

Moreover, there may exist some bugs in a one-line statement, which can be quite tedious to spot manually. Many developers call them "stupid" because they think fixing these bugs is simple [29]. Unfortunately, studies on these kinds of bugs are very limited and most of the developers cannot spot them by themselves [29]. Detecting these kinds of bugs and suggesting fixes for them can be another research direction for our work. Another possible direction of our research would be to analyze different characteristics of developers (i.e. developers' age, experience, duration between commits, timestamp of performing commits) to understand which of these things triggers the possibility of writing risky code. If we create a plug-in or tool that can detect such relationships, it can warn them and assist them to resolve the issues before committing to reduce the chance of introducing bugs into the central system.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300, 2019.

[3] Andrea Arcuri. On the automation of fixing software bugs. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1003–1006, Leipzig, Germany, 2008. ACM. Doctoral symposium session.

[4] Catal C. Banerjee S. Application of artificial immune systems paradigm for developing software fault prediction models. *Evolutionary computation and optimization algorithms in software engineering*, page 76–93, 2010.

[5] 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 *3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, page 7–12, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Fiorenza Brandy. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year, 2013.

[7] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and Regression Trees.* CRC Press, 1984.

[8] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers, 2013.

[9] Gerardo Canfora, Andrea Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective cross-project defect prediction. In *IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, 03 2013.

[10] Rich Caruana, Nikos Karampatziakis, and Ainur Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *25th International Conference on Machine Learning*, ICML '08, pages 96–103, New York, NY, USA, 2008. ACM.

[11] G. Catolino. Just-in-time bug prediction in mobile applications: The domain matters! In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 201–202, 2017.

[12] G. Catolino, D. Di Nucci, and F. Ferrucci. Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 99–110, 2019.

[13] Bruno Cunha. 11 reasons to learn bash. `https://www.dataquest.io/blog/why-learn-the-command-line/`. [Online; Retrived 23-December-2012].

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

[15] Steven Davies, Marc Roper, and Murray Wood. A preliminary evaluation of text-based and dependency-based techniques for determining the origins of bugs. In *2011 18th Working Conference on Reverse Engineering*, WCRE '11, page 201–210, USA, 2011. IEEE Computer Society.

[16] J. Ekanayake, Jonas Tappolet, H. Gall, and A. Bernstein. Time variance and defect prediction in software projects. *Empirical Software Engineering*, 17:348–389, 2011.

[17] Liliana Favre. Modernizing software amp; system engineering processes. In *2008 19th International Conference on Systems Engineering*, pages 442–447, 2008.

[18] eelke folmer, Martijn Welie, and Jan Bosch. Bridging patterns: An approach to bridge gaps between se and hci. *Information and Software Technology*, 48:69–89, 02 2006.

[19] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21, 05 2014.

[20] R.L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111, 2001.

[21] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra. Automatic program repair. *IEEE Software*, 38(04):22–27, jul 2021.

[22] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. Software bug prediction using machine learning approach. *International Journal of Advanced Computer Science and Applications*, 9, 01 2018.

[23] S. G. HART. Development of nasa-tlx (task load index) : Results of empirical and theoretical research. *Human Mental Workload*, 1988.

[24] Tim Kam Ho. Random decision forest. In *3rd International Conference on Document Analysis and Recognition*, pages 278–282, Montreal, QC, 1995.

[25] Stack Overflow Insights. Stack overflow developer survey 2021 - integrated development environment. `https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environment`, 2021. [Online; Retrieved 01-October-2012].

[26] Quinlan JR. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, 1993.

[27] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[28] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[29] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *17th International Conference on Mining Software Repositories*, MSR '20, page 573–577, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. Jitbot: An explainable just-in-time defect prediction bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 1336–1339, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Foutse Khomh, Bram Adams, Jinghui Cheng, Marios Fokaefs, and Giuliano Antoniol. Software engineering for machine-learning applications: The road ahead. *IEEE Software*, 35(5):81–84, 2018.

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

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

[34] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, page 81–90, USA, 2006. IEEE Computer Society.

[35] Takashi Kokubun. Gitstar ranking. `https://gitstar-ranking.com/`, 2014. [Online; Retrieved 20-December-2012].

[36] Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. Are faults localizable? In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 74–77, 2012.

[37] Scott Matteson. Report: Software failure caused $1.7 trillion in financial losses in 2017, 2018.

[38] Karl Meinke and Amel Bennaceur. Machine learning for software engineering: Models, methods, and applications. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 548–549, 2018.

[39] Babak Memarian and Panagiotis Mitropoulos. Work factors affecting task demands of masonry work. *47th ASC Annual International Conference*, 10 2021.

[40] Tom Mitchell. *Machine Learning.* McGraw Hill, 1997. ISBN: 0-07-042807-7.

[41] Ditte Hvas Mortensen. User research: What it is and why you should do it. `https://www.interaction-design.org/literature/article/user-research-what-it-is-and-why-you-should-do-it`. [Online; Retrived 25-September-2012].

[42] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In *5th International Symposium on Software Visualization*, SOFTVIS '10, page 5–14, New York, NY, USA, 2010. Association for Computing Machinery.

[43] Shrikanth N C, Suvodeep Majumder, and Tim Menzies. Early life cycle software defect prediction. why? how? *International Conference on Software Engineering*, 11 2021.

[44] Jaechang Nam, Sinno Pan, and Sunghun Kim. Transfer defect learning. In *International Conference on Software Engineering*, pages 382–391, 05 2013.

[45] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390, 2018.

[46] M.Q. Patton. Utilization-focused evaluation. *4th edition. Thousand Oaks, CA: Sage*, 2008.

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

[48] QALab. Tips to avoid the huge cost of fixing a software bug. `http://www.qalab.co/tips-to-avoid-the-huge-cost-of-fixing-a-software-bug.html`. [Online; Retrieved 01-October-2012].

[49] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *International Conference on Software Engineering*, pages 432–441, 05 2013.

[50] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[51] Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: Analytics and risk prediction of software commits. In *2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 966–969, New York, NY, USA, 2015. Association for Computing Machinery.

[52] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210 – 229, 1959.

[53] Jacek undefinedliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery.

[54] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 326–337, New York, NY, USA, 2019. Association for Computing Machinery.

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

[56] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering*, 46(8):836–862, 2020.

Appendix A

Survey Questions

The questions asked on the Google form for Study 1 are listed below:

- How many years of experience do you have in software development?
- Do you often find bugs in your code after you commit your version of code?
- Does it often introduce new problems in the future?
- Would you like to detect bugs in your work before performing the commit operation?
- When do you think detecting bug-inducing commits are helpful?
- If you think you want to detect them in some other stage, when do you think it should be?
- What kind of support do you think might help you detect bug-inducing commits?
- If you think any other kind of support is better, what should it be?
- If you have your own bug-inducing commit algorithm, would you like support where you can add your bug-inducing algorithms in a plug-in to customize it?
- If your answer is positive or negative, can you please explain the reason behind your answer?

The questions asked on the Google form for Study 3 to understand the time frame and measure the task load of the developers are listed below:

- How many years have you been working as a software developer?
- How many years have you been using GitHub as your version control system?
- While performing git operation, which of the following options you prefer? An IDE based plug-in or a command line based tool?
- Which operating system are you using as your development environment?
- How much time did you spend installing the plug-in?
- How much time did you spend installing the tool?
- How much time did the plug-in need to detect if your commit is bug-inducing or not?
- Do you think the suggestion made you look into your code twice before committing?
- How much time you spend choosing the features and algorithm?
- How much time was needed to train your model?
- Did you train the model with different features and models and compare their accuracy?

All of the following questions are asked in a Likert scale for the VS Code plug-in, the command line tool and for the configuration options:

- How much mental and perceptual activity was required?
- How much physical activity was required?
- How much time pressure did you feel due to the pace of tasks or task elements?

- How successful were you in performing the task?
- How hard did you have to work (mentally and physically) to accomplish your level of performance?
- How irritated, stressed, and annoyed versus content, relaxed, and complacent did you feel during the task?

The questions asked on the Google form for Study 3 to compare out toolchain with the state-of-the-art are listed below:

- How many years have you been working as a software developer?
- Which of the two tools gives you a good overview of the project's commits?
- Which of the two tools helps you to analyze your current commit just-in-time?
- Which of the two tools helps you to prevent your bug inducing commit just-in-time?
- Please add your opinion on the two tools, and which one do you prefer to detect bug-inducing commit just-in-time?
- Which of the two tools gave you more options for customizing your features and algorithms?

Appendix B

Dataset

The datasets we have used in our work can be found at the following link:
`https://usaskca1-my.sharepoint.com/:f:`
`/g/personal/nao816_usask_ca/EqdPBCRL01NKkpcbHfCMNvEBnq2XaPWwDuls7jAv3Qcnow?e=lubsdJ`