# Toward Better Understanding and Supporting of Continuous Integration (CI) Practices

by

Islem SAIDANI

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE IN PARTIAL FULFILLMENT FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, AUGUST 09, 2022

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

## ACKNOWLEDGEMENTS

First and foremost, praise and thanks goes to my LORD for the many blessing undeservingly bestowed upon me.

I would like to express my deepest appreciation to my parents who have believed in me when no one did. Without your sacrifices and love, I would not have finished this thesis.

I would also like to thank my second mother, Wiem, for her continued support and encouragement through my entire life. I am also very grateful to all my brothers and sisters in law who have supported me even from long distances. Never forget my nieces who brighten my days with their beautiful smiles.

Special thanks to all my friends, Sabrine, Fatma, Wafa, Khadija, Safa, Malek and others, who helped me get through the difficult times and believed in me. Can’t forget the Ouesleti family who have warmly welcomed and helped me. Thank you!

I am deeply grateful to my advisor, Prof. Ali Ouni, for giving me his trust, scientific guidance throughout the course of this thesis, engaging me in new ideas and demanding a high quality of work.

I would also like to express my gratitude to all my co-authors who have contributed to this thesis completion.

I am deeply grateful to all members of the jury for agreeing to read the manuscript and to participate in the defense of this thesis.

**Vers une meilleure compréhension et application de l'Intégration Continue (IC)**

Islem SAIDANI

## RÉSUMÉ

La croissance rapide de l'industrie du développement de logiciels pose de nouveaux défis aux développeurs de logiciels, car ils doivent répondre rapidement aux besoins des utilisateurs. En effet, les systèmes logiciels subissent de fréquentes modifications pour ajouter de nouvelles exigences, corriger les bogues découverts et s'adapter aux nouveaux changements technologiques et environnementaux. Ainsi, s'assurer que ces changements sont effectués de manière contrôlée est d'une importance cruciale. À cette fin, les entreprises de développement de logiciels ont massivement adopté l'Intégration Continue (IC) afin de réduire les risques d'erreur et d'augmenter la vitesse déploiement pour obtenir un avantage concurrentiel. L'IC vise à aider les développeurs à intégrer les changements de code constamment et rapidement grâce à des processus de construction et de test automatisés. Néanmoins, comme toute solution, l'adoption de l'IC induit plusieurs défis tels que la perte de productivité et les retards de livraison. Dans le cadre de cette thèse, nous visons à soutenir l'adoption de l'IC en abordant deux problèmes principaux. En premier lieu, nous nous attaquons au manque de connaissances empiriques sur l'adoption et les défis de l'IC. Ensuite, nous abordons le problème de construction lié à sa longue durée et à son échec.

Tout d'abord, nous avons examiné de manière empirique les défis auxquels sont confrontés les développeurs sur la base des discussions dans Stack Overflow, un forum Questions & Réponses populaire. Grâce à cette étude, nous avons révélé que la construction logicielle est un obstacle majeur auquel les développeurs sont confrontés lorsqu'ils utilisent l'IC. Deuxièmement, nous avons montré, à travers une étude empirique, comment l'adoption de l'IC peut avoir un impact sur les efforts d'assurance qualité. Nous avons constaté que l'adoption de l'IC a le potentiel de changer la façon dont les développeurs appliquent la refactorisation du code. Ensuite, nous nous sommes attaqués au problème d'échec de construction, en développant deux solutions: la première est basée sur l'adaptation de l' algorithme génétique NSGA-II, une approche de programmation génétique multi-objectifs. Cette approche permet de générer des règles à partir des données historiques des constructions et dont la sortie prédit si la construction en question est plus susceptible de réussir ou d'échouer. La deuxième approche utilise des réseaux de neurones récurrents basés sur la mémoire à long court terme LSTM-RNN pour construire des modèles de prédiction des résultats de construction. Le problème est composé d'une seule série de résultats de construction et un modèle doit apprendre de la série d'observations passées pour prédire le prochain résultat de construction dans la séquence. De plus, nous avons adapté l'algorithme génétique mono-objectif GA pour régler les hyper-paramètres de nos modèles LSTM. Les résultats de validation révèlent que les deux approches proposées ont montré de meilleures performances prédictives que les techniques existantes. Enfin, nous avons introduit un nouvel outil automatisé, basé sur l'adaptation de l'algorithme génétique multi-objectifs SPEA2, pour détecter les changements qui ne nécessitent pas de déclencher la construction, c'est-à-dire qui peuvent être ignorés. Cette approche a surpassé les techniques existantes et a été approuvée par une évaluation industrielle.

**Mots-clés:** Intégration Continue, Construction logicielle, Génie logiciel basé sur la recherche heuristique, Génie logiciel empirique, apprentissage approfondi

**Toward Better Understanding and Supporting of Continuous Integration (CI) Practices**

Islem SAIDANI

## ABSTRACT

The rapid growth of the software development industry raises new challenges to software developers as they need to respond quickly to the users' needs in a world of complex and continuous change. Indeed, software systems undergo frequent changes to add new user requirements, fix discovered bugs, and adapt to new technological and environment changes. Thus, ensuring that these changes are done in a controlled way is of crucial importance. To this end, software development companies have massively adopted Continuous Integration (CI) in order to reduce the scope for error and increasing the speed to market to gain a competitive advantage. CI aims at supporting developers in integrating code changes constantly and quickly through an automated build and test processes. Nevertheless, like any solution, CI brings with it challenges like productivity loss and release delays. In this thesis, we aim to support the adoption of CI by addressing two main problems. In the first place, we tackle the lack of empirical knowledge about CI adoption and challenges. Then, we address the problem of CI builds related to its long time and failure.

First, we empirically examined the challenges faced by CI developers based on the discussions in Stack Overflow, a popular Q&A forum. Through this study, we revealed that software build is a major barrier that developers face when using CI. Second, we showed through an empirical study, how CI adoption can impact the quality assurance efforts. We found that adopting CI has the potential to change the way developers apply code refactoring. Then, we tackled the build failure problem, by developing two solutions: The first is based the adaption of Non-dominated Sorting Genetic Algorithm (NSGA-II), a Multi-Objective Genetic Programming (MOGP) approach which allows generating rules from historical data of CI builds and whose binary output predicts whether the input build is most likely to succeed or fail. The second approach uses Long Short-Term Memory (LSTM)-based Recurrent Neural Networks (RNN) to construct prediction models for CI build outcome prediction. The problem is comprised of a single series of CI build outcomes and a model is required to learn from the series of past observations to predict the next CI build outcome in the sequence. In addition, we tailored Genetic Algorithm (GA) to tune the hyper-parameters for our LSTM models. The validation results reveal that the two proposed approaches showed better predictive performances than the state-of-art techniques. Lastly, we introduced a novel automated tool, based on the adaption of Strength-Pareto Evolutionary Algorithm (SPEA2), to detect changes that do not require to trigger the build, *i.e.,* can be skipped. This approach outperformed existing techniques and was approved through an industrial evaluation.

**Keywords:** Continuous Integration, Software Build, Search-based Software Engineering, Empirical Software Engineering, Deep Learning

# TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# LIST OF ALGORITHMS

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ETS | École de Technologie Supérieure |
| CI | Continuous Integration |
| MOGP | Multi-Objective Genetic Programming |
| ML | Machine Learning |
| DL | Deep Learning |
| NSGA | Non-dominated Sorting Genetic Algorithm |
| SPEA | Strength-Pareto Evolutionary Algorithm |
| IBEA | Indicator-Based Evolutionary Algorithm |
| GA | Genetic Algorithm |
| RF | Random Forest |
| DT | Decision Tree |
| SVC | Support Vector Classification |
| LR | Logistic Regression |
| NB | Naive Bayes |
| LSTM | Long Short-Term Memory |
| MRA | Multiple Regression Analysis |
| MOGP | Multi-Objective Genetic Programming |
| PR | Pull Requests |
| GS | Grid Search |

| | |
|---|---|
| RS | Random Search |
| TPE | Tree-structured Parzen Estimator |
| BOHB | Bayesian Optimization HyperBand |
| PSO | Particle Swarm Optimization |
| EMO | Evolutionary Multi-Objective Optimisation |
| SMOTE | Synthetic Minority Oversampling Technique |

# CHAPTER 1

# INTRODUCTION

## 1.1 Research context

Due to fierce competition in the global software industry, the companies are under increasing demands to continuously deliver high-quality products at low risk. Thus, improving the software development process can give a company a competitive advantage over its competitors. In this context, modern processes have gravitated towards agile methodologies to realize value through improved performance and meet the needs of today's complex software development. However, in such context, software change management represents a fast-paced task of extreme complexity (Yu, Vasilescu, Wang, Filkov & Devanbu, 2016a). In fact, developers are struggling to find a balance between continuous delivery, and how to ensure that these quick changes impact the development process positively rather than hinder it. A recent movement in the agile development industry attempts to support an efficient change management process by relying on *Continuous Integration*.

Continuous integration (CI) (Duvall, Matyas & Glover, 2007a) is an indispensable step of modern software engineering practices (Brandtner, Giger & Gall, 2014) that aims to automatically control the gate between development and deployment. First, it provides the mechanism through which changes are evaluated for risk and then applied. Each time a change is requested to the software system, the CI pipeline performs automated tests to ensure that the change will not break the system, and makes changes that pass the automated tests available to be deployed with appropriate authorization controls. Its philosophy, as put forward by Fowler (2006), is to regularly integrate the changes introduced by different developers into a shared repository branch. This best practice has proved to enhance team productivity by freeing developers to do more thought provoking work without compromising quality with the early detection of defects (Vasilescu, Yu, Wang, Devanbu & Filkov, 2015). For these valuable benefits, CI has been widely adopted in the open-source as well as the software industry markets (Yu, Yin, Wang, Yang & Wang, 2016b; Vassallo, Palomba & Gall, 2018b; Vasilescu *et al.*, 2015). For instance, it has been reported that at least 40% of the 34,000 most popular open-source projects on GitHub use CI (Hilton, Tunnell, Huang, Marinov & Dig, 2016). From the academic side, the study of CI has become an active research topic and triggered many research papers to study its impacts (Bernardo, da Costa & Kulesza, 2018; Ståhl, Mårtensson & Bosch, 2017; Gupta, Khan, Gallaba & McIntosh, 2017; Vasilescu *et al.*, 2015; Gupta *et al.*, 2017), usage (Vassallo *et al.*, 2018b; Yu *et al.*, 2016a; Vassallo, Palomba, Bacchelli & Gall, 2018a; Abdalkareem, Mujahid, Shihab & Rilling, 2019) and costs (Widder, Vasilescu, Hilton & Kästner, 2018; Hilton *et al.*, 2016; Vassallo *et al.*, 2017; Luo, Zhao, Ma & Chen, 2017).

## 1.2 Problem statement

While CI can be considered as a well-established discipline of change management, introducing changes under such context still be is risky and can lead to productivity loss, release delays and cost impacts. In the next subsections, we highlight the different challenges, to be addressed in this thesis project, that are mainly related to the lack of empirical knowledge of CI adoption and the complexity of CI build process.

### 1.2.1 Lack of empirical knowledge about CI

Like any software development process, CI brings many challenges (Widder *et al.*, 2018) and its adoption can have side effects on other software development practices (Zhao, Serebrenik, Zhou, Filkov & Vasilescu, 2017). Although there is an increasing amount of literature has been published on CI adoption and its challenges, there are many knowledge gaps that need to be filled. We highlight these open issues, as follows:

**Problem 1. Limited empirical knowledge about CI challenges**

The existing studies highlighting the challenges faced by developers, relied on surveys of CI practitioners (Hilton *et al.*, 2016; Hilton, Nelson, Tunnell, Marinov & Dig, 2017), and case studies (Colomo-Palacios, Fernandes, Soto-Acosta & Larrucea, 2018; Ståhl *et al.*, 2017). However, interviews and case studies are a limited resource and cannot be generalized due to the necessarily limited number of people/projects they gather information from. For instance, while we know that developers struggle with CI builds (Luo *et al.*, 2017), it is still unclear what build tools/servers are challenging the most. There are other sources of information, besides interviews and case studies, for understanding the challenges of CI; the popular Q&A forums such as Stack Overflow [1]. We therefore believe that an analysis about CI discussions on SO can help the research community and CI stakeholders in better understanding of developers' concerns and hence improving the adoption of CI.

## Problem 2. Lack of empirical knowledge about the impacts of adopting CI on quality assurance practices

A promising feature of CI is quality assurance (Duvall *et al.*, 2007a; Vasilescu *et al.*, 2015; Hilton *et al.*, 2017) that is usually performed through code Refactoring (Fowler, Beck, Brant, Opdyke & Roberts, 1999). Indeed, as an agile method, the incremental nature of CI requires the code to be continuously refactored in order to maintain high quality (Stamelos & Sfetsos, 2007) and keep the quality gates (*i.e.,* steps required to ensure the reliability of code changes (Schermann, Cito, Leitner & Gall, 2016)) always green (Vassallo *et al.*, 2018b). Otherwise, it may be hard for development teams to understand, maintain and extend their code (Szóke, Antal, Nagy, Ferenc & Gyimóthy, 2014). Moreover, the absence of Continuous Refactoring (CR) may result in the need for large refactorings (Stamelos & Sfetsos, 2007) that, like any other complex change, may hinder the CI build progress and requires more debugging effort (Zhang, Chen, Chen, Peng & Zhao, 2019). Although, recent work (Vassallo *et al.*, 2018b) emphasized the potential of CI in changing the way developers perceive and apply Refactoring. we still lack empirical evidence to confirm or refute this assumption. Such empirical knowledge can help evaluating CI impacts as well as guiding the design of tools and processes to control code quality assessment.

[1] http://stackoverflow.com/

### 1.2.2 CI build failure and long time

There is a consensus that build failure is a major barrier that developers face when adopting CI (Luo *et al.*, 2017) which may prevent them from proceeding further with development. Additionally, the build process is typically time and resource-consuming and can cause delays in the product release dates. In this thesis by article, we identify two problems related to CI builds.

**Problem 3. CI build failure**

It is crucial to preemptively detect whether a change is likely to trigger a build failure before integrating it to the system. This would help to save time and effort needed for build resolution and thus enhancing the developers' productivity. Existing solutions leverage the history of previous CI builds in order to train Machine Learning (ML) based techniques. Although these works have advocated that predicting CI build outcome is possible and beneficial, the CI build failure prediction is not yet resolved as the accuracy on the failure prediction (*i.e.,* minority class) is not accommodated.

**Problem 4. The long build time**

As stated before, the build process is typically time-consuming. One of the main reasons for such delays is that some simple changes (*i.e.,* can be skipped) trigger the build, which represents an unnecessary overhead and particularly painful for large projects. This challenge motivated the research on developing techniques (Abdalkareem *et al.*, 2019; Abdalkareem, Mujahid & Shihab, 2020) to speed up CI process by detecting commits that do not require the system's build (*e.g.,* commits affecting non-source files). However, these existing techniques are not practically effective which suggests that the problem is not yet resolved.

## 1.3 Research objectives

The main objective of this thesis is to **enhance CI adoption by providing a better understanding of its practice and proposing solutions to optimize its process**. It will be achieved with four specific objectives that are summarized as follows:

- **Objective #1:** Take advantage of the richness of information provided by Q&A forums to gain a deep understanding of developers' challenges when adopting CI in practice.
- **Objective #2:** Explore and understand the impacts of CI on the code quality assurance efforts.
- **Objective #3:** Reduce the expenses of CI build failures by predicting its outcome.
- **Objective #4:** Speed-up the CI build process by detecting the commits that can be skipped during the build.

## 1.4 Main contributions

To overcome the previously identified problems, we propose the following contributions: (cf. Figure 1.1):

### Contribution 1: Empirical study about CI challenges in Stack Overflow

We propose the first empirical study of CI challenges that is based on Q&A forums. Indeed, we collect CI related posts from Stack Overflow (SO), the most popular crowd-sourced forum, and analyze those posts through quantitative and qualitative analyzes. First, to study the trends of CI discussions, we investigate the metadata of CI questions, users and tags. Then, we extract the CI main topics using Latent Dirichlet Allocation (LDA) tuned with Genetic Algorithm (GA). Finally, we investigate the most popular and difficult topics faced by developers and perform a qualitative analysis based on a statistical sample of unanswered questions to get further insights into the CI challenges. Additionally, we provide practical implications of our findings for researchers, developers, tool builders and educators.

**Contribution 2: Empirical study about the impact of CI on Refactoring practice**

We provide the first empirical evidence of the impact of CI on Refactoring by designing three novel research questions to understand the evolution of Refactoring practices, in terms of frequency, size and involved developers. We collect a large corpus of commits and Refactoring operations extracted from open-source projects that adopt CI and analyze the changes before and after CI adoption using Multiple Regression Analysis (MRA). We also provide practical implications of our findings for future research on the Refactoring of modern systems, that can be useful to raise developer's awareness of Refactoring in the context of CI and recommend Refactoring tools suitable to CI context.

**Contribution 3: Two approaches to predict CI build failure**

We first address problem of CI build failure by introducing a novel search-based approach based on Non-dominated Sorting Genetic Algorithm (NSGA-II) to generate CI build failure prediction rules. Our approach aims at finding the best combination of CI built features and their appropriate threshold values, based on two conflicting objective functions to deal with both failed and passed builds. We conduct an empirical study of our approach compared to different existing techniques. Additionally, we provide qualitative evidence of the potential reasons behind build failure through a novel feature ranking approach. The approach is implemented as a command line-based tool called BF-Detector (Saidani, Ouni, Chouchen & Mkaouer, 2021a) that is available as a standalone Java *jar* file in order to facilitate its integration within CI frameworks. Finally, we improve the prediction of BF-Detector by introducing bad smells related information (Saidani & Ouni, 2021).

Then, we propose another formulation for CI build failure prediction problem as a time series problem using Long short-term memory (LSTM) the artificial Recurrent Neural Network (RNN). To the best of our knowledge, this is the first attempt to use deep learning LSTM-based approach to learn CI build failures. The built model can be trained efficiently using a representative subset of the CI build outcomes, which requires no feature engineering. Moreover, we use Genetic

Algorithm (GA) to optimize the hyper-parameters of our models in order to achieve optimal performance. To evaluate our approach, we compare it under different validation scenarios against existing solutions.

**Contribution 4: Search-based approach to detect CI Skip commits**

We introduce a novel search-based approach based on the adaptation of Strength-Pareto Evolutionary Algorithm (SPEA2) (Zitzler *et al.*, 2001) to generate CI skip prediction rules. We evaluate our approach and investigate the performance using different validation scenarios while comparing its predictive performance against the baseline approaches. Furthermore, we evaluate our approach through an empirical study conducted with industrial experts. In addition, we provide qualitative evidence of the potential reasons behind CI skip using our proper detecting rules.

## 1.5 Thesis organization

As shown in Figure 1.1, this thesis is composed of six chapters. Chapter 2 provides a review of the literature on previous research that is relevant to the main themes of this dissertation. In Chapter 3, we present our empirical study about CI challenges and discussions on Stack Overflow. Then, in Chapter 4, we present our empirical study about the impacts about CI on Refactoring practice. Chapter 5 and Chapter 6 reports our search-based and deep-learning based contributions for the prediction of CI build failure respectively. In Chapter 7, we introduce our search-based approach for the detection of CI Skip commits. Finally, we summarize the main contributions and reporting the recommendations for future research in the Conclusion.

## 1.6 Publications

The following is a list of our publications related to this dissertation:

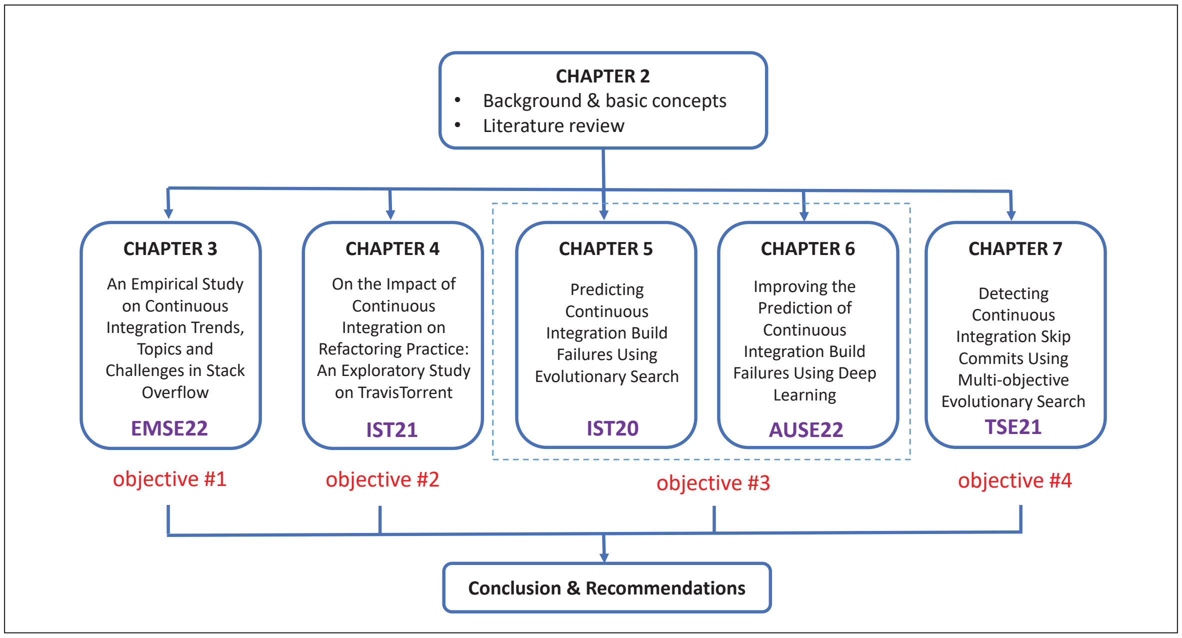

CHAPTER 2
Background & basic concepts
Literature review
CHAPTER 3
An Empirical Study on Continuous Integration Trends, Topics and Challenges in Stack Overflow
EMSE22
CHAPTER 4
On the Impact of Continuous Integration on Refactoring Practice: An Exploratory Study on TravisTorrent
IST21
CHAPTER 5
Predicting Continuous Integration Build Failures Using Evolutionary Search
IST20
CHAPTER 6
Improving the Prediction of Continuous Integration Build Failures Using Deep Learning
AUSE22
CHAPTER 7
Detecting Continuous Integration Skip Commits Using Multi-objective Evolutionary Search
TSE21
objective #1
objective #2
objective #3
objective #4
Conclusion & Recommendations

Figure 1.1 Thesis organisation & contributions

## Articles in Journals

1. Saidani, Islem and Ali Ouni. "An Empirical Study on Continuous Integration Trends, Topics and Challenges in Stack Overflow." paper submitted in 2022 to the journal of Empirical Software Engineering (EMSE).
2. Saidani, Islem, Ali Ouni, and Mohamed Wiem Mkaouer. "Improving the prediction of continuous integration build failures using deep learning." Journal of Automated Software Engineering, 29, no. 1 (2022): 1-61. Available at: https://link.springer.com/article/10.1007/s10515-021-00319-5
3. Saidani, Islem, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. "On the impact of Continuous Integration on Refactoring practice: An exploratory study on TravisTorrent." Information and Software Technology 138 (2021): 106618. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0950584921000914
4. Saidani, Islem, Ali Ouni, and Wiem Mkaouer. "Detecting skipped commits in continuous integration using multi-objective evolutionary search." IEEE Transactions on Software Engineering (2021). Available at: https://ieeexplore.ieee.org/document/9622158/

5. Saidani, Islem, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. "Predicting continuous integration build failures using evolutionary search." Information and Software Technology 128 (2020): 106392. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0950584920301579

**Articles in Refereed Conferences and Workshops**

1. Saidani, Islem, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. "Bf-detector: an automated tool for ci build failure detection." In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tool Track, pp. 1530-1534, 2021. Available at: https://dl.acm.org/doi/10.1145/3468264.3473115
2. Saidani, Islem, and Ali Ouni. "Toward a Smell-aware Prediction Model for CI Build Failures." In IEEE/ACM International Workshop on Refactoring (IWOR), pp. 18-25. IEEE, 2021. Available at: https://ieeexplore.ieee.org/document/9680290
3. Saidani, Islem, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. "On the prediction of continuous integration build failures using search-based software engineering." In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, pp. 313-314. 2020. Accepted as a poster, available at: https://dl.acm.org/doi/pdf/10.1145/3377929.3390050

# CHAPTER 2

# STATE OF THE ART

## 2.1 Introduction

This chapter provides a literature review on research work related to this thesis. We first provide the background required to understand this thesis. Then, we survey the related work that is relevant to the main themes of this research work.

## 2.2 Background

In this section, we provide the necessary background for CI process, search-based approaches and LSTM-RNN approach.

### 2.2.1 CI process

CI aims to build healthier software systems by developing and testing in smaller increments without compromising software quality. The basic notion of CI, as described by Folwer (Fowler, 2006) is to support developers' work by automating the code compilation, dependencies collection and tests running. This process is an enduring check on the quality of contributed code that mitigates the risk of “breaking the build" as regressions can be detected and fixed immediately.

CI has a well-defined life-cycle when generating builds. The main phases of the CI build life-cycle are depicted in Figure 2.1. First of all, a contributor forks, *i.e.,* clones, the project repository, makes some changes, as creating a new feature or by fixing some bugs, on the code base (1). When the work is done, the contributor submits the changes to the original repository (2). At this point, the CI service carries out a series of tasks to build and test these changes (3). Then, it provides immediate feedback on the outcome of the test to the core team (4), *i.e.,* developers who dispose of write access to a project's code repository (Vasilescu *et al.*, 2015).

When one or more of those tasks fail, the build is considered *failed*, otherwise it will be *passed* and core team members proceed to do a code review and, if necessary, the submitter would be requested for modifications. After a cycle of code reviews, automatic building and testing, if everyone is satisfied, the submitted changes will be merged to the mainline branch.

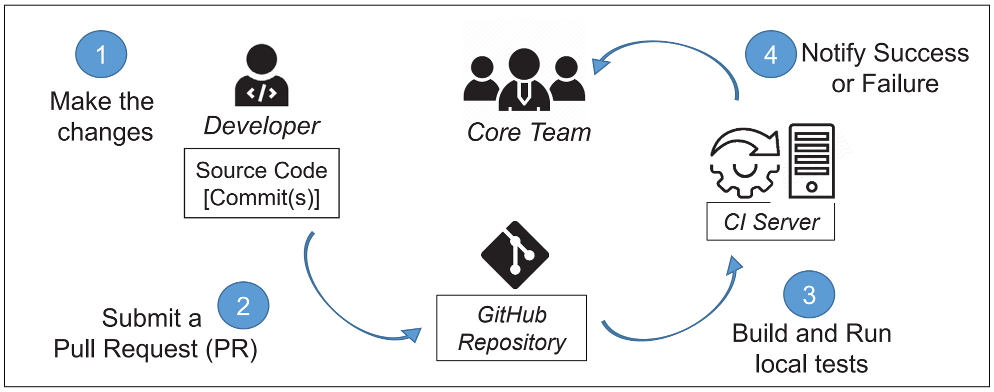


Figure 2.1 CI build process

### 2.2.2 Search-based techniques

Different mono- and multi-objective metaheuristic techniques are used in this thesis. We provide in this section the necessary background for unfamiliar readers with metaheuristics. More specifically, we used the following metaheuristics: Genetic Algorithm (GA), Non-dominated Sorting Genetic Algorithm (NSGA-II) and Strength-Pareto Evolutionary Algorithm (SPEA2).

#### 2.2.2.1 Genetic Algorithm

GA is a widely used computational search technique, that has proven good performance in solving many software engineering problems (Harman, Mansouri & Zhang, 2012; Harman, McMinn, De Souza & Yoo, 2010; Mkaouer *et al.*, 2015; Ouni, Kessentini, Sahraoui, Inoue & Deb, 2016). GA is inspired by Darwinian evolution and aims at finding -near- optimal solutions by simulating a natural evolutionary process (Goldberg, 1989). Algorithm 2.1 provides a high level pseudo-code of GA. It starts by randomly creating an initial population $P_0$ of individuals encoded using a specific representation. Then, a child population $Q$ is generated from the population of parents $P_0$ using genetic operators (crossover and mutation). The whole population $Q$ is sorted according to their performance computed by a fitness function and the worst solutions will be excluded based on the elitism mechanism, *i.e.,* only the fittest solution will survive and will be transmitted to the next population. This process will be repeated until reaching the last iteration according to a stop criterion.

Algorithm 2.1 High level pseudo code of the Genetic Algorithm (GA) (Davis, 1991)

```
1: Create an initial population $P_0$;
2: EvalPopulation($P_0$); /* Evaluates the population $P_0$ */
3: $t = 0$;
4: while stopping criteria not reached do
5:     $Q \leftarrow$ create-new-pop($P_t$); /* Create new solutions from $P_t$ */
6:     /* EvalPopulation($P_t$); /* Evaluate the new solutions */
7:     $P_{t+1} \leftarrow ApplyGeneticOperators(P_t \cup Q)$; /* Next generation population*/
8:     t = t+1;
9: end while
```

### 2.2.2.2 NSGA-II

NSGA-II (Deb & Jain, 2013) has proven good performance in solving many software engineering problems (Harman *et al.*, 2012, 2010; Mkaouer *et al.*, 2015; Ouni *et al.*, 2016). As described in Algorithm 2.2, NSGA-II starts by randomly creating an initial population $P_0$ of individuals encoded using a specific representation (line 1). Then, a child population $Q_0$ is generated from the population of parents $P_0$ (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population $R_0$ of size $N$ (line 5). *Fast-non-dominated-sort* (Deb *et al.*, 2002) is the technique used by MOGP to classify individual solutions into different dominance levels (line 6) (Deb *et al.*, 2002). The whole population that contains $N$ individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front $F_0$ get assigned dominance level of 0. Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front $F_1$ of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. Fronts are added successively until the parent population $P_{t+1}$ is filled with $N$ solutions (line 8). When MOGP has to cut off a

Algorithm 2.2 High level pseudo code of NSGA-II (Deb *et al.*, 2002)

```
1: Create an initial population $P_0$
2: Create an offspring population $Q_0$
3: $t = 0$
4: while stopping criterion not reached do
5:     $R_t = P_t \cup Q_t$
6:     F = fast-non-dominated-sort($R_t$)
7:     $P_{t+1} = \emptyset \ and \ i = 1$
8:     while $\mid P_{t+1} \mid + \mid F_i \mid \leqslant N$ do
9:         Apply crowding-distance-assignment($F_i$)
10:        $P_{t+1} = P_{t+1} \cup F_i$
11:        $i = i + 1$
12:    end while
13:    $Sort(F_i, \prec n)$
14:    $P_{t+1} = P_{t+1} \cup F_i[N- \mid P_{t+1} \mid]$
15:    $Q_{t+1}$ = create-new-pop($P_{t+1}$)
16:    t = t+1
17: end while
```

front $F_i$ and select a subset of individual solutions with the same dominance level, it relies on the crowding distance (Deb *et al.*, 2002) to make the selection (line 9). This parameter is used to promote diversity within the population. The front $F_i$ to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of $F_i$ are chosen (line 14). Then, a new population $Q_{t+1}$ is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to a stop criterion (line 4).

#### 2.2.2.3 SPEA2

SPEA2 is as an intelligent search-based algorithm, which has been widely adopted to solve many software engineering problems (Zhao, Lei, Ma, Liu & Zhang, 2016; Garcia & Trinh, 2019; Sofianopoulos & Tambouratzis, 2011). As described in Algorithm 2.3, SPEA2 starts with an initial population $P_0$ and an empty archive $\bar{P_0}$ (line 1). The external archive is a collection of high-quality solutions to be maintained and used exclusively for mating of future generations. The archive size is typically, but not necessary, equal to the population size. Then, the following steps are performed in each iteration $t$. The fitness assignment (line 3), which is based on the

*Pareto dominance* principle for both the population and the archive in order to quantify the quality of candidate solutions in the current population. Note that a non-dominated solution is a solution that has fitness values such that no other solution within the set has better fitness values across all objectives. During the environmental selection step (line 4), all non-dominated solutions from the population and archive are copied to the archive of the next generation: $\bar{P}_{t+1}$. If the non-dominated set size fits exactly into the archive ($|\bar{P}_{t+1}| = N$) the environmental selection step is completed. Otherwise, there can be two situations: Either the archive $\bar{P}_{t+1}$ is too small or too large. In the first case, the best $N - |\bar{P}_{t+1}|$ dominated individuals in the previous archive and population are copied to the new archive. In the second case, an archive truncation procedure is employed to iteratively remove individuals from $\bar{P}_{t+1}$ until $|\bar{P}_{t+1}| = N$. Next, if the stopping condition is not met then mating selection is performed (line 5). Binary tournament selection with replacement on $\bar{P}_{t+1}$ is used to fill the mating pool. Finally crossover and mutation operators are applied to the mating pool and set $\bar{P}_{t+1}$ to the resulting population. The generation counter is increased (line 7) and this process will be repeated until some condition is satisfied. (line 2).

Algorithm 2.3 Pseudo code of SPEA2 (Zitzler *et al.*, 2001)

```
Require: N: population size, $\bar{N}$: archive size, T: maximum number of generations
Ensure: A: non-dominated set
1: Initialization: Generate an initial population $P_0$, create an empty archive (external set)
   $\bar{P}_0 = \emptyset$. Set $t = 0$
2: while $t < T$ or another stopping criterion is not reached do
3:     Fitness assignment: Calculate fitness values of individuals in $P_t$ and $\bar{P}_t$
4:     Environmental selection: Copy all non-dominated individuals in $P_t$ and $\bar{P}_t$ to $\bar{P}_{t+1}$
       IF size of $\bar{P}_{t+1} > \bar{N}$ THEN reduce the size of $\bar{P}_{t+1}$
       ELSE IF if size of $\bar{P}_{t+1} < \bar{N}$ THEN fill $\bar{P}_{t+1}$ with dominated individuals from $P_t$ and $\bar{P}_t$
5:     Mating selection: IF the stopping condition is not satisfied
       THEN perform binary tournament selection with replacement on $\bar{P}_{t+1}$in order to fill
       the mating pool ELSE Stop.
6:     Variation: Apply crossover and mutation into $\bar{P}_{t+1}$
7:     t = t+1
8: end while
9: Termination: $A = \bar{P}_{t+1}$
```

### 2.2.3 LSTM-RNN

Long Short-Term Memory (LSTM) networks (Hochreiter & Schmidhuber, 1997a) are a special type of Recurrent Neural Networks (RNNs) that have recently emerged as effective models capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (Hochreiter & Schmidhuber, 1997b) and widely applied in language modeling (Sundermeyer, Schlüter & Ney, 2012), machine translation (Cui, Wang & Li, 2015), speech recognition (Graves, Jaitly & Mohamed, 2013), classification (Athiwaratkun & Stokes, 2017; Wang, Huang, Zhu & Zhao, 2016b; Graves & Schmidhuber, 2005) and many other real-world problems (Bouktif, Fiaz, Ouni & Serhani, 2018, 2020).

LSTM networks were designed to overcome the difficulty of training RNNs due to the vanishing gradient problem (Pascanu, Mikolov & Bengio, 2013). In a nutshell, it was observed that the gradients in RNNs tend to get smaller with back-propagation which forces the network to interrupt the learning process. In addition to hidden state and memory vectors, LSTMs introduce three gating mechanisms (Karpathy, Johnson & Fei-Fei, 2015) namely (i) forget gate for deletion of less important information from memory, (ii) input gate to add new information to cell state and (iii) output gate which decides what to output from memory. These gates allow efficient management of LSTM internal cell memory.

Figure 2.2 shows the information flow and the set of gates within LSTM cells (colah's blog, 2020). In this diagram, the pink circles represent point-wise operations (e.g. "+" operation for addition), while the yellow boxes stand for neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied to different locations.

## 2.3 Related Work

This section presents the research around this thesis. In particular, the related work can be divided broadly into two research areas: (1) studies on CI practice, impacts & challenges and (2) work related to CI builds.

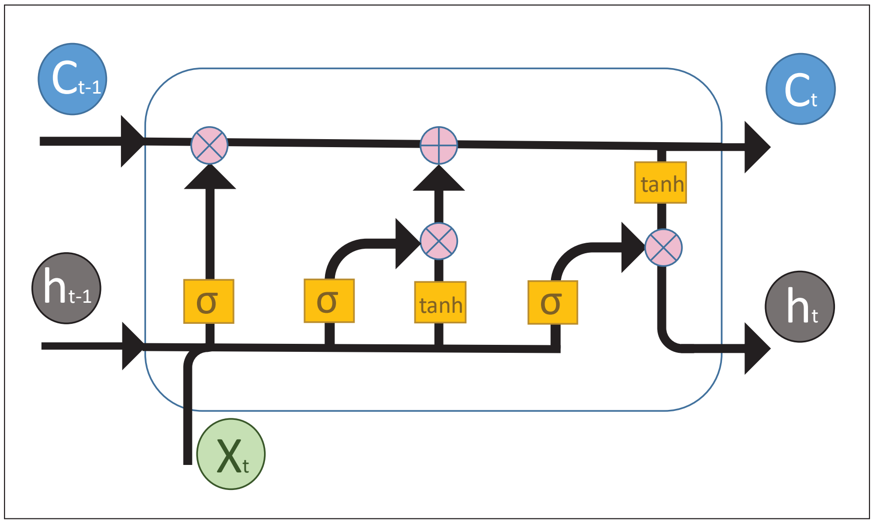


Figure 2.2 An overview of information flow in LSTM

### 2.3.1 CI practice, impacts & challenges

#### 2.3.1.1 CI practice and impacts

Many research works have focused on studying the outcomes of the adoption of CI, related mainly to teams' productivity, development practices and code quality. For the sake of clarity and completeness of the reporting, we summarize these works in Table 2.1.

Many research works have shown that CI adoption can be beneficial. For instance, Vasilescu *et al.* have found, using multiple regression modeling, that CI improves the number of processed Pull Requests (PRs) (*i.e.,* a submitted candidate code change to be merged into the mainline repository) and reduces the quantity of rejected ones. These results indicate a significant improvement in the team's productivity. Hilton *et al.* claimed also that CI improves team's productivity. Indeed, they found that after adopting CI (i) the studied CI projects release twice more than those that do not use CI and (ii) the PR is accepted faster. Yu *et al.* studied the acceptance and latency of PR in CI context. Using regression models in a sample of 10 GitHub projects that use Travis CI, the authors found that the availability of the CI pipeline is a dominant factor in hastening the PR evaluation process. Yu *et al.* studied the nature of CI detected defects

Table 2.1 A summary of the literature on the impact of CI adoption on software quality/development

| **Study** | **Studied impact(s)** | **Methodology** | **Results** |
|---|---|---|---|
| Vasilescu *et al.* | Quality and Productivity | Regression Analysis of 246 Travis CI projects | CI improves the productivity without an observable diminishment in code quality. |
| Hilton *et al.* | Productivity | Mining 1,529,291 builds from Travis CI | • Projects that use CI release more than twice as often as those that do not use CI.<br>• The PR is accepted sooner. |
| Yu *et al.* | Productivity | Regression Analysis of top 10 Travis CI projects | the presence of CI is a dominant factor for both PR acceptance and latency |
| Yu *et al.* | Quality | Regression Analysis of 246 Travis Ci projects | • CI failures are not highly correlated with eventual bugs,<br>• A mature CI process is associated with better fault detection<br>• The use of CI in a PR does not necessarily mean having a request of good quality. |
| Zhao *et al.* | development practices | Regression Analysis of 575 Travis CI projects | CI adoption is associated with:<br>• An increase in the number of merged commits<br>• the "commit small" guideline is followed to some extent<br>• An increasing trend in the number of closed PR<br>• An increase in the PR latency<br>• A drop trend in the number of issues closed issues. |
| Bernardo *et al.* | Productivity | Regression Analysis of 162,653 PR of 87 Travis CI projects | CI does not always reduce the time for delivering PRs |
| Rahman, Agrawal, Krishna & Sobran | Productivity and quality | Mining 150 OSS and 123 proprietary projects | Closed bugs, closed issues, and frequency of commits, significantly increased after adoption of CI for OSS projects, but not for proprietary projects. |

and social factors are associated with them and how they relate to eventual bugs. To this end, they performed both quantitative and qualitative analysis: regression modeling and a qualitative study of 50 PRs. As a result, they found that CI failures are not highly correlated with eventual bugs and that mature CI process is associated with better fault detection. Zhao *et al.* used regression discontinuity design (Imbens & Lemieux, 2008) to quantitatively evaluate the effect of adopting CI on development practices, such as code writing and submission, issue and PR closing. The main result of their study is that CI practice aligns with the “commit often" guideline (Fowler, 2006) while merged commits seem to be getting smaller as recommended by Fowler.

While studies mentioned above suggest that the adoption of CI increases the release frequency of a software project, other works did not observe such an increase in their quantitative analyses. For instance, Bernardo *et al.* have observed, by training regression models, that CI does not always reduce the time for delivering merged PRs. Their models also reveal that PRs, which are merged more recently in a release cycle, experience a slower delivery time. Rahman *et al.* have observed for the studied OSS projects some CI benefits *e.g.,* improvements in bug and issue resolution. However, for the proprietary projects, they could not make similar observations.

To sum up, previous work showed that CI adoption can have some side effects but it is unclear how it can impact the quality assurance efforts like code Refactoring. Hence, we fill this knowledge in Chapter 4, by conducting the first empirical study to investigate the impacts of CI on Refactoring activities.

#### 2.3.1.2 CI challenges

Despite its valuable benefits, CI adoption brings with it many challenges. In the following, we review the most relevant papers that discuss CI challenges and bad practices. These works are summarized in Table 2.2.

Debbiche *et al.* (2014) interviewed 11 developers at a major communication company to investigate what challenges they faced when adopting CI. Their case study revealed a list of challenges including testing, code dependencies and tools and infrastructures. Laukkanen *et al.*

Table 2.2 Literature review summary of CI challenges

| Paper | Main Challenges | Methodology |
|---|---|---|
| Elazhary *et al.* (2021) | • Build can be a bottleneck<br>• Longer builds<br>• Dependency management is complex<br>• UI difficult to test<br>• Developer works in isolation until merge<br>• No testing in deployment builds | • case study of three small- to medium-sized companies interviews<br>• Activity log mining |
| Widder, Hilton, Kästner & Vasilescu (2019) | • Lack of support for a project's language<br>• Long builds frustrate users or make CI impractical<br>• CI configuration files can be confusing<br>• Troubleshooting build failures is difficult | • 12 interviews from Travis CI leavers |
| Vassallo, Proksch, Gall & Di Penta (2019b) | • Slow Build<br>• Skip Failed Tests<br>• Broken Release Branch<br>• Late Merging | • Survey with 124 developers |
| Pinto, Castor, Bonifacio & Rebouças (2018) | • Configuring the build environment<br>• Inadequate testing and time pressure<br>• False sense of confidence | • Survey with 158 CI users |
| Hilton *et al.* (2017) | • Troubleshooting a CI build failure<br>• Long build time<br>• Automating the build process<br>• Lack of support for the desired workflow<br>• Setting up a CI server or service<br>• Lack of tool integration<br>• Security and access controls | • Survey with 51 developers |
| Laukkanen, Itkonen & Lassenius (2017) | • Complex and inflexible build<br>• Large commits<br>• Merge conflicts<br>• Broken builds<br>• Long-running branches<br>• Ambiguous test result<br>• Time-consuming testing<br>• Lack of experience and motivation<br>• Slow integration approval | • Systematic Literature Review |
| Shahin, Babar & Zhu (2017) | • Build and test time<br>• Lack of awareness and transparency on build and test results<br>• Lack of expertise and skill<br>• Lack of suitable tools and technologies<br>• Coordination and collaboration challenges<br>• Difficulty to change legacy culture<br>• Lack of proper test strategy<br>• Merging conflicts<br>• Security and scalability issues in deployment pipeline | • Systematic literature Review |
| Laukkanen, Paasivaara & Arvonen (2015) | • Lack of Time<br>• Unstable Tests<br>• Slow Tests<br>• Insufficient Testing Environments<br>• Agreement on Tools<br>• Global Distribution of the Organization | • Interviewing 27 stakeholders at Ericsson |
| Debbiche, Dienér & Svensson (2014) | • Difficulty to cope with CI culture<br>• Difficulty with Tools and Infrastructure<br>• Regression Feedback Time<br>• Test Automation<br>• Unstable Test Cases<br>• Too Many Manual Tests<br>• Preserving Quality | • 11 interviews at a major telecommunication company |

(2015) interviewed 27 stakeholders at Erickson and found that the main challenges are due to the lack of time, unstable tests, tools and team organization. Shahin *et al.* (2017) used Systematic Literature Review (SLR) method for reviewing the peer-reviewed papers on continuous practices published between 2004 and 2016. Their findings revealed a list of critical factors including testing effort and time, team awareness and transparency, good design principles and appropriate infrastructures that should be carefully considered when introducing continuous practices in a given organization. Another SLR was performed by Laukkanen *et al.* (2017) who identified forty problems related to CI/CD adoption. The most critical reported problems were related to testing, merging conflicts and system design. Hilton *et al.* (2017) have found, by interviewing 51 developers, that the latter face many trade-offs between speed and assurance, between better access and information security, and between more CI configuration options and better flexibility in use. Additionally, the main challenges they face are related to build failure and time, lack of tool support and the setting of CI servers. Pinto *et al.* (2018) surveyed 158 CI users in order to investigate work practices and challenges in CI. Their main results reveal that CI main challenges are (i) the build environment, (ii) inadequate testing and time pressure and (iii) false sense of confidence in CI servers' feedback. Widder *et al.* (2019) conducted a mixed method, including a survey and statistical modeling from 6,239 projects to identify the reasons behind CI abandonment. There results revealed that many developers find that the most difficult issues to resolve are (1) build failures, (2) complex tool setups like Docker and (3) long build times. Vassallo *et al.* (2019b) survey with 124 developers revealed four relevant anti-patterns (*i.e.,* practices that contradict CI principles) of CI namely slow build, skipping failed tests, broken release branch and late merging. Elazhary *et al.* (2021) found, by performing a case study and activity log mining, that CI main challenges are related to builds, dependency management and testing.

The main conclusions to drive from this review is that it is confirmed that CI adoption brings many challenges to the development team. The previous works that studied CI challenges focused on interviewing/surveying a selected set of developers. However, surveys/interviews are a limited resource of information and therefore cannot be generalized. At the same time, there are other sources of information for examining CI challenges and SO is one of them. This Q&A website is a popular venue for developers who seek advice to resolve many technical problems/issues. To the best of our knowledge, there is no study that mined SO to investigate CI challenges. We fill this gap with an empirical study, where we analyze developers discussions to investigate CI trends, topics and challenges. This study is presented in Chapter 3.

### 2.3.2 CI builds

#### 2.3.2.1 Prediction of CI failure

Many research works have introduced prediction models to predict the build outcome. For the sake of clarity, we summarise them in Table 2.3.

Xia & Li (2017) compared nine ML models to construct CI prediction models of 126 open source projects hosted on GitHub. Their experiments were based on both cross-validation and online scenarios. In cross-validation, their models achieved an Area Under the ROC Curve (AUC) score of over 70%. However, under the online scenario, they observed a tendency for their prediction scores to decrease up to 60% of AUC. In both scenarios, they found that Decision Tree (DT) and Random Forest (RF) achieved the best performance scores. Ni & Li (2017) employed AdaBoost (ADA) to predict CI build failures of 532 CI projects. This adaption achieved an AUC of 75%, using 50% of the dataset as training set and last 50% instances as test set. Hassan & Wang (2017) proposed the prediction model of CI build failure on three build systems, namely Ant, Maven and Gradle, under the cross-project prediction and cross-validation scenarios. Using RF, they achieved over 90% of AUC scores for the considered build systems. Additionally, the cross-validation provided better results. However, when we looked at the provided dataset, we found that there is a large number of redundant lines that may influence the validity of the reported results. When looked at their dataset, we found that the dataset is perfectly balanced (45% of failed builds) which is not the case in practice as it is generally known that failed builds are much less to occur than passed ones (Xie & Li, 2018). Xia *et al.* (2017a) conducted an empirical study to evaluate the predictive performance of six common ML models including RF and DT considering cross-project validation. For dataset selection, they compared three methods namely Random Selection, Burak Filter based on build-level and Bellwether Strategy based on project-level. According to the results of their experiments, they found that Bellwether strategy

Table 2.3 An overview of the literature on the prediction of CI build outcome

| **Ref** | **Results** | **Considered Models** | **Summary** |
|---|---|---|---|
| Xia, Li & Wang (2017a) | DT is the best classifier with F1-Score = 33% considering Bellwether strategy | DT, Gradient Boosting (GB) , RF, LR, KNN, NB | Empirical study to compare two data selection filters including Burak Filter and Bellwether Strategy. |
| Ni & Li (2017) | The prediction achieved a score of 74% in terms of accuracy. ADA was the best performer. | ADA, DT and NB | The authors adapted ADA algorithm to predict CI build failure and compared this approach to NB and DT. The considered scenario was not mentioned. |
| Xia & Li (2017) | The prediction performance in cross validation scenario achieved 55% in terms of F1 in median. When it comes to online scenario, the prediction performance falls to 30%. In both scenarios, DT and RF showed the best performance. | 9 ML classifiers including DT, RF etc. | The authors used 9 classifiers to construct prediction models and investigated the performance of both cross-validation and online predictions. |
| Luo *et al.* (2017) | SVM and LR have the highest average prediction accuracy of 88% | SVM +DT+RF+LR | The authors compared 4 ML models considering 10-cross validation. |
| Hassan & Wang (2017) | Cross-validation scenario provided better results with an average F-Measure score of 92% compared to 87% achieved cross-projects. | RF | The idea is to split the data based on the build systems (Ant, Maven, and Gradle) and then perform cross-project and cross-validation based predictions. |

performs better than the two other methods. And among the used models, they found that DT classifier performs the best achieving a score of 17% for F1-score on average. Luo *et al.* (2017) have used the features of TravisTorrent dataset to predict the result of a build. Additionally, they compared Support Vector Machine (SVM), DT, RF and Logistic Regression (LR). Based on 10-fold cross-validation, the results reveal that LR and SVM were the best performers.

#### 2.3.2.2 Optimization of CI build time

Abdalkareem *et al.* (2019) are the first to examine the CI commits that can be CI skip and determine the reasons behind it. Additionally, they proposed a rule-based technique to automatically detect and label the commits to be skipped by using five rules related mainly to non-source files (e.g.documentation) and cosmetic changes (*e.g.,* source code formatting). Based on a corpus of ten open-source Java projects that use Travis CI, this technique has reached a moderate score of 58% in terms of F1-score. A related effort for improving CI skip detection, by Abdalkareem *et al.* (2020), proposed a ML-based technique to raise the issue related to the moderate performance of the rule-based approach. By conducting an empirical study based on the same dataset of (Abdalkareem *et al.*, 2019), the used DT model achieved higher F1-score of 79% within project validation and 55% under cross-project validation. Additionally, they investigated the most prominent features and found that the number of developers who changed the modified files and the terms used in the written commit messages are the best indicators of CI skip commits.

#### 2.3.2.3 Summary of research on CI builds

To sum up, there is a consensus that CI build failure is a major barrier to face when adopting CI. To this end, many research studies have been conducted to prevent the failure by predicting its outcome. Although these research efforts have advocated that predicting CI build failure is possible, these works achieved a limited prediction accuracy that is sometimes comparable to the performance of random guessing. Another main issue to classic ML-based approaches is related to the imbalanced distribution of build results. This challenges their applicability due to the performance bias that can occur when an imbalanced distribution of class examples is used.

Furthermore, this imbalanced nature of the training data was rarely discussed in existing works. However, in CI context, a good accuracy on the failed builds prediction is more important than the passed builds accuracy. The existence of these issues suggest that the build failure prediction problem is not yet resolved. In Chapters 5 and 6, we propose novel approaches to resolve the problem of CI build prediction.

The second issue related to CI builds is its long duration. Previous works attempted to mitigate the problem by proposing ML based approaches that detect the changes that can be skipped. But similarly to CI build outcome problem, the applicability of ML is limited. Hence, to address these limitations, we proposed in Chapter 7, a novel approach that improves the state-of-the-art approaches.

## 2.4 Conclusion

In this chapter, we first presented the background of our thesis and the main techniques used to implement our proposed tools. Then, we presented the research works that are related to this thesis and how we plan to address their limitations.

# CHAPTER 3

# AN EMPIRICAL STUDY ON CONTINUOUS INTEGRATION TRENDS, TOPICS AND CHALLENGES IN STACK OVERFLOW

Islem Saidani[a] , Ali Ouni[a]

[a] Department of Software Engineering and IT, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

Paper submitted for publication, March 2022

## Abstract

During the last few years, Continuous Integration (CI) has become a common practice in open source and industrial environments in order to reduce the scope for errors and increase the speed to market through automated build and test processes. However, despite this wide adoption throughout the years, little is known about the challenges developers discuss. Analyzing the discussions of developers is required to understand what researchers, educators and practitioners should focus on, and how discussion communities can be helpful to shed the light on CI challenges. In this study, we examine Stack Overflow (SO), the most popular crowd-sourced forum, to understand the challenges developers face under CI context. We collect a corpus of 27,728 CI related developers posts from SO and analyze those posts through a mixed-method with quantitative and qualitative analyzes. To study the trends of CI discussions, we investigated the metadata of CI questions, users and tags. Then, we extract the CI main topics using Latent Dirichlet Allocation (LDA) tuned with Genetic Algorithm (GA). Finally, we investigate the most popular and difficult topics faced by developers and perform a qualitative analysis based on a statistical sample of unanswered questions to get further insights into CI challenges. The LDA clustering reveals that developers face challenges with six main topics namely Build, Testing, Version Control, Configuration, Deployment and CI Culture. Particularly, we found that the build topic is the most popular among the studied topics and that version control and testing topics are the most difficult for SO community. Our study uncovers insights about CI challenges and adds evidence to existing knowledge about CI issues related especially to software build. Based on the results of our study, we conclude several implications for researchers, *e.g.,* need for more effort to investigate the reasons behind the reported issues, educators, *e.g.,* teach CI principals and philosophy, and practitioners, *e.g.,* take the difficult topics into consideration when distributing the tasks.

**Keywords.** Continuous Integration,Stack Overflow, Topic modeling, Latent Dirichlet Allocation, Empirical Study

## 3.1 Introduction

Continuous Integration (CI) is a set of modern software development practices that is widely adopted in industry and open-source environments (Vasilescu *et al.*, 2015). CI advocates to continuously integrate code changes, by automating the process of build and testing (Fowler, 2006), which reduces the cost and risk of delivering defective changes. Nevertheless, introducing changes under such context still be is risky and can lead to productivity loss (Bernardo *et al.*, 2018), release delays (Widder *et al.*, 2019) and cost impacts (Laukkanen *et al.*, 2015).

Prior studies that have examined CI challenges relied mainly on surveys/interviews of a selected number of stakeholders (Debbiche *et al.*, 2014; Hilton *et al.*, 2017; Pinto *et al.*, 2018). Although interviewing developers provides great insight, it has a major limitation: it cannot be generalized due to the limited number of the interviewed persons. Therefore, there is a need to investigate CI challenges on a large scale. At the same time, we observe that discussions related to CI are becoming increasingly prevalent in online developer forums, to find answers to their CI related issues. Stack Overflow (SO)[1] is one of the most popular Q&A sites for developers, by recording over 19 million questions in 2020 (Openja, Adams & Khomh, 2020). For instance, in one of the SO posts[2], a developer asked: 💬*"I try to setup CI for Go app and Jenkins.. So, my questions are (1) If my app will contain much more code file, packages etc. should I still build it as go*

[1] https://stackoverflow.com/

[2] https://stackoverflow.com/questions/34731416

*build main.go? (2) How correctly give name for go build output file to add it to the artifacts? (3) Should I use some kind of make file/script etc to collect dependencies on build machine? What is best practice here?".* This question received an answer only after over four years despite being viewed 1k times. Which may indicate that the issue faced by this developer is difficult to resolve.

We therefore believe that an analysis about CI discussions on SO can help the research community and CI stakeholders in better understanding developers' concerns and hence improving the adoption of CI. To this aim, we conduct in this paper a large-scale empirical study of 27,728 CI related posts on SO to address the following Research Questions (RQs):

- **RQ1: (Trends) How have CI discussions grown since the creation of Stack Overflow?** We aim to gain insights into the temporal trends of CI discussions. Specifically, we study the volume of questions (11,641) and their answers (16,087), the users responsible for creating such posts (17,992) and also the tags (2,806) associated with CI questions. Results show that developers frequently use SO to seek help with their CI problems. We also found that SO users are showing more interest in CI over the years and that most CI questions tags are around CI servers and platforms.
- **RQ2: (Topics) What topics are discussed around CI?** We leverage Latent Dirichlet Allocation (LDA) technique to identify the key topics that developers discuss on SO. Additionally, we use an advanced parameter tuning technique based on Genetic Algorithm (GA) to find the optimal parameters of LDA algorithm. Our tuned LDA reveals that CI discussions cover six main topics namely *Build, Testing, Version Control, Configuration, Deployment* and *CI Culture*. The primary driver behind these questions is enhance the usage of CI tools/infrastructures in the development process. Specifically, around 40% of discussions are related to build process suggesting this phase is key a concern within CI projects development.
- **RQ3: (Challenges) Which topics are the most popular and difficult among CI related questions?** We exploit the information provided in SO to discover the topics being the most popular and difficult to answer CI questions. In addition, we examine a significant simple of unanswered questions to gain further insights into the CI challenges. Our findings reveal that

build related questions are the most popular (in terms of view, favorite and score questions). Additionally, questions related to version control involve the highest rates of unanswered questions, while testing questions require the longest time to receive accepted answers. When examined the unanswered questions, we found that those questions usually receive responses in the form of comments where users either suggest solutions to address the problem or ask for more clarification.

The main contributions of this study can be summarized as follows:

- We conduct the first empirical study to mine and extract 27,728 SO posts related to CI to better understand the challenges, trends and topics around CI.
- We perform a mixed-method study, though quantitative and qualitative analyses, to shed light on characteristics of CI related topics and the usage of CI tools/infrastructures.
- We provide practical implications of our findings for researchers, developers, tool builders and educators.
- We publicly provide a replication package (Saidani, 2021) that contains the dataset and scripts in order to replicate our results.

The remainder of this paper is organized as follows. In Section 3.2, we describe our research methodology and reveal the main findings of our study. In Section 3.3, we discuss the implications of our work. Then, we review threats to validity in Section 3.4, and finally we address the conclusions to draw in Section 3.5.

## 3.2 Empirical Study Design

The main *goal* of this study is to obtain and share insights to CI stakeholders regarding how CI is discussed in practice by analyzing Stack Overflow (SO) posts. Figure 3.1 depicts an overview of our methodology. Our methodology comprises two main steps: (*1*) CI posts extraction and (*2*) analysis method. In the following, we present the details of each of these two steps.

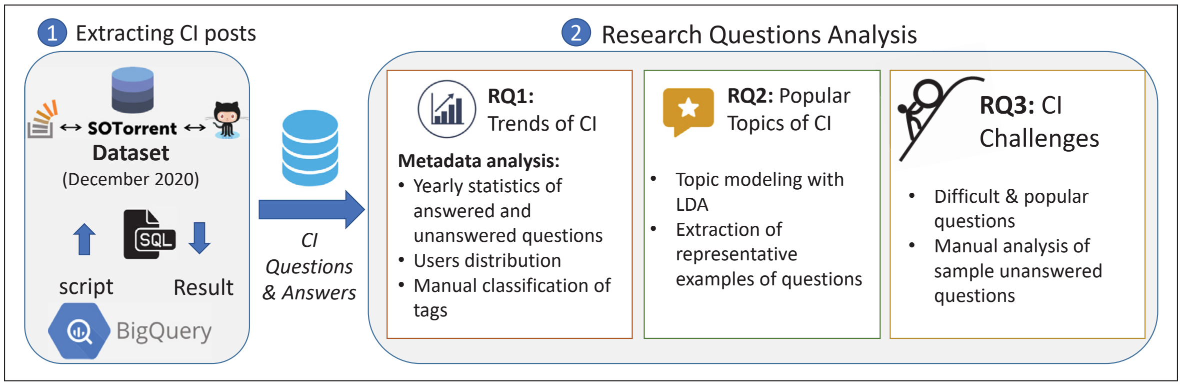

1 Extracting CI posts
SOTorrent Dataset
(December 2020)
SQL
script
Result
BigQuery
CI Questions & Answers
2 Research Questions Analysis
RQ1: Trends of CI
Metadata analysis:
• Yearly statistics of answered and unanswered questions
• Users distribution
• Manual classification of tags
RQ2: Popular Topics of CI
• Topic modeling with LDA
• Extraction of representative examples of questions
RQ3: CI Challenges
• Difficult & popular questions
• Manual analysis of sample unanswered questions

Figure 3.1 Overview of the our study methodology

### 3.2.1 Extracting CI posts

#### 3.2.1.1 SOTorrent Database

In this study, we mined CI questions & answers based on SOTorrent dataset (Baltes, Dumani, Treude & Diehl, 2018). We particularly utilized the latest dump, dated on December 2020, available on Google BigQuery [3] that allows to execute SQL queries on various public datasets. The dataset contains information about questions, answers and their metadata such as the creation date, score and view count. In the following we briefly describe the information used in our study:

- *Posts:* Among the different types of posts in the dataset, we only consider the questions and answers. For each question, we find the title and body fields. However, the answer type can only contain a body. Only one answer can be marked as “accepted" if the developer asking the question decides to.
- *OwnerUserId:* This field is used as an identifier of the post’s creator.
- *Tags:* When creating the question, the developer must select at least one tag and a maximum of five tags to describe that question.
- *View Count:* This metric counts the number of times the question was viewed.

[3] https://cloud.google.com/bigquery

- *Favorite Count:* We use this metric to count the number of times SO users have marked the question as a favorite.
- *Score:* This metric is based on the up-votes the post (question or answer) receives.

#### 3.2.1.2 CI posts

To extract relevant discussions on SO, we query questions from SOTorrent dataset being tagged with *“continuous integration"* or that contain this term in their title texts. Similarly to Peruma *et al.* (2022), we excluded searching for the term in the body of the post to avoid the increase of false positives in our dataset. Table 3.1 summarizes the collected data. As shown in the table, we extracted 11,641 CI related questions, from which 1,981 (17.1%) questions did not receive an answer, 5,382 (46.2%) had an accepted answer, while 4,278 (36.7%) questions received at least one, but not accepted, answer.

Table 3.1 Statistics about the collected data

| **Item** | **Count** |
|---|---:|
| Number of posts | 27,728 |
| Number of questions | 11,641 |
| Number of answered questions | 9,660 |
| Number of accepted answers | 5,382 |
| Number of distinct tags | 2,806 |
| Number of distinct users | 17,992 |
| Average number of tags per question | 4 |
| Average number of answers per question | 1.4 |

### 3.2.2 Research Questions Analysis & Results

In this paper, we address three Research Questions (RQs). As described in Section 3.1, RQ1 aims to examine the temporal trends and growth of CI posts, users as well as the tags used to describe the questions. In RQ2, we classify the CI discussions into topics based on Natural Language Processing (NLP) techniques. Then, in RQ3, we are based on results of RQ2 to identify the most popular and challenging topics being discussed among developers. In the next subsections, we explain the rationale behind each RQ and our approach to produce and analyze the results. then we present our main findings.

#### 3.2.2.1 RQ1. (Trends) How have CI discussions grown since the creation of SO?

**Motivation.** In this RQ, we are particularly interested to study the growth of CI discussions over the years. This preliminary investigation helps us understand the extent to which developers seek help on CI problems.

**Approach.** First, we explore the volume of CI posts (*i.e.,* questions and answers) Then, we want to analyze the developers' involvement in CI discussions and whether a subset of SO users are responsible for the majority of CI related questions and answers. Then, in order to determine the concepts and technologies being associated with CI questions, we manually review the tags and classify them in order to determine the different categories these tags fall under.

**Results.**

*Trends of Questions.* Figure 3.2 shows the yearly growth of CI questions with and without an accepted answer. In this figure, the red bars represent the questions without an accepted answer, while the blue bars are questions with an accepted answer. The total number of questions is represented by the grey bars. Similarly to Peruma *et al.* (2022), we eliminate SO data of the years 2008 (SO was launched this year) and 2020 as they contain incomplete information.

A first look at these bar-plots shows that, as the years pass, developers are showing more interest in CI since the number of the total questions increases each year. The only exception is for the year 2014 in which the number of questions decreased by 36 as compared to 2013. Regarding questions with an accepted answer, except for the years 2013, 2014 and 2018, we found that the related number of posts is also growing; while the number of questions without accepted answers is always increasing. Additionally, from the year 2014, the number of questions without accepted answers outnumber questions with an accepted answer.

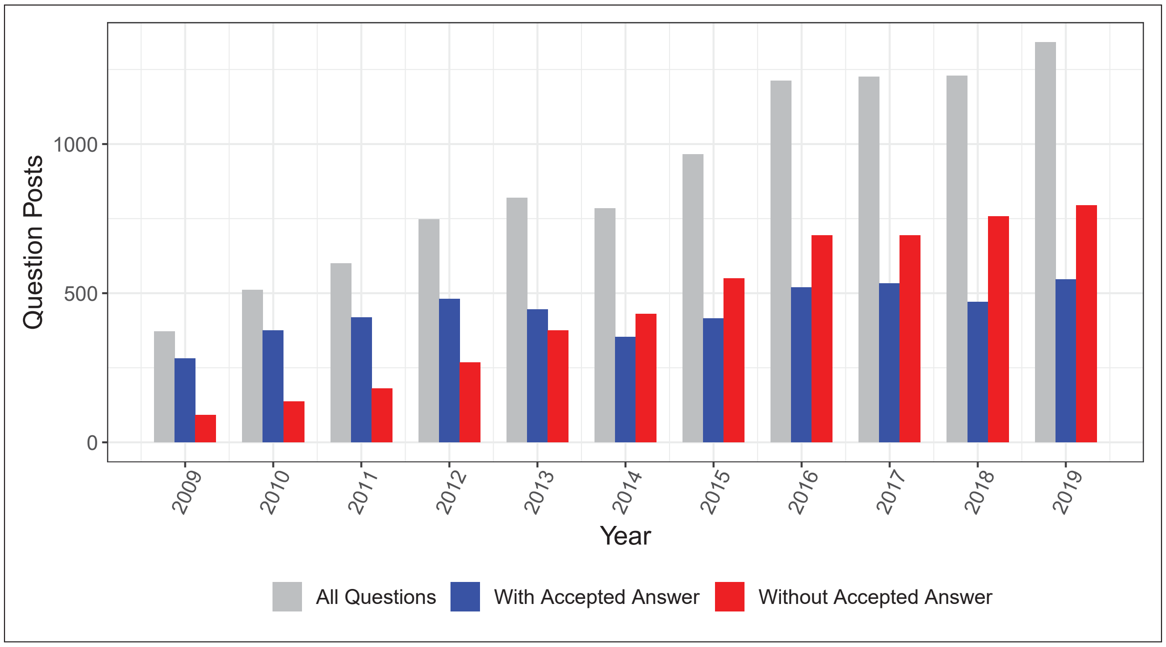


Figure 3.2 Count of CI questions and their answers per year

*Trends of Users.* Next, we look at the involvement of the SO community members in CI discussions. We investigate unique users who post questions and answers and analyze their trends over the years. Figure 3.3 reveals an increasing trend in the number of unique users involved in CI discussions, except for the year 2014 (similarly to the number of CI questions shown in Figure 3.2). Hence, we believe that the dip in questions and users in 2014 is an interesting phenomenon that would require further investigation to explain the reasons behind it.

Next, we investigate the distribution of unique users for both questions and answers. As shown in Table 3.2, we see that 9,566 distinct users created 11,641 CI related questions (Table 3.1). As for answers, we observe that 4,525 distinct users posted accepted answers, while 5,589 distinct users are responsible for non-accepted answers. We also see that 87.36% of the users asked at most one question; which is applied to accepted and non-accepted answers. Indeed, 88% and 92% of the users were associated with only one accepted and non-accepted answer respectively.

*Trends of Tags* As a further step to study the trends of CI discussions, we investigate the categories under which fall the tags associated with CI-related questions. In total, our dataset contains

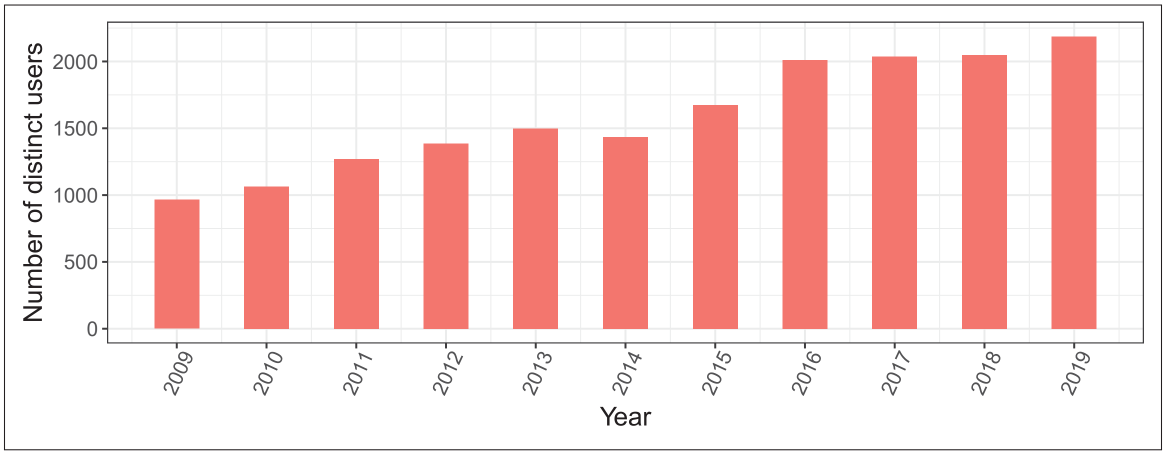


Figure 3.3 Count of distinct users involved in CI discussions per year

2,806 distinct tags excluding *"continuous integration'"* tag. The top-10 tags are all related to servers (jenkins 7.9%, teamcity 2.3%, azure-devops 2.1%, hudson 1.6%), platforms (docker 1.7%, gitlab 2.3%, github 1.4%), tools (git 2.7%), programming languages (Java 1.5%) and concepts (continuous-deployment 3.12%). These popular tags represent 27.02% of the tags in the dataset as shown in Table 3.3.

Next, we plot the yearly growth of these popular tags as shown in Figure 3.4. Recall that we ignore the years 2008 and 2020 due to their incomplete data. Regarding CI servers, we observe that, except for "azure-devops", all the related posts are decreasing from a particular year. For instance, we see that the volume of "jenkins" tag shrinks from the year 2017 while the number of "hudson" tagged posts show a steep decline from 2012. Additionally, the popularity of posts related to "github", "gitlab", "docker" and "continuous-deployment" seems to be in constant increase over the years. We also see that the curve of "git" and "java" tags are relatively constant.

Finally, we manually classify the tags to determine the most popular technologies/concepts used in CI discussions. To this aim, we manually reviewed a statistically significant sample of tags composed of 538 tags which represents a confidence level and interval of 99% and 5% respectively. As a result, we clustered the tags into seven categories namely *Platforms/Servers*, *Tools/IDEs*, *Programming Languages*, *Framework/Library/API*, *Concepts*, *Operating Systems*,

Table 3.2 Distribution of the number of posts created by a user

| **Number of posts created by a user** | **count** | **Percentage** |
|---:|---:|---:|
| *Questions -total distinct users* | *9,566* | |
| 1 | 8,357 | 87.36% |
| 2 | 844 | 8.82% |
| 3 | 195 | 2.04% |
| 4 | 94 | 0.98% |
| 5 | 32 | 0.33% |
| Others | 44 | 0.46% |
| *Accepted Answers - total distinct user:* | *4,525* | |
| 1 | 3,996 | 88.31% |
| 2 | 376 | 8.31% |
| 3 | 93 | 2.06% |
| 4 | 35 | 0.77% |
| 5 | 10 | 0.22% |
| Others | 15 | 0.33% |
| *Non-Accepted Answers - total distinct user:* | *5,589* | |
| 1 | 5,154 | 92.22% |
| 2 | 336 | 6.01% |
| 3 | 64 | 1.15% |
| 4 | 15 | 0.27% |
| 5 | 8 | 0.14% |
| Others | 12 | 0.21% |

and Other. Figure 3.5 shows the distribution of the instances associated with each tag category. Most of the tags (40.15%) are related to platforms and servers which include CI servers (*e.g.,* Jenkins [4], Teamcity [5]) and platform products such as Docker [6] and Amazon Web Services [7] The next most popular category is related to general concepts including CI principals (*e.g.,* continuous deployment, continuous delivery and build automation), repository (*e.g.,* pull request, versioning) and programming (*e.g.,* variables, command-line). The third most popular tags fall under the Tools/IDEs category (17.85%). In this category, we found that tools are related

[4] https://www.jenkins.io/

[5] https://www.jetbrains.com/teamcity/

[6] https://www.docker.com/

[7] https://aws.amazon.com/

Table 3.3 Top-10 most popular tags

| Tag | Occurrence | Percentage |
|---|---:|---:|
| *jenkins* | 2,647 | 7.98% |
| *continuous-deployment* | 1,035 | 3.12% |
| *git* | 899 | 2.71% |
| *teamcity* | 771 | 2.32% |
| *gitlab* | 770 | 2.32% |
| *azure-devops* | 697 | 2.10% |
| *docker* | 589 | 1.77% |
| *hudson* | 534 | 1.61% |
| *java* | 528 | 1.59% |
| *github* | 490 | 1.48% |
| Others | 24,210 | 72.98% |

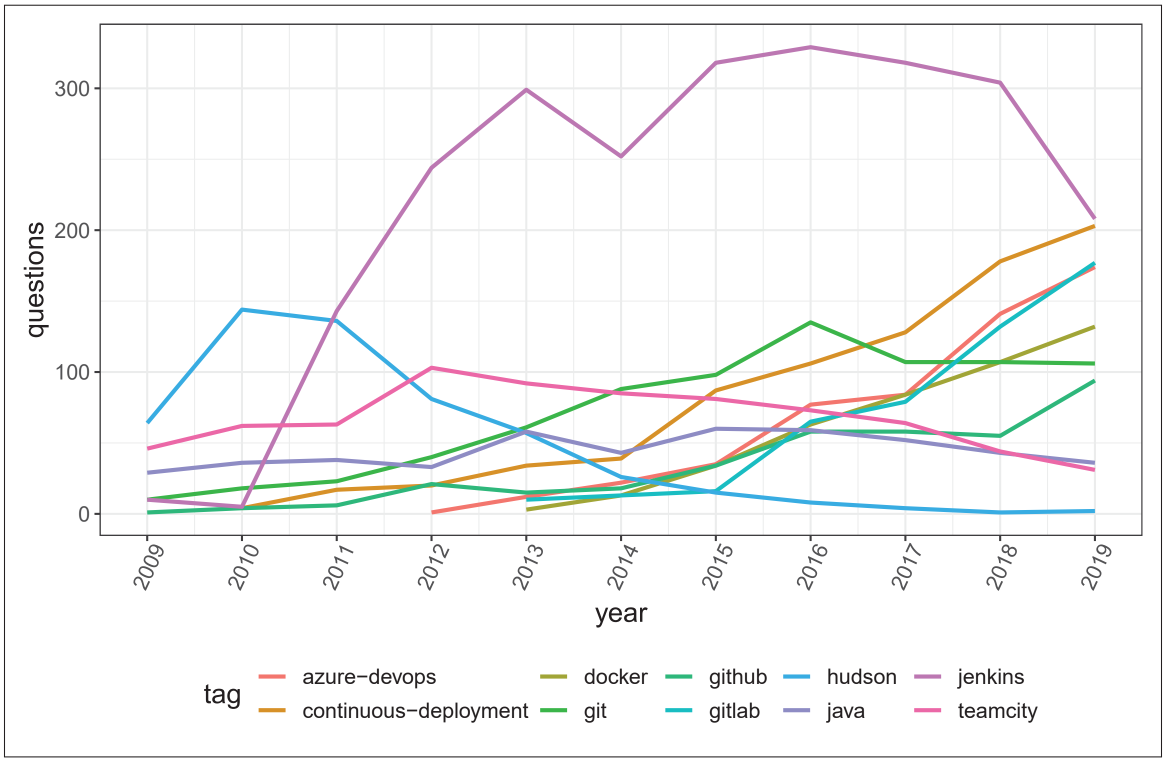


Figure 3.4 Yearly growth of the popular tags associated with CI posts

to version control (*e.g.,* Git [8]), build (*e.g.,* Maven [9], MSbuild [10]), testing (*e.g.,* Selenium [11],

[8] https://git-scm.com/

[9] https://maven.apache.org/

NUnit [12]), configuration (*e.g.,* chef-recipe [13]) and code analysis (*e.g.,* FindBugs [14] and Phpcs [15]). The tagged posts include popular IDEs such as Xcode [16] and Visual Studio [17] Tags related to programming languages represent only 8.5% of the tags in our sample set. In this category, we found that the top-3 most popular languages are Java, python and C#. The category of frameworks/libraries/APIs which represents 5.7% of the tags include popular frameworks such as Node.js [18], Ruby on Rails [19] and Angular [20] With regards to Operating Systems (OS) category, we found that the popular tags are mobile-based (*e.g.,* Android and iOS).

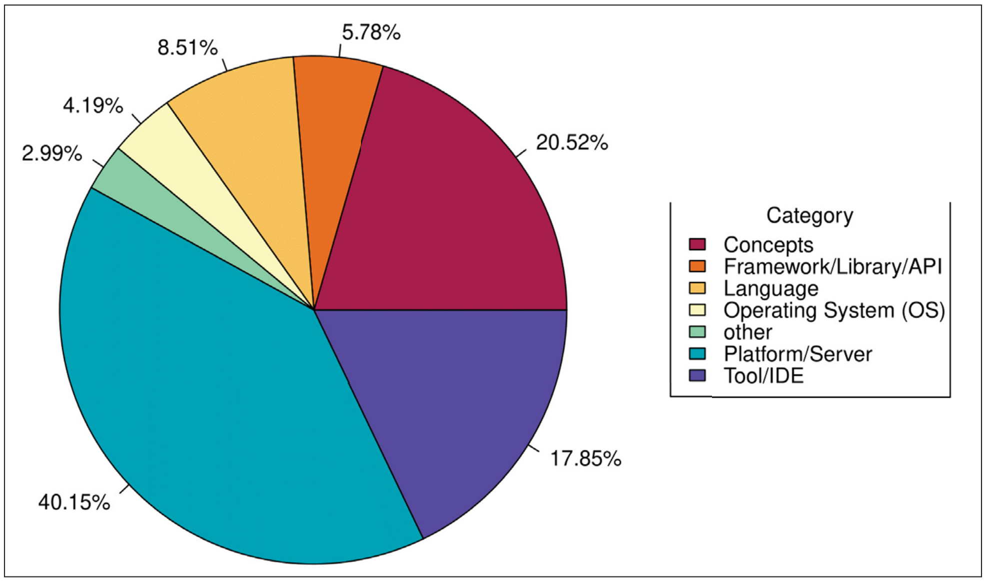


Figure 3.5 Distribution of the tags' categories

[10] https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild?view=vs-2022

[11] https://www.selenium.dev/

[12] https://nunit.org/

[13] https://www.chef.io/products/chef-infra

[14] http://findbugs.sourceforge.net/

[15] https://github.com/squizlabs/PHP_CodeSniffer

[16] https://developer.apple.com/xcode/

[17] https://visualstudio.microsoft.com/fr/

[18] https://nodejs.org/en/

[19] https://rubyonrails.org/

[20] https://angular.io/

**Summary for RQ1.** The findings of RQ1 reveal that SO is widely used by developers to seek help with CI related problems. We found that the CI community is showing more interest in CI over the years by posting over 1,000 questions in 2019. Additionally, we see a rise in questions around *Jenkins* which indicates a need for support for this CI service. Finally, our analysis of the questions tags reveals that 40% of posted questions are related to servers and platforms followed by tags associated with general concepts (20%).

#### 3.2.2.2 RQ2. (Topics) What topics are discussed around CI?

**Motivation.** We aim in RQ2 to identify the main issues faced by developers when adopting CI. This can be helpful for tool builders in the design of their tools documentation and development, for CI research community to gain focus their research efforts on the most interesting improvement areas and also for educators to pay particular attention to the most challenging topics.

**Approach.** In order to explore the high level issues facing CI developers, we use Latent Dirichlet Allocation (LDA) technique (Blei, Ng & Jordan, 2003) to aggregate and discover what is being asked in the CI posts. LDA has been recognized as one of the best unsupervised Machine Learning (ML) techniques that has shown its effectiveness in clustering a large volume of text documents (Yang, Lo, Xia, Wan & Sun, 2016; Openja *et al.*, 2020). In order to cluster CI posts, we build a corpus in which each row is composed of a question's title and body. Additionally, it is necessary to preprocess the text in order to filter out irrelevant information. We start by removing all the code snippets (*i.e.,* enclosed in $< code >$ tag), HTML tags (such as *<p>* and *</p>*) and URLs from the corpus. Then, we proceed with the removal of numbers and punctuation marks as they add a little value to the text relevance. We also remove English stop words and add an extra set of stop words composed of the frequently occurring words such as “question", “answer”, etc. The full list of removed words is available in our replication package (Saidani, 2021). Finally, we apply lemmatization of words to convert the word to its meaningful base form (which is called Lemma) similarly to previous studies (Peruma *et al.*, 2022; Abdellatif, Costa, Badran, Abdalkareem & Shihab, 2020). For example, “development" is mapped to “develop".

To obtain optimal results with LDA, it is necessary to tune its parameters (Peruma *et al.*, 2022; Yang *et al.*, 2016). The first required input for LDA, is the estimated number of *topics* to be generated. Selecting a low value can result in general topics while a high value would produce a long list of detailed topics that could contain noise. Moreover, LDA depends highly on the number of *iterations* that define when the train reaches its end. Hence, we tune the maximum number of iterations through the corpus when inferring the topic distributions. The same uncertainty about the amount of these parameters also exists for the *chunk size* and *passes* as they affect how well/poorly the model can perform. While the chunk size defines the umber of documents to be used at one in each training cycle, the number of passes defines how many times the algorithm is supposed to pass over the whole corpus. Hence, it is important to apply the parameters' tuning (Tantithamthavorn, McIntosh, Hassan & Matsumoto, 2018a) for LDA. One the other hand, finding the suitable LDA configuration can be seen as a combinatorial problem where the selection is made from a very large space of choices. Therefore, we use an advanced tuning technique, Genetic Algorithm (GA), to effectively explore this large search space and find the optimal set of parameter values of LDA. GA is a widely used computational search technique, that has proven good performance in solving many software engineering problems (Saidani, Ouni & Mkaouer, 2022; Mkaouer *et al.*, 2015; Ouni *et al.*, 2016).

In our adaptation of GA, we selected four parameters to be optimized presented in Table 3.4 along with their values ranges. Additionally, we compute and evaluate the LDA models performance using the *Topic Coherence* metric (Röder, Both & Hinneburg, 2015) which measures how similar are the topic words are to each other.

Table 3.4 Configuration space for the parameters of LDA

| **Parameters** | **Search Space** |
|---|---|
| Number of topics | range [2,50] |
| Number of iterations | range [10,5000] |
| Chunk size | range [10,2000] |
| Passes | range [1,100] |

Finally, since LDA does provide meaningful names for its generated topics, we manually label the topic names. To implement LDA algorithm, we use the python package *Gensim* [21]

**Results.**

As a result of tuning our LDA model using GA, we found that the (near) optimal configuration consists of fixing the passes, iterations and chunk size to 200, 2,000 and 500 respectively as depicted in Table 3.5. Additionally, the LDA clustering yields six main topics associated with SO CI questions, that represent the main stages of CI/CD pipeline, namely *Build*, *Testing*, *Version Control*, *Configuration*, *Deployment* and *CI Culture* as represented in Table 3.6. For each topic, we show a partial set of the associated words (unigrams and bigrams).

Table 3.5 Best Configuration of LDA as revealed by GA

| **Parameters** | **Optimal Value** |
|---|---|
| Number of topics | 6 |
| Number of iterations | 2,000 |
| Chunk size | 500 |
| Passes | 200 |

Table 3.6 LDA topics and part of their corresponding words

| **Topic** | **Related Key Words (unigrams & bigrams)** |
|---|---|
| *Build* | build, job, error, server, branch, fail, jenkins, teamcity, msbuild, time |
| *Testing* | test, run, test-case, code, file, xcode, script, integration-test, unit-test, selenium |
| *Deployment* | deploy, server, environment, pipeline, jenkins, use, application, web, version, azure-devops |
| *Configuration* | configur, file, setup, error, gitlab-ci, install, yml, resource, jenkins, name |
| *Version Control* | branch, git, gitlab, github, repository, master, change, push, commit, pull-request |
| *CI Culture* | want, need, know, use, way, favorite, important, best, tool company, different |

[21] https://radimrehurek.com/gensim/

Next, to determine the distribution of each topic, we assign for each question in our dataset, the most dominant such as only topic can be selected.

As shown in Figure 3.6, we clearly see that, Build topics are the most frequently occurring topics with 39.5% (4,601 questions). In the following, we describe in detail each topic and include some representative examples of associated questions. It should be noted that our analysis of the sub-topics is based on the frequent terms in each topic as well as our manual investigation.

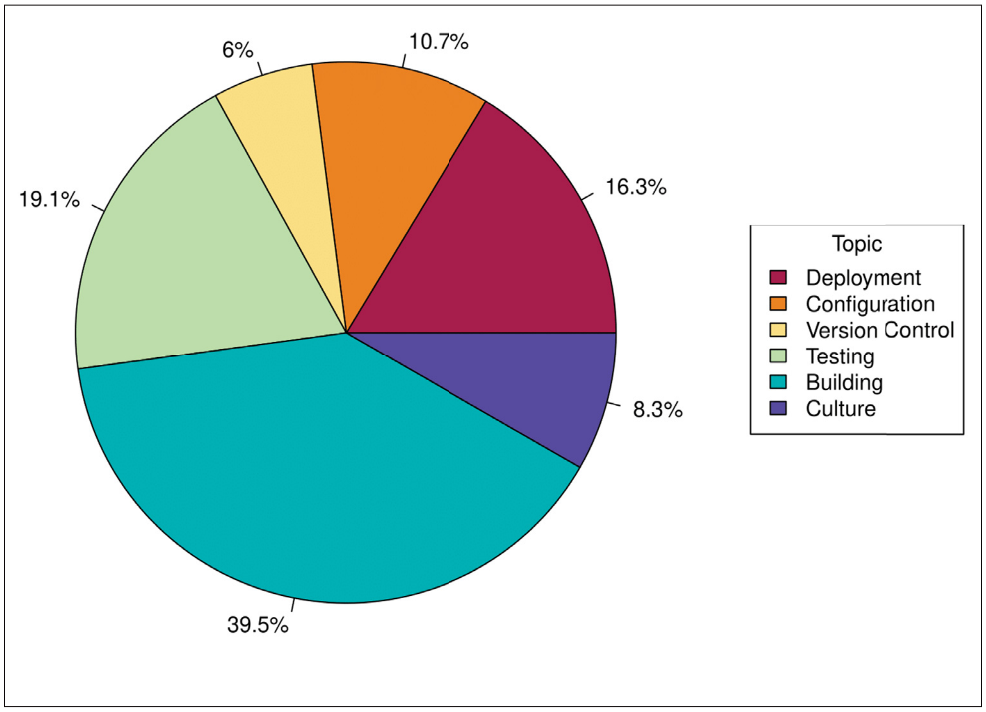


Figure 3.6 Breakdown of the frequency of occurrence for each CI topic

***1. Build:*** As stated by previous research works (Saidani, Ouni, Chouchen & Mkaouer, 2020a; Saidani *et al.*, 2022; Hilton *et al.*, 2017), software build represents a major barrier that developers face when adopting CI. To this aim, developers turn to SO for assistance with build issues. In fact, we encounter in the list of associated words, terms like “issue" and “fail" highlighting that developers seek help mainly with fixing the broken builds (*e.g.,* Quote 1). This suggests that developers struggle to obtain working solutions for build resolution.

**How do I fix this incorrect CI build failure?**

💬*““.. When the two dependencies change and all projects rebuild we get incorrect failures like the one above. How can we stop from getting these failures?"*

Quote 1: An example of question where the user seeks help with fixing the build failure [22]

Another common challenge is related to the build performance since the word “time" appeared in the top-10 words as shown in Table 3.6. As stated in previous works (Ghaleb, da Costa & Zou, 2019a; Saidani, Ouni & Mkaouer, 2021c), the builds can take hours and even days in CI context. This affects both the speed and cost of software development (Luo *et al.*, 2017) as well as the productivity of developers who seek help to speed up CI build as shown in Quote 2.

**Build in VSO takes a long time**

💬*“.. However, the execution of the builds takes forever. I have created a small test solution (one class with one property) and a test project (with a single test, using NUnit), and the build takes more than 20 minutes to complete. Is there any way we can speed things up in VSO"*

Quote 2: A sample question highlighting the need to speed up CI build [23]

Servers/infrastructures are an essential for adopting CI as they allow the automation of build process. The presence of the terms “jenkins", “teamcity", indicate that these CI servers may not be easily utilized by stakeholders. For instance, the user in Quote 3, is asking how to speed-up the build with Jenkins. This question, despite being viewed 7k times, did not receive a right answer after 8 years of being published.

[22] https://stackoverflow.com/questions/7837902

[23] https://stackoverflow.com/questions/29848548

**Jenkins - How to run post-build action without re-running the job?**

💬*“I have a lengthy Jenkins job with a failing post-build action. How can I repeatedly run the post-build action without re-running the whole job?"*

Quote 3: Sample question related to Jenkins[24]

Build systems were developed to automate the code compilation and they are an essential part of CI process. Hence, more effort is needed to improve build systems. In particular, according to Table 3.6, more attention should be given to MsBuild, the Microsoft build tool, that contains difficult-to-understand features which leads to commonly occurring errors (Hashimi & Hashimi, 2006). For instance, in Quote 4, the developer is seeking help to solve an MsBuild error. This question is among the most popular questions in our dataset by reaching 73k views. This finding aligns with Openja *et al.* (2020) results.

**build .NET application in Jenkins using MSBuild**

💬*“.. I’ve added MSBuild plugin ..But my build processes are failing by showing the below error message"*

Quote 4: Sample question related to MSBuild[25]

***2. Testing:*** Figure 3.6 shows that Testing is the second most dominating topic representing 19% of CI questions that developers ask in SO. This topic deals mainly with questions that actively discuss the basics of testing. Indeed, when observing the frequent terms in this topic, we encounter terms related to unit testing, test cases and integration testing. For example, in Quote 5, the user wants to know how to run integration tests. The accepted answer suggested to *“run integration tests automatically on TeamCity agent after main build is completed"*.

[24] https://stackoverflow.com/questions/15191539

[25] https://stackoverflow.com/questions/10227967

**How to run integration tests?**

💬*“..For plain unit tests, we use TeamCity for continuous integration. How do you run the integration unit tests and when do you run them?"*

Quote 5: Sample question related to integration tests[26]

Some questions that may be interesting to investigate are questions related to mobile testing. Specifically, we found that *Xcode*, an Integrated Development Environment (IDE) for mac-OS, is a frequent term in testing topic. Indeed, mobile apps undergo frequent updates to introduce new features, fix reported issues or adapt to new technological or environment changes. Hence, introducing changes in this context is risky and can harmfully affect the application rating and competitiveness. Thus, ensuring that the changes can by safely integrated (*i.e.,* by testing them) is of crucial importance. As a result of our analysis, we found that developers search information about how to solve their problems related to testing or setting a mobile test environment as shown in the question of Quote 6.

**Travis CI Android Tests: no connected devices**

💬*“I am trying to set up Travis for Android. Running the build seems to work so far, but when it comes to the tests, it complains about no connected devices!."*

Quote 6: Sample question related to mobile testing[27]

In addition to mobile apps, developers seem to be interested to the continuous testing of Web applications. Specifically, we found that *Selenium*, a set of tools and libraries for supporting Web browser automation, is a frequent term in this topic. In Quote 7, we highlight an example of a question about running Selenium tests with CI:

[26] https://stackoverflow.com/questions/2552506

[27] https://stackoverflow.com/questions/31264136

**How to run Selenium tests with CI (Continuous Integration)?**

💬*"..I'm using Selenium for automated testing my websites. I have around 100 test cases and I want to run them every day by making Test Suite automatically.. "*

Quote 7: Sample question to testing Web applications with Selenium[28]

***3. Deployment:*** Continuous Deployment (CD) is an essential practice consisting of automatically deploying every change into the production environment (Shahin *et al.*, 2017). We found that this topic covers 16% of discussions about CI. When analyzing the related questions, we found that more than 70% of the discussions on this topic include *"how to"* suggesting that generally, users are seeking help with performing specific tasks about deployment. This result aligns with the findings of Openja *et al.* (2020) who find that the majority of *"how"* questions are related to Deployment topic.

A manual investigation of a significant sample of questions reveals that users enquire about the deployment pipelines and their associated tools. Indeed, the success of implementing CI/CD practices depends heavily on the appropriate selection of tools/infrastructures (Shahin *et al.*, 2017). By observing the frequent terms in this topic, we encounter "jenkins" and "azure-devops", two popular CI/CD servers. While we found 357 (or 18% of deployment questions) Jenkins tagged questions, Azure (previously known as Team Foundation Server (TFS)) tagged questions represent 16.7%. Quotes 8 and 9 show the most popular questions about these two tools (in terms of views count).

**JENKINS how to deploy artifacts to maven repo?**

💬*"I use Jenkins 1.500 and I looking for plugin that will provide possibility to deploy artifacts to maven repository, in previous version of jenkins it was possible in post build actions using maven-plugin but for now that option disappear."*

Quote 8: Sample question asking about deploying artifacts with Jenkins[29]

[28] https://stackoverflow.com/questions/5627070

[29] https://stackoverflow.com/questions/14694696

**How to set Azure pipeline variable from PowerShell**

💬*“..am trying to set the Azure pipeline variable value in PowerShell. I have created one variable winversion in the Azure pipeline. Now, in a PowerShell task, I want to assign some values to the winversion variable. My simple question is how can I change the value of an Azure PipeLine variable at run time?"*

Quote 9: Sample question asking how to setup Azure pipeline[30]

Web deployment is a relevant sub-topic in Deployment since the terms “web" and “application" appear the frequent LDA terms in this topic. When analyzing the deployment questions, we found that 17% of discussions encounter the term “web". An example of Web deployment question is presented in Quote 10.

**Is there any stable tool for complete Web deployment & CI**

💬*“I’ve spent a plenty of hours trying to find a full stable solution for an application deployment (in my case it’s php). There are a lot of SO answers, where phing / capistrano / hudson are being proposed, but such propositions make me feel sad."*

Quote 10: Sample question about Web deployment[31]

***4. Configuration:*** Configuration management is an essential part of CI/CD pipeline and discussions on this topic cover around 10% of CI questions in our dataset. By observing the frequent terms in this topic, we realize that most of the concerns about configuration is related to the appropriate setup of infrastructures/servers. Indeed, we encounter the terms “jenkins" and “gitlab-ci". By analyzing the questions in this topic, we found many questions related to configuration issues with Jenkins. For instance, in Quote 11, the user is asking how to configure Jenkins to run on port 80. This question despite being viewed 77k times, did not receive any accepted answer.

[30] https://stackoverflow.com/questions/55472792

[31] https://stackoverflow.com/questions/35995990

**How to configure Jenkins to run on port 80**

💬*“..Is this because jenkins is running as the jenkins user on a privileged port? If so, how do I fix this? Any other ideas a welcome"*

Quote 11: Sample question about the configuration of Jenkins[32]

Gitlab CI is a popular service that allows build up CI/CD pipelines. According to our analysis, it seems that configuring the Gitlab CI tool is not trivial. For example, in Quote 12, the developer is facing an issue with the configuration file (*i.e.,* gitlab-ci.yml) of Gitlab CI that does not execute the scripts of build, deployment and testing. In the accepted answer, it was mentioned that the developer should use local paths in order to recognize the directory of the scripts.

**gitlab-ci.yml not executing shell script**

💬*“I set up gitlab-ci for my project, and inserted the following yml script..Do I have the wrong setup? Obviously its not the syntax and the permissions are allright, otherwise i'd get an error.What could this be?"*

Quote 12: Sample question about the configuration of Gitlab-CI[33]

***5. CI Culture:*** To take full advantage of CI, a set of guiding principles should be applied. In this topic that represents 8% of discussions in our dataset, developers discuss how to properly embed CI in their companies/projects. For instance, the developer in Quote 13 seeks to gain deeper understanding of the importance of using CI. The accepted answer of this question includes: 💬*“Using CI is a useful skill to have, but you want to avoid developing any bad habits that wouldn't translate to a team environment"*.

[32] https://stackoverflow.com/questions/9330367

[33] https://stackoverflow.com/questions/33807387

**Is Continuous Integration important for a solo developer?**

💬*“h’ve never used CI tools before, but from what I’ve read, I’m not sure it would provide any benefit to a solo developer that isn’t writing code every day. First - what benefits does CI provide to any project? Second - who should use CI? Does it benefit all developers?"*

Quote 13: Sample question for CI culture topic aims to understand how CI is important[34]

This topic includes also discussions in which developers establish comparisons about different tools of CI. For example, in Quote 14, the developers needs to choose between two popular CI servers namely Jenkins and Travis CI. This question has been viewed 132k times.

**Jenkins vs Travis-CI. Which one would you use for a Open Source project?**

💬*“For my project I need to choose between Jenkins and Travis-CI. I’ve been using Jenkins for years but I’ve also read good reviews about Travis-CI. Which one would you use for an Open Source project? What are the main benefits or advantages of both?"*

Quote 14: Sample question for CI culture topic aims to compare between Jenkins and Travis CI[35]

***6. Version Control:*** A core practice of CI is that all developers commit to the mainline (or master) branch daily. This topic is related to challenges in setting up repository branches and maintaining their synchronization and covers around 6% of questions in our dataset. By observing the topic related words, we found “gitlab" and “github" are among the most used terms which indicates that developers are facing challenges when using these two version control systems. For example, in Quote 15, the user wants to find out a way to force other developer to mention the issue in the commit message on GitHub that seems to be a missed feature in GitHub. Additionally, the question did not receive any accepted answer after 9 years.

[34] https://stackoverflow.com/questions/130592

[35] https://stackoverflow.com/questions/32422264

**How to avoid developers to commit without mention the issue on commit message on Github**

💬*".. Our project is currently hosted on GitHub, and we have a well configured Jenkins CI Server too. The doubt is: "how we can force our developers to mention a issue before commit?"*

Quote 15: Sample question for Version Control topic[36]

**Summary for RQ2.** the LDA clustering results in six topics including Build, Testing, Version Control, Configuration, Deployment and CI Culture. Specifically, around 40% of discussions are related to build process suggesting this phase is a key concern within CI projects development.

#### 3.2.2.3 RQ3. (Challenges) Which topics are the most popular and difficult among CI related questions?

**Motivation.** In the last RQ, we aim to provide further insights by investigating the most popular topics that attract CI community the most.

**Approach.** Similarly to previous studies (AlOmar *et al.*, 2020; Peruma *et al.*, 2022; Openja *et al.*, 2020), we consider the (1) view count, (2) favorite count, and (3) score of CI questions as metrics to measure the topic popularity. The high scores obtained for these metrics, the most popular is the CI topic. In addition, we aim to identify the most difficult/challenging for developers. Particularly, we look at questions that do not have any (accepted) answers as well as the median time the community takes to provide an acceptable answer to a question. Finally, we perform a qualitative analysis by examining a significant sample of unanswered questions (*i.e.,* questions without an accepted and non-accepted answer post). This sample include 498 questions and represents a confidence level of 99% and an interval of 5% for each topic, from a total of 1,981 unanswered questions.

[36] https://stackoverflow.com/questions/13704498

**Results.** Table 3.7 shows the results of our CI topics' popularity and difficulty.

*Popularity*. Looking at the table, we clearly see that *Build* topic is the most the most popular in terms of favorite, view and score counts while *Configuration* topic is the least popular. This finding confirms that build issues are the main concerns of CI developers who turn to SO community to seek for help. Next, we compare the popularity of CI questions against the other discussions on SO that are not part of our dataset. As a result, we found that CI questions have a higher average of scores (2.94) compared to the average score of non-CI questions (2.1). However, the views and favorites of non-CI questions are higher by reaching average values of 2472.2 (compared to 1996.5) and 2.7 (compared to 0.89) respectively. This suggests that CI discussions are not among the popular discussions of SO.

*Difficulty.* When it comes to the questions' difficulty, we found that *Version Control* discussions have the highest rate of unanswered questions among the six topics by reaching 21% and 59% of the questions having no answers or any accepted answer respectively. With regards to time needed to receive an accepted answer, we found that *Testing* questions take over 18 hours in median to receive an accepted answer. These results suggest that these two topics are most challenging for CI developers to answer. At the same time, questions related *CI Culture* seem to be the least challenging by achieving the lowest average percentage of questions without answers (17%) and of median hours to receive an accepted answer (only 5 hours). This may be explained by the fact that usually this type of questions are non-technical (*e.g.,* comparison between two CI servers) and not specific to the developer's project, which make it easier for developers to answer. Overall, we see that CI questions receive accepted answers within a short period of time (11 hours) and that only 17% of the questions remain unanswered. In the following, we investigate some of these unanswered questions.

*Unanswered questions* As a final step of our analysis, we examine the unanswered questions (*i.e.,* questions without any accepted or non-accepted answer). To this aim, we manually reviewed a statistically significant sample of unanswered from each topic with a confidence level and interval of 99% and 5% respectively. Specifically, we examined these unanswered questions for

Table 3.7 Popularity and difficulty of CI topics

| **Topic** | **Popularity Metrics** | | | **Difficulty Metrics** | | | | |
|---|---|---|---|---|---|---|---|---|
| | **Average Counts** | | | **Questions without answers** | | **Questions without accepted answer** | | **Median hours to an accepted answer** |
| | **Favorite** | **View** | **Score** | **count** | **Percentage** | **count** | **Percentage** | |
| *Culture* | 0.92 | 2082.8 | 3.23 | 164 | 17% | 536 | 55% | 5.0 |
| *Version Control* | 0.88 | 2045.9 | 3.04 | 148 | **21%** | 416 | **59%** | 5.0 |
| *Deployment* | 0.86 | 1380.5 | 2.16 | 349 | 18% | 1102 | 58% | 12.5 |
| *Testing* | 0.94 | 1642.0 | 3.00 | 411 | 18% | 1235 | 56% | **18.0** |
| *Build* | **0.95** | **2488.8** | **3.28** | 669 | 15% | 2295 | 50% | 10.0 |
| *Configuration* | 0.61 | 1654.6 | 2.45 | 240 | 19% | 675 | 54% | 10.0 |
| *Average for all CI questions* | *0.89* | *1996.5* | *2.94* | *1981* | *17%* | *6259* | *54%* | *11* |

incompleteness (*e.g.,* lack of source code, concrete examples, etc.) and ambiguity (unclear or short question) to identify the reasons behind the poor interaction of users with these questions.

The result of our examination reveals that about 60% of these questions were not completely ignored by users but rather received a least one comment (up to 10 comments per question). We found that these comments are either suggesting some solutions/answers to address the questions (*e.g.,* QuestionID = 45439753 [37]) or requesting for more clarification (*e.g.,* in QuestionID = 55527078 [38]). In other comments, the users mention that is no solution for the problem (*e.g.,* QuestionID = 8041785 [39]). Moreover, since our dataset covers the questions until March 2020, we found some questions being already answered after this date (*e.g.,* QuestionID = 38680366 [40]). Finally, we found rare cases where the unanswered questions are not related to CI which means that in these questions, the developers missuses the “continuous integration" tag in the question (*e.g.,* QuestionID = 40231075 [41]). Hence, it seems that most of these questions are not actually ignored by the CI community.

By manually reviewing the rest of the unanswered questions, we found that, for Build topic, the users are seeking help to resolve builds issues related to Jenkins (*e.g.,* QuestionID = 2426220 [42]). With regards to Testing topic, we found that the questions are often focused on testing mobile apps using Xcode IDE that adopt CI/CD principals (*e.g.,* QuestionID = 24678804 [43]). Same observation is made for questions about Version Control. Indeed, in most of unanswered questions, developers are asking about nonexistent versioning features (*e.g.,* QuestionID = 55952407 [44]). Considering the remaining topics, we found that users are asking about better

[37] https://stackoverflow.com/questions/45439753

[38] https://stackoverflow.com/questions/55527078

[39] https://stackoverflow.com/questions/28741142

[40] https://stackoverflow.com/questions/38680366

[41] https://stackoverflow.com/questions/40231075

[42] https://stackoverflow.com/questions/17156569

[43] https://stackoverflow.com/questions/24678804

[44] https://stackoverflow.com/questions/55952407

ways to apply CI which may not be possible with the current tools (*e.g.,* QuestionID = 10242059 [45]).

**Summary for RQ3.** The analysis of the most popular/difficult topics reveals that questions related to build issues are the most popular while those related version control and testing are challenging. By reviewing a significant sample of unanswered questions, we found that these questions usually receive responses from SO community in the form of comments which either seek for more details/clarifications or suggesting to address the problem.

## 3.3 Discussion and Takeaways

In the previous section, we showed that SO is widely used by developers to seek help with infrastructures and servers, and receive a response within a short period of time (11 hours on average). In our study, we supplement our quantitative analysis with a qualitative analysis based on a statistically significant set of CI questions to gain a deeper understanding of CI challenges. In this section, we discuss how our findings can help the researchers, tool builders, developers and educators.

### 3.3.1 Takeaways for researchers

We believe that our results can help CI researchers to further evolve the field. While we consider that all the revealed topics are important and worth to be studied, we think that the most popular/difficult topics should be given more attention by researchers and can be important directions of research. We particularly encourage researchers to consider further investigations on the following problems:

**More research on build (failure, time and systems).** Despite the significant number of studies that investigated the factors behind build failures (Rausch, Hummer, Leitner & Schulte, 2017; Beller, Gousios & Zaidman, 2017; Luo *et al.*, 2017) and the proposed automatic prediction tools

[45] https://stackoverflow.com/questions/10242059

(Hassan & Wang, 2017; Xie & Li, 2018; Saidani *et al.*, 2020a, 2022), CI developers still seek assistance to deal with build failures (as revealed by our findings) ; which suggests that more effort is required to investigate/prevent the reasons behind the build breakage. Future research can leverage the history of discussions on SO to examine the possible reasons behind the build failure. Moreover, our findings in RQ2 reveal that the long CI build time is another challenge faced by developers who seek help with solutions to speed up CI process. Recently, some solutions have been proposed to partially address the problem, by detecting the changes that can be skipped during the build (Abdalkareem *et al.*, 2019; Saidani *et al.*, 2021c). Nevertheless, more effort is need to speed up the time need for changes that cannot be skipped. Finally, as revealed in RQ2, the discussions around build systems (*e.g.,* MSbuild) are emergent. This potentially indicates a need for research in build systems as they are an important part of CI practices, as there is a limited research being conducted on them (Barua, Thomas & Hassan, 2014).

**Investigate the issues of Deployment.** Our analysis has revealed that deployment is a popular topic in CI discussions. Similarly to CI, Continuous Deployment (CD) practice helps reducing errors and speeding up the development process (Shahin *et al.*, 2017) which motivates the need to study its specific challenges. Having provided a methodology to investigate CI challenges, our study can be extended to conduct an in-deeper investigation of deployment challenges that being faced by developers.

**Research on the configuration of CI environments.** Our study revealed that the configuration of CI servers/infrastructure a large concern for developers. While is it been reported that CI systems are vulnerable to misconfiguration (Gruhn, Hannebauer & John, 2013), little is known how the features are misused in CI specification files. Hence, we encourage researchers to conduct empirical studies on this matter in order to improve the management and verification of CI configurations.

**More attention should be paid to Web/mobile platforms.** In RQ2, we found Web-related keywords (*e.g.,* "Selenium") in testing and deployment topics (Table 3.6). Moreover, the emergence of the mobile development is also apparent in these two topics (*e.g.,* "Xcode" and "application" keywords). This suggests a trend towards the CI of software for Web and mobile platforms. At the same time, we note the very limited academic research being done today in supporting the adoption in Web/mobile domains. Hence, we believe that a more research effort is required to study the specific needs, related to CI adoption, of developers in these platforms.

### 3.3.2 Takeaways for tool builders and vendors

**Enhance the user experience.** Developers use a variety of tools, including servers, Version Control Systems (VCS), testing frameworks and build systems, to support the CI of their software changes. In this paper, we showed that most of the tags in the CI questions are related to infrastructures and tools. Moreover, we revealed that some tools/infrastructures are emergent in developers' discussions. We therefore believe that tool builders must ensure that their products exhibit an optimal user experience. Specifically, a particular attention should be paid to tools/infrastructures being mentioned in this paper (*e.g.,* Jenkins, TeamCity, Docker etc.) as optimizing these tools would potentially reduce the obstacles for developers.

**Improve testing activity.** Getting people to write tests has been broadly recognized as difficult (Widder *et al.*, 2019). This finding is confirmed by our study as developers usually inquire about the basics of testing in the context of CI. Hence, we believe that testing tools builders should summarize more detailed instructions to help developers create their tests more easily and quickly.

### 3.3.3 Takeaways for practitioners and developers

**Recommendation to team leads.** Based on our results, we suggest that team leads should take the difficult topics into consideration when distributing the tasks between the development team members. Specifically, tasks related to testing and version control might be assigned and/or reviewed by experienced developers.

**Better choice for appropriate tools/infrastructures.** The wide adoption of CI depends largely on the high availability of tools/infrastructures. We encourage project stakeholders and developers to pay more attention to the selection of appropriate tools and infrastructures (and their configurations) for build CI pipelines in order to mitigate its challenges.

**Lack of expertise and skill.** CI/CD pipelines can be difficult for newcomers. The SO community might need to propose incentives to encourage CI experts to further contribute. Our findings motivate more experienced developers to explore the CI-related problems discussed by practitioners and solutions for these problems.

**Misuse of version control system (VCS).** The misuse of VCS can hinder the CI process (*e.g.,* big commits can lead to build failure (Saidani *et al.*, 2020a)). Developers must avoid bad practices (Zampetti *et al.*, 2020) related to the repository organization and be aware of its impacts on the CI process.

### 3.3.4 Takeaways for educators

**Towards a better understanding of CI practices/principals.** 8% of CI questions discuss how to get into the philosophy Of CI. This finding is an indicator of the of the limited availability of knowledge and training for CI and therefore suggests that it is still challenging for developers to practice CI effectively. Hence, educators are encouraged to ensure that CI principals/practices are covered and practiced in course materials and laboratories.

**More focus on different CI topics.** We highly recommend educators to pay a particular attention to the most popular and difficult topics based on specific experiences occurred to developers and discussed in Stack Overflow (*e.g.,* build, testing, version control, etc.) and ensure that these issues are covered in course materials and/or practiced by students in labs and/or assignments.

## 3.4 Threats to validity

**Threats to internal validity** are concerned with the factors that could have affected the validity of our results. First, we entirely relied on the existence of the term “continuous integration" in the tag or in the title to identify CI related posts in SO. There is the possibility that we may have missed other CI posts by excluding synonymous terms. However, we highly decreased the false positives (*i.e.,* selected questions that are not really related to CI). Indeed, in rare cases, we found that “continuous integration" term was miss-used by developers when posting the question. Hence, broadly speaking, we believe that our approach resulted in a significantly relevant dataset.

**Threats to construct validity** are mainly related to the rigor of the study design. The use of LDA to cluster CI discussions can be considered as a threat to the construct validity of our study. However, as mentioned earlier in the paper, this technique has been widely used in similar contexts (Peruma *et al.*, 2022; Openja *et al.*, 2020). Additionally, the search space used to tune LDA parameters could introduce some bias in our results as considering different ranges/parameters may yield to different results. To mitigate this threat, we used an advanced technique, Genetic Algorithm (GA) widely used for the automatic tuning of Machine Learning (ML) techniques (Saidani *et al.*, 2022; Yang & Shami, 2020; Wicaksono & Supianto, 2018). Nevertheless, future replication of this work should explore other ranges/parameters and their impacts on LDA performance. In terms of qualitative analysis, we rely heavily on manual analyzing. Due to the large volume of our data, we selected significant sample of tags (RQ1) and unanswered questions (RQ3) as representative data for our manual analysis with a confidence level and interval of 99% and 5% respectively.

**Threats to external validity** concerns the possibility to generalize our results. To conduct our study, we collected data from Stack Overflow (SO), the most popular Q&A forum (Openja *et al.*, 2020), within a period of 12 years (from 2008 to 2020). However, we cannot guarantee the generalizability of our findings to other forums/websites. Future work should replicate our study considering other technology-based question and answer websites.

## 3.5 Conclusion and Future Work

In this work, we present the first empirical study to investigate the trends, topics and challenges discussed by developers on Stack Overflow (SO) when adopting Continuous Integration (CI). By analyzing the asked questions, we show that SO has been widely used by developers to seek assistance with CI challenges. Specifically, Servers and Platforms are tagged the most in CI questions. With regards to CI topics, we found, using tuned LDA modeling, that discussions on CI can be categorized into six topics where 40% of the questions are about “Build" topic. Next, we investigate the characteristics of answers in terms of popularity (*e.g.,* number of views) and difficulty (*e.g.,* hours to receive an accepted answer) and find that “Build" topics are the most popular, while “Version Control" and “Testing" topics seem to be the most difficult. Based on our quantitative and qualitative analysis, we also distill many takeaways for different stakeholders.

We plan, as a future work to conduct a survey with different CI stakeholders from both open-source and industry in order to complement our empirical study with further insights into CI adoption and its related challenges; *e.g.,* we can ask the interviewed persons about the topics we observed in our empirical study and how difficult they are for them.

# CHAPTER 4

# ON THE IMPACT OF CONTINUOUS INTEGRATION ON REFACTORING PRACTICE: AN EXPLORATORY STUDY ON TRAVISTORRENT

Islem Saidani[a] , Ali Ouni[a] , Mohamed Wiem Mkaouer[b] , Fabio Palomba[c]

[a] Department of Software Engineering and IT, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[b] Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester,
NY 14623, United States
[c] University of Salerno, Via Giovanni Paolo II, 132, 84084 Fisciano SA, Italy

Paper published in *Information and Software Technology*, May 2021

## Abstract

The ultimate goal of Continuous Integration (CI) is to support developers in integrating changes into production constantly and quickly through automated build process. While CI provides developers with prompt feedback on several quality dimensions after each change, such frequent and quick changes may in turn compromise software quality without Refactoring. Indeed, recent work emphasized the potential of CI in changing the way developers perceive and apply Refactoring. However, we still lack empirical evidence to confirm or refute this assumption. We aim to explore and understand the evolution of Refactoring practices, in terms of frequency, size and involved developers, after the switch to CI in order to emphasize the role of this process in changing the way Refactoring is applied. We collect a corpus of 99,545 commits and 89,926 Refactoring operations extracted from 39 open-source GitHub projects that adopt Travis CI and analyze the changes using Multiple Regression Analysis (MRA). Our study delivers several important findings. We found that the adoption of CI is associated with a drop in the Refactoring size as recommended, while Refactoring frequency as well as the number (and its related rate) of developers that perform Refactoring are estimated to decrease after the shift to CI, indicating that Refactoring is less likely to be applied in CI context. Our study uncovers insights about CI theory and practice and adds evidence to existing knowledge about CI practices related especially

to quality assurance. Software developers need more customized Refactoring tool support in the context of CI to better maintain and evolve their software systems.

**Keywords.** Continuous Integration, Refactoring, Exploratory Study, Mining Software Repositories, Multiple Regression Analysis

## 4.1 Introduction

A major challenge in modern software engineering is ensuring the quality of increasingly large and complex software systems. To this end, software development companies have massively adopted Continuous Integration (CI) in order to deliver software with fewer defects and shorter release cycles. CI aims at supporting developers in integrating changes, into a shared repository, more frequently (and even daily) and the key to making this possible, according to Fowler, is automating the build and test processes. For its valuable benefits, such as significant improvements in productivity (Vasilescu *et al.*, 2015), CI has been promoted as the leading edge of software engineering practices (Hilton *et al.*, 2016).

To take full advantage of CI, a set of guiding principles have been introduced to support developers adopting CI in practice (Shahin *et al.*, 2017; Duvall *et al.*, 2007a; Vassallo *et al.*, 2018b,a; Zampetti *et al.*, 2020). For instance, as advocated by Duvall *et al.*, CI users should continuously inspect code quality, which includes performing Static Code Analysis (SCA), in order to maintain the code of good health. Another key principle is Continuous Refactoring (CR) (Chen & Babar, 2014; Vassallo *et al.*, 2018b) which consists of *“searching for Refactoring opportunities at every completed change and to perform Refactoring immediately, without postponing it"* (Vassallo *et al.*, 2018b). Indeed, as an *Agile* method, the incremental nature of CI requires the code to be continuously refactored in order to maintain high quality (Stamelos & Sfetsos, 2007) and keep the quality gates, steps required to ensure the reliability of code changes (Schermann *et al.*, 2016), always green (Vassallo *et al.*, 2018b). Otherwise, it may be hard for development teams to understand, maintain and extend their code (Szóke *et al.*, 2014). Moreover, the absence of CR may result in the need for large refactorings (Stamelos & Sfetsos, 2007) that, like any other complex change, may hinder the CI build progress and requires more debugging effort (Zhang *et al.*, 2019). Hence, it is encouraged to partition the large change into many smaller ones of few hours each (Zhao *et al.*, 2017).

From the academic side, the adoption of Refactoring techniques for CI has received some attention and automatic tools were proposed (Wyrich & Bogner, 2019; Alizadeh, Ouali, Kessentini & Chater, 2019), while others used the outcome of SCA tools to detect Refactoring opportunities (Wedyan, Alrmuny & Bieman, 2009; Mens & Tourwé, 2004; Ouni *et al.*, 2016). However, in practice, there is a lack of empirical knowledge of how Refactoring is applied in CI context. The only preliminary study was conducted by Vassallo *et al.* through a survey with CI developers. Their findings point out the potential of CI to change the way developers adopt Refactoring as it is commonly known that the late is often not applied (Negara, Chen, Vakilian, Johnson & Dig, 2012; Silva, Tsantalis & Valente, 2016; Murphy-Hill, Parnin & Black, 2012) and performed only by specific developers (Tsantalis, Guana, Stroulia & Hindle, 2013). However, there is no empirical evidence confirming this assumption.

In this paper, we want to investigate the possible impact of CI on the way Refactoring is applied in practice. First, we study whether CI adoption has increased the likelihood of applying Refactoring more frequently to answer the following question **(RQ1):** *Does CI impact the Refactoring frequency?*. Second, we study whether the size of Refactoring changes would decrease after the switch to CI. This leads us to our **RQ2:** *Does the adoption of CI affect the Refactoring change size?* Third, we study the relationship between adopting CI and the involvement of developers in Refactoring activities. Particularly, we ask our last research question **(RQ3)**: *How are developers involved in code Refactoring before and after the adoption of CI?*

We present an extension of Vassallo *et al.* work and conduct the first exploratory study involving a benchmark of 99,545 commits and 89,926 Refactoring operations during four year development of 39 Open-Source Software (OSS) projects centered around the adoption of Travis CI, a widely used CI service (Vasilescu *et al.*, 2015). Using Multiple Regression Analysis (MRA), we show that the adoption of CI is associated with a drop in the Refactoring size, which aligns with the "small Refactoring" guideline (Stamelos & Sfetsos, 2007), while Refactoring frequency as well as the number (and its related rate) of developers that perform Refactoring are estimated to decrease after the shift to CI, indicating that Refactoring is less likely to occur and, in contrast with the earlier findings (Vassallo *et al.*, 2018b), Refactoring is not spread in CI context. Our MRA also indicates that these trends will continue over time but with different variations between projects with different sizes, ages and releasing frequency. Based on these findings, we conjecture that software developers may need more customized Refactoring tool support in the context of CI to better maintain and evolve their software systems.

In summary, this paper makes the following contributions:

1. **Empirical evidence of the impact of CI on Refactoring:** We designed three novel research questions and conducted an empirical study that allowed us to provide the first in-depth answers to questions about the impacts of CI adoption on Refactoring practices.
2. **Data collection and analysis:** We collected and analyzed a benchmark of 99,545 commits and 89,926 Refactoring operations from 39 long-lived OSS projects. Then, we analyzed the data using MRA to capture any effects of CI adoption.
3. **A research road-map:** We provide practical implications of our findings for future research on the Refactoring of modern systems. We believe that novel techniques should be innovated to (i) raise developer's awareness of Refactoring in the context of CI, (ii) recommend micro-refactoring operations in order to avoid build failure and (iii) support newcomers when performing code quality tasks.

**Replication Package.** The dataset used in our study is publicly available for future replication and extension purposes (Saidani, 2020a).

**Structure of the paper.** The remainder of this paper is organized as follows. We present our research methodology in section 4.2, while present and analyze the obtained results in Section 4.3. In Section 4.4, we discuss the obtained results and their implications. Then, we review threats to validity in Section 4.5, and finally we address the conclusions to draw in Section 4.6.

## 4.2 Study Design and Methodology

The *goal* of this study is to investigate the possible impact of CI adoption on Refactoring activities by analyzing how developers change the way they refactor their software systems in practice. In this section, we define our research questions and present the design of our study.

### 4.2.1 Research questions

The study aims at addressing the following research questions:

**RQ1. Does CI impact the Refactoring frequency?** In this first RQ, we are particularly interested in investigating how frequently developers refactor their software systems after the adoption of CI. Our motivation is based on the fact that the aim of CI is to get changes into production as quickly as possible, without compromising software quality. We speculate that without continuous Refactoring, such frequent and quick changes during the CI process may negatively affect some quality attributes such as readability, understandability, flexibility, etc. (Vassallo *et al.*, 2018a). Indeed, Refactoring is known to have a paramount importance to deliver a high-quality software product, by removing defects and reducing technical debt (Fowler *et al.*, 1999) which are introduced by quick and often unsystematic development (Stamelos & Sfetsos, 2007).

**RQ2. Does the adoption of CI affect the Refactoring change size?** In this research question, we want to assess the size of the changes related to Refactoring through the software system before and after the adoption of CI. Indeed, Refactoring is recommended to be small in size (Stamelos & Sfetsos, 2007) as this would (i) help developers track the progress, (ii) reduce the risk of introducing complexity or defects during Refactoring and (iii) avoid breaking the build (Zhang *et al.*, 2019). Hence, we expect that after adopting CI, developers would integrate Refactoring related changes with smaller chunks.

**RQ3. How are developers involved in code Refactoring before and after the adoption of CI?** The motivation of this research question stems from previous research works (Tsantalis *et al.*, 2013) confirming that Refactoring is performed by specific developers that usually have a key role in the management of the project. In this study, we want to analyze whether CI raises the code authorship, *i.e.,* motivation to program the code with high quality by performing the Refactoring (Wang, 2009).

### 4.2.2 Methodology

Figure 4.1 provides an overview of our research methodology to address our defined research questions. Our methodology comprises three main steps: (*i*) context selection, (*ii*) Refactoring data extraction, and (*iii*) analysis method. In the following, we present the details of each of these three steps.

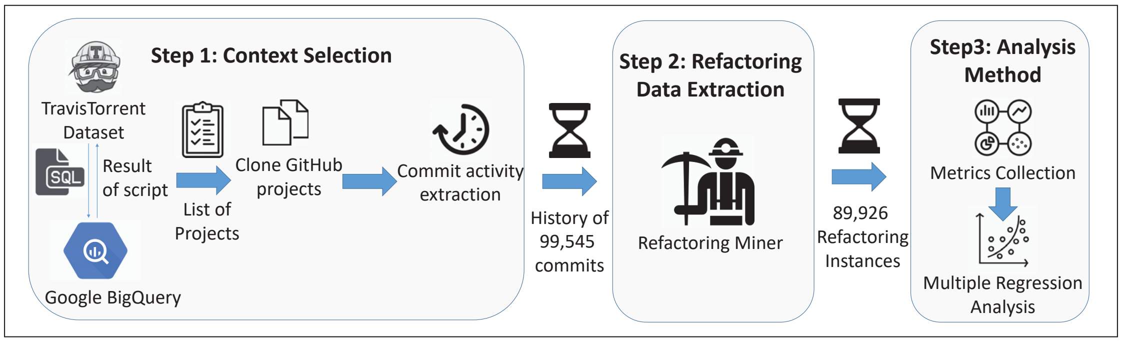

Step 1: Context Selection
TravisTorrent Dataset
SQL
Result of script
List of Projects
Clone GitHub projects
Commit activity extraction
Google BigQuery
History of 99,545 commits
Step 2: Refactoring Data Extraction
Refactoring Miner
89,926 Refactoring Instances
Step3: Analysis Method
Metrics Collection
Multiple Regression Analysis

Figure 4.1 An overview of our research methodology to study the Refactoring practices in CI

#### 4.2.2.1 Context Selection

We gather our dataset from 39 OSS projects hosted on GitHub which have switched to Travis CI, a widely used CI system, at some point during their life-cycle. To answer our research questions, we mined these projects based on the latest TravisTorrent dump dated on 2017/02/08[1] and using the Big Query Google Tool [2] to query pieces of information such as the programming language and the repository URL. The choice of the subject systems was driven by the following criteria:

[1] https://travistorrent.testroots.org/

[2] https://cloud.google.com/bigquery

- Projects with sufficiently long historical code change records, *i.e.,* at least two years before and after the adoption of CI to get deep insights into the possible impacts and feed our regression models with sufficient data.
- Projects that have a consistent change activity during the studied period *i.e.,* having at least one merged commit in the mainline branch each month for the studied period. We chose a monthly partition following previous studies on the impact of CI (Yu *et al.*, 2016a; Gupta *et al.*, 2017; Rahman *et al.*, 2018; Vasilescu *et al.*, 2015; Zhao *et al.*, 2017) because (*i*) it leads to more meaningful results than providing only one value per year and (*ii*) to fit our regression models and control for time variable. Hence, we avoid biasing our results with zero values due to projects not being active during some months (thus no Refactoring activities will take place).
- We also restricted our analysis to Java projects as we rely on the *RefactoringMiner* tool (Tsantalis, Mansouri, Eshkevari, Mazinanian & Dig, 2018a), an automated tool for detecting Refactoring activities applied in software projects during their development life-cycle (Section **-B).

Thereafter, we cloned all project repositories and extracted all their commits change history to be used in next steps. We recorded a total of 99,545 commits on the mainline branch for the studied projects. Table 4.1 reports the analyzed projects, the number of commits, Refactoring related commits and contributors. Moreover, we report other historical statistics about the projects such as the age in months and the number of releases. All the data collected and used in our exploratory study is publicly available for replication purposes in our comprehensive replication package (Saidani, 2020a).

#### 4.2.2.2 Refactoring Data Extraction

We use in our study the tool *RefactoringMiner*[3], a commit-based Refactoring detection tool that is based on the UMLDiff algorithm (Xing & Stroulia, 2005) for computing the differences between object-oriented models (Tsantalis, Mansouri, Eshkevari, Mazinanian & Dig, 2018b).

[3] https://github.com/tsantalis/RefactoringMiner

Table 4.1 Systems involved in the study

| Project | Description | Historical Statistics | | | Considered in the study | | |
|---|---|---|---|---|---|---|---|
| | | Age | Total Commits | Total Contributors | # of releases | # of commits | # of ref. commits |
| *airlift/airlift* | Framework for building REST services | 37 | 2,371 | 73 | 227 | 905 | 114 |
| *apache/pdfbox* | Mirror of Apache PDFBox | 75 | 8,120 | 24 | 49 | 4,340 | 801 |
| *apache/storm* | Mirror of Apache Storm | 42 | 7,321 | 477 | 39 | 4,722 | 412 |
| *aws/aws-sdk-java* | The official AWS SDK | 36 | 1,835 | 188 | 787 | 291 | 110 |
| *chocoteam/choco3* | A Java library for Constraint Programming | 35 | 3,819 | 30 | 28 | 2,328 | 565 |
| *dropwizard/dropwizard* | A library for RESTful web services | 26 | 3,905 | 444 | 143 | 2,240 | 290 |
| *druid-io/druid* | A real-time analytics database. | 25 | 7,330 | 381 | 428 | 5,308 | 881 |
| *DSpace/DSpace* | A digital asset management system | 134 | 9,209 | 213 | 95 | 2,370 | 236 |
| *FasterXML/jackson-databind* | Data-binding package | 25 | 4,237 | 184 | 115 | 2,225 | 545 |
| *FenixEdu/fenixedu-academic* | Student Information System | 123 | 36,934 | 162 | 345 | 5,517 | 644 |
| *geoserver/geoserver* | Open source software server | 27 | 7,568 | 295 | 123 | 3,562 | 575 |
| *GeoWebCache/geowebcache* | Caching server | 101 | 2,134 | 75 | 122 | 461 | 80 |
| *google/error-prone* | Static analysis tool | 35 | 3,597 | 222 | 31 | 1,497 | 307 |
| *google/guava* | Google core libraries | 61 | 4,995 | 319 | 87 | 2,383 | 393 |
| *grails/grails-core* | Grails Web Application Framework | 106 | 16,152 | 329 | 189 | 5,513 | 405 |
| *igniterealtime/Openfire* | A XMPP server | 115 | 7,931 | 183 | 158 | 1,377 | 201 |
| *jOOQ/jOOQ* | Light database-mapping software library | 23 | 7,135 | 84 | 73 | 4,137 | 697 |
| *jpos/jPOS* | Open source library/framework | 179 | 4,378 | 74 | 49 | 713 | 62 |
| *junit-team/junit* | A testing framework | 155 | 2,002 | 207 | 23 | 843 | 137 |
| *lenskit/lenskit* | Recommender toolkit | 40 | 5,884 | 50 | 52 | 3,575 | 588 |
| *maxcom/lorsource* | Website engine | 64 | 6,759 | 89 | 1 | 3,500 | 415 |
| *mybatis/mybatis-3* | SQL mapper framework | 32 | 2,399 | 142 | 29 | 1,220 | 146 |
| *nutzam/nutz* | Web Framework | 47 | 5,379 | 94 | 57 | 1,897 | 256 |
| *oblac/jodd* | An open-source Java utility library | 53 | 5,055 | 63 | 54 | 2,446 | 597 |
| *orbeon/orbeon-forms* | Open source web forms solution | 90 | 22,092 | 36 | 50 | 4,844 | 304 |
| *owncloud/android* | Android App | 28 | 6,141 | 91 | 92 | 3,607 | 511 |
| *perfectsense/brightspot-cms* | Enterprise user experience platform | 32 | 5,678 | 49 | 23 | 4,557 | 298 |
| *proofpoint/platform* | Security Awareness & Education Platform | 49 | 3,132 | 69 | 216 | 1,203 | 187 |
| *sparklemotion/nokogiri* | Web parser | 36 | 4,013 | 195 | 147 | 1,585 | 81 |
| *spring-data-commons* | shared infrastructure across the Spring Data | 47 | 1,891 | 91 | 155 | 714 | 147 |
| *tananaev/traccar* | GPS Tracking System | 58 | 5,214 | 113 | 37 | 3,162 | 388 |
| *TGAC/miso-lims* | An open-source LIMS for NGS sequencing centres | 58 | 3,209 | 25 | 219 | 2,908 | 450 |
| *tinkerpop/blueprints* | A Property Graph Model Interface | 28 | 1,532 | 64 | 19 | 1,414 | 322 |
| *tinkerpop/rexster* | A Graph Server | 26 | 1,476 | 26 | 17 | 1,400 | 259 |
| *twall/jna* | Java Native Access | 171 | 3,112 | 170 | 52 | 1,272 | 125 |
| *Unidata/thredds* | A middleware | 86 | 9,780 | 63 | 60 | 3,739 | 1,122 |
| *weld/core* | Integrations for Servlet containers and Java SE | 71 | 7,534 | 108 | 160 | 2,351 | 501 |
| *xtreemfs/xtreemfs* | Distributed Fault-Tolerant File System | 66 | 4,742 | 52 | 20 | 2,175 | 255 |
| *zxing/zxing* | Barcode scanning library | 74 | 3,434 | 143 | 27 | 1,244 | 118 |
| | **Median** | **49** | **4,995** | **94** | **60** | **2,328** | **307** |
| | **Average** | **64.5** | **6,395.6** | **146.1** | **117.9** | **2,552.4** | **372.4** |
| | **Total** | **-** | **249,429** | **5,697** | **4,598** | **99,545** | **14,525** |

Table 4.2 presents the list of Refactoring operations that can be detected by RefactoringMiner with their respective number of Refactoring instances identified in the 39 projects involved in our study. We selected the RefactoringMiner tool as it provides high precision of 98% and recall of 87% (Tsantalis *et al.*, 2018a), implements the detection of over 32 Refactoring operations, and has been widely used in recent empirical studies (Tan & Bockuisch; Silva *et al.*, 2016; Vassallo, Grano, Palomba, Gall & Bacchelli, 2019a; AlOmar, Mkaouer, Ouni & Kessentini, 2019).

Table 4.2 Analyzed Refactoring operations statistics with their different levels

| **Refactoring Operation** | **Level** | **Instances** | **Projects** |
|---|---|---|---|
| Move Class | Class | 13,312 | 38 |
| Rename Method | Method | 10,749 | 39 |
| Rename Variable | Block | 9,527 | 39 |
| Rename Attribute | Field | 7,341 | 39 |
| Rename Parameter | Block | 6,706 | 39 |
| Extract Method | Method | 6,154 | 39 |
| Pull Up Attribute | Field | 5,780 | 38 |
| Move Method | Method | 5,527 | 39 |
| Move Attribute | Field | 3,691 | 39 |
| Pull Up Method | Method | 3,414 | 39 |
| Extract Variable | Block | 2,964 | 39 |
| Rename Class | Class | 2,855 | 39 |
| Inline Method | Method | 2,009 | 39 |
| Push Down Method | Method | 1,077 | 36 |
| Extract Class | Class | 997 | 39 |
| Move And Rename Class | Class | 915 | 37 |
| Move Source Folder | Package | 655 | 31 |
| Inline Variable | Block | 653 | 39 |
| Push Down Attribute | Field | 602 | 30 |
| Extract Super-class | Class | 553 | 37 |
| Parameterize Variable | Method | 479 | 38 |
| Replace Variable With Attribute | Block | 461 | 36 |
| Extract Interface | Class | 324 | 32 |
| Change Package | Package | 305 | 28 |
| Extract Subclass | Class | 126 | 32 |
| Move And Rename Attribute | Field | 32 | 13 |
| Replace Attribute | Field | 24 | 8 |
| | **Total** | 89,926 | 39 |

#### 4.2.2.3 Analysis Method

**Used Metrics:**

To address **RQ1**, we define two measures including the number of Refactoring commits per month (NRC) and the Refactoring rate (RRC) as follows:

- **NRC:** *Number of Refactoring Commits* It counts the number of commits that have at least one Refactoring operation applied, in each month. In this study, we only consider commits that are merged into the mainline development branch as local git commits may not be subjected to CI by Travis as stated by previous works (Zhao *et al.*, 2017; Vasilescu *et al.*, 2015)
- **RRC:** *Rate of Refactoring Commits* which computes the ratio of Refactoring commits (NRC) among the total number of merged commits (NC) per month. This measure gives insights about the extent to which developers tend to refactor their code during the development of their projects.

To answer **RQ2**, we capture the change size of a Refactoring commit. For this aim, we define the following measures. Note that each mean value bellow is computed over all Refactoring commits in the considered month.

- **RB:** *Refactoring Breadth* The average number of files where at least one Refactoring operation was applied per commit. To compute this metric, we used a predefined method of RefactoringMiner called "detectAtCommit" which returns all the needed information about the involved classes.
- **RBR:** *Refactoring Breadth Rate* The average rate of Refactoring breadth per commit. The rate refers to the number of files related to Refactoring divided by the total number of modified files.

To answer **RQ3**, we assess the extent to which developers are involved in Refactoring activities before and after the adoption of CI by defining the following metrics:

- **NRefDev:** *Number of Refactoring Developers* Counts the number of developers who applied at least one Refactoring per month.
- **RRD:** *Rate of Refactoring Developers* The ratio of the number of committers who applied Refactoring in their commit changes divided by the total number of committers.

**Multiple Regression Analysis**

To evaluate the effects of the adoption of CI (RQs 1-3), we use Multiple Regression Analysis (MRA) (Edwards, 1985) as a method for analyzing the relationship between a set of explanatory variables (predictors, *e.g.,* the time in months) and a response (outcome, *e.g.,* the rate of Refactoring commits), while controlling for known covariates (*e.g.,* project age) that might influence the response. Solving the regression gives us the coefficients for each predictor. If the coefficient is significant, it can help us reason about the treatment (*e.g.,* the adoption of CI in our case) and its effects, if any, while controlling for confounding variables. In our study, we perform our MRA to estimate the trends in our set of metrics (Section 4.2.2.3) marked as *Y* before the adoption of CI, and the changes in the trend after the adoption CI as follows:

$$y_t = \alpha + \beta * time\_before\_ci_t + \gamma * time\_after\_ci_t + \delta * ci\_is\_adopted_t + \epsilon$$

Here $y_t$ is the trend (*i.e.,* the predicted value) in the outcome variable Y in each time $t$; *time_-before_ci* indicates the time in months at time $t$ from the start of the observation and coded 0 after CI adoption(*i.e.,* from -24 to -1); *time_after_ci* counts the number of months at time $t$ after the CI adoption and coded 0 before the adoption (*i.e.,* from 1 to 24); *ci_is_adopted* indicates whether CI is adopted at time $t$ ($ci\_is\_adopted = 1$) or not ($ci\_is\_adopted = 0$). Using this model, we can capture any divergence (regression) in the slopes (decrease/increase) before and after the adoption of CI. Moreover, we consider the following confounding variables ($\epsilon$):

- **Total number of commits (TotalComm)** Following Zhao *et al.*, we consider the total number of commits in a project's history as an indicator for project activity/size.
- **Total number of developers (TotalDev)** We also consider the total number of developers as a proxy for the project's community size.
- **Project age at the time of CI adoption (AgeAtCI)** in months. Mature projects may be less affected by the adoption of CI than other projects (Zhao *et al.*, 2017).

- **Number of releases (NReleases)** We manually checked the timeline of each project to collect its number of releases. We want to inspect the releasing frequency on Refactoring practice as it is known that projects with frequent releases may have the chance to fix bugs faster (Habchi, Rouvoy & Moha, 2019) and hence apply more refactorings.

We implement the MRA using the function *lm* from *lmerTest*[4] package in R. Log transform predictors (Cohen, West & Aiken, 2014) are used to stabilize the variance and improve model fit. To avoid multicollinearity phenomenon in which one predictor variable can be linearly predicted from the others (Cohen *et al.*, 2014), we consider the Variance Inflation Factor (package *car*[5] in R). To improve robustness, the top 3% of the data was filtered out as outliers in order to avoid inflating the model's fit (Vasilescu *et al.*, 2015). For each model, we report (*i*) the coefficients that describe the mathematical relationship between each independent variable and the dependent variable and higher values suggests higher effect, (*ii*) $\rho - values$ that provide the significance level of the coefficients, (*iii*) the sum of squares which computes the variance explained by each variable, and (*iv*) the standard error which indicates how wrong the regression model using the units of the response variable; smaller values are better to provide evidence of the fitted model.

## 4.3 Study Results

In this section, we present and discuss the results of our study to answer our research questions RQ1-3. All the data collected and used in our study is publicly available for replication and extension purposes in our comprehensive replication package (Saidani, 2020a).

For the sake of clarity, the key metrics used in our study are shown in Table 4.3. The results of our MRA are presented and discussed in the next section.

4 https://cran.r-project.org/web/packages/lmerTest/index.html

5 https://cran.r-project.org/web/packages/car/car.pdf

Table 4.3 Summary of the study measures

| **Metric** | **Description** |
|---|---|
| NRC | Number of Refactoring Commits |
| RRC | Rate of Refactoring Commits |
| RB | Refactoring Breadth |
| RBR | Refactoring Breadth Rate |
| NRefDev | Number of Refactoring Developers |
| RRD | Rate of Refactoring Developers |
| TotalComm | Total number of commits |
| TotalDev | Total number of developers |
| AgeAtCI | Project age at the time of CI adoption |
| NReleases | Number of releases |

### 4.3.1 RQ1: Trends in Refactoring frequency after the adoption of CI

We start by quantifying the trends in the number of Refactoring commits (NRC) and Rate of Refactoring Commits (RRC) using the Multiple Regression Analysis (MRA) as described in Section 4.2.2.3. Table 4.4 summarizes the regression analysis results for Refactoring frequency measures. For each variable, we report its coefficients (*Coeff*) and corresponding sum of squares (*Sum Sq*), a measure of variance for each variable and the standard error of the regression (*Error*) which represents the average distance between the observed values and the regression line. The statistical significance is indicated by stars symbols. We consider coefficients to be important if they are statistically significant ($\rho < 0.05$).

From the obtained results in Table 4.4, the NRC model confirms a statistically significant negative baseline trend in the response with *ci_is_adopted* which means that the number Refactoring commits would decrease after introducing CI. The coefficient for time is negative, suggesting a decreasing baseline trend in terms of Refactoring commits after the adoption of CI. However, the model does not detect any effect for the time before the adoption of CI since the coefficient *time_before_ci* is not statistically significant. Overall, the trend remains descending (the sum of the coefficients for *time_after_ci* and *ci_is_adopted* is negative): less Refactoring commits after the adoption of CI.

Table 4.4 MRA results for Refactoring frequency in terms of Number of Refactoring Commits (NRC) and Rate of Refactoring Commits (RRC)

| **Metric** | **NRC Model** | | | | **RRC Model** | | | |
|---|---|---|---|---|---|---|---|---|
| | *Coeff* | *Error* | $\rho - value$ | *Sum Sq* | *Coeff* | *Error* | $\rho - value$ | *Sum Sq* |
| Intercept | -11.76 | 6.39 | . | | 0.34 | 0.11 | ** | - |
| ci_is_adopted | -2.20 | 0.51 | *** | 548.9 | -0.01 | $7.9{*}10^{-3}$ | * | 0.041 |
| time_before_ci | 0.01 | 0.02 | | 9.25 | $-7.9{*}10^{-4}$ | $4.0{*}10^{-4}$ | . | 0.027 |
| time_after_ci | -0.12 | 0.02 | *** | 631.8 | $-7.9{*}10^{-4}$ | $4.0{*}10^{-4}$ | . | 0.027 |
| log(TotalComm) | 4.07 | 0.72 | *** | 947.4 | -0.01 | 0.01 | | 0.004 |
| log(TotalDev) | -0.72 | 0.61 | | 42 | -0.01 | 0.01 | | 0.019 |
| log(AgeAtCI) | -3.12 | 0.83 | *** | 423.1 | -0.02 | 0.01 | | 0.013 |
| log(NReleases) | 0.09 | 0.43 | | 1.51 | 0.01 | $7.7{*}10^{-3}$ | | 0.016 |
| $R^2$ | 0.17 | | | | 0.07 | | | |

***: $\rho < 0.001$, **: $\rho < 0.01$, *: $\rho < 0.05$, '.': $\rho < 0.1$, ' ': $\rho \geq 0.1$

Next, we assess the confounding variables namely the project size in terms of total number of commits, developers, project age, and number of releases. As reported in Table 4.4, the NRC model confirms a statistically significant, positive, baseline trend in the response with project size ($TotalComm$) which explains an important amount of variability in the response (*Sum Sq* = 947.4). This finding suggests that Refactoring is performed more frequently within bigger projects as they are more active and have larger codebase. For example, in *Unidata/thredds* project for which we recorded a 9,780 of commits, developers merged 20 Refactoring commits per month on median, while in *airlift/airlift* project with 2,371 commits, developers tend to merge about 2 Refactoring commits per month on median for the studied period. Also, the model reveals a particular trend for older projects ($AgeAtCI$) to apply less code refactorings. This finding is quite surprising since it is commonly admitted that as projects age, the maintenance focus is generally shifted to bug-fixing (Zhao *et al.*, 2017) or quality assurance to master the increasing software complexity (Lehman, 1996) which is usually performed through the assistance of Refactoring (Fowler, 2018). Moreover, we observe that the team size ($TotalDev$: the total number of commit authors over the entire history) has no statistically significant effect which means that projects with larger committers base do not necessarily apply more the Refactoring (cf. Table 4.4). For example, *apache/storm* project which has the larger base of contributors in our dataset with 477 contributors, developers tend to merge 6 Refactoring commits per month while in *TGAC/miso-lims* with 25 contributors, we recorded a median number of Refactoring commits of 9 in the studied period. Furthermore, we found no evidence for the releasing frequency ($NReleases$) to affect Refactoring frequency estimators which means that, for the studied projects, a higher releasing frequency does not necessarily imply that developers apply more Refactoring.

Looking at the rate of Refactoring commits, *i.e.,* the RRC model, we see that the only significant predictor is the CI adoption variable suggesting that the rate of Refactoring commits would decrease after introducing CI with a slight decrease trend of 0.01. The model reveals no evidence for time variables to be effective as the coefficients are not significant. With regard to the confounding variables, we observe that all the studied project characteristics have no significant effect.

**Summary for RQ1.** Our MRA study results suggest that the adoption of CI can result into a decrease in terms of Refactoring frequency. However, the regression analysis reveals that projects with larger size are less sensitive to this trend. Moreover, the MRA models suggest the more aged is the project, the less performed is the Refactoring.

### 4.3.2 RQ2: Trends in Refactoring change size after the adoption of CI

In this research question, we are particularly interested in exploring the possible effects of CI on Refactoring breadth. Hence, we analyze by using MRA models the relationships between CI related variables and metrics related to Refactoring churn and breadth while controlling for confounding variables. The MRA models for Refactoring breadth are summarized in Table 4.5.

First, Table 4.5 reveals a significant drop in the number of changed files related to Refactoring after the adoption of CI since $ci\_is\_adopted$ variable is statistically significant but with no significant effect for the time which indicates that this trend may change over time. This result reveals that *refactoring tends to be less diffused after the adoption of CI*. Looking at the confounding variables, we see through the RB model that in aged projects, Refactoring changes tend to affect fewer files since the relative coefficient (-0,6) is negative while in the RBR model, this effect was not significant. Additionally, we found no evidence for the project size to affect the Refactoring breadth. Moreover, we see that the rate of Refactoring breadth slightly decreases after CI with a higher frequency of releasing as suggested in the RBR model indicating that $NReleases$ predictor has a significant effect on the response variables.

Table 4.5 MRA analysis results for the Refactoring breadth (RB) and the Refactoring breadth rate (RBR)

| **Metric** | **RB Model** | | | | **RBR Model** | | | |
|---|---|---|---|---|---|---|---|---|
| | *Coeff* | *Error* | $\rho - value$ | *Sum Sq* | *Coeff* | *Error* | $\rho - value$ | *Sum Sq* |
| Intercept | 1 | 1.9 | | | 0.3 | 0.2 | | |
| ci_is_adopted | -0.6 | 0.2 | ** | 51.6 | -0.04 | 0.02 | * | 0.22 |
| time_before_ci | $5{*}10^{-3}$ | 0.01 | | 1.1 | $9{*}10^{-4}$ | $10^{-3}$ | | 0.03 |
| time_after_ci | -0.02 | 0.01 | * | 33.8 | $-2{*}10^{-3}$ | $10^{-3}$ | * | 0.24 |
| log(TotalComm) | 0.4 | 0.2 | . | 22.75 | 0.04 | 0.02 | | 0.13 |
| log(TotalDev) | -0.1 | 0.1 | | 2.1 | $-{*}10^{-3}$ | 0.02 | | $2{*}10^{-3}$ |
| log(AgeAtCI) | -0.6 | 0.2 | * | 36.8 | -0.02 | 0.02 | | 0.02 |
| log(NReleases) | 0.2 | 0.1 | | 14.7 | -0.03 | 0.02 | * | |
| $R^2$ | 0.05 | | | | 0.05 | | | |

***: $\rho < 0.001$, **: $\rho < 0.01$, *: $\rho < 0.05$, '.': $\rho < 0.1$, ' ': $\rho \geq 0.1$

**Summary for RQ2.** Our MRA results reveal that the Refactoring tend to affect less files after the adoption of CI. Additionally, our model suggests a slight drop in the relative rate after the adoption of CI but with different variations between projects especially with higher releasing frequency.

### 4.3.3 RQ3: How are developers involved in Refactoring activities?

In this research question, we analyze using MRA whether the adoption of CI impacts the way developers are involved in Refactoring activities. The statistical model for the number of Refactoring developers and its relative rate are summarized in Table 4.6.

Looking at the number of Refactoring developers (NRefDev) model, we observe that the *time_after_ci* and *ci_is_adopted* predictors exhibit negative coefficients scores of -0.02 and -0.39, respectively. We first note such a slight increasing trend in the number of Refactoring

Table 4.6 The MRA analysis results for the Number of Refactoring Developers (NRefDev) and the Rate of Refactoring Developers (RRD)

| **Metric** | **NRefDev Model** | | | | **RRD Model** | | | |
|---|---|---|---|---|---|---|---|---|
| | *Coeff* | *Error* | $\rho - value$ | *Sum Sq* | *Coeff* | *Error* | $\rho - value$ | *Sum Sq* |
| Intercept | -4.4 | 1.8 | * | | 0.82 | 0.28 | ** | |
| ci_is_adopted | -0.39 | 0.1 | *** | 17.2 | -0.12 | 0.02 | *** | 1.83 |
| time_before_ci | 0.01 | $5*10^{-3}$ | *** | 17.4 | $-2*10^{-3}$ | $10^{-3}$ | * | 0.33 |
| time_after_ci | -0.02 | $5*10^{-3}$ | *** | 19.6 | $-5*10^{-3}$ | $10^{-3}$ | *** | 1.12 |
| log(TotalComm) | 0.7 | 0.2 | *** | 17.5 | 0.04 | 0.03 | | 0.14 |
| log(TotalDev) | 0.2 | 0.1 | | 2.05 | -0.1 | 0.02 | *** | 0.88 |
| log(AgeAtCI) | -0.39 | 0.2 | | 3.3 | -0.08 | 0.03 | * | 0.29 |
| log(NReleases) | 0.14 | 0.12 | | 1.66 | $7*10^{-3}$ | 0.01 | | 0.01 |
| $R^2$ | 0.2 | | | | 0.14 | | | |

***: $\rho < 0.001$, **: $\rho < 0.01$, *: $\rho < 0.05$, '.': $\rho < 0.1$, ' ': $\rho \geq 0.1$

developers prior to the adoption of CI, although the trend slows down following the adoption of CI. In addition, our model results indicate that the variable counting for the project size ($TotalComm$) behaves consistently with NRC model 4.4: bigger projects tend to have larger base of Refactoring developers. Moreover, we found no evidence for the contributor base ($TotalDev$) to have any effects which indicates that a large number of contributors does not imply having more developers to apply Refactoring. Neither the age nor the releasing frequency seems to have any significant effect.

With regard to RRD model, we observe a significant negative for time variable before the adoption of CI which remains decreasing after the switch to CI considering its related predictors. When we look at the confounding variables, we observe also a significant negative trend for the variables accounting for the age. Another important result to highlight is the significant negative effect of the contributor base ($TotalDev$) on the rate of Refactoring developers which indicates that the more involved developers are in the project, the less is the rate of those who apply Refactoring. Overall, this model suggests that the rate of Refactoring developers tends to decrease as the time passes and this trend is slightly accelerated after the adoption of CI.

To get more insights, we provide an example extracted from our dataset namely MYBATIS-3 project[6] which is an SQL mapper framework for Java. During its development, the *Mybatis* project version control system involves, for the studied period, 8 developers before the adoption of CI and 40 developers after its adoption. Figures 4.2a and 4.2b show the percentage of the Refactoring operations performed by *Mybatis* developers before and after the adoption of CI, respectively. While all the developers have applied at least one Refactoring before CI (8/8 developers), Refactoring activities were performed by a limited number of developers after the adoption of CI (7/40 developers). Additionally, the top-one Refactoring developer, namely "developer1" (a core team member), performed 72% of the Refactoring commits before and after the adoption of CI. He is also the top-one committer with 67% and 52% of the commits before and after the adoption of CI, respectively. These observations are consistent with previous results (Tsantalis *et al.*, 2013) claiming that most of the applied refactorings are generally performed by specific developers (usually core team members). Moreover, we can confirm a previous assumption about developers attraction in the context of CI (Vasilescu *et al.*, 2015).

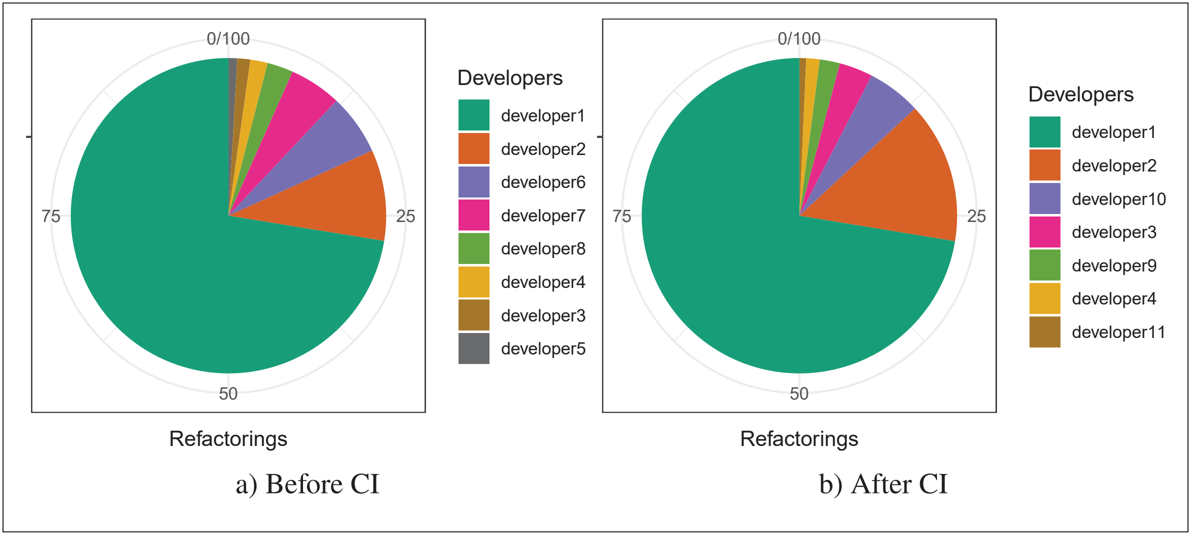


Figure 4.2 Distribution of Refactoring contributors in the project *mybatis/mybatis-3*

6 https://github.com/mybatis/mybatis-3

**Summary for RQ3.** MRA results reveal a decreasing trend for the rate of Refactoring developers especially with the adoption of CI with a considerable negative effect for aged projects and those with larger number of contributors. This may be due to the fact that the Refactoring is usually performed by particular developers and as the contributors base gets larger after the adoption of CI, the Refactoring rate will decrease.

## 4.4 Discussion

In this section, we further discuss the main findings of our study along with outlining their practical implications for future research on the Refactoring of modern systems.

**Refactoring is less applied in CI.** Our results reveal that the Refactoring frequency tend to decrease after the shift to Travis CI. This finding was surprising as CI principles may suggest developers to refactor their code more frequently to improve software quality. This may be due to the fact that CI developers may not consider quality degradation to affect the success of the build process as stated by Vassallo *et al.*. Based on this finding, we believe that future research effort should be devoted to build techniques able to increase the developer's awareness of Refactoring in the context of CI, for instance through improved visualization approaches that may graphically show to developers how a certain Refactoring action, conducted at build-time, would be beneficial for the quality of source code.

**Towards Just-In-Time Refactoring recommendation.** Our results for RQ2 reveal that developers tend to make smaller Refactoring changes to software projects, as they have a lower Refactoring breadth, which is consisting with “refactor smaller" (Stamelos & Sfetsos, 2007) and “commit smaller" (Fowler, 2006) guidelines. We believe that this finding would encourage tool builders to conceive Refactoring recommendation systems that can be adopted in a CI environment and able to recommend micro-refactoring operations or, even better, small local Refactoring operations that targets specific files touched by developers during a code change (*i.e.,* commit). These just-in-time Refactoring tools would (*i*) avoid changing the program design radically, and (*ii*) allow developers reviewing the recommendations, and their relative impacts and hence easily decide whether to apply or ignore them. Such tools could be in the form of Refactoring recommendation systems or bots that can be integrated into existing CI systems. While some preliminary research has been conducted toward this direction (Lambiase, Cupito, Pecorelli, De Lucia & Palomba, 2020; Pantiuchina, Bavota, Tufano & Poshyvanyk, 2018; Ujihara, Ouni, Ishio & Inoue, 2017; Alizadeh *et al.*, 2019; Wyrich & Bogner, 2019), we believe that additional effort is needed to build Refactoring tools that more properly reflect the developer's needs in the context of CI development.

**Support for newcomers to better practice Refactoring.** To survive and thrive, a software project must attract, support and retain new developers and help them be productive. However, our findings show that newcomers may be reluctant to practice Refactoring activities in the project: these are perfectly in line with the results reported by previous studies on the barriers that newcomers face when joining a new project (Steinmacher, Wiese, Chaves & Gerosa, 2013; Steinmacher, Conte, Gerosa & Redmiles, 2015). However, our study shows that an additional barrier consists of newcomers not being able to refactor source code to improve its quality. Based on this result, we envision a novel category of tools that may support newcomers when performing code quality tasks: more specifically, tool builders should provide development teams with more practical tools and/or techniques for supporting newcomers during the integration in the development team as well as instruments that community shepherds may use to identify the developers having adequate skills to properly guide the newcomers in their Refactoring phases.

## 4.5 Threats to validity

A number of possible threats might affect the validity of our empirical study.

**Threats to Internal validity** concern factors that could have influenced our results (Palomba, Panichella, Zaidman, Oliveto & De Lucia, 2017). From the list of Cook, we consider that one threat to internal validity can be related to *instrumentation*: We opted for RefactoringMiner, an open-source tool, to collect Refactoring data. This tool has a high F1-score of 81% according to recent experiments conducted in (Tan & Bockuisch). To alleviate any potential threats with

RefactoringMiner, we are planning to replicate our study with other existing tools such as *RefDiff*, a state-of-the-art Refactoring detection tool that has shown a high accuracy (Silva & Valente, 2017). More interestingly, to enable other researchers to verify and extend our study, we provide our replication package along with detailed results available for the research community (Saidani, 2020a). Another threat is related to *confounding variables*. To mitigate this issue, we included controls in our models, to capture project size, age, community base and releasing trends that could have confounded the relationship between CI adoption and Refactoring practice.

**Construct threats to validity** are mainly related to the fact that some projects may leave CI systems after its adoption (Widder *et al.*, 2018). To address this issue, we manually checked whether Travis-CI was disabled/abandoned by investigating all the commits in which the CI configuration file was modified and found that none of our studied projects has abandoned CI during the studied period. Another potential threat could be related to selecting projects that used another CI system before adopting Travis-CI. Hence, we mitigated this issue by inspecting the existence of any other CI configuration file (*e.g.,* “.appveyor.yml" for AppVeyor CI system) before the adoption of Travis-CI. In this investigation, we considered AppVeyor,[7] Circle-CI,[8] and Drone.[9] Additionally, we checked that our studied projects never used a self-hosted CI system (*i.e.,* using their CI service locally) like Jenkins,[10] Team-city,[11] or Easy-CIS,[12] by inspecting whether the commit messages contained the name of the above mentioned CI systems.

**Conclusion threats to validity** refer to issues that affect our ability to draw the correct conclusions and the way we estimated Refactoring practice. The fact that developers did not apply more frequently/intensively Refactoring, this does not mean that they did not search for Refactoring opportunities. In other terms, developers could have some recommendations of Refactoring (or checked manually Refactoring opportunities) but find them not relevant, so they may end up

[7] https://www.appveyor.com/

[8] https://circleci.com/

[9] https://drone.io/

[10] https://jenkins.io/

[11] https://www.jetbrains.com/teamcity/

[12] http://easycis.aspone.cz/

not applying them. It is worth remarking that we have studied Refactoring practice by looking at the actions actually performed by developers over the history of the considered software projects. Yet, we cannot exclude that developers still employed Refactoring recommendation tools (*e.g.,* JDeodorant (Fokaefs, Tsantalis, Stroulia & Chatzigeorgiou, 2011) or Aries (Bavota, De Lucia, Marcus, Oliveto & Palomba, 2012)) to get suggestions on how to improve source code quality. However, we were interested in understanding how the actual application of Refactoring changes from before to after the adoption of CI. As such, the investigation of whether Refactoring recommendation tools have been used is out of the scope of our paper.

**External threats to validity** concern the generalizability of our results. First, we conducted this study based on a large dataset of 99,545 commits from 39 GitHub projects consistently active during our 48-month observation period. This filtering was required to fit our models and control for *time* variable as well as to avoid biasing our conclusions due to an inflation of zero values in our data. We also made restrictions, since we depend on RefactoringMiner, to Java projects. To our knowledge, current available Refactoring detection tools are dedicated to Java language (Tan & Bockuisch). Moreover, we only-considered Travis CI, the most popular CI service on GitHub (Hilton *et al.*, 2016). These three constraints allowed the statistical investigation of active projects that have introduced CI since years: as such, the results of our study apply to projects having similar characteristics and might therefore be used by developers of those projects to reason about continuous integration has changed the way they apply Refactoring.

We cannot speculate on the validity of our results when considering projects having different characteristics, e.g., non-active projects, or written in different programming languages. Similarly, we would like not to raise opinions on the applicability of the results to software systems following different programming practices. Our future research agenda includes a replication of our study on a different and more varied set of software projects.

## 4.6 Conclusion

We presented in this paper the *first empirical study* that investigates the possible impacts of continuous integration (CI), a quality-driven process, on changing the way developers practice Refactoring. To analyze potential CI impacts, we (1) employed different heuristics estimating Refactoring commits frequency, size and involved developers, (2) used Multiple Regression Analysis (MRA) to estimate CI impacts on Refactoring practice while controlling for different confounding variables and (3) analyzed the change in Refactoring tactics two years before and after the adoption of CI.

Based on data extracted from a sample of 39 GitHub projects deploying CI, our results revealed that the Refactoring change size tends to decrease as recommended. However, the frequency and Refactoring authors tend to drop during the two years following the CI adoption. These findings lend support to previous research efforts claiming the presence of barriers, related especially to lack of time and knowledge, preventing developers from adopting Refactoring techniques/tools in CI context. We believe that software developers need more customized Refactoring tool support in the context of CI to better maintain and evolve their software systems.

Our future work will include extending our study to other open-source and industrial projects from different programming languages and application domains. We also plan to conceive Refactoring tools that can support CI developers in their quality enhancement efforts.

# CHAPTER 5

# PREDICTING CONTINUOUS INTEGRATION BUILD FAILURES USING EVOLUTIONARY SEARCH

Islem Saidani[a] , Ali Ouni[a] , Moatez Chouchen[a] , Mohamed Wiem Mkaouer[b]

[a] Department of Software Engineering and IT, École de Technologie Supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[b] Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester, NY 14623, United States

Paper published in *Information and Software Technology*, August 2020

## Abstract

Continuous Integration (CI) is a common practice in modern software development and it is increasingly adopted in the open-source as well as the software industry markets. CI aims at supporting developers in integrating code changes constantly and quickly through an automated build process. However, in such context, the build process is typically time and resource-consuming which requires a high maintenance effort to avoid build failure. The goal of this study is to introduce an automated approach to cut the expenses of CI build time and provide support tools to developers by predicting the CI build outcome. In this paper, we address problem of CI build failure by introducing a novel search-based approach based on Multi-Objective Genetic Programming (MOGP) to build a CI build failure prediction model. Our approach aims at finding the best combination of CI built features and their appropriate threshold values, based on two conflicting objective functions to deal with both failed and passed builds. We evaluated our approach on a benchmark of 56,019 builds from 10 large-scale and long-lived software projects that use the Travis CI build system. The statistical results reveal that our approach outperforms the state-of-the-art techniques based on machine learning by providing a better balance between both failed and passed builds. Furthermore, we use the generated prediction rules to investigate which factors impact the CI build results, and found that features related to (1) specific statistics about the project such as team size, (2) last build information in the current build and (3) the types of changed files are the most influential to indicate the potential failure of a given build. This paper proposes a multi-objective search-based approach for the problem of CI build failure prediction. The performances of the models developed using our MOGP approach were statistically better than models developed using machine learning techniques. The experimental results show that our approach can effectively reduce both false negative rate and false positive rate of CI build failures in highly imbalanced datasets.

**Keywords.** Continuous Integration, Build Prediction, Multi-Objective Optimization, Search-Based Software Engineering, Machine Learning

## 5.1 Introduction

Continuous integration (CI) (Duvall, Matyas & Glover, 2007b) is a set of software development practices that are widely adopted in industry and open source environments (Vasilescu *et al.*, 2015). A typical CI system, such as Travis CI (CI, 2021), advocates to continuously integrate code changes, introduced by different developers, into a shared repository branch. The key to making this possible, according to Fowler (Fowler, 2006), is automating the process of building and testing, which reduces the cost and risk of delivering defective changes. From the academic side, the study of CI adoption has become an active research topic and it has already been shown that CI improves developers' productivity (Hilton *et al.*, 2016), helps to maintain code quality (Vasilescu *et al.*, 2015) and allows for a higher release frequency (Zhao *et al.*, 2017).

However, despite its valuable benefits, CI brings its own challenges. Hilton et al. (Hilton *et al.*, 2017) revealed that build failure is a major barrier that developers face when using CI. A build failure, *i.e.,* failing to compile the software into machine executable code, represents a blocker that prevents developers from proceeding further with development, as it requires an immediate action to resolve it. In addition, the build resolution may take hours or even days to complete, which severely affects both, the speed of software development and the productivity of developers (Abdalkareem *et al.*, 2019). Such challenges motivated researchers and practitioners to develop techniques for preemptively detecting when a software state is most likely to trigger a failure when built, and thus developers can take the necessary preventive actions to avoid it.

Existing studies leverage the history of previous build success and failures in order to train machine learning (ML) models. Such models learn from the CI builds history and use the domain knowledge to extract features and predict the outcome of a given input build. For instance, Foyzul and Wang (Hassan & Wang, 2017) used Random Forest (RF), for the binary classification of build outcome, and Ni and Li (Ni & Li, 2017) adapted the cascaded classifiers to improve the accuracy of CI build prediction. Although these works have advocated that predicting CI build outcome is possible and beneficial, none of them accommodated for the imbalanced distribution of the successful and failed classes when building their prediction models. This challenges their applicability due to the performance bias that can occur when an imbalanced distribution of class examples is used in the learning process (Bhowan, Johnston & Zhang, 2011; Bhowan, Zhang & Johnston, 2010; Bhowan, Johnston, Zhang & Yao, 2013; Saidani, Ouni, Chouchen & Mkaouer, 2020c). Hence, the minority class instances, *i.e.,* the failed builds class in our case, is much more likely to be miss-classified. However, in CI context, a good accuracy on the failed builds prediction is more important than the passed builds accuracy. Also, increasing the accuracy of the builds failure class (known as probability of detection) can result in maximizing also the number of incorrectly classified failed builds (*i.e.,* false alarms) which makes these two objectives in conflict (Malhotra & Khanna, 2017; Bhowan *et al.*, 2011).

To deal with the above mentioned challenges, Evolutionary Multi-Objective Optimization (EMO) (Harman *et al.*, 2012; Nam, Fu, Kim, Menzies & Tan, 2017; Ouni, Kessentini, Sahraoui & Boukadoum, 2013; Chen, Nair, Krishna & Menzies, 2018; Kessentini & Ouni, 2017) have been found useful for developing software engineering predictive models (Eckart, Marco & Lothar, 2001; Jin & Sendhoff, 2008). Researchers have advocated that the use of (EMO) is appropriate because it allows adapting the fitness function to evolve classifiers with good classification ability across both the minority and majority classes, *e.g*, balance between failed and passed builds. This is accomplished by treating the conflicting objectives independently in the learning process using the notion of *Pareto Dominance*. Additionally, to deal with the imbalanced nature of the dataset, a Multi-Objective Genetic Programming (MOGP) approach (Zhao, 2007), that promotes diversity between solutions equally on both minority and majority classes, allows the imbalanced training data to be used directly in the learning process *i.e.,* without relying on sampling techniques to re-balance the data (Bhowan, Johnston, Zhang & Yao, 2012; Bhowan *et al.*, 2013) which advocates that MOGP approaches are more suitable for binary classification tasks with imbalanced data (Bhowan *et al.*, 2011).

In this paper, we introduce a novel MOGP approach to predict CI build outcome. The idea is based on the adaption of the Non-dominated Sorting Genetic Algorithm (NSGA-II) (Deb *et al.*, 2002) with a tree-based solution representation, in order to generate rules from historical data of CI builds using two competing objectives in the learning process, namely the probability of detection and the probability of false alarms. As a solution to this binary classification problem, a candidate rule is expressed as a combination of metrics and their appropriate threshold values; and should cover as much as possible the build results from the base of build results. In a nutshell, our approach takes as input, a given build, calculates a set a metrics that are fed into our rule, previously generated using the history of builds, and whose binary output predicts whether the input build is most likely to succeed or fail, based on its likelihood to the successful or failed builds.

To evaluate our approach, we conducted an empirical study on a benchmark composed of 56,019 build instances from 10 open source projects that use the Travis CI system, one of the most popular CI systems. We compare our predictive performance to existing Genetic Programming (GP) algorithms and three widely-used ML techniques namely Random Forest, Decision Tree and Naive Bayes. The statistical results reveal that our approach advances the state-of-the art by outperforming existing prediction models. Moreover, we examine the most important features, used by our generated rules, in indicating the correct CI build outcome, in order to provide the practitioners with useful insights on how to avoid build failures. In summary, the contributions of this work are the following:

- A novel formulation of the CI build prediction as a multi-objective optimization problem to handle imbalance nature of CI builds as well as to achieve a good predictive performance on both classes (passed and failed). To the best of our knowledge, this is the first attempt to use a search-based approach for the CI build prediction.

- An empirical study of our MOGP technique compared to different existing approaches based on a benchmark of 10 large and long-lived projects. The obtained results reveal that our proposal is more efficient than existing techniques with a median of AUC (Area Under The Curve) of 68% compared to 61% achieved by existing ML techniques for which we applied re-sampling. Additionally, our approach is able to strike a better balance between both failed and passed builds achieving an improvement of at least 15% for the balance metric (Malhotra, 2015). These are interesting and actionable results considering the highly imbalanced nature of the studied projects with an average failure rate of 19% in the minority class.
- A qualitative evidence of the potential reasons behind build failure through a novel feature ranking approach. The rules analysis shows that the metrics related to (1) specific statistics about the project such as team size, (2) last build information in the current build and (3) the types of changed files are the most influential to indicate the potential failure of a given build.
- A comprehensive dataset (Saidani, 2020d) collected from 10 long-lived software projects, containing over 56,019 records of build results.

**Replication Package.** The comprehensive dataset collected and used in our study is publicly available in (Saidani, 2020d) for future replications and extensions. Also, we provide all details about the validation results as well as illustrative examples of the generated rules available for the research community.

**Paper Organization.** The remainder of this paper is organized as follows. We present our approach in section 5.2. Section 5.3 shows the experimental setup of our empirical study. Section 5.4 presents the results and findings of our studied research questions. Section 5.5 discusses the implications of our findings for developers, researchers and tool builders. Section 5.6 reviews the threats to the validity of our results. Finally, Section 5.7 concludes the paper and outlines avenues for future work.

## 5.2 Search-based Prediction of CI build failure

In this section, we describe our approach that uses multi-objective GP based on an adaptation of NSGA-II.

### 5.2.1 Approach Overview

Figure 5.1 provides an overview of our proposed approach to generate rules for CI builds outcome prediction. In our study, we start from the observation that it is more beneficial for CI developers to identify good practices to follow in order to avoid build failures rather than simply detecting whether the build will succeed or fail. Thus, the *goal* of the proposed approach is to generate a set of rules, as a combination of CI-related metrics extracted from various sets of information about CI builds. As described in Figure 5.1, the first step of our approach consists of collecting a set of examples of build results (failed and succeeded builds) information based CI-related (cf. Section 5.2.3). Then, in the second step, we take these inputs to generate a set of predictive rules that predict as much as possible the CI builds outcome with high accuracy.

The multi-objective GP algorithm is the key element of our approach. First, it starts by generating a set of solutions. Every solution is composed of a set of prediction rules *i.e.,* combination of threshold values assigned to each metric. These combination of metrics-thresholds are connected with logical operators. All the generated solutions in the population are evaluated using two objectives to (1) maximize the true positive rate, and (2) minimize the false positive rate. Change operators are applied, at every iteration, to generate new solutions. After repeating this process until reaching a stop criteria, the best solution is returned by the algorithm. In our experiments, the stop criteria is when reaching a maximum number of generations. All parameters configuration details are described later in Section 5.3.5.

### 5.2.2 NSGA-II adaptation

In this section, we describe in details our search-based approach.

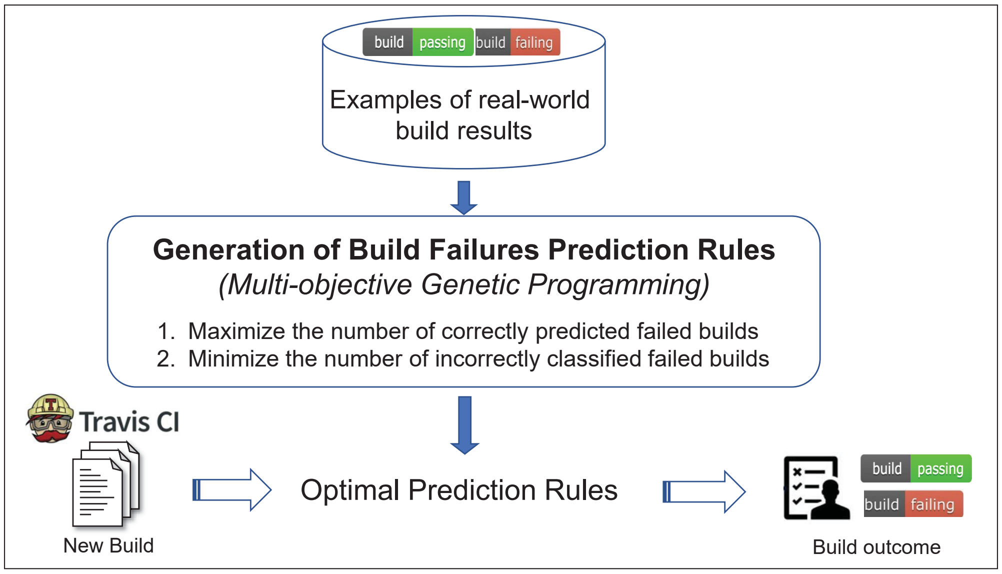

build passing
build failing
Examples of real-world build results
Generation of Build Failures Prediction Rules
(Multi-objective Genetic Programming)
1. Maximize the number of correctly predicted failed builds
2. Minimize the number of incorrectly classified failed builds
Travis CI
New Build
Optimal Prediction Rules
build passing
build failing
Build outcome

Figure 5.1 An overview of our approach

The following three subsections describe more precisely our adaption of GP to the CI build failure problem.

**i. Solution/Individual representation:** Our adaptation to the NSGA-II algorithm is to adopt it with the generic model of GP learning to the space of programs. Unlike other evolutionary search algorithms, in GP, solutions are themselves programs following a tree-like representation instead of fixed length linear string formed from a limited alphabet of symbols (Koza & Koza, 1992). For the build failures prediction problem, a candidate solution, *i.e.,* a prediction rule, is represented as an IF – THEN clause with the following template:

**IF** (Combination of metrics and their thresholds) **THEN** RESULT.

The IF clause describes the conditions under which a build is said to be succeeded or failed. The condition corresponds to a logical expression that combines some metrics and their threshold values using logical operators (OR, AND). A solution is encoded as a tree where each terminal belongs to the set of metrics described in Table 5.1 and their corresponding thresholds are generated randomly. Each internal-node belongs to the connective set C = {AND, OR}. Figure 5.2 shows an illustrative example of a solution.

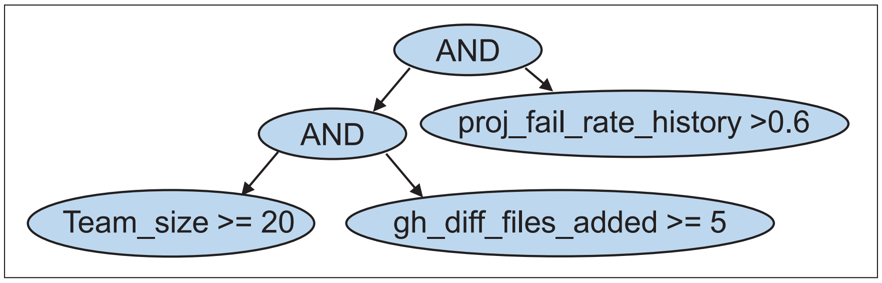


Figure 5.2 A simplified example of solution encoding for CI build failure prediction

This rule predicts the build failure in case the *fail rate history* is greater than 0.6 and the *files added* are higher than or equal to 5 and the *team size* is higher than or equals to 20.

**IF** proj_fail_rate_history > 0.6 AND team_size ≥ 20 AND gh_diff_files_added ≥ 5
**THEN** Failure.

**ii. Generation of an initial population:** To generate an initial population composed of *n* solutions, we start by defining the maximum tree length (should not exceed a predefined threshold). The actual tree length will vary with the number of metrics to use for failure prediction that vary from 1 to 33 (the number of considered metrics, cf. Table 5.1). Notice that a high tree length value does not necessarily mean that the results are more precise since, usually, only a few metrics are needed to predict the failure. Because the individuals will evolve with different tree lengths (structures), with the root (head) of the trees unchanged, we randomly assign for each one:

- One metric and threshold value to each leaf node. The threshold values are ranged between lower and upper bounds of the metric in question (*e.g*, if the number of team sizes is between 1 and 10, the threshold will be randomly selected according this metric distribution). These upper bounds are fixed based on the training set. We also assign a mathematical operator (≥, ≤, =) that depends on the metric category. Note that "=" is only used for categorical metrics (*e.g*, gh_is_pr), ≥ and ≤ are applied only with continuous (*e.g*, committer_fail_history) or discrete metrics (*e.g*, gh_team_size).
- A logic operator (AND, OR) to each function node.

It is worth to mention that during individual generation or evolution, the infeasible rules that contain nodes with the a condition and its negation in the same sub-tree like for example "*gh_is_pr = 1 AND gh_is_pr = 0*" are automatically rejected.

**iii. Genetic operators:** Crossover and mutation are defined as follows.

*Crossover:* is used to combine the genetic information of two parents. In this adaptation, we use single-point crossover operator. A sub-tree is extracted from each parent. Then, the crossover operator exchanges the nodes and their relative sub-trees between parents. Figure 5.3 shows an example of the crossover process. In fact, two parent solutions, namely P1 and P2, are combined to generate two new child solutions. The right sub-tree of P1 is swapped with the left sub-tree of P2.

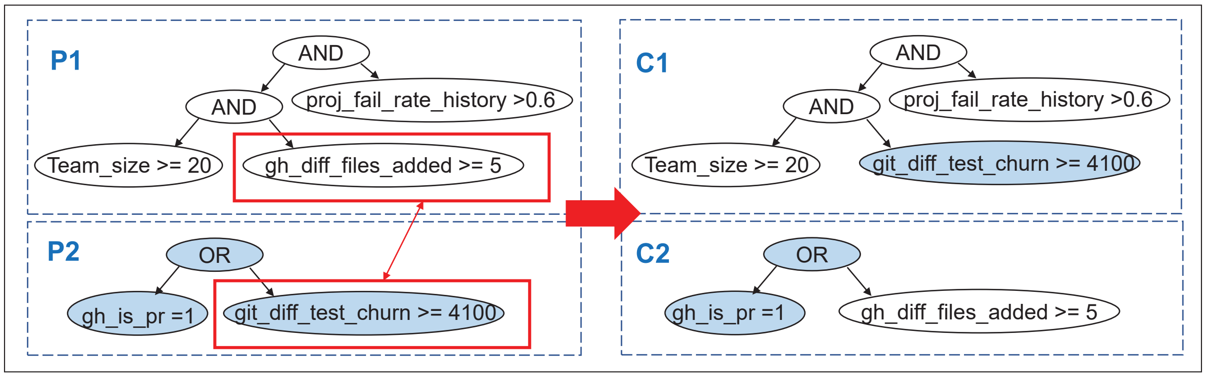


Figure 5.3 An example of crossover operator

For example, after applying the crossover operator the new rule C2 to predict build failure will be:

**IF** gh_is_pr = 1 OR gh_diff_files_added ≥ 5 **THEN** Failure.

*Mutation:* it can be applied either to a function node or a terminal node. In this problem, the mutation operator first randomly selects a node in a randomly selected tree. Then, if the selected node is a terminal, it is replaced by another terminal (metric or another threshold value). If the selected node is a function (logical operators), it is replaced by a new function (*e.g*, OR becomes

AND). Then, the node and its sub-tree are replaced by a new randomly generated sub-tree. To illustrate the mutation process, consider again the example that corresponds to a candidate rule to predict CI build failure. Figure 5.4 illustrates the effect of a mutation that deletes the note containing proj_fail_rate_history feature, leading to the automatic deletion of node AND (no left sub-tree).

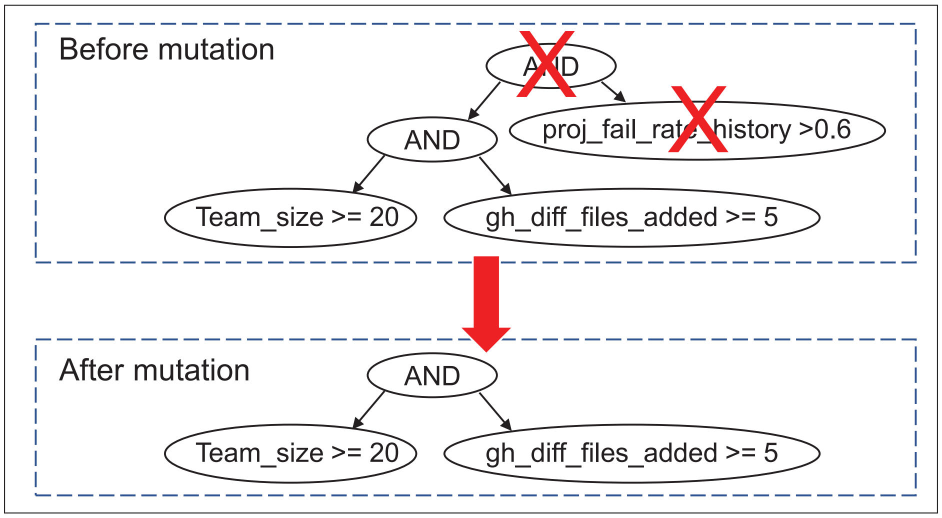


Figure 5.4 An example of mutation operator

Thus, after applying the mutation operator the new rule will be:

**IF** team_size ≥ 20 OR gh_diff_files_added ≥ 5 **THEN** Failure.

**iii. Multi-criteria solution evaluation (fitness function):** An appropriate fitness function should be defined to evaluate how good is a candidate solution. According to Harman and Clark (Harman & Clark, 2004), search-based algorithms used from prediction can use performance measures to identify better solutions in the search process. To evaluate the fitness of each solution, we use two objective functions to be optimized, based on two well-known metrics, the true positive rate and false positive rates (Malhotra & Khanna, 2017):

- (1) Maximize the True Positive Rate (TPR), also known as the probability of detection (PD). PD is an indicator of the percentage of builds that are correctly classified as failed. The higher the value of PD, the better is the solution.

$$PD = \frac{TP}{TP + FN} \times 100$$

where TP and FN are the number of true positives and the number of false positives, respectively.

- (2) Minimize the False Positive Rate (FPR), also known as probability of false alarm (FP), which is the ratio of false positives (*i.e.,* incorrectly classified failed builds) to the actual number of passed builds. The lower the value of PF, the better is the solution.

$$PF = \frac{FP}{FP + TN} \times 100$$

where FP and TN are the number of false positives and the number of true negatives, respectively.

**iv. Pareto-front selection:** Multi-objective algorithms such as NSGA-II do not produce a single solution like GA, but a set of non-dominated solutions called Pareto-optimal solutions. These solutions provide a trade-off between the prediction accuracy of both failed and passed build classes. In the CI built prediction problem, the best solutions are those who represent the Pareto-front that maximize the TPR and minimize the FPR. Hence a solution is chosen based on its preferences in terms of trade-off. To this end, and in order to fully automate our approach, we extract a single default best solution from the returned set of solutions. Since in our case the ideal solution (True Pareto) has the best TPR value (equals to 1) and the best FPR value (equals to 0), we select the nearest solution to the ideal one in terms of Euclidean distance. The following equation is used to choose the solution (noted *BestSol*) (Ouni, Kessentini, Sahraoui & Hamdi, 2012; Ouni, Kessentini, Inoue & Cinnéide, 2017) that corresponds of the best compromise between TPR and FPR:

$$BestSol = \min_{i=1}^{n} \sqrt{(1 - TPR[i])^2 + FPR[i]^2}$$

where $n$ is the number of solutions in the Pareto front returned by NSGA-II.

### 5.2.3 Dataset and CI-related Metrics

To collect our data, we use TravisTorrent (Beller, Gousios & Zaidman, 2017), which is a publicly available dataset that contains information about Travis-CI builds of several projects hosted in GitHub. By combining the data from Travis-CI and GitHub, detailed features, *i.e.,* metrics can be extracted and used for predictions (Xia *et al.*, 2017a; Ni & Li, 2017; Xia & Li, 2017; Luo *et al.*, 2017; Xie & Li, 2018). Table 5.1 lists the build metrics used to generate our prediction rules. Besides the existing TravisTorrent features (marked as $T$ in the third column), we also generated other features marked as $G$ which were extracted from existing research. During feature selection, we considered 10 categories described as follows:

- **Change size.** These features measure how the change made is distributed across the different aspects, including the commits and code.
- **Files change.** These features compute the changes (deletion, addition or modification) at the file level.
- **Cooperation.** These metrics estimate the level of cooperation in terms of comments and code revisions.
- **Triggering Commit.** In this group, we collect some information about the commit that triggered the build, to know whether the build is managed by a core member or as part of pull requests which may increase the risk of breaking the build. We are also interested in collecting other temporal factors such as the day of the week.
- **Change Type.** In this group, we count different types of files changed in built commits using file extensions. The changes may be related to source, documentation, configuration or other files.

- **Test Change.** These features measure the test changes which represent additional indicators on the quality of the build code.
- **Link to last build.** This set of features estimates the project's stability which may lead to a better prediction.
- **Committer experience.** These metrics estimate the committer experience related mainly to the number of passed/failed builds that may reflect her/his level of experience.
- **Project statistics.** This group of features captures some additional information about the committer and the project experience which may indicate the quality of the current build.
- **Test Density.** This set of features is dedicated to estimate the project familiarity with testing, one of the core goals of CI (Fowler, 2006).

By using these metrics, we collected a total of 56,019 records of build results. However, it is worth mentioning that some builds were filtered out from the original dataset since no information about the last build was found. Additionally, since TravisTorrent dataset organizes the build results at the job level, we aggregate the results of all jobs related to a build and provide one outcome using the build identifier in the TravisTorrent dataset. This is required to avoid biasing our results due to duplicated builds. Also, we eliminated builds that have a status of "Error" or "Cancel" from our dataset since we only focus on builds that have a "pass" or "fail" status. For a broader public for reproducibility and extension, we provide our data available (Saidani, 2020d).

## 5.3 Validation

In this section, we report the results of a large-scale empirical study on a benchmark of 56,019 build instances. The comprehensive dataset collected and used in our study is publicly available in (Saidani, 2020d) for future replications and extensions.

Figure 5.5 provides an overview of our experimental design used in the validation of our approach. First, we evaluate our predictive performance against existing approaches in the two first questions. At this step, we run search-based algorithms and non deterministic ML

Table 5.1 CI-related Metrics extracted from literature

| Category | Metric | Source | Description | Reference |
|---|---|---|---|---|
| **Change size** | git_num_all_built_commits | T | # of commits contained in this single build | (Xia *et al.*, 2017a),(Ni & Li, 2017) |
| | gh_num_commits_on_files_touched | T | # of unique commits on the files touched in the built commits | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| | git_diff_src_churn | T | # of lines of code changed in all built commits | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| **Files change** | gh_diff_files_added | T | # of files added in all built commits | (Xia *et al.*, 2017a),(Ni & Li, 2017),(Xia & Li, 2017) |
| | gh_diff_files_deleted | T | # of files deleted by all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| | gh_diff_files_modified | T | # of files modified by all built commits | (Xia *et al.*, 2017a),(Ni & Li, 2017),(Xia & Li, 2017) |
| **Cooperation** | gh_num_commit_comments | T | # of comments of all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| | num_of_distinct_authors | T | # of distinct authors in all built commits | (Xia *et al.*, 2017a),(Xie & Li, 2018) |
| | Total_Number_Of_Revisions | G | # of revisions on all the files touched by the current build | (Xie & Li, 2018) |
| **Triggering commit** | gh_by_core_team_member | T | Whether the commit that has triggered the build was authored by a core team member | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| | gh_is_pr | T | Whether this build was triggered as part of a pull request on GitHub. | (Luo *et al.*, 2017) |
| | day_week | G | Day of week of the first commit for the build | (Hassan & Wang, 2017) |
| **Change type** | gh_diff_src_files | T | # of src files changed by all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017) |
| | gh_diff_doc_files | T | # of documentation files changed by all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| | gh_diff_other_files | T | # of files which are neither source code nor documentation. | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| | num_config_files | G | # of configuration files (*.xml, *.yml, etc) edited in this commit. | (Ni & Li, 2017),(Hassan & Wang, 2017) |
| **Test Change** | git_diff_test_churn | T | # of lines of test code changed in all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| | gh_diff_tests_added | T | # of test cases added in all built commits | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| | gh_diff_tests_deleted | T | # of test cases deleted in all built commits | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| **Link to last build** | prev_built_result | G | Result of last build | (Ni & Li, 2017), (Hassan & Wang, 2017),(Xie & Li, 2018) |
| | same_committer | T | Indicates whether the committer is the same as last build | (Ni & Li, 2017) |
| | elapsed_days_last_build | T | Counts the days since last build | (Ni & Li, 2017) |
| | git_prev_commit_resolution_status | T | = it could be "build found", "merge found" or "no previous build" | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| **Committer Experience** | committer_fail_history | G | The fail rate of the builds by the current committer in the past | (Ni & Li, 2017) |
| | committer_fail_recent | G | Similar to committer history, but measuring only his last five builds | (Ni & Li, 2017) |
| | committers_avg_exp | G | The average number of builds the committers made in the project before this build | (Ni & Li, 2017) |
| **Project History** | project_fail_history | G | The fail rate of the all the project's previous build | (Ni & Li, 2017) |
| | project_fail_recent | G | Similar to project fail history but using only last five builds | (Ni & Li, 2017) |
| | gh_team_size | T | # of developers that committed from the moment the build was triggered and 3 months back. | (Xia & Li, 2017),(Luo *et al.*, 2017),(Hassan & Wang, 2017) |
| | gh_sloc | T | # of source lines of code, in the entire repository at the time of this build. | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| **Test Density** | gh_test_lines_per_kloc | T | # of lines in test cases per 1000 gh_sloc. | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| | gh_test_cases_per_kloc | T | # of test cases per 1000 gh_sloc. | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| | gh_asserts_cases_per_kloc | T | # of assertions per 1000 gh_sloc. | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |

T: TravisTorrent, G: Generated

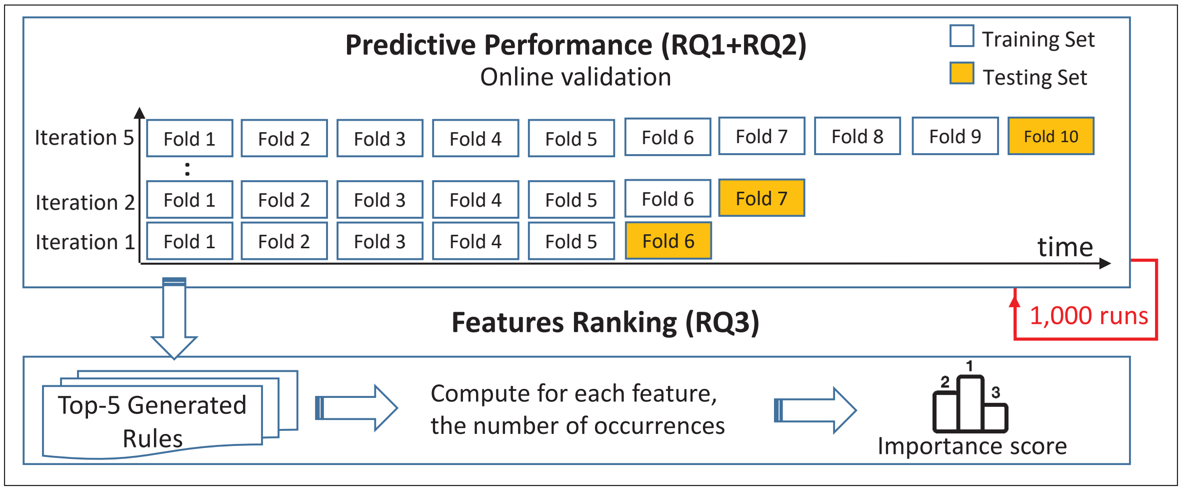


Figure 5.5 Experimental design

techniques used in this empirical study 1,000 times to deal with the stochastic nature of these algorithms. To validate the predictive performance, we consider online validation (Xia & Li, 2017). Next step in this validation is related to a qualitative study of the most important metrics to indicate CI build outcome. In the following, we describe each step in detail.

### 5.3.1 Research Questions

We designed our experiments to answer three research questions:

- **RQ1. (SBSE validation)** How does the proposed NSGA-II perform compared to Random Search (RS), mono-objective algorithm (GA) and other Multi-Objective algorithms?
- **RQ2. (Performance evaluation with ML)** How does our approach perform compared to ML techniques?
- **RQ3. (Features analysis)** What features are most important to predict CI build failures?

### 5.3.2 Analysis method

#### 5.3.2.1 Prediction performance

The first *goal* of our empirical study is to evaluate the performance of our approach for the CI build failure prediction problem compared to existing techniques (RQ1+RQ2).

**RQ1** is a “standard" question asked in any Search-Based Software Engineering (SBSE) formulation (Harman & Jones, 2001). First, we compare our SBSE formulation against Random Search (RS) (Harman *et al.*, 2010; Karnopp, 1963) is the simplest form of search-based algorithms. It may fail to find optimal solutions that occupy small proportion of the overall search as it is unguided without efficient use of genetic operators (Harman *et al.*, 2010). In this RQ, we aim in the first place as a *sanity check* to evaluate the need for an intelligent method such as NSGA-II that can outperform RS. In addition, it is important also to determine if considering separate conflicting objectives to be optimized (multi-objective) is an appropriate formulation compared to aggregating them in a single objective. Hence, we compared NSGA-II to mono-objective GP where a single fitness function, *Fit(mono)*, is used. *Fit(mono)* is defined as follows:

$$Fit(mono) = \frac{PD + (1 - PF)}{2} \tag{5.1}$$

In order, to make our results comparable, we compute the well-known evaluation metric Area Under the ROC Curve (**AUC**). This measure indicates how much a prediction model/rule is capable of distinguishing between classes. A larger AUC value indicates better prediction performance. For binary classification, AUC is defined as follows (Cervantes, Li & Yu, 2013):

$$AUC = \frac{1 + \frac{PD}{100} - \frac{PF}{100}}{2} \in [0, 1] \tag{5.2}$$

Moreover, it is important to account for imbalance in a data set. Indeed, various researchers (Li, Zhang, Wu & Zhou, 2012; Menzies, Greenwald & Frank, 2006; Malhotra, 2015) advocate the use of the **balance** metric to assess the performance of models that were initially trained using imbalanced training data. Balance measure computes the Euclidean distance between the optimum couple (PD=100, PF=0) to a specific pair of (PD, PF) (Menzies *et al.*, 2006). Higher balances are desirable for a model. The balance metric is defined as follows.

$$Balance = 1 - \sqrt{\frac{(0 - \frac{PF}{100})^2 * (1 - \frac{PD}{100})^2}{2}} \in [0, 1] \tag{5.3}$$

The main merit of the AUC and balance is their robustness toward imbalanced data.

#### 5.3.2.2 Algorithms performance

We evaluate the performance of NSGA-II over other MOEAs to identify the most effective algorithm in multi-objective optimization. Thus, we compare our approach with NSGA-III (Deb & Jain, 2013), Indicator-Based Evolutionary Algorithm (IBEA) (di Pierro, Khu & Savic, 2007) and Strength-Pareto Evolutionary Algorithm (SPEA2) (Zitzler *et al.*, 2001), as they are among the most popular MOEAs and have been widely utilized in SBSE (Harman *et al.*, 2012,?; Harman, 2007; Ouni, 2020; Saidani *et al.*, 2020c; Mkaouer *et al.*, 2015). Additionally, all the search-based algorithms used in this paper are implemented using the MOEA framework (Hadka, 2013), an open source framework for developing and experimenting with MOEAs (Hadka, 2014).

Since the underlying goal of MOEAs is to determine a set of alternative solutions known as Pareto front approximations (Hadka, 2014), we aim to compare the performance of each algorithm using Zitzler et al. (Zitzler, Thiele, Laumanns, Fonseca & Da Fonseca, 2003) measures, based on three different performance aspects for multi-objective optimization (1) the quality of the generated Pareto fronts, (2) the convergence to the exact Pareto front, and (3) the diversity of the produced solutions. In particular, we consider the following metrics :

- **Hyper-volume (HV):** calculates the volume of the space dominated by all the solutions *i.e.,* convergence of a solution set. A larger HV value indicates better performance. This metric is widely accepted as it guarantees that any approximation set that achieves more HV value for a particular MOP, it should contain more Pareto optimal solutions (Riquelme, Von Lücken & Baran, 2015).

- **Generational Distance (GD):** measures the average distance between each Pareto front solution and the true Pareto front. Smaller is GD, better is the MOEA *i.e.,* closer it is from the Pareto optimal. This metric occupied the second position, after HV, of the most used MOGP performance metrics (Riquelme *et al.*, 2015).
- **Spacing (SP):** is the most popular uniformity indicator (Li & Yao, 2019). It measures the standard deviation of distance from each solution to its closest neighbor in the obtained set. A lower SP value is preferable as it indicates that the solution provides a better Pareto front representation and hence it can be considered to possess better quality.

These indicators are automatically computed, on the testing set, using the MOEA Framework tool which provides the statistical analysis and displays the minimum, median and maximum values of each performance indicator.

To answer **RQ2**, we compare the prediction performance of NSGA-II with three widely-used ML techniques in previous CI and software engineering research (Xia *et al.*, 2017a; Ni & Li, 2018; Xia & Li, 2017; Luo *et al.*, 2017; Hassan & Wang, 2017; Ni & Li, 2018; Santolucito, Zhang, Zhai & Piskac, 2018), namely Decision Tree (DT), Random Forest (RF) and Naive Bayesian (NB). We use both prediction metrics, *balance* and *AUC*, as described for RQ1.

**ML preprocessing:** First, data scaling is performed in order to standardize the range of variables. Then we rely on Synthetic Minority Oversampling Technique (SMOTE) method (Chawla, Bowyer, Hall & Kegelmeyer, 2002), to re-sample the training data. Note, that we did not re-sample the testing dataset since we want to evaluate ML techniques in a real-life scenario, where the data is imbalanced.

**Validation scenario:** We conduct an online validation in which builds are ordered and predicted chronologically. Similar to prior work (Xia & Li, 2017), we ranked for each selected project, the builds according to its start time and broke the whole set of a given project into ten folds.

Then, we used the latter five folds as testing sets: At each iteration i ($1 \leq i \leq 5$), the test set fold j ($6 \leq j \leq 10$), the former j-1 folds were selected as training set to train the model. It is worthy to mention, that we verified for each project and validation iteration, the existence of failed builds. To get more details about the failure rate in each validation iteration, please consider our replication package (Saidani, 2020d).

#### 5.3.2.3 Feature Ranking

The goal of **RQ3** is to analyze the factors influencing build failures which will be valuable for developers to prevent potential build failures in their projects. While existing research works (Rausch *et al.*, 2017; Beller *et al.*, 2017; Luo *et al.*, 2017) attempted to give insights into CI build failure by applying correlation analysis to discover the relationship between the selected features and the build outcome. In this paper, we address this problem by exploring the interpretable knowledge provided by our generated rules. Since we use online validation, the analysis produces 5 rules for each project. Thus, the same feature may occur multiple times in the near-optimal rules. The higher the number of occurrences of a feature, the more important is the feature in identifying failed builds. In addition, to give a more general view, we aggregate the results of features ranking for each project and feature category (cf. Section 5.2.3).

### 5.3.3 Subjects Selection

Our experiments are based on TravisTorrent dataset [1], from which we selected top-10 Java and Ruby, the only supported languages in this dataset (Beller *et al.*, 2017), projects according to the number of build records (after removing inadequate rows as described in Section 5.2.3). An overview about the studied projects is reported in Table 5.2. It is noteworthy that the data in all these projects is highly imbalanced. Our replication package is publicly available at (Saidani, 2020d).

[1] https://travistorrent.testroots.org/

Table 5.2 Studies projects statistics

| **Project Name** | **Language** | **# of Builds** | **Failure Rate** | **Age at CI (days)** |
|---|---|---|---|---|
| *CloudifySource/cloudify* | java | 4,568 | 0.25 | 220 |
| *gradle/gradle* | java | 3,822 | 0.08 | 1,833 |
| *Graylog2/graylog2-server* | java | 3,341 | 0.12 | 470 |
| *mitchellh/vagrant* | ruby | 3,569 | 0.14 | 765 |
| *openMF/mifosx* | java | 2,252 | 0.07 | 2 |
| *opf/openproject* | ruby | 5,913 | 0.35 | 287 |
| *rails/rails* | ruby | 11,732 | 0.30 | 2,354 |
| *rapid7/metasploit-framework* | ruby | 6,391 | 0.07 | 2,571 |
| *ruby/ruby* | ruby | 11,814 | 0.21 | 5,099 |
| *SonarSource/sonarqube* | java | 2,317 | 0.24 | 1,013 |
| **Average** | – | 5,602 | 0.19 | 1,461 |

Cloudify[2] is a cloud-enablement platform that on-boards applications to public and private clouds without architectural or code changes. Gradle[3] is a popular build tool with a focus on build automation and support for multi-language development. It offers a flexible model that can support the entire development lifecycle from compiling and packaging code to publishing web sites. Graylog2-server[4] is an open source log management system that centrally captures, stores, and enables real-time search and log analysis against terabytes of machine data from different component in the IT infrastructure. Vagrant[5] is a tool for building and distributing development environments that provides easy workflow for developers and leverages a declarative configuration file which describes all software requirements, packages, operating system configuration, users, and so on. Mifosx[6] is an open technology platform for financial inclusion that provides core functionalities to deliver financial services. OpenProject[7] is one of

[2] https://github.com/CloudifySource/cloudify

[3] https://github.com/gradle/gradle

[4] https://github.com/Graylog2/graylog2-server

[5] https://github.com/hashicorp/vagrant

[6] https://github.com/openMF/mifosx

[7] https://github.com/opf/openproject

the leading open source web-based project management systems. Rails[8] is a web application framework that provides several features needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. Metasploit[9] is a penetration testing platform that enables to write, test, and execute exploit code with a suite of tools to test security vulnerabilities, enumerate networks, execute attacks, and evade detection. Ruby[10] is an interpreted object-oriented programming language often used for web development. Finally, SonarQube[11] is a platform for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities on several programming languages.

### 5.3.4 Inferential Statistical Test methods Used

When applied to the same problem instance, search-based algorithms, DT and RF techniques may provide different results on each run. To deal with this stochastic nature, it is important to assess their effectiveness by performing several runs, at least 1,000 runs as suggested by Arcury and Briand guidelines (Arcuri & Briand, 2011) as well as recent works (Zhang, Harman, Ochoa, Ruhe & Brinkkemper, 2018; Paixao, Harman, Zhang & Yu, 2017; Ferrucci, Harman, Ren & Sarro, 2013; Almarimi *et al.*, 2019; Ouni *et al.*, 2016; Boukharata, Ouni, Kessentini, Bouktif & Wang, 2019; Mkaouer *et al.*, 2015). In addition, it is also essential to use the statistical tests that provide support for/rejection of the conclusions derived by analyzing the obtained results. In this paper, we employ Wilcoxon signed rank test (Wilcoxon, Katti & Wilcox, 1970) in order to detect significant performance differences between the algorithms under comparison ($\alpha$ is set at 0.05). In this validation, each iteration is repeated 1,000 times, for each algorithm and each project. It is worth mentioning that for RQ3, we choose the rule with the median value through 3,000 runs of each iteration.

[8] https://github.com/rails/rails

[9] https://github.com/rapid7/metasploit-framework

[10] https://github.com/ruby/ruby

[11] https://github.com/SonarSource/sonarqube

We also use Vargha-Delaney A (VDA) (Vargha & Delaney, 2000), a non-parametric effect size measure which is widely used in SBSE (Nejati & Gay, 2019). The A measure indicates the probability that one technique will achieve better performance than another technique. When the A measure is 0.5, the two techniques are equal. When the A measure is above or below 0.5, one of the techniques outperforms the other (Thomas, Hemmati, Hassan & Blostein, 2014). Vargha-Delaney statistic also classifies the magnitude of the obtained effect size value into four different levels (*negligible, small, medium, and large*) (Scalabrino, Grano, Di Nucci, Oliveto & De Lucia, 2016).

### 5.3.5 Parameter Tuning and Setting

First, we investigated a number of calibration of different parameters in order to effectively set the parameters of each technique used in the study. To facilitate the replication of our results, we report in Table 5.3 our algorithmic parameter tuning. The initial populations of all the search-based algorithms were randomly generated. The process is stopped when the maximum number of generations, set to 500, is reached. The maximum depth of the tree (*i.e.,* rules) is fixed to 10.

Table 5.3 Algorithms parameters

| **Algorithms** | **Parameters** | **Values** |
|---|---|---|
| NSGA-II, NSGA-III IBEA, SPEA2, GA, RS | Population size | 100 |
| | Maximum number of generations | 500 |
| | Maximum depth of the tree | 10 |
| | Crossover probability * | 0.9 |
| | Mutation probability * | 0.1 |
| RF | Maximum depth of the tree | 10 |
| | Number of estimators | 200 |
| DT | Maximum depth of the tree | 10 |
| NB | Used NB classifier | Gaussian naive Bayes |

* Not applied to RS

The three ML techniques analyzed in the study are DT, RF and NB. The parameter settings for DT method include maximum depth of 10. RF's parameter setting involves using a maximum tree depth of 10 and number of estimators of 200. For NB classifier selection, we use Gaussian Naive Bayesian (John & Langley, 2013) as the majority of the handled data is continuous.

## 5.4 Experimental results

This section presents the experimental results obtained for RQ1-3.

### 5.4.1 RQ1. Results for GP comparison

In this RQ, we report the results comparing the performance of NSGA-II the other search-based technique in order to determine the most effective GP technique for CI build prediction. Figure 5.6 plots the results while Table 5.4 highlights the statistical tests results of this comparison.

As shown in Figure 5.6, we clearly see that NSGA-II outperformed RS as well as GA and this by an increase of 17% and 35% in terms of AUC and balance respectively. In fact, both mono-objective algorithms achieved a median score of 31% in terms of balance, while GA was slightly better in terms of AUC with a score of 51% compared to 50% achieved by RS. Additionally, the Wilcoxon test results showed that over 50,000 experiment instances (5 iterations $\times$ 1,000 runs $\times$ 10 projects), NSGA-II was significantly better than GA and RS, with large VDA effect sizes. This provides evidence that the use of multi-objective formulation for the prediction problem is more suited as it can provide a better compromise between PD and PF.

With regards to other MOEAs, NSGA-II was the best in terms of AUC in all the studied projects while it showed better predictive performance in nine out of ten projects in terms of balance. Overall, the statistical tests results reveal that NSGA-II is significantly the best among other MOEAs with small effect sizes. Next, we compare the performance of multi-objective optimization for the different MOEAs. Table 5.5 shows the results of MOEAs comparisons based on the hyper-volume (HV), Generational Distance (GD) and Spacing (SP) as described in Section 5.3.2.1. The experiment shows that, in median, NSGA-II was significantly the best in terms of HV, GD and SP. In fact, NSGA-II achieved a median score of 0.99 in terms of HV, while the other algorithms achieved 0.96 which means that NSGA-II is better to cover the volume of the space dominated by its solutions. In terms of GD, NSGA-II is also better to achieve a closer distance between its Pareto front solutions and the true Pareto front with a score of $4 * 10^{-3}$ compared to 0.01 for NSGA-III, SPEA2 and IBEA. Regarding SP, NSGA-II achieves also the best spacing between the generated solutions with median SP score of 0.05. Hence, these results motivate our choice to use NSGA-II as a search method

Furthermore, we show the Pareto front of each algorithm in Figure 5.7 from the *mitchellh/vagrant* project. We observe that NSGA-II tends to evolve more near-optimal solutions in the middle region of the identified Pareto front with a good spread of solutions along the front, pushing it outwards toward the ideal point (*i.e.,* high true positive rate and low false positive rate). We observe also that NSGA-III and IBEA have less non-dominated solutions in the middle of the Pareto front. However, for both extremes of the Pareto front we observe that most of algorithms reach similar regions of the search space. On the other hand, we observe that IBEA achieves less interesting solutions in its Pareto front. For the CI build failures prediction problem, optimal solutions within the extreme edges of the Pareto front are typically less desirable than solutions in the middle region. That is, solutions in the middle region provide the optimal trade-off between both objective functions (TPR and FPR) while solutions from the extreme edge region represent predictions rules with either high true positive rate (TPR) or low false positive rate (FPR).

Table 5.4 Statistical tests results of NSGA-II compared to other search-based techniques

| **NSGA-II** | Measures | **vs. RS** | **vs. GA** | **vs. IBEA** | **vs. NSGA-III** | **vs. SPEA2** |
|---|---|---|---|---|---|---|
| **AUC** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
|  | *A estimate* | 0.98 | 0.97 | 0.62 | 0.62 | 0.57 |
|  | *Magnitude* | Large | Large | Small | Small | Small |
| **Balance** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
|  | *A estimate* | 0.98 | 0.98 | 0.62 | 0.62 | 0.59 |
|  | *Magnitude* | Large | Large | Small | Small | Small |

### 5.4.2 RQ2. Results for the comparison with ML

Figures 5.8 and 5.9 show the boxplots comparing the results of all the executed experiments iterations to compare NSGA-II with ML algorithms (DT, NB, and RF) in each studied project.

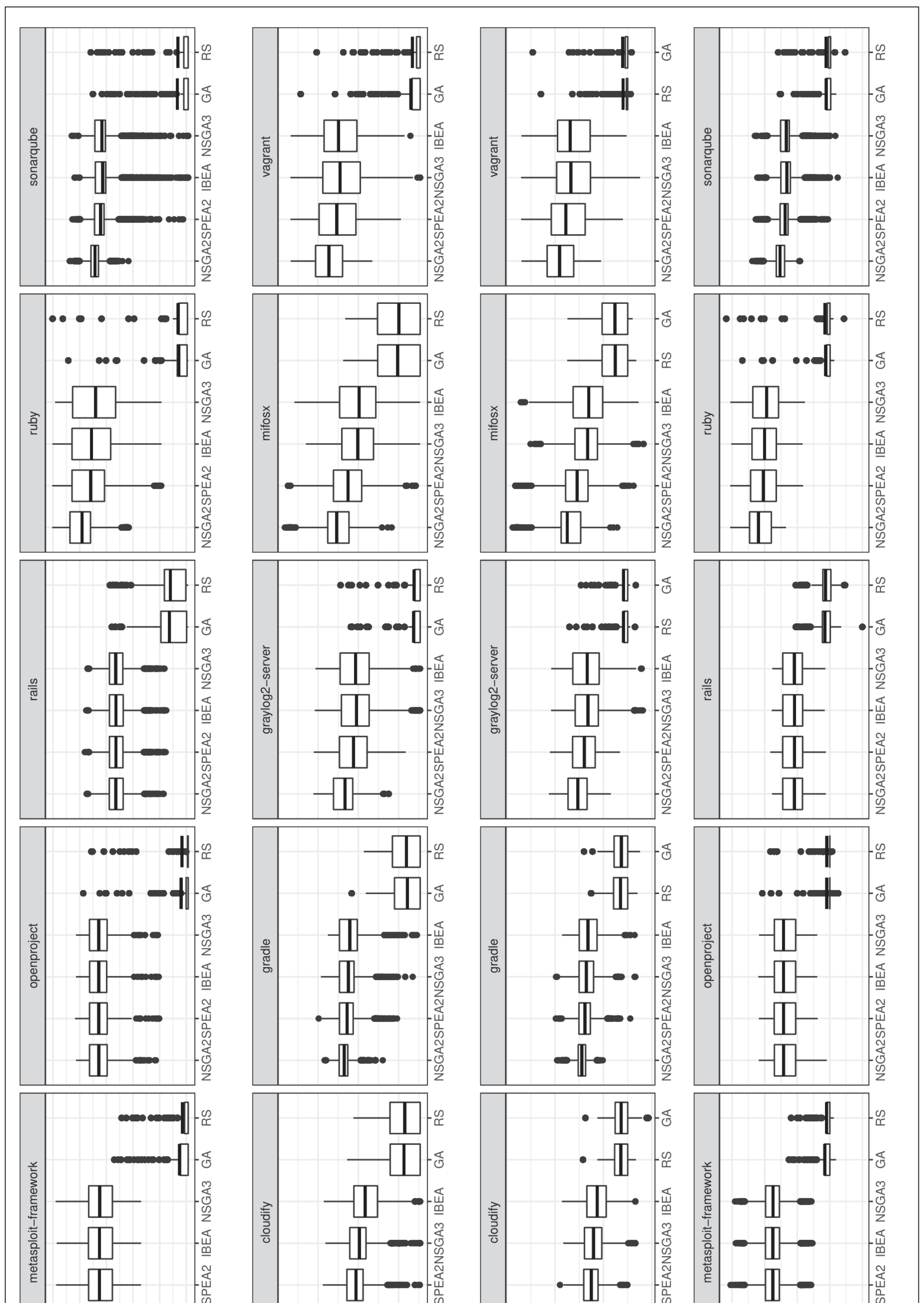


Figure 5.6 Boxplots comparing scores of the search-based algorithms for the 5,000 experiment instances (1,000 runs × 5 validation iterations) in each project

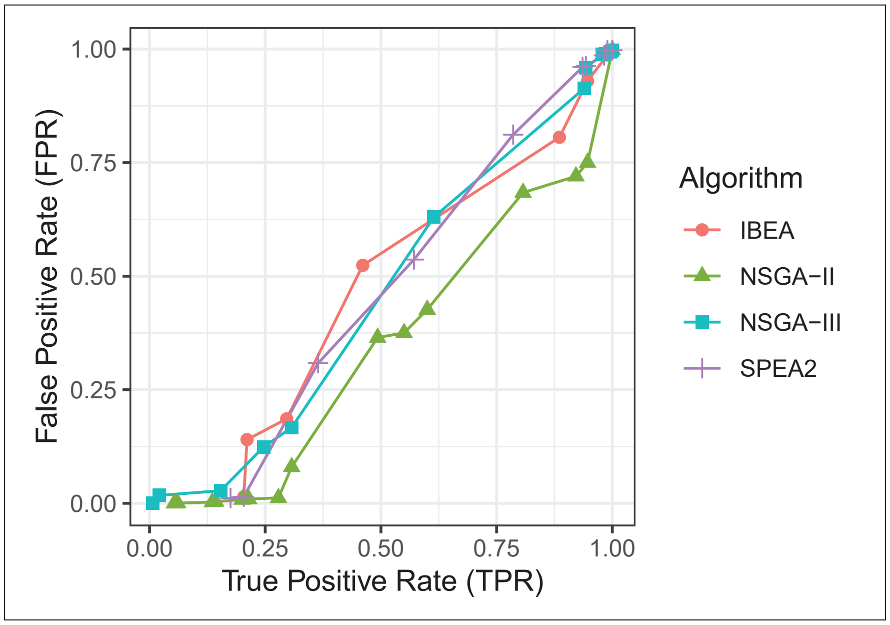


Figure 5.7 An example of Pareto Front extracted from *mitchellh/vagrant* project

Table 5.5 Performance metrics achieved by each of the MOEAs in terms of hyper-volume (HV), generational distance (GD), and spacing (SP)

| | **NSGA-II** | **SPEA2** | **NSGA-III** | **IBEA** |
|---|---|---|---|---|
| *HV* | **0.99** | 0.96 | 0.96 | 0.96 |
| *GD* | **0.004** | 0.01 | 0.01 | 0.01 |
| *SP* | **0.05** | 0.10 | 0.24 | 0.21 |

Table 5.7 reports the average (of 5 online validation iterations) balance and AUC scores while Table 5.6 shows the statistical comparisons of these experiments. Note that NSGA-II, RF and DT were executed 1,000 times for each experimentation instance to deal with their stochastic nature. Then we computed the median values of each experiment. Also, in the figures, the horizontal black lines indicate the average values of the corresponding scores.

As we can see, our NSGA-II technique achieves an average AUC of 69% and an average balance of 66%. Although the achieved results may seem modest performance numbers, they are quite significant given the high imbalanced nature of the data (*i.e.,* only a small portion of the builds

Table 5.6 Statistical tests results of NSGA-II compared to ML techniques

| **Metric** | **Statistics** | **NSGA-I vs. RF** | **NSGA-II vs DT** | **NSGA-I vs NB** |
|---|---|---|---|---|
| **AUC** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.76 | 0.92 | 0.92 |
| | *Magnitude* | Large | Large | Large |
| **Balance** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.80 | 0.91 | 0.93 |
| | *Magnitude* | Large | Large | Large |

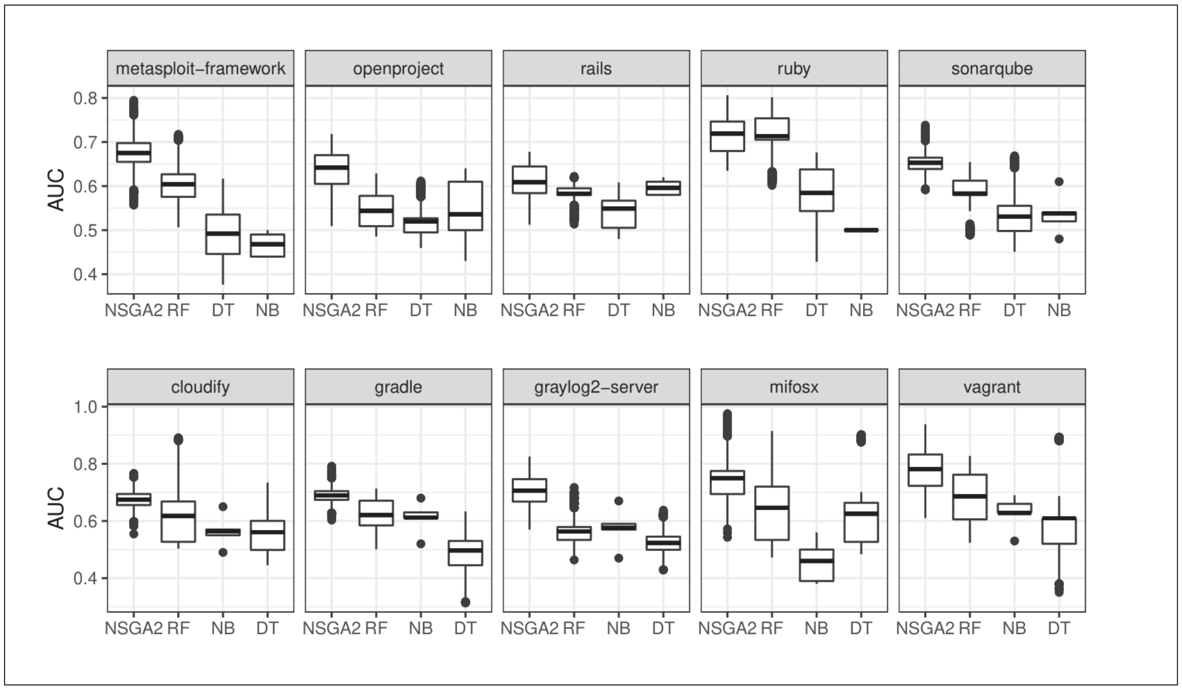


Figure 5.8 Boxplots comparing the achieved AUC values for NSGA-II and each of the machine learning techniques, DT, NB and RF

are failed) as can be noticed from Table 5.2. Moreover, we see from Table 5.7 that for the 10 studied projects, the best AUC and balance values were achieved by the NSGA-II algorithm. On the other hand, for the different projects, the statistical analysis provide evidence that our approach performs better than the ML techniques with a large VDA's effect size and A estimate > 0.5 for both balance and AUC.

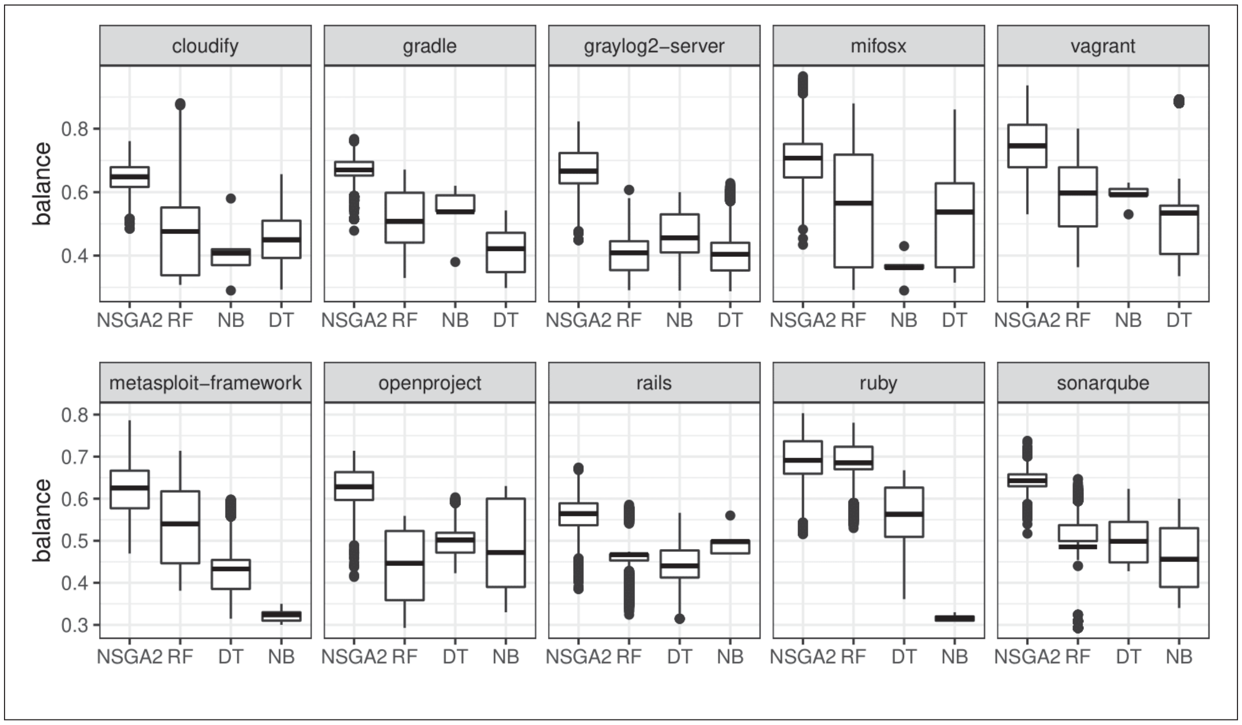


Figure 5.9 Boxplots comparing the achieved balance values for NSGA-II and each of the machine learning techniques, DT, NB and RF

For instance, in the *Graylog2/graylog2-server* project in which the number of failed builds represent only 12%, our approach achieved 71% in terms of AUC compared to 58% for NB, 56% for RF and 52% for DT which represents an improvement of 13% over ML. Also, in *mitchellh/vagrant* project, in which we obtained the best results, our approach outperforms ML techniques by achieving 78% in terms of AUC compared to 69%, 63% and 60% for RF, NB and DT, respectively.

Based on these results, we can conjecture that NSGA-II performs better in comparison with ML techniques even without need for features scaling or relying on any re-sampling technique. This could be justified by the fact that NSGA-II had a better trade-off (*i.e.,* balance and AUC) between both positive (*i.e.,* failed) and negative (*i.e.,* passed) accuracies, which indicates that our approach is advantageous over ML when developing prediction rules for imbalanced datasets. Although the results reveal that GP shows less sensitivity to deal with imbalanced data than

Table 5.7 Performance of NSGA-II vs ML techniques

| **Project** | **AUC** | | | | **Balance** | | | |
|---|---|---|---|---|---|---|---|---|
| | *NSGA-II* | *DT* | *RF* | *NB* | *NSGA-II* | *DT* | *RF* | *NB* |
| cloudify | **0.67** | 0.55 | 0.62 | 0.56 | **0.65** | 0.43 | 0.47 | 0.41 |
| gradle | **0.69** | 0.50 | 0.62 | 0.61 | **0.67** | 0.42 | 0.51 | 0.54 |
| graylog2-server | **0.71** | 0.52 | 0.56 | 0.58 | **0.67** | 0.41 | 0.41 | 0.46 |
| metasploit-framework | **0.68** | 0.49 | 0.60 | 0.47 | **0.63** | 0.44 | 0.54 | 0.32 |
| mifosx | **0.75** | 0.62 | 0.64 | 0.46 | **0.72** | 0.53 | 0.55 | 0.36 |
| openproject | **0.64** | 0.52 | 0.54 | 0.53 | **0.63** | 0.50 | 0.45 | 0.47 |
| rails | **0.61** | 0.55 | 0.58 | 0.60 | **0.56** | 0.44 | 0.47 | 0.50 |
| ruby | **0.72** | 0.58 | 0.71 | 0.50 | **0.69** | 0.56 | 0.68 | 0.31 |
| sonarqube | **0.65** | 0.53 | 0.58 | 0.54 | **0.64** | 0.50 | 0.49 | 0.45 |
| vagrant | **0.78** | 0.60 | 0.69 | 0.63 | **0.75** | 0.53 | 0.60 | 0.59 |
| **Median** | **0.68** | 0.54 | 0.61 | 0.55 | **0.66** | 0.47 | 0.50 | 0.46 |
| **Average** | **0.69** | 0.55 | 0.61 | 0.55 | **0.66** | 0.48 | 0.52 | 0.44 |

ML, we advocate the use of HyBridized Techniques (HBT) which have been found useful by combining the advantages of search-based and ML techniques to produce better results (Malhotra & Khanna, 2017).

### 5.4.3 RQ3. Results for Feature Analysis

In this RQ, we want to better understand what features contributed to achieving higher performances. Figure 5.10 shows the results of feature ranking for each project while Table 5.8 provides a summary for the all studied projects. Broadly speaking, the figure did not reveal any significant variation between features categories with regard to the rate of occurrences. However, among all projects, the most important feature types are project history, link to the last build and change type.

**Project History** features are the most prominent features for six projects namely *cloudify*, *graylog2-server*, *vagrant*, *openproject*, *sonarqube* and *ruby*. For these projects, a closer examination reveals that the statistics of the project have a clear indication of the build outcome. For instance, in *openproject* project, our rules expose that one of the conditions to cause build

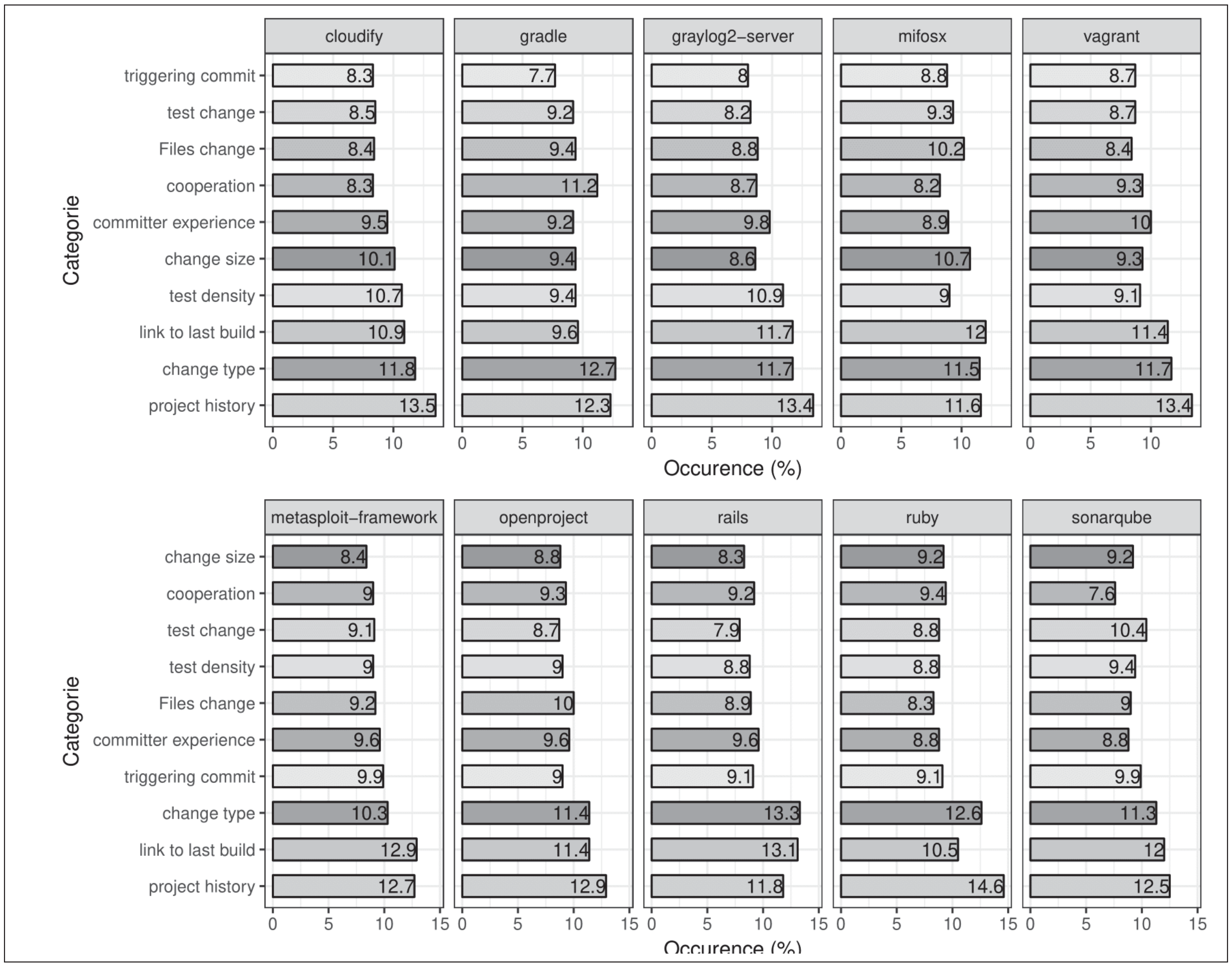


Figure 5.10 Features ranking for each project

failure is having a historical failure rate higher than 34% which alone covers around 63% of the builds in this project. A similar behavior was observed in *sonarqube* project as well. This result lends support to previous research efforts (Ni & Li, 2017) claiming that the statistics about the project are the most useful features in predicting the build outcome.

**Link to last build** is another features category that seems to be important, which appears the most in *metasploit-framework* and *mifosx* projects. For instance, in *metasploit-framework* most of our generated rules classify the instances that failed along from the previous one. On the other side, in this project, there exist 500 failed builds of which 124 occurred consecutively (about 25%) which pr1ovides additional support for our rules. As stated previously (Hassan & Wang,

Table 5.8 A summary of the features ranking for all the studied projects

| Category | Occurrence (%) |
|---|---:|
| Project history | 12.77 |
| Link to last build | 12.08 |
| Change type | 11.78 |
| Committer experience | 9.40 |
| Triggering commit | 9.34 |
| Files change | 9.20 |
| Cooperation | 9.04 |
| Test density | 8.97 |
| Test change | 8.75 |
| Change size | 8.67 |

2017; Ni & Li, 2017; Rausch *et al.*, 2017), it is apparent that phases of build instability perpetuate failures.

**Change type** features are the most occurring among two projects namely *rails* and *gradle*. This suggests that changes to specific types of files can affect the build outcome. For example, in *rails* project, there exists 2,567 builds where changes to only source code files introduced build failures which represent 72% of failed builds.

Other features are also important in indicating CI build outcome. For instance, metrics about test change represent also an important percentage of appearance in *sonarqube* project. However, statistics about the triggering commit seem to be less important and the least appearing in three projects which indicates that these features are not highly related to the build outcome.

## 5.5 Discussion

In this section, we discuss our findings and their implications for developers, researchers and tool builders.

### 5.5.1 For CI developers

***We can help developers to take the necessary preventive actions to avoid breaking the build.*** We have shown that our approach is able to predict the CI build results, however, the key innovation of our approach is that it is able to provide an explainable prediction model, and also some modalities to be respected in order to avoid build failures. For instance, Figure 5.11 shows an example of a prediction rule that was generated by our tool to predict the failure in the *mitchellh/vagrant* project with high AUC and balance scores of 92%. In this rule, it is suggested that, among different conditions, if the number of modified files (FM) in the current build is less 10 then your CI build is likely to fail. As an alternative to avoid such build failures, the developer may opt to reduce the number of modified files in a commit or may also split the number the files into two or more build pushes to reduce the change complexity, and thus reduce potential build failures. More interestingly, we plan to extend our approach with further support to software developers by suggesting change fixes for their failed CI builds based on the violated conditions in the generated tree-based rules.

Hence, such explainable models show indeed that it is possible to pinpoint the root cause of a CI build failure using our search-based approach. Moreover, it is worth noting that it may be possible to reduce the complexity of the generated prediction rules (*e.g*, tree size and/or depth) in order to provide easier explainable models for CI developers with smaller slice and less complexity, but with of cost of scarifying with some accuracy. Indeed, as part of our future work, we plan to extend our approach into a multi-objective approach to find the best trade-off between the model *accuracy* and *complexity*, which are in conflicting considerations.

***Usage scenario of our tool.*** Figure 5.12 provides a typical usage scenario of our tool in practice. When a developer commits a change to the repository (1) our tool is triggered to predict potential build failures. Once triggered, (2) the user is invited to choose whether to load the previously generated rule or generate new rule. This decision can be made if the current rule is not no more up-to-date, *i.e.,* after a number of builds. After the generation/loading of the prediction rule, our tool analyzes the changes made in the commit's files and compute the CI metrics to determine

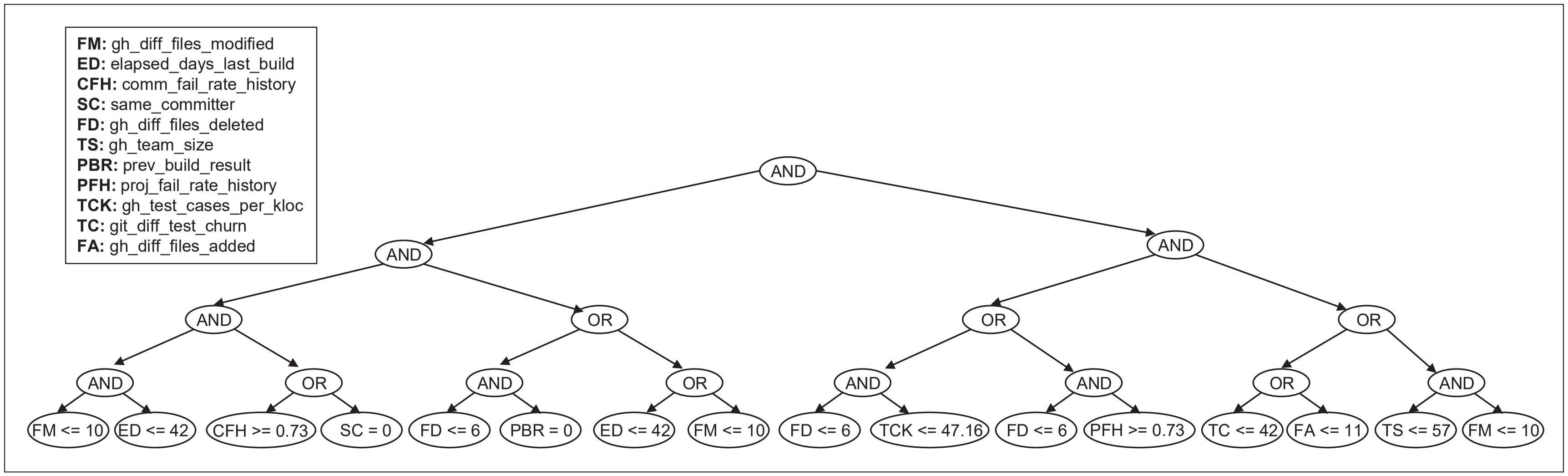


Figure 5.11 An example of CI build failure prediction rule for the *mitchellh/vagrant* project

whether the build would pass/fail. Finally, the prediction is provided to the developer with the required explanation to guide the developer in his retro-actions if needed. In this way, developers can cut off the expenses of CI build process by saving the build generation time and effort.

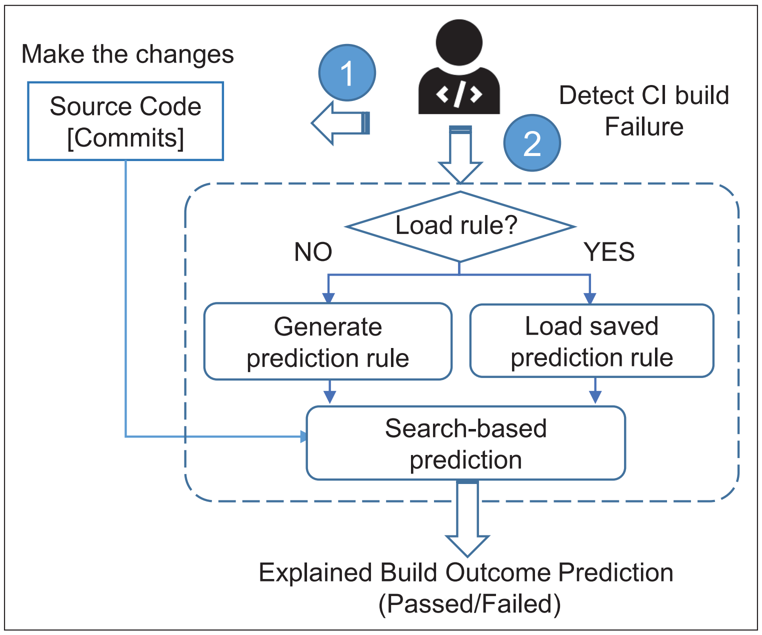


Figure 5.12  A usage workflow of our approach

***Build verification is fast.*** We envisage our solution being used by developers, in their daily CI workflow to check whether their changes will break the build. One of the benefits of using our approach is that also, like ML techniques, we can save the learning model to be used for the prediction or updated later when more data is available over time as the project evolves. Thus, it is important to assess the scalability of our approach from the data point of view. To this end, we conducted an experiment to assess the ability of our search-based approach to scale to larger datasets. Figure 5.13 reports the results of our experiment. We find that our search-based approach scales linearly *i.e.,* depends on the size of the learning set, as shown in the figure. For instance, with a dataset composed of 10,629 our tool can train the model within 9 minutes approximately, which is considered reasonable from computation point of view. However, from a developer point of view it is worth noting that the training on the dataset is required only once to build the model that will be used later for the prediction. The prediction consists of simply checking whether the conditions that appear in the prediction rule (*e.g*, Figure 5.11) are violated or not which takes typically few seconds. Thereafter, the tool can update the model with more data after a number builds that could be configured by the developer.

Note that in this work, all the experiments are executed on a computer equipped with an Intel Core i7-7700k 4.2 GHZ CPU and 16GB memory.

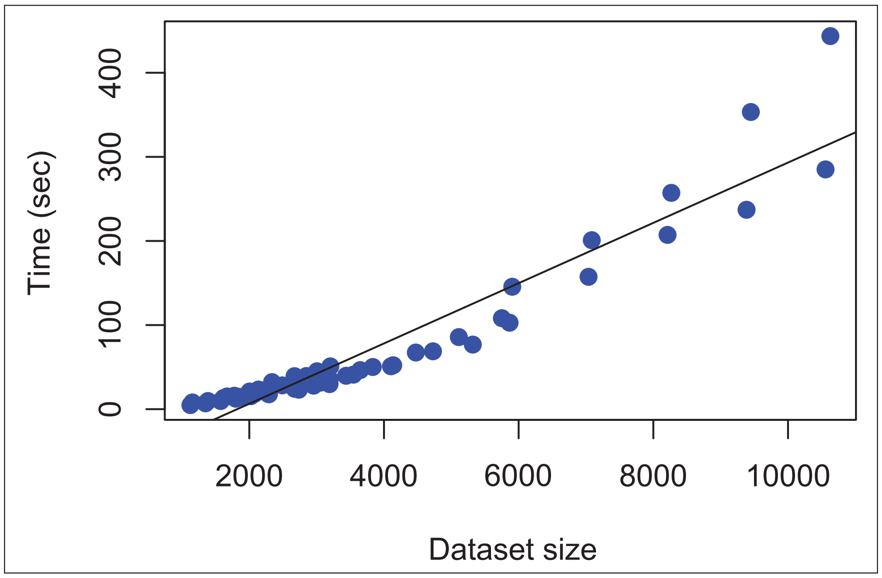


Figure 5.13 The impact of the training dataset size on the NSGA-II execution time to build the prediction model

### 5.5.2 For researchers

***The reasons behind build failure need more in-depth studies.*** Although, in this paper, we showed that failure prediction is possible with encouraging scores, we believe that by enhancing the feature engineering, we can obtain better results. Hence, the results may encourage CI researchers to investigate other measurable internal and external metrics and factors that could be correlated with the build outcome.

***Retro-actions to fix a failed build.*** As discussed earlier in Section 5.5.1, our explainable model for build failures prediction can provide a valuable support to developers on how to proceed to fix their failed builds based on the violated rules or conditions. Moreover, looking at what rules or specific conditions were violated in a build failure represent a crucial information and valuable knowledge to be used as a starting point to prepare or recommend retro-action plans to fix the failed build. Thus, such valuable information may encourage researchers to develop automated build failure fix approaches, which is indeed one of our future research works. Furthermore, providing such information on the build failures may increase learning within developers and provide them with better understanding on the root causes of such build failures. Moreover, documenting such violations may also increase knowledge transfer from developers.

***Researchers could investigate periodicity in build failure.*** Our features analysis lends support to previous a research efforts (Rausch *et al.*, 2017) showing that many failed builds occurred consecutively which indicate that if the build failed, the next build is more likely to fail as well. This finding may encourage researchers to get insights into the periodic trends of build failure which would help us to enhance the prediction accuracy.

### 5.5.3 For tool builders

***Rules updating strategies should be considered when building CI build prediction tools.*** As we can see from our obtained models, the final prediction rules (features, and threshold values) can differ from one project to another, and from datasets in the same project. Ideally, the prediction rules/models should be updated regularity as the project evolves, *i.e.,* after a given number of builds, or generalized among other projects. However, in a real world setting, prediction models may achieve less performance when applied to different data and different contexts (Choetkiertikul *et al.*, 2018; Abdalkareem *et al.*, 2020; Zhang, Zheng, Zou & Hassan, 2016). For instance, Figure 5.14 highlights a simplified example, considering a rule R1 for project A and R2 for project B. In the same rule R1, we see that the feature denoting the source churn (*git_diff_src_churn*) can be associated to two different thresholds, in different sub-trees, *i.e.,* a failed build if the source churn is $\geq 200$ and the elapsed days $\leq 2$; or if the number of modified files $\leq 2$ and the source churn $\geq 500$. However, in project B, the rule indicates a failed build if one the three mentioned conditions is met. Hence, it would be difficult to generalize the threshold values to be recommended for each feature. However, generating new prediction rules frequently during the development process could be costly (Janssen, Moons, Kalkman, Grobbee & Vergouwe, 2008). As an alternative solution to leverage this issue is to update the existing prediction rule by combining the information that is captured in the original rule with the information of the recent data, hence updating the existing rule (*e.g.,* threshold values, or features). Hence, an interesting feature to consider with tools builders is to adjust the prediction rules on-the-fly based on the developers preferences or when the prediction accuracy starts to decrease over time. We are planning to integrate this feature in our approach as part of our future work.

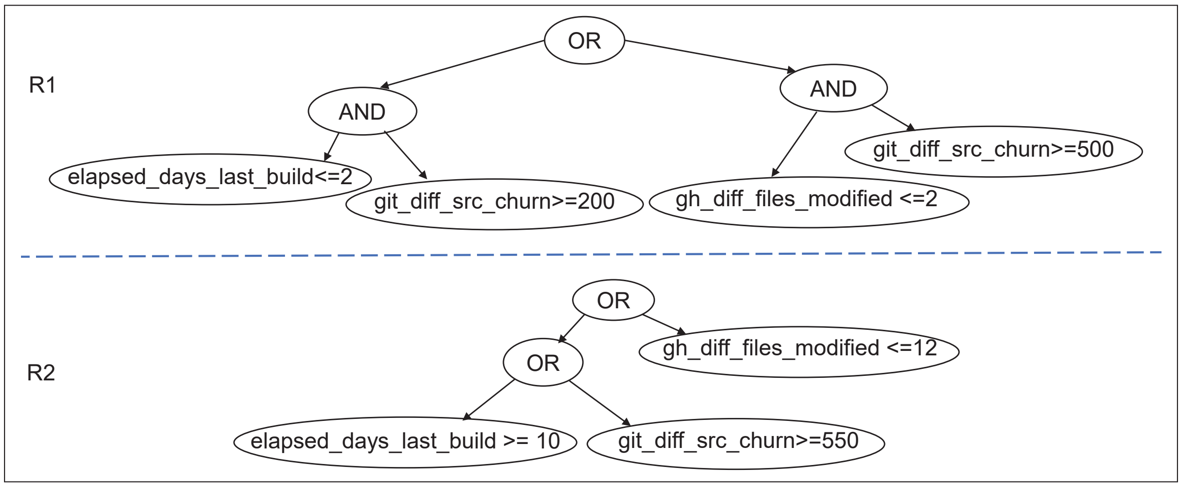

R1
OR
AND
AND
elapsed_days_last_build<=2
git_diff_src_churn>=200
gh_diff_files_modified <=2
git_diff_src_churn>=500
R2
OR
OR
gh_diff_files_modified <=12
elapsed_days_last_build >= 10
git_diff_src_churn>=550

Figure 5.14 An example showing how the prediction rules can be less effective when applied to other projects

***Tool for recommending relevant files for build failures localization.*** Our features ranking analysis showed that change type features, such as the number of configuration files touched in the built commits, are prominent to detect build failures in the studied projects. On another hand, developers may follow a tedious process to localize the file causing the failure. Hence, tool builders should supply development teams with tools to identify potential files in order to accelerate the build fixing process.

## 5.6 Threats to validity

This section describes the threats to the validity of our experiments.

**Internal validity.** One threat to internal validity is related to training and test sets selection. As an attempt to mitigate this issue, we considered online validation which is a realistic scenario as it considers the chronological order of CI builds and mimics what happens during the continuous integration process. Future work is planned to validate our approach considering other scenarios such as cross-project validation. Another threat to validity can be related to the stochastic nature of the meta-heuristic algorithms (Harman *et al.*, 2012; Arcuri & Briand, 2011). To mitigate this threat, we performed 1,000 runs of each experimentation instance and considered the median value in each validation iteration. Moreover, we have double checked our experiments as well as the datasets collected from TravisTorrent through manual inspection, still there could be errors that we did not notice.

**Construct validity.** Threats to construct validity can be related to the set of used metrics and performance measure. We basically used standard performance metrics such as AUC and balance that are widely accepted in predictive models in software engineering (Malhotra & Khanna, 2017). As for the used measurements, we used standard features from TravisTorrent data set and other generated features related especially to historical build failure that commonly used in the literature (Xia *et al.*, 2017a; Ni & Li, 2018; Xia & Li, 2017; Luo *et al.*, 2017; Hassan & Wang, 2017; Ni & Li, 2018; Santolucito *et al.*, 2018). Although our approach is not closely coupled with the features used in this paper, we plan to extend our measurements to other code level metrics and other external factors as an attempt to see their impact on the prediction performance. Another potential threat could be related to the selection of the prediction techniques. Although we used different search-based techniques, *i.e.,* NSGA-II, NSGA-II, SPEA2, GA, and random search, and different machine learning techniques, *i.e.,* DT, RF and NB, which are the most applied in existing solutions for build prediction and several other software engineering problems (Xia *et al.*, 2017a; Luo *et al.*, 2017; Santolucito *et al.*, 2018; Hassan & Wang, 2017). To mitigate this threat, we plan as part of our future work to conduct a large scale empirical study with other search-based and machine learning techniques.

**Conclusion validity.** We have carefully chosen non-parametric tests, namely Wilcoxon and Vargha-Delaney A, in the study as they do not require data normality assumptions and also for being the most used statistical tests in SBSE research community (Nejati & Gay, 2019). The suitability of the used statistical non-parametric methods with data ordinality, along with no assumption on their distribution raises our confidence about the significance of the analyzed statistical relationships. Moreover, to increase the confidence in the study results, we used two widely-acknowledged prediction performance measures, *i.e.,* balance and AUC, and three performance measures, *i.e.,* hyper-volume (HV), generational distance (GD) and spacing (SP) to evaluate the obtained results from the considered algorithms.

**External validity.** Our experimental results might have concerns of generalizability, since we performed the experiments with ten open source projects that use TravisTorrent as their CI host tool. While TravisTorrent is one of widely used CI tools, our results could not be generalized to other CI tools and other open-source or industrial projects. As future work, we plan to extend our study on other open source and industrial projects as well as other CI tools. We also plan to provide our approach as bot to be integrated into code review and CI tools to help developers predicting their build failure risks.

## 5.7 Conclusions and Future Work

In this article, we introduced a new search-based approach for CI build failure prediction. In our genetic programming (GP) adaptation, prediction rules are represented as a combination of metrics and threshold values that should correctly predict as much as possible the failed builds extracted from a base of real world examples. Considering online validation, the statistical analysis of the obtained results provides evidence that our approach outperforms three Machine Learning (ML) techniques, for which we applied re-sampling, as well as Random Search and mono-objective Genetic Algorithm, based on a benchmark of 56,019 CI builds of ten projects that use Travis CI. Regarding the most important indicators used by our generated rules, we found that features related to (1) specific statistics about the project such as team size, (2) last build information in the current build and (3) the types of changed files are the most influential to indicate the potential failure of a given build.

While the obtained results are considered promising, it could be further validated with larger sample size with a variety of CI systems to conclude about the general applicability of our methodology. Moreover, we believe that by using a more personalized group of features with external factors, the prediction performance could be further improved, which we plan to explore in the future. Also, we plan also to extend our approach by adopting HyBridized Techniques (HBT) which have been found useful by combining the advantages of search-based and ML techniques to produce better results.

# CHAPTER 6

# IMPROVING THE PREDICTION OF CONTINUOUS INTEGRATION BUILD FAILURES USING DEEP LEARNING

Islem Saidani[a] , Ali Ouni[a] , Mohamed Wiem Mkaouer[b]

[a] Department of Software Engineering and IT, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[b] Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester,
NY 14623, United States

Paper published in *Automated Software Engineering*, December 2021

## Abstract

Continuous Integration (CI) aims at supporting developers in integrating code changes constantly and quickly through an automated build process. However, the build process is typically time and resource-consuming as running failed builds can take hours until discovering the breakage; which may cause disruptions in the development process and delays in the product release dates. Hence, preemptively detecting when a software state is most likely to trigger a failure during the build is of crucial importance for developers. Accurate build failures prediction techniques can cut the expenses of CI build cost by early predicting its potential failures. However, developing accurate prediction models is a challenging task as it requires learning long- and short-term dependencies in the historical CI build data as well as extensive feature engineering to derive informative features to learn from. In this paper, we introduce *DL-CIBuild* a novel approach that uses Long Short-Term Memory (LSTM)-based Recurrent Neural Networks (RNN) to construct prediction models for CI build outcome prediction. The problem is comprised of a single series of CI build outcomes and a model is required to learn from the series of past observations to predict the next CI build outcome in the sequence. In addition, we tailor Genetic Algorithm (GA) to tune the hyper-parameters for our LSTM model. We evaluate our approach and investigate the performance of both cross-project and online prediction scenarios on a benchmark of 91,330 CI builds from 10 large and long-lived software projects that use the Travis CI build system.

The statistical analysis of the obtained results shows that the LSTM-based model outperforms traditional Machine Learning (ML) models with both online and cross-project validations. *DL-CIBuild* has shown also a less sensitivity to the training set size and an effective robustness to the concept drift. Additionally, by considering several Hyper-Parameter Optimization (HPO) methods as baseline for GA, we demonstrate that the latter performs the best.

**Keywords.** Continuous Integration, Build Prediction, Travis CI, Genetic Algorithm, Long Short Term Memory, Machine Learning, Hyper-Parameters Optimization , Concept Drift.

## 6.1 Introduction

Continuous integration (CI) (Duvall *et al.*, 2007b) is a set of software development practices that are widely adopted in commercial and open source environments (Vasilescu *et al.*, 2015). A typical CI system, such as Travis CI (CI, 2021), a widely-used cloud-based platform for providing CI services to software projects, advocates to continuously integrate code changes, introduced by different developers, into a shared repository branch. The key to making this possible, according to Fowler (Fowler, 2006), is automating the process of building and testing, which reduces the cost and risk of delivering defective changes. From the academic side, the study of CI adoption has become an active research topic and it has already been shown that CI improves developers' productivity (Hilton *et al.*, 2016; Saidani *et al.*, 2021c; Saidani, Ouni, Chouchen & Mkaouer, 2020b), helps to maintain code quality (Vasilescu *et al.*, 2015; Saidani, Ouni, Mkaouer & Palomba, 2021b; Saidani *et al.*, 2021a) and allows for a higher release frequency (Zhao *et al.*, 2017).

However, despite its valuable benefits, CI brings its own challenges. Hilton *et al.* (2017) revealed that build failures represent major barriers that developers face when using CI . A build failure, *i.e.,* failing to compile the software into machine executable code, represents a blocker that hinders developers from proceedings further with development, as it requires immediate action to resolve it. Indeed, Ghaleb *et al.* (2019a) have shown that long build duration is not always associated with passed builds as expected and Hilton *et al.* (2016) found that passed builds can run faster than failed builds. For example, in the TravisTorrent dataset (Beller *et al.*, 2017), failed Travis CI builds run 12 hours while passed builds can take 8 hours on average. In addition to the long build duration issue, the resolution may take hours or even days to complete, which severely affects both, the speed of software development and the productivity of developers (Vasilescu *et al.*, 2015).

Such challenges motivated researchers and practitioners to develop techniques for preemptively detecting when a software state is most likely to trigger a failure when built. In recent years, numerous prediction methods have been developed to leverage the history of previous build success and failures in order to train Machine Learning (ML) models. Such models learn from the CI builds history and use the domain knowledge to extract features and predict the outcome of a given input build. For instance, Hassan & Wang (2017) used Random Forest (RF), for the binary classification of build outcome , while Ni & Li (2017) adapted AdaBoost (ADA) to improve the accuracy of CI build prediction. Although these techniques have advocated that predicting CI build failures is possible in practice and beneficial, the applicability of these approaches is limited due to three main challenges:

- *Feature engineering:* Traditional ML techniques rely on a set of features manually designed for characterizing a given problem, *e.g.,* CI builds. Generally, the feature engineering task is tedious, time-consuming, error-prone and requires substantial expertise in the field (Li, Gong, Yu & Zhou, 2018; Shan *et al.*, 2016; Bouktif *et al.*, 2018; Sundsøy, Bjelland, Reme, Iqbal & Jahani, 2016). Additionally, the accuracy of prediction models depends highly on the relevance of the selected features. Broadly speaking, the build failure prediction problem is not yet resolved as the reasons behind the build failure is still ambiguous.
- *Temporal information:* Previous work on CI build prediction is focused on TravisTorrent-based measures (*e.g.,* number of all built commits, number of distinct authors, etc.) to predict new CI build results in the future, without taking temporal information into account, *i.e.,* chronological order of CI build outcomes. As a result, these works achieved a limited prediction accuracy. The CI builds outcome data is by nature a time series data (Atchison, Berardi, Best, Stevens & Linstead, 2017) where the temporal dimension is of crucial

importance. However, such time series data can be highly erratic and complex with much noise and high dimensionality especially with unexpected or repetitive build failures over time (Längkvist, Karlsson & Loutfi, 2014).

- *Data imbalance:* Another innate issue to classic ML-based approaches is related to the imbalanced distribution of class examples as failed builds are typically likely to occur less than passed ones (Xie & Li, 2018). This challenges their applicability due to the performance bias that can occur when an imbalanced distribution of class examples is used (Bhowan *et al.*, 2011, 2010, 2013). Furthermore, this imbalanced nature of the training data was rarely discussed in existing works. However, in CI context, a good accuracy on the failed builds prediction is more important than the passed builds accuracy.

These challenges make Deep Learning (DL) time series models suitable for this kind of problems (Längkvist *et al.*, 2014). Indeed, DL methods make no assumption about the underlying pattern in the data and are also more robust to noise (which is common in time series data), making them an ideal choice for time series analysis of CI builds. Additionally, DL models are known to decrease the reliance on engineered features to address classification problems (Ordóñez & Roggen, 2016).

In this paper, we introduce *DL-CIBuild*, a novel approach to predict CI build failure. In particular, Long Short-Term Memory (LSTM) network is trained on sequential data in which each series observation is the history of build results during a specific time period. The time series prediction produced by LSTM models are then used to estimate the outcome of future builds. Moreover, as naive selection of hyper-parameter values may compromise the effectiveness of any DL adaptation, we opt for an automated hyper-parameter optimization (Tantithamthavorn *et al.*, 2018a; Jebnoun, Braiek, Rahman & Khomh, 2020). In particular, we rely on Genetic algorithm (GA) to find the optimal set of parameter values to build a model with optimal prediction accuracy. Furthermore, to handle the data imbalance, we apply Threshold Moving (Zhou & Liu, 2005) to move the classification threshold such that more failed builds can be classified correctly (Zheng, 2010).

To evaluate our approach, we conducted an empirical study on a benchmark composed of 91,330 builds records from 10 open source projects that use the Travis CI system, one of the most popular CI systems (Hilton *et al.*, 2017). We compare our predictive performance to five widely-used ML techniques namely Random Forest (RF), Decision Tree (DT), AdaBoost (ADA), Logistic Regression (LR) and Support Vector Classification (SVC) for which we applied resampling. The statistical results reveal that our approach advances the state-of-the art by outperforming existing prediction models.

In summary, the contributions of this work are the following:

- We introduce a new formulation of the CI build failure prediction as a time series problem using LSTM-RNN, and implement it with a tool called *DL-CIBuild*. To the best of our knowledge, this is the first attempt to use deep learning LSTM-based approach to learn CI build failures. The built model can be trained efficiently using CI build outcomes, which requires no feature engineering. Moreover, we use GA to optimize the hyper-parameters of our models for optimal performance.
- We conduct an empirical study to evaluate our LSTM-RNN based technique compared to different existing approaches based on a benchmark of 10 large open source projects with a total number of 91,330 builds. First, we validated the efficiency of GA for Hyper-Parameters Optimization (HPO) against four HPO methods such as Particle Swarm Optimization (PSO). Additionally, the obtained results of the predictive performance comparison reveal that *DL-CIBuild* is more efficient than existing ML techniques in terms of AUC, F1-score and accuracy which indicates that our approach is able to strike a better balance between both failed and passed builds accuracies. These results are further enhanced under cross-project validation by achieving a median of 72%, 57% and 78% of AUC, F1-score and accuracy, respectively. The obtained results indicate that *DL-CIBuild* is a promising solution to deal with the lack of data in software projects. Moreover, we conducted a sensitivity analysis suggesting that our approach has less sensitivity than ML techniques with regards to the dataset size. Last but not least, we showed that *DL-CIBuild* is robust to concept drift (Widmer & Kubat, 1996).

- We provide our comprehensive dataset package available for future replications and extensions (Saidani, 2020c). Our replication package contains the CI build dataset, the source code of *DL-CIBuild*, all the scripts used to run and reproduce the experiments with the necessary documentation.

The remainder of this paper is organized as follows. Section 6.2 provides the motivation for time-series prediction We present our approach in Section 6.3. Section 6.4 shows the experimental setup of our empirical study while Section 6.5 presents the obtained results. Section 6.6 discusses the results implications on CI developers, researchers and tool builders. Section 6.7 reviews the threats to the validity of our results. Finally, Section 6.8 concludes the paper and outlines avenues for future work.

## 6.2 Motivating example

Figure 6.1 depicts an example of CI build outcome fluctuations over time for Jruby GitHub project (Jruby, 2019) that uses the Travis CI system. As shown in the figure, we can observe that there exist data patterns and an explicit dependency on the time variable that may have a strong association with build outcome. In practice, it is common that after a CI build failure, developers proceed to take the right actions to fix the cause of the build failure which may lead to a sequence of build failures followed by a succeeded build, as can be seen in Figure 6.1. Conversely, after a sequence of successful builds, build failures often happen in an unexpected manner over time. Moreover, previously experienced bugs and failures can be the root cause of new failures during the CI build, while other unforeseen failures may happen in an independent manner resulting into temporal dynamic behavior and complex non-linear dependencies between failures. Thus, individual observations, *i.e.,* build results, cannot be predicted independently on each other. This makes the CI build failures a time-series data involving a sequence of observations over regularly spaced intervals. Indeed, time series data consists typically of sampled data points taken from a continuous, real-valued process over time, such as the CI build process.

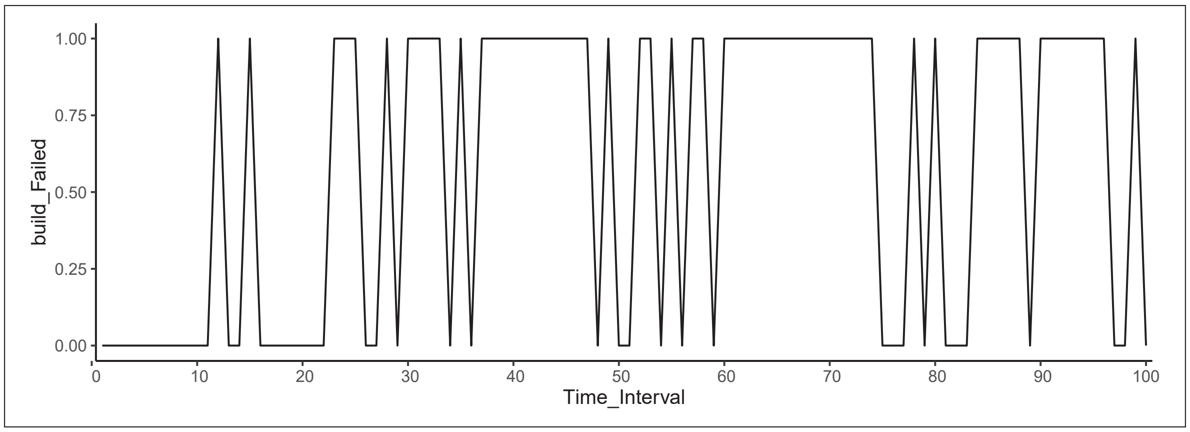


Figure 6.1 A snapshot of build outcome fluctuations in JRuby between 2016-08-01 and 2016-08-31. "1" means a failed build, and "0" for a passed build

This motivates us to formulate the CI build failure prediction as a time series problem of using LSTM deep learning to learn from past observations in order to identify temporal patterns that best describe the inherent structure and temporal process embodied in the series and thus increase the predictive performance.

## 6.3 Our Proposed Approach

In this section, we present our approach *DL-CIBuild* for CI build failure prediction. We first explain how we built our LSTM-RNN model to learn CI build failures, then we describe our genetic algorithm-based method to optimize the model hyper-parameters.

### 6.3.1 Methodology Overview

The main *goal* of our approach is to help developers cutting off such expenses by effectively predicting the CI build outcome before they happen. Indeed, CI build failures are generally time and resource-consuming and can cause disruptions in the development process and delays in the software product release dates (Ghaleb *et al.*, 2019a). In particular, we handle the problem of CI build failures as a time series prediction problem by estimating the outcome of a given build based on the history of observed build processes. We use LSTM-based Recurrent Neural

Network (RNN) to model the CI build process sequential data. Figure 6.2 provides an overview of our proposed approach.

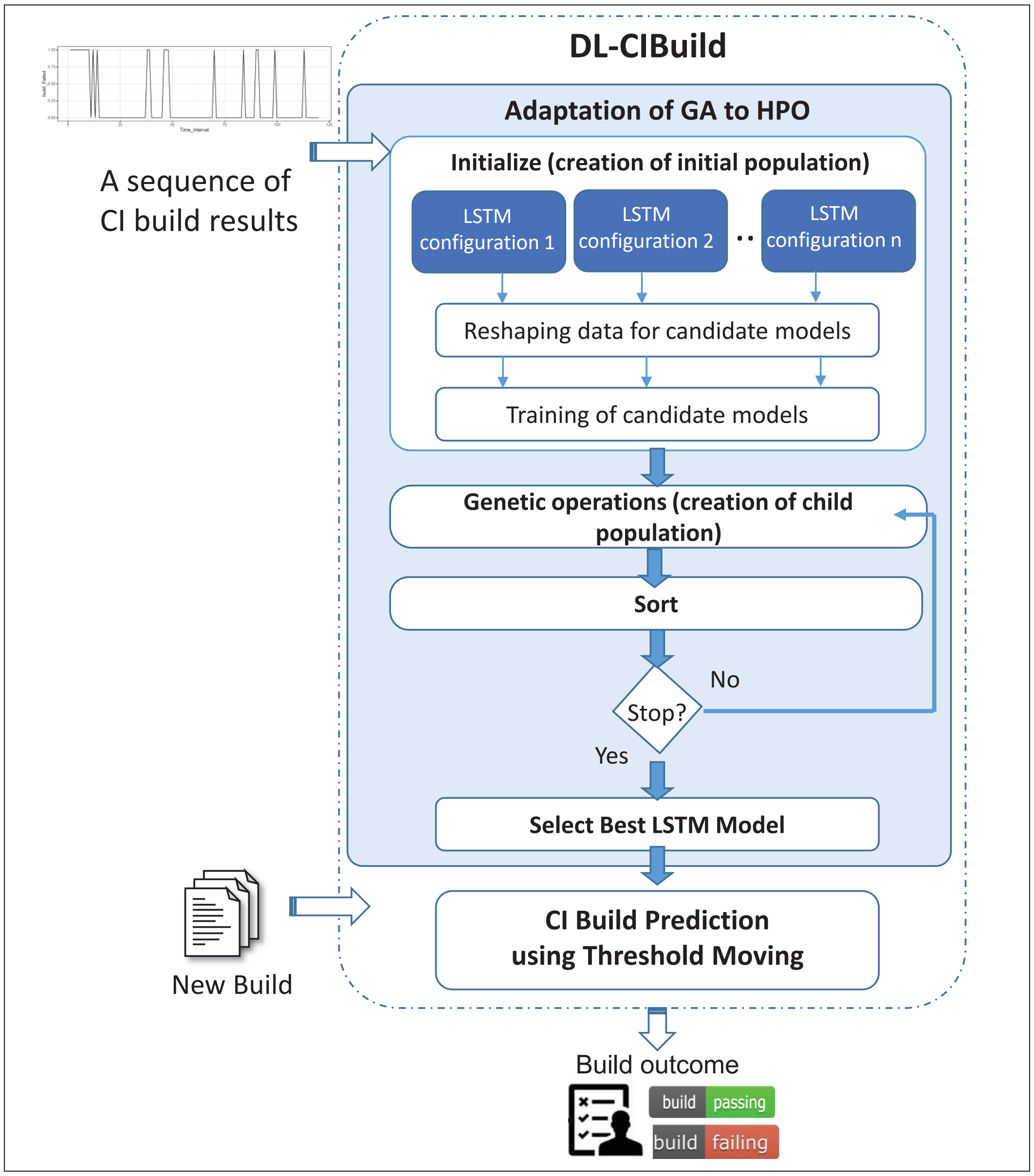

DL-CIBuild
Adaptation of GA to HPO
A sequence of CI build results
Initialize (creation of initial population)
LSTM configuration 1
LSTM configuration 2
LSTM configuration n
Reshaping data for candidate models
Training of candidate models
Genetic operations (creation of child population)
Sort
No
Stop?
Yes
Select Best LSTM Model
CI Build Prediction using Threshold Moving
New Build
Build outcome
build passing
build failing

Figure 6.2 *DL-CIBuild* overview

Our framework starts by adapting Genetic Algorithm (GA) to determine the appropriate hyper-parameters for the LSTM model. These parameters are then used to build the architecture of the final LSTM model. During this Hyper-Parameters Optimization (HPO), the input data,*i.e.,* a sequence of CI build results, is prepared using reshaping; then the candidate models are trained according to their generated configurations. The training data is extracted from the history of CI builds that are typically recorded using the CI build system used by a software project, *e.g.,* Travis CI. At the end of HPO, the optimal model, *i.e.,* providing the best score, is selected. In the prediction phase, our optimal model is used to predict if an unknown build would fail or succeed. The hyper-parameters to be tuned, the data preprocessing and the adaptation of GA are described in the following sub-sections.

### 6.3.2 LSTM model construction and hyper-parameters tuning

We first need to design and configure our LSTM-RNN model by choosing the architecture, setting up the initial hyper-parameters, and selecting the mathematical components such as activation functions, loss functions, and gradient-based optimizers (Jebnoun *et al.*, 2020; Bouktif *et al.*, 2020). Obtaining good results using LSTM networks is not trivial, as it requires consideration of the tuning of many parameters. Unfortunately, applying LSTM models may not produce acceptable or optimal results, not only because of the nature of the analyzed data but also due to the naive selection of its hyper-parameter values. Table 6.1 lists the parameters to be optimized for the LSTM model.

To construct the model, the first LSTM parameters to be tuned are the numbers of hidden layers and neurons per layer. As a neural network, LSTM depends highly on the settings of these parameters. There is no final definite rule of how many nodes (*i.e.,* hidden neurons) or how many layers one should be choosing, and generally, practitioners perform a trial and error approach to get the best results. The same uncertainty about the amount of these parameters also exists for the number of $epochs$ and the $batchsize$ as they affect how well/poorly the model can perform and also can help to prevent over-fitting. Another important parameter to be optimized is the optimizer. Among the optimizers, there exists stochastic root mean square propagation (*RMSprop*) and adaptive moment estimation (*adam*). Last but not least we have to decide, the probability of $dropout$ which stands for a regularization method where input and connections to LSTM neurons are partially excluded from activation and weight updates in order to avoid over-fitting. Not that for each layer, we set the same dropout probability.

As LSTM input data is essentially a set of past observation sequences, it is important to identify the most relevant time steps to feed the model. This can allow the LSTM model to capture the valuable information contained with different timescales.

Table 6.1 List of parameters for the LSTM model

| Category | N° | Parameter | Description |
|---|---|---|---|
| Hyper-parameters | 1 | Number of units | Number of neurons in each LSTM layer. |
| | 2 | Number of layers | Number of hidden layers to be used to train the model. |
| | 3 | Batch Size | Number of samples to be propagated through the network before updating the internal parameters. |
| | 4 | Number of epochs | Number of times that the learning algorithm will work through the entire training set. |
| | 5 | Optimizer | Type of optimizer used to update weights during training. |
| | 6 | Dropout probability | Sets the rate of input units to drop in order to avoid over-fitting. |
| Input parameters | 7 | Time Step | Number of previous observations that are used to predict the next result outcome. |

Finding the suitable configuration is, on the one hand, a combinatorial problem where the selection is made from a very large space of choices; on the other hand, it is a learning problem where the hyper-parameters should reflect the CI build domain knowledge, such as the size of the software project, the number of developers, the adopted testing methods, the used CI system, influential time lags, seasonality, and other socio-technical factors that could differ from one project to another. We describe in the next subsection how GA is used to find the suitable hyper-parameters for our LSTM model.

### 6.3.3 GA Adaptation for HPO of LSTM

In this section, we describe how we adapted Genetic Algorithm (GA) for LSTM model configuration problem, then we provide the hyper-parameters to be optimized for our LSTM model. In particular, as described in Section 2.2.2.1, for any attempt to use GA in a real-world problem, a number of key elements need to be defined such as the solution representation, genetic operators and the fitness function.

#### 6.3.3.1 Individual representation

A candidate solution, *i.e.,* a set of parameters configurations, is represented as an array where each cell corresponds to a randomly generated value for a specific parameter as depicted in Figure 6.3. The initial population is composed of *N* solutions created randomly.

| | | | | | | |
|---|---|---|---|---|---|---|
| Number of units = 64 | Number of layers = 3 | Batch Size =25 | Number of epochs=5 | Optimizer= 'Adam' | Dropout probability= 0.1 | Time Step=60 |

Figure 6.3 An example of solution encoding for the GA

#### 6.3.3.2 Genetic operators

To evolve a population of solutions, genetic operators such as crossover and mutation are used. We formulate our genetic operators as follows.

- *Crossover:* is used to combine the genetic information of two parents. In our adaptation, we use the standard single-point crossover operator. A sub-list is extracted from each parent. Then, the crossover operator exchanges the two sub-lists between parents. Figure 6.4 shown an example of the crossover operator applied to Parents 1 and 2 to produce two offspring solutions Child 1 and Child 2.
- *Mutation:* The mutation operator aims at adding slight modifications to a candidate solution. In our adaptation, the mutation operator first randomly selects one or more cells from a given candidate solution. Then, the selected cell(s) will be replaced by new randomly generated values. Figure 6.5 shows an example of mutation operators where three random cells from a parent solution are selection, *i.e.,* the *number of layers*, the *number of epochs* and the *optimizer*, and randomly replaced by other values.

#### 6.3.3.3 Solution evaluation (fitness function):

Each candidate solution should be evaluated to assess how good it is in solving the problem at hand. An appropriate fitness function should be defined to evaluate the fitness of a candidate

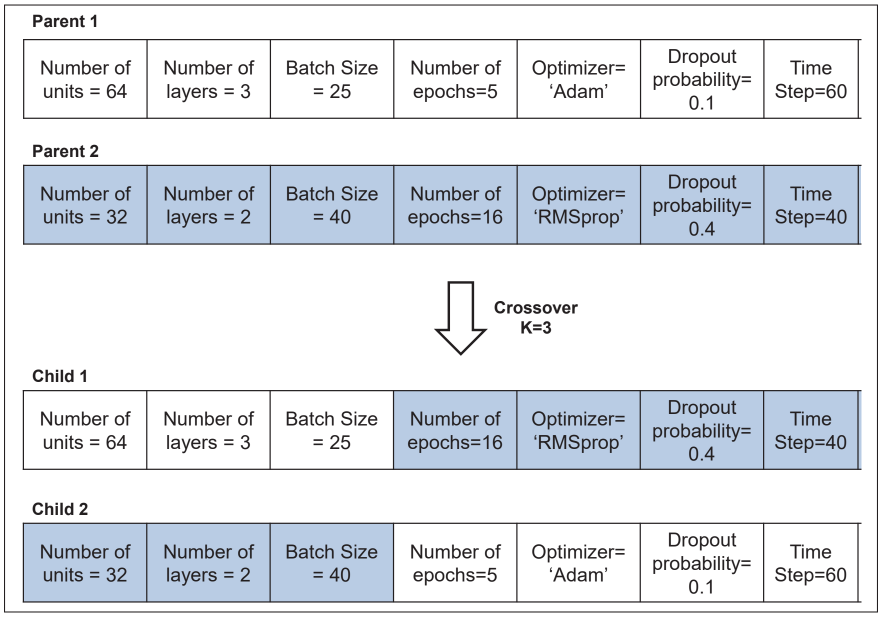


Figure 6.4 An example of crossover operator for the GA

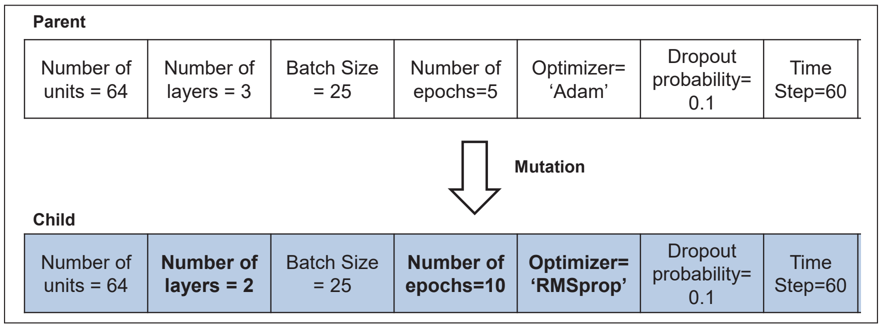


Figure 6.5 An example of mutation operator for the GA

solution, *i.e.*, the selected hyper-parameters to build out model. In this paper, we aim to optimize the architecture of our LSTM-RNN model by minimizing the validation loss (Li, Dong, Wang & Xu, 2020).

### 6.3.4 Data preprocessing

To train the model, we must first transform the data to a specific encoding that could be modeled with LSTM. The input data for LSTM, which consists of set of CI builds results, needs to be reshaped into a 3D array with the following dimensions *[samples, time steps, features]*, where the *samples* are the input data, *time steps* are the number of previous observations (which is tuned by GA) used to predict the next build result and *features* is the number of features considered to feed the network which corresponds to 1 as we use a single LSTM model. In Figure 6.6, we provide an example of input data (*i.e.,* a sequence of builds outcomes) and how it reshaped with a time step = 5.

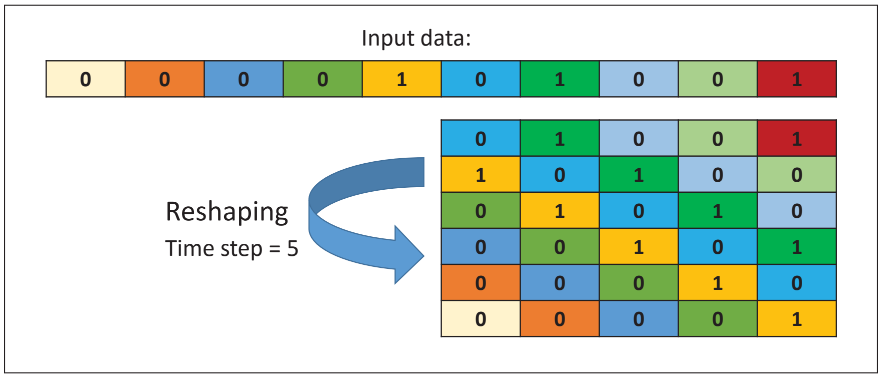


Figure 6.6 An example of data preprocessing

### 6.3.5 CI Build Prediction based on Threshold Moving strategy

In our case, LSTM works as a binary classifier where the output is the probability of class membership (*i.e.,* the probability of new build to fail) and this must be interpreted before it can be mapped to a class label (*e.g.,* failed or passed). Hence, it is crucial to set the decision threshold above which all values are mapped to one class and all other values are mapped to another class. Threshold moving has brought the attention from the DL research community (Collell, Prelec & Patil, 2018; Buda, Maki & Mazurowski, 2018; Krawczyk & Woźniak, 2015; Zhou & Liu, 2005; Zheng, 2010) as a solution to handle the imbalanced distribution of class examples in time series data, which is indeed the case of build failure prediction (Xie & Li, 2018). This solution refers to tuning the threshold used to map probabilities to class labels as the default value (=0.5) can lead to a poor predictive performance when the data is imbalanced (Provost, 2000).

## 6.4 Empirical Study Setup

In this section, we describe the design of empirical study that we performed to evaluate our approach, *DL-CIBuild*, based on the TravisTorrent dataset (Beller *et al.*, 2017).

Figure 6.7 provides an overview of our experimental design. First, we start by selecting the suitable optimization technique for our DL approach (RQ1). Then to validate the predictive performance, we compare our results with five widely-used Machine Learning (ML) techniques including Decision Tree (DT) (Quinlan, 2014), Random Forest (RF) (Breiman, 2001), AdaBoost (ADA) (Schapire, 2013; Ni & Li, 2017), Support Vector Classification (SVC) (Hsu, Chang, Lin *et al.*, 2003) and Logistic Regression (LR) (Bishop). We first consider online validation (Xia & Li, 2017) (RQ2). Then, we investigate the generalizability of identifying CI build failures by applying cross-project validation using the *Bellwether strategy* (Xia *et al.*, 2017a) (RQ3). Lastly, we evaluate the sensitivity of our DL approach to the training size while comparing its performance against the other ML techniques (RQ4). In the following, we describe our validation in detail.

### 6.4.1 Replication Package

We provide our replication package available at (Saidani, 2020c). Specifically, we provide a comprehensive dataset, the source code of *DL-CIBuild* and the benchmark models (*i.e.,* RF, ADA, DT, LR and SVC). We also provide detailed instructions on how to run the code and replicate all the experiments we reported in this paper for future replications and extensions.

### 6.4.2 Data

Our experiments are based on TravisTorrent dataset, from which we selected top-10 projects according to the number of build records. An overview about the studied projects is reported in Table 6.2. It is worth noting that the dataset is highly imbalanced as reported in Table 6.2 with an average failure rate of 0.3.

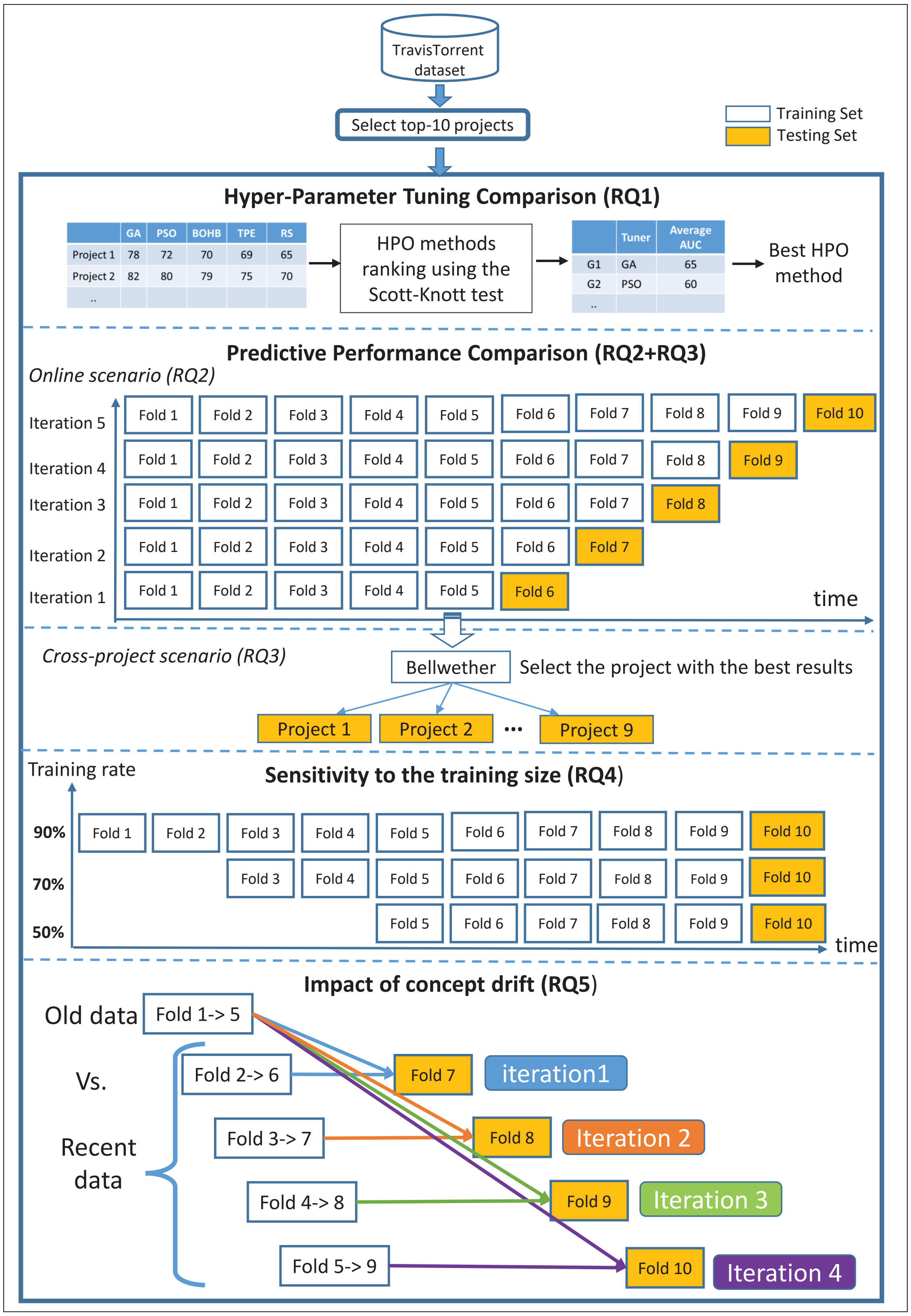


Figure 6.7 Experimental design

Table 6.2 Studies projects statistics

| **Project** | **Language** | **Number of builds** | **Failure** (%) | **Age at CI** (in days) |
|---|---|---|---|---|
| rails/rails | Ruby | 19,447 | 35 | 2,354 |
| ruby/ruby | Ruby | 15,388 | 22 | 5,099 |
| jruby/jruby | Ruby | 12,085 | 62 | 1,074 |
| rapid7/metasploit-framework | Ruby | 8,839 | 8 | 2,571 |
| apache/jackrabbit-oak | Java | 8,205 | 42 | 102 |
| opf/openproject | Ruby | 7,088 | 36 | 287 |
| CloudifySource/cloudify | Java | 5,742 | 26 | 220 |
| Graylog2/graylog2-server | Java | 5,199 | 11 | 470 |
| SonarSource/sonarqube | Java | 4,690 | 27 | 1,013 |
| openSUSE/open-build-service | Ruby | 4,647 | 29 | 341 |

*Rails*[1] is a web application framework that provides several features needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. *Ruby*[2] is an interpreted object-oriented programming language often used for web development. *JRuby*(Jruby, 2019) is an implementation of Ruby on the JVM. *Metasploit*[3] is a penetration testing platform that enables to write, test, and execute code with a suite of tools. *Jackrabbit Oak*[4] is a scalable, high-performance hierarchical content repository designed for use as the foundation of modern web sites and content applications. *OpenProject*[5] is one of the leading open source web-based project management systems. *Cloudify*[6] is a cloud-enablement platform that on-boards applications to public and private clouds without architectural or code changes. Graylog2-server[7] is a log management system that centrally captures, stores, and enables real-time search and log analysis. *Vagrant*[8] is a tool for building and distributing development environments with a declarative

[1] https://github.com/rails/rails

[2] https://github.com/ruby/ruby

[3] https://github.com/rapid7/metasploit-framework

[4] https://github.com/apache/jackrabbit-oak

[5] https://github.com/opf/openproject

[6] https://github.com/CloudifySource/cloudify

[7] https://github.com/Graylog2/graylog2-server

[8] https://github.com/hashicorp/vagrant

configuration file. SonarQube[9] is a popular platform for continuous inspection of code quality. Finally, the *Open Build Service*[10] is a generic system to build and distribute binary packages from sources in an automatic, consistent and reproducible way.

### 6.4.3 Research Questions

We designed our experiments to answer five research questions:

**RQ1. (Hyper-Parameter Optimization Comparison) How effective is GA for HPO compared to existing techniques?**

**Motivation.** To fit our approach into the CI build prediction problem, we must first tune their hyper-parameters. Since there are many different HPO methods with different use cases, it is crucial to evaluate the need for an intelligent method such as GA.

**Approach.** In order to verify the performance of GA, we compare it with four methods. These techniques are selected based were chosen based on their popularity (Yang & Shami, 2020), diversity (belonging to different families) and availability in Python:

1. *Random Search (RS):* A HPO technique that belongs to the family of model-free algorithms. This method was proposed to overcome certain limitations of Grid Search (GS) related mainly to the computational costs random search (RS). RS is similar to GS; but, instead of testing all values in the search space, RS randomly selects a pre-defined number of samples between the upper and lower bounds. To implement RS, we use *Hyperopt* (HpBandSter, 2021b) (the HPO Python framework) while selecting *rand.suggest* algorithm. With regards to its performance, Bergstra & Bengio (2012) argued that RS is more effective than GS.
2. *Tree-structured Parzen Estimators (TPE):* is a Bayesian optimization (BO) based method that, unlike GS and RS, determines the future evaluation points based on the previously-obtained results. This technique has been widely applied in practice (Xia, Liu, Li & Liu,

[9] https://github.com/SonarSource/sonarqube

[10] https://github.com/openSUSE/open-build-service

2017b; Guo, Hu, Wu, Peng & Wu, 2019). We use the implementation of this technique as provided by *Hyperopt* using the algorithm *tpe.suggest*.

3. *Bayesian Optimization HyperBand (BOHB):* is a multi-fidelity optimization technique that uses a subset of the original to solve the constraint of limited time and resources. It has been shown that BOHB outperforms many other optimization techniques when tuning SVM and DL models (Falkner, Klein & Hutter, 2018). To implement this technique, we use *HpBandSter* Python library (HpBandSter, 2021a).
4. *Particle Swarm Optimization (PSO):* is another meta-heuristic conceived by Shi & Eberhart (1998) that has been widely adopted for complex HPO problems (Tharwat & Hassanien, 2019; Lorenzo, Nalepa, Kawulok, Ramos & Pastor, 2017). Note that PSO is supported in *Optunity* HPO framework (Optunity, 2019) as the default option.

In order to ensure a fair comparison, some constraints should be satisfied. First, we evaluate the different HPO methods using the same hyper-parameter configuration space. Table 6.3 summarizes the configuration space for LSTM model. Additionally, since the studied HPO methods evaluate the candidate configurations based on an objective function to be optimized, we use the same function to be optimized namely the *validation loss*. The training loss functions are threshold-independent metrics *i.e.,* not sensitive to imbalanced data (Tantithamthavorn *et al.*, 2018a).

Table 6.3 Configuration space for the hyper-parameters of LSTM

| **Hyper-parameters** | **Search Space** |
|---|---|
| Number of units | range [32,512] |
| Number of layers | range [1,7] |
| Batch Size | range [4,256] |
| Number of epochs | range [2,10] |
| Optimizer | [‘adam’, ‘rmsprop’] |
| Dropout probability | range [0.01,0.3] |
| Time step | range [30,120] |

After that, to deal with the stochastic nature of HPO methods, we repeat each experiment 31 times and the median performance is reported as the performance estimate, as recommended by

Arcuri & Briand (2011). On the other hand, we set the maximum number of iterations to 50 for RS, TPE, PSO and BOHB; while we set the number of generations and population size to 5 and 10 respectively for GA ($5 * 10 = 50$).

In the next step, the performance metrics are selected. For each experiment on the selected ten datasets, online validation (cf. Section 6.4.3) is considered to evaluate the studied HPO methods. First, the Area Under the ROC Curve (AUC) (cf. Section 6.4.4) is used as the classification performance metric. Additionally, the computational time (CT), the total time needed to complete an experimentation, is also used as the efficiency metric (Yang & Shami, 2020; Wicaksono & Supianto, 2018; Xia *et al.*, 2017b; Tantithamthavorn *et al.*, 2018a). Note that AUC is computed on the testing set while CT is calculated on the training set as the HPO methods are applied on this set.

It is also worth mentioning that in this work, all the experiments are executed on a computer equipped with an Intel Core i7-8700k CPU  3.20 GHz and using 64-bit based Windows.

**RQ2. (Within-project validation) How does our *DL-CIBuild* approach perform compared to ML techniques within projects?**

**Motivation.** The first *goal* of our empirical study is to evaluate the performance of our DL-based approach for the CI build failures prediction problem against existing ML techniques. Thus, we want to investigate the efficiency of considering the time series dataset which consists of a sequential data of CI build outcomes against the use of ML techniques trained on state-of-the-art CI related features to assist developers in automatically identifying build failure.

**Approach.** We conduct an online validation in which builds are ordered and predicted chronologically. Similar to prior work by Xia and Li (Xia & Li, 2017), we ranked for each selected project, the builds according to its start time and broke the whole set of a given project into ten folds. Then, we used the latter five folds as testing sets: At each iteration $i$ ($1 \leq i \leq 5$), the test set fold $j$ ($6 \leq j \leq 10$), the former $j - 1$ folds are selected as training set to train the model. It is worthy to mention, that we verified for each project and validation iteration, the existence of failed builds.

**RQ3. (Cross-project validation) How effective is our approach compared to ML techniques when applied on cross-projects?**

**Motivation.** Building a model to predict CI build failure requires having labeled data to train on. However, in real world situation, many projects do not have sufficient historical labeled data to build a classifier (Xia *et al.*, 2017a) (*e.g.,* small or new project) which may prevent the project team from using a prediction tool. In this research question, we investigate to what extent a build failure prediction can be generalized through cross-project prediction.

**Approach.** Cross-project validation is a the-state-of-art technique to solve the lack of training data in software engineering (Xia *et al.*, 2017a). Specifically, we adopt *Bellwether* strategy (Krishna, Menzies & Fu, 2016) as the project-level filter. The Bellwether strategy is a recently introduced source filtering method that can further improve prediction results of existing filtering methods, as reported by Xia et al. (Xia *et al.*, 2017a). In this strategy, the *Bellwethers* are selected as the best source projects according to previous prediction result, and considered as the source projects in the following cross-project prediction. In this section, we select the bellwether as the project providing the best results within online validation (RQ1).

**RQ4. (Sensitivity to training size) How effective is our approach when varying the training set size?**

**Motivation.** After validating the effectiveness of *DL-CIBuild* under two validation scenarios, we want to go further by showing the effects of the training data on the effectiveness of our technique compared to ML techniques which remains unknown. Knowing the impact of the size of training set is important, as it allows us to estimate the performance of *DL-CIBuild* when a small amount of data is provided. Also, given the same amount of data, the best scores we get, the more useful an approach is.

**Approach.** Using the same dataset described in Section 6.4.2, we train and evaluate our approach against baseline techniques based on different training sizes. Similarly to RQ2, we split the data into 10 folds sorted by the time of the build; then, we vary the size of the training set while using the same testing fold in each experiment. In the first experiment, 50% of the datasets are used to construct the predictive models. In the second experiment, the datasets used for the model construction are increased to 70%. In the third experiment, the datasets used for model construction are increased to 90%. In the testing phase, we compare the predictive performance as described in the next section.

**RQ5. (Concept drift) To which extent is our approach robust to concept drift?**

**Motivation.** As the time passes, data can change. In some cases, the performance of the prediction models can degrade because the learned relationship between the input and output variables is no longer valid. This problem is called *concept drift* (Widmer & Kubat, 1996; Tsymbal, 2004) which should be detected and addressed to ensure the successful application of ML/DL based techniques (Singh, Walenstein & Lakhotia, 2012; Ekanayake, Tappolet, Gall & Bernstein, 2009; Zenisek, Holzinger & Affenzeller, 2019). In this paper, we aim to investigate whether the CI build failure prediction drifts over time using *DL-CIBuild*. This would help us assess the need of the model's retraining to prevent degradation in performance. On the other hand, if the drift is found to be negligible, this would indicate the robustness of our proposed approach.

**Approach.** To study the possible concept drift, we train and test the predictive performance of our approach over time against baseline techniques. As shown in Figure 6.7, we first split the data into 10 folds sorted by the time of the build. In the first iteration, we train the models using folds 1 to 5 (old data) and folds 2 to 6 (recent data) and compare the predictive performance on the fold 7. In the second iteration, we compare the old data (*i.e.,* folds from 1 to 5) to the folds 3 to 7 and test both data on fold 8 etc. In this way, we assess the effectiveness of the approaches based on data from two different time periods in order to assess whether the predictive performance drifts over time.

### 6.4.4 Evaluation Metrics

To evaluate the predictive performance (*i.e.,* RQ2-5), we first compute the widely-used performance evaluation metric **F1-score** which is defined as follows:

$$F1\text{-}score = 2 * \frac{Precision * Recall}{Precision + Recall} \in [0,1] \tag{6.1}$$

In our study, the recall is the percentage of correctly classified failed builds relative to all of the builds that actually failed while the precision is the percentage of detected failed builds that actually failed. These metrics are defined as follows:

$$Recall = \frac{TP}{TP + FN} \in [0,1] \tag{6.2}$$

$$Precision = \frac{TP}{TP + FP} \in [0,1] \tag{6.3}$$

where *TP* is the number of failed builds that are correctly classified as CI failed; *FP* denotes the number of passed builds classified as failed; and *FN* measures the number of classes of actual CI failed builds that identified as passed.

The second metric we consider in this study the **Accuracy**. It refers to the proportion of correct predictions made by the model. Formally, Accuracy is defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \in [0,1] \tag{6.4}$$

Moreover, it is important to account for imbalance in a data set as generally failed builds are much less to occur than past ones in typical software projects (Xie & Li, 2018). Hence, we consider AUC measure which indicates how much a prediction model/rule is capable of distinguishing between classes. A larger AUC value indicates better prediction performance. The main merit of the AUC is its robustness toward imbalanced data. For binary classification, AUC is defined as follows (Cervantes *et al.*, 2013):

$$AUC = \frac{1 + \frac{TP}{TP+FN} - \frac{FP}{FP+TN}}{2} \in [0, 1] \tag{6.5}$$

### 6.4.5 Machine learning benchmark

We compare the prediction performance of our *DL-CIBuild* approach with five widely-used ML techniques in previous CI and software engineering research (Xia *et al.*, 2017a; Ni & Li, 2018; Xia & Li, 2017; Luo *et al.*, 2017; Hassan & Wang, 2017; Ni & Li, 2018; Santolucito *et al.*, 2018), namely Decision Tree (DT), Random Forest (RF), AdaBoost (ADA), Support Vector Classification (SVC) and Logistic Regression (LR). The initial input to these models is a set of features comprising 21 state-of-the-art CI features from TravisTorrent dataset (Xia *et al.*, 2017a; Ni & Li, 2018; Xia & Li, 2017; Luo *et al.*, 2017; Hassan & Wang, 2017; Ni & Li, 2018; Santolucito *et al.*, 2018). These features are summarized in Table 6.4.

#### 6.4.5.1 Data pre-processing

Data pre-processing is a vital step to obtain better performance of ML models which comprises data cleansing, normalization, and structure change(Hastie, Tibshirani & Friedman, 2009). As ML models are sensitive to the scale of the inputs, the data are normalized in the range $[0, 1]$ by using feature scaling. Also, to mitigate the issue related to the imbalanced nature of the dataset, we rely on Synthetic Minority Oversampling Technique (SMOTE) method (Chawla *et al.*, 2002), to resample the training data. Note, that we did not resample the testing dataset since we want to evaluate ML techniques in a real-life scenario, where the data is imbalanced. Additionally, for the sake of fairness, we apply Threshold Moving (TM) to all ML techniques.

Table 6.4 CI-related features extracted from TravisTorrent

| Metric | Description | Reference |
|---|---|---|
| git_num_all_built_commits | Number of commits contained in this single build | (Xia *et al.*, 2017a),(Ni & Li, 2017),(Xia & Li, 2017),(Luo *et al.*, 2017),(Xie & Li, 2018) |
| gh_num_commits_on_files_touched | Number of unique commits on the files touched in the built commits | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| git_diff_src_churn | Number of lines of code changed in all built commits | (Xia & Li, 2017),(Luo *et al.*, 2017),(Hassan & Wang, 2017),(Xie & Li, 2018) |
| gh_diff_files_added | Number of files added in all built commits | (Xia *et al.*, 2017a),(Ni & Li, 2017),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_diff_files_deleted | Number of files deleted by all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_diff_files_modified | Number of files modified by all built commits | (Xia *et al.*, 2017a),(Ni & Li, 2017),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_num_commit_comments | Number of comments of all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| num_of_distinct_authors | Number of distinct authors in all built commits | (Xia *et al.*, 2017a),(Xie & Li, 2018) |
| gh_by_core_team_member | Whether the commit that has triggered the build was authored by a core team member | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_is_pr | Whether this build was triggered as part of a pull request on GitHub. | (Luo *et al.*, 2017) |
| gh_diff_src_files | Number of src files changed by all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017) |
| gh_diff_doc_files | Number of documentation files changed by all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_diff_other_files | Number of files which are neither source code nor documentation. | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |
| git_diff_test_churn | Number of lines of test code changed in all built commits | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) ,(Hassan & Wang, 2017) |
| gh_diff_tests_added | Number of test cases added in all built commits | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_diff_tests_deleted | Number of test cases deleted in all built commits | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_team_size | Number of developers that committed from the moment the build was triggered and 3 months back. | (Xia & Li, 2017),(Luo *et al.*, 2017),(Hassan & Wang, 2017) |
| gh_sloc | Number of source lines of code, in the entire repository at the time of this build. | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017),(Xie & Li, 2018) |
| gh_test_lines_per_kloc | Number of lines in test cases per 1000 gh_sloc. | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_test_cases_per_kloc | Number of test cases per 1000 gh_sloc. | (Xia & Li, 2017),(Luo *et al.*, 2017) |
| gh_asserts_cases_per_kloc | Number of assertions per 1000 gh_sloc. | (Xia *et al.*, 2017a),(Xia & Li, 2017),(Luo *et al.*, 2017) |

#### 6.4.5.2 Parameter tuning for Machine Learning techniques

We use the best HPO method as the one to be revealed in RQ1. In order to facilitate the replication of our results, we provide the selected main parameters and their respective search spaces for ML techniques as shown in Table 6.5.

Table 6.5 Configuration space for the hyper-parameters of ML models

| Model | Hyper-parameters | Search Space |
|---|---|---|
| ***SVC*** | C | ['linear', 'rbf'] |
| | kernel | range [1,10] |
| | max of iterations | range [200,5000] |
| ***DT*** | Criterion | ['gini', 'entropy'] |
| | max depth | range [10,100], None |
| | min samples split | range [2,10], None |
| | min samples leaf | range [1,5], None |
| | max features | ['sqrt', 'log2', None] |
| ***RF*** | Number of estimators | range [50,600] |
| | max depth | range [10,100], None |
| | Criterion | ['gini', 'entropy'] |
| | min samples split | range [2,10], None |
| | min samples leaf | range [1,5], None |
| | max features | ['sqrt', 'log2', None] |
| ***ADA*** | random state | [None,0] |
| | Number of estimators | range [50,600] |
| | Algorithm | ['SAMME', 'SAMME.R'] |
| | learning rate | range [0,1] |
| ***LR*** | max of iterations | range [200,5000] |
| | penalty | ['l1','l2','none'] |
| | solver | ['newton-cg', 'lbfgs', 'sag','saga','liblinear'] |

### 6.4.6 Inferential Statistical Test methods Used

When applied to the same problem instance, ML and LSTM models may provide different results on each run. To deal with this stochastic nature, it is important to assess their effectiveness by performing a large number of runs, at least 30 runs as suggested in (Arcuri & Briand, 2011).

Additionally, it is essential to use the statistical tests that provide support for/rejection of the conclusions derived by analyzing the obtained results.

#### 6.4.6.1 Statistical tests for RQ1, RQ4 and RQ5

To perform multiple comparison tests, we cluster the approaches using Scott-Knott Effect Size Difference (ESD) method (Tantithamthavorn, McIntosh, Hassan & Matsumoto, 2017, 2018b). Scott-Knott partitions the set of treatment means (*e.g.,* means of model performance) into statistically distinct groups with non-negligible difference (*i.e.,* $\rho{-}value < 0.05$). This clustering algorithm has been widely applied to different software engineering domains such as ranking the classification techniques (Ghotra, McIntosh & Hassan, 2015) and comparing HPO methods (Tantithamthavorn *et al.*, 2018a). We use the implementation of the Scott-Knott test provided by the *ScottKnott* R package (Jelihovschi, Faria & Allaman, 2014). The Scott-Knott test ranks each approach exactly once, however several approaches may appear within one rank.

#### 6.4.6.2 Statistical tests for RQ2 and RQ3

W employ Wilcoxon signed rank test (Wilcoxon *et al.*, 1970) in order to detect significant performance differences between the algorithms under comparison ($\alpha$ is set at 0.05). We also use the Cliff's delta, $\delta$, a non-parametric effect size measure for ordinal data (Cliff, 1993) to assess the difference magnitude. The effect size is considered negligible when $\mid \delta \mid < 0.147$, small when $0.147 \leq \mid \delta \mid < 0.33$, medium when $0.33 \leq \mid \delta \mid < 0.474$ and large otherwise (Romano, Kromrey, Coraggio & Skowronek, 2006).

### 6.5 Experimental Results

In this section, we present the results of our empirical study with respect to the five research questions.

### 6.5.1 RQ1. Results of HPO comparison

The experiments of applying GA and other four different HPO methods when applied to LSTM models are summarized in Table 6.6. This table shows the average performance of each HPO methods evaluated based on AUC and the Computational Time (CT).

With regards to AUC scores, we clearly see that meta-heuristics methods, GA and PSO showed significantly better performances than other HPO methods. Using PSO, the LSTM model can achieve 60% in terms of AUC, while with GA, it can achieve a better performance with an improvement of 5%. This confirms that meta-heuristic techniques are more suitable to complex search spaces as stated by previous studies (Yang & Shami, 2020). Then, we see that BOHB method have shown a better performance than TPE as excepted since BOHB combines the advantages of Bayesian optimization and Hyperband by using TPE as a standard surrogate model. Lastly, we have found that TPE and RS obtained 53% and 52% in terms of AUC respectively but with no significant difference.

With the same search space size, we have found that BOHB is faster than other HPO methods. Conversely, BOHB does not yield the best performance in our experiments. On the other hand, the computation time of RS and TPE is on average better than meta-heuristic algorithms due to their lower algorithmic complexities (Yang & Shami, 2020). In addition, PSO is faster than GA since the latter requires an additional computational time dedicated to genetic operations (*i.e.,* mutation and cross-over). But statistically, the difference is not significant (same ranking group).

Table 6.6 The ranking of the HPO methods for LSTM, divided into distinct groups that have a statistically significant difference in the mean

| **AUC** | | | **CT** | | |
|---|---|---|---|---|---|
| *Method* | *Rank* | *Avg (%)* | *Method* | *Rank* | *Avg(sec)* |
| GA | 1 | 65 | BOHB | 1 | 386 |
| PSO | 2 | 60 | RS | 2 | 520 |
| BOHB | 3 | 56 | TPE | 3 | 765 |
| TPE | 4 | 53 | PSO | 4 | 1,670 |
| RS | | 52 | GA | | 1,750 |

**Summary for RQ1.** *DL-CIBuild* can achieve higher predictive performance when using GA as HPO method with an improvement of 5% compared to PSO. But this comes with a higher computational time. Nevertheless, we use GA as HPO method in order to guarantee near-to-optimal configurations for LSTM models. For the sake of a fair comparison, we also use GA as HPO method for ML models.

### 6.5.2 RQ2. Results of online validation

Table 6.7 reports the average (of 5 online validation iterations) AUC, F1 and accuracy scores for each studied project. Note that *DL-CIBuild*, all the involved techniques are executed 31 times to deal with their stochastic nature. Then, we computed the median values of each experiment. Moreover, Table 6.8 shows the statistical comparisons of these experiments.

With regards to AUC, we clearly see that, for nine out of ten projects, the best scores are obtained by *DL-CIBuild* achieving on median 65% with an improvement of 7% over ML techniques. On the other hand, for the different projects, the statistical analysis provides evidence that our approach performs better than the ML techniques with *large* Cliff's delta effect sizes. For instance, in the *ruby* project for which we obtained the best AUC results, our approach achieved 74% in terms of AUC compared to 59% for RF, 58% for LR, 57% for ADA, 56% for SVC and 55% for DT; which represents an improvement of 15% over ML for this project. However, in the *sonarqube* and *metasploit-framework* project, RF was slightly better than *DL-CIBuild*. One explanation for this results could be related to the fact that the CI-related features are more efficient to predict the failure than the temporal information for these projects.

Overall, the results for AUC reveal that *DL-CIBuild* can reach a better trade-off (*i.e.,* balance) between both positive (*i.e.,* failed) and negative (*i.e.,* passed) accuracies, by applying threshold moving, than all the ML techniques even with resampling. This result lends support to previous results confirming that threshold-moving is a better choice in training cost sensitive neural networks (Zhou & Liu, 2005).

Looking at F1-scores, we also see that *DL-CIBuild* achieved the best results for 6 out of 10 projects with a median score of 49% with an improvement of 10% as compared to the results achieved by SVC (the best ML performing technique). The statistical tests reveal that *DL-CIBuild* outperforms ML with medium (compared to ADA, DT, LR and SVC) to large effect sizes (with RF). Exceptionally, in *sonarqube* project, we found evidence for LR algorithm to be better than *DL-CIBuild*. But overall, we clearly see that LR achieved poor performances in terms of F1-scores of 33% in median. This is especially the case for *graylog2-server* and *metasploit-framework* projects as LR turns out to be inefficient to correctly detect failed builds.

Broadly speaking, F1-score results demonstrate a compelling superiority of *DL-CIBuild* to identify more failed builds than ML techniques.

As for the accuracy scores, the obtained results also show that *DL-CIBuild* is a better performer than the five considered ML techniques, with a significant improvement of 11% in median, and large effect sizes as shown in Table 6.8. Additionally, the accuracy scores of our approach range from 63% to 85% while achieving in median a high score of 72% and for 9 out of 10 projects, the accuracy values of *DL-CIBuild* exceed those of ML techniques.

To sum up, it is worth noting that, due to the highly imbalanced nature of the analyzed data (*i.e.,* only a small portion of the builds are failed) as can be seen from Table 6.2, the achieved AUC, F1 and accuracy results by *DL-CIBuild* are considered significant. Furthermore, we can see from the statistical results, that ML modest performance may not be only related to the nature of the dataset as we applied resampling to the training data using SMOTE, but this could be related to the complex and erratic temporal dependencies between the builds that are hard to capture with traditional ML techniques. Thus, DL-based time series models seem more appropriate to such a problem.

Table 6.7 Performance of *DL-CIBuild* vs ML techniques under online validation

| | AUC | | | | | | F1 | | | | | | Accuracy | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Projects** | *DL-CIBuild* | *DT* | *LR* | *ADA* | *RF* | *SVC* | *DL-CIBuild* | *DT* | *LR* | *ADA* | *RF* | *SVC* | *DL-CIBuild* | *DT* | *LR* | *ADA* | *RF* | *SVC* |
| cloudify | **72** | 52 | 58 | 58 | 61 | 53 | **52** | 23 | 26 | 28 | 34 | 29 | **85** | 51 | 62 | 72 | 72 | 41 |
| graylog2-server | **64** | 53 | 60 | 57 | 59 | 59 | **30** | 12 | 14 | 12 | 14 | 14 | **72** | 59 | 53 | 42 | 59 | 61 |
| jackrabbit-oak | **61** | 52 | 54 | 57 | 57 | 54 | 52 | **56** | 31 | 54 | 45 | 48 | **63** | 51 | 42 | 55 | 48 | 49 |
| jruby | **69** | 53 | 55 | 53 | 54 | 53 | **77** | 70 | 68 | 55 | 41 | 61 | **72** | 61 | 59 | 60 | 47 | 55 |
| metasploit-framework | 60 | 55 | 63 | 57 | **64** | 53 | 22 | 17 | 23 | 19 | **24** | 15 | **81** | 75 | 70 | 56 | 79 | 70 |
| open-build-service | **67** | 53 | 60 | 59 | 56 | 58 | **46** | 27 | 38 | 36 | 31 | 35 | **77** | 56 | 49 | 48 | 55 | 54 |
| openproject | **62** | 52 | 53 | 53 | 53 | 51 | 45 | 41 | 31 | 37 | 39 | **47** | **70** | 55 | 55 | 57 | 58 | 47 |
| rails | **66** | 52 | 54 | 52 | 54 | 51 | **52** | 32 | 35 | 16 | 34 | 43 | **69** | 61 | 60 | 65 | 63 | 34 |
| ruby | **74** | 55 | 58 | 57 | 59 | 56 | **64** | 51 | 49 | 43 | 40 | 50 | **77** | 54 | 47 | 57 | 63 | 54 |
| sonarqube | 57 | 57 | 62 | 60 | **63** | 59 | 35 | 36 | **44** | 40 | 43 | 34 | 66 | 63 | 70 | 70 | 71 | **77** |
| **Median** | **65** | 53 | 58 | 57 | 58 | 53 | **49** | 34 | 33 | 37 | 37 | 39 | **72** | 57 | 57 | 57 | 61 | 54 |
| **Average** | **65** | 53 | 58 | 56 | 58 | 55 | **47** | 36 | 36 | 34 | 35 | 38 | **73** | 58 | 57 | 58 | 62 | 54 |

Table 6.8 Statistical tests results of *DL-CIBuild* compared to ML techniques under online validation

| ***DL-CIBuild*** | | vs. ADA | vs. DT | vs. LR | vs. RF | vs. SVC |
|---|---|---|---|---|---|---|
| **Accuracy** | *p-value* | $10^{-9}$ | $10^{-9}$ | $10^{-11}$ | $10^{-6}$ | $10^{-10}$ |
| | *Effect Size* | Large | Large | Large | Large | Large |
| **AUC** | *p-value* | $10^{-11}$ | $10^{-16}$ | $10^{-9}$ | $10^{-9}$ | $10^{-14}$ |
| | *Effect Size* | Large | Large | Large | Large | Large |
| **F1** | *p-value* | $10^{-5}$ | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-3}$ |
| | *Effect Size* | Medium | Medium | Medium | Large | Medium |

**Summary for RQ2.** *DL-CIBuild* can achieve higher predictive performance than state-of-the-art ML techniques with a statistical significance under online-validation. Instead it achieved, in median, 65% and 49% in terms of AUC and F1-score respectively while reaching 72% of the overall classification accuracy. Moreover, we find that *jruby* project results outperform all the other projects by achieving the best scores in median. Thus, we select this project as the source (*i.e.,* training set) project in the following cross-project prediction.

### 6.5.3 RQ3. Results of cross-projects validation

As mentioned earlier, *jruby* project exhibited the highest prediction capability among the studied projects by achieving the best scores on average (and in median) and it is considered as the *Bellwether* for cross-project strategy. Hence, we train *DL-CIBuild*, based on *jruby* project, using our evaluation metrics, the Area Under the ROC Curve (AUC), F1-score, and accuracy values, to measure the performance of our classifier. Table 6.9 presents the effectiveness of cross-project modeling compared to ML techniques while Table 6.10 reports the statistical tests results.

First, the results show that *DL-CIBuild* achieves a performance of AUC value of 72% in median which ranges from 63-82%. Six projects out of nine show good performance results (≥ 70%) and *cloudify* achieves high AUC value of 82%. Compared to within-project validation, these results show that our approach can achieve with cross-projects a significant improvement of 7% in median over online validation with a large effect size. Except for *ruby* whose AUC score slightly decreased from 74% to 73%, may be because the data in this project is larger than the bellwether data, all the studied projects show a better performance which indicates that *DL-CIBuild* a very promising solution to mitigate the lack of data, especially for new software projects. Additionally, we observe an improvement of 17% in median over ML techniques whose results are worse than their within-project scores. The statistical tests results show that the difference is significant with large effect sizes.

The same observations for AUC can be applied to F1-score for which we recorded for *DL-CIBuild* a significant improvement of 8% compared to within-project results with a medium effect size. Also, *DL-CIBuild* is the best technique across all the studied projects by achieving in median 57% compared to ML techniques that showed modest to low F1-scores of 37% for SVC, 35% for LR and ADA, 34% for RF and 31% for DT. The statistical tests results show that *DL-CIBuild* is significantly better with large effect sizes, as reported in Table 6.10. Another observation to report from these results is that all ML techniques have shown a drop in F1-scores; which confirms previous findings in the literature who pointed out that ML techniques are less effective for cross-project prediction (Choetkiertikul *et al.*, 2018; Abdalkareem *et al.*, 2020; Zhang *et al.*, 2016). This result shows that when building ML techniques under cross-project prediction, the target project has a low collinearity with the source project features.

Looking at the classification accuracy, we see that the scores are significantly improved compared to within project results, with a small effect size, for eight projects out of nine by achieving in median 78% (and 80% on average) and the accuracy values range from 68-89%. Similarly to online validation, *DL-CIBuild* obtained better accuracy results compared to ML with significant differences and large effect sizes.

Table 6.9 Performance of *DL-CIBuild* vs ML techniques under cross-project

| | **AUC** | | | | | | **F1** | | | | | | **Accuracy** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Projects** | *DL-CIBuild* | *DT* | *LR* | *ADA* | *RF* | *SVC* | *DL-CIBuild* | *DT* | *LR* | *ADA* | *RF* | *SVC* | *DL-CIBuild* | *DT* | *LR* | *ADA* | *RF* | *SVC* |
| cloudify | **82** | 52 | 50 | 52 | 56 | 51 | **76** | 31 | 2 | 36 | 34 | 41 | **89** | 52 | 74 | 39 | 66 | 29 |
| graylog2-server | **77** | 53 | 50 | 55 | 55 | 53 | **51** | 16 | 12 | 20 | 20 | 21 | **87** | 65 | 42 | 66 | 54 | 23 |
| jackrabbit-oak | **74** | 50 | 54 | 57 | 53 | 57 | **70** | 46 | 55 | 59 | 48 | 59 | **76** | 49 | 52 | 53 | 53 | 53 |
| metasploit-framework | **63** | 52 | 60 | 54 | 53 | 52 | **29** | 13 | 21 | 12 | 14 | 15 | **86** | 65 | 72 | 76 | 79 | 41 |
| open-build-service | **70** | 51 | 56 | 52 | 52 | 52 | **57** | 28 | 35 | 35 | 43 | 26 | **78** | 52 | 65 | 47 | 43 | 63 |
| openproject | **63** | 51 | 55 | 53 | 52 | 51 | **51** | 33 | 34 | 31 | 25 | 22 | **68** | 57 | 63 | 60 | 62 | 58 |
| rails | **72** | 50 | 55 | 51 | 52 | 50 | **62** | 43 | 40 | 43 | 39 | 51 | **77** | 42 | 59 | 46 | 53 | 35 |
| ruby | **73** | 50 | 55 | 51 | 52 | 52 | **61** | 28 | 37 | 23 | 23 | 37 | **85** | 42 | 46 | 60 | 63 | 31 |
| sonarqube | **64** | 51 | 55 | 57 | 52 | 50 | **46** | 42 | 41 | 41 | 36 | 43 | **77** | 34 | 52 | 58 | 46 | 28 |
| **Median** | **72** | 51 | 55 | 53 | 52 | 52 | **57** | 31 | 35 | 35 | 34 | 37 | **78** | 52 | 59 | 58 | 54 | 35 |
| **Average** | **71** | 51 | 54 | 53 | 53 | 52 | **56** | 31 | 31 | 33 | 31 | 35 | **80** | 51 | 58 | 56 | 58 | 40 |

Table 6.10 Statistical tests results of *DL-CIBuild* under cross-projects compared to its achieved within-project results as well as ML techniques

| ***DL-CIBuild*** | | ***vs. online validation*** | ***vs. DT*** | ***vs. LR*** | ***vs. ADA*** | ***vs. RF*** | ***vs. SVC*** |
|---|---|---|---|---|---|---|---|
| **AUC** | *p-value* | < 0.001 | < 0.001 | < 0.001 | < 0.001 | < 0.001 | < 0.001 |
| | *Effect Size* | Large | Large | Large | Large | Large | Large |
| **F1** | *p-value* | < 0.001 | < 0.001 | < 0.001 | < 0.001 | < 0.001 | < 0.001 |
| | *Effect Size* | Medium | Large | Large | Large | Large | Large |
| **Accuracy** | *p-value* | 0.01 | < 0.001 | 0.002 | 0.002 | 0.002 | < 0.001 |
| | *Effect Size* | Small | Large | Large | Large | Large | Large |

**Summary for RQ3.** Our findings show a substantial improvement for *DL-CIBuild* compared to online validation results by achieving 72%, 57% and 78% in terms of AUC, F1-score and accuracy, respectively. These results indicate that our approach is effective when learning from a cross-project training corpus. We explain these results by the fact that in a cross-project setting, our approach is fed with more data. Moreover, our proposed approach still outperforms state-of-the-art ML techniques.

### 6.5.4 RQ4. Results of the sensitivity to training size

We have validated the effectiveness of *DL-CIBuild* in terms of AUC, F1-score and accuracy through the RQ2 and RQ3. In this experiment, we want to go further by assessing the extent to which our approach can perform when varying different amounts of data compared to other ML techniques.

Figure 6.8 presents the performance (in terms of AUC, F1 and accuracy) on the test dataset, of the studied approaches, after training for 31 times with 50%, 70% and 90% of the dataset. Additionally, Table 6.11 shows the rank differences for all of the studied approaches when varying the training sizes.

Looking at the plotted boxplots of *DL-CIBuild*, we observe that, for all the computed measures, the performance of our approach increases up to 90% of the datasets for which the best scores were recorded. For instance, increasing the training from 50% to 90% of the datasets, results in an improvement of 3%, 5% and 9% in terms of AUC, F1 and accuracy in median respectively. Moreover, Table 6.11 shows that there is a clear separation of AUC, F1 and accuracy scores of *DL-CIBuild* into distinct Scott-Knott ranks for 50% and 90% of training data. However, the scores seem comparable when training on 70% and 90% of the datasets; which means that our approach plateaus out from 70%. Nevertheless, we can conjecture that an advantage of using our approach in practice is that, as the project ages and more CI build records are available, *DL-CIBuild* will reach higher scores.

As compared to ML techniques, we clearly see that *DL-CIBuild* is better across different training set sizes. Moreover, Table 6.11 shows that for AUC scores, *DL-CIBuild* is statistically better than other techniques even when trained only on 50% of the datasets. As for F1 and accuracy scores, we see that *DL-CIBuild* share the same ranking with other ML but achieves better scores. Overall, we conjecture that, for different training sizes, our approach is more suitable than the ML techniques.

Table 6.11 The ranking of the approaches when varying the training size, divided into distinct groups that have a statistically significant difference in the average (Avg)

| **AUC** | | | **F1** | | | **Accuracy** | | |
|---|---|---|---|---|---|---|---|---|
| *Approach* | *Avg(%)* | *Rank* | *Approach* | *Avg(%)* | *Rank* | *Approach* | *Avg(%)* | *Rank* |
| *DL-CIBuild*-90 | 64 | 1 | *DL-CIBuild*-90 | 48 | 1 | *DL-CIBuild*-90 | 69 | 1 |
| *DL-CIBuild*-70 | 63 | | *DL-CIBuild*-70 | 46 | | *DL-CIBuild*-70 | 68 | |
| *DL-CIBuild*-50 | 61 | 2 | *DL-CIBuild*-50 | 43 | 2 | RF-50 | 64 | 2 |
| RF-50 | 60 | 3 | RF-50 | 39 | | RF-70 | 62 | |
| ADA-70 | 60 | 4 | LR-70 | 37 | 3 | DT-50 | 62 | |
| RF-90 | 60 | | LR-50 | 37 | 4 | ADA-50 | 62 | |
| RF-70 | 59 | | RF-90 | 36 | | ADA-70 | 62 | |
| ADA-90 | 59 | | LR-90 | 36 | | DT-90 | 60 | |
| LR-70 | 59 | | RF-70 | 35 | | RF-90 | 60 | |
| LR-50 | 59 | | ADA-50 | 35 | | ADA-90 | 60 | |
| ADA-50 | 59 | | DT-70 | 35 | | *DL-CIBuild*-50 | 60 | |
| LR-90 | 58 | | SVC-90 | 34 | | DT-70 | 59 | |
| DT-90 | 56 | 5 | DT-50 | 34 | | SVC-50 | 57 | 3 |
| DT-50 | 56 | | DT-90 | 33 | | SVC-70 | 56 | |
| DT-70 | 56 | | SVC-70 | 32 | | LR-70 | 56 | |
| SVC-50 | 56 | | ADA-90 | 31 | | LR-50 | 55 | |
| SVC-70 | 55 | | ADA-70 | 31 | | SVC-90 | 53 | |
| SVC-90 | 54 | | SVC-50 | 31 | | LR-90 | 51 | |

**Summary for RQ4.** The sensitivity analysis shows that our approach is more effective in CI build failure prediction than other ML techniques considering different training sizes. Although *DL-CIBuild* is able to work well for reduced amount of training data, its performance can be further improved within larger datasets.

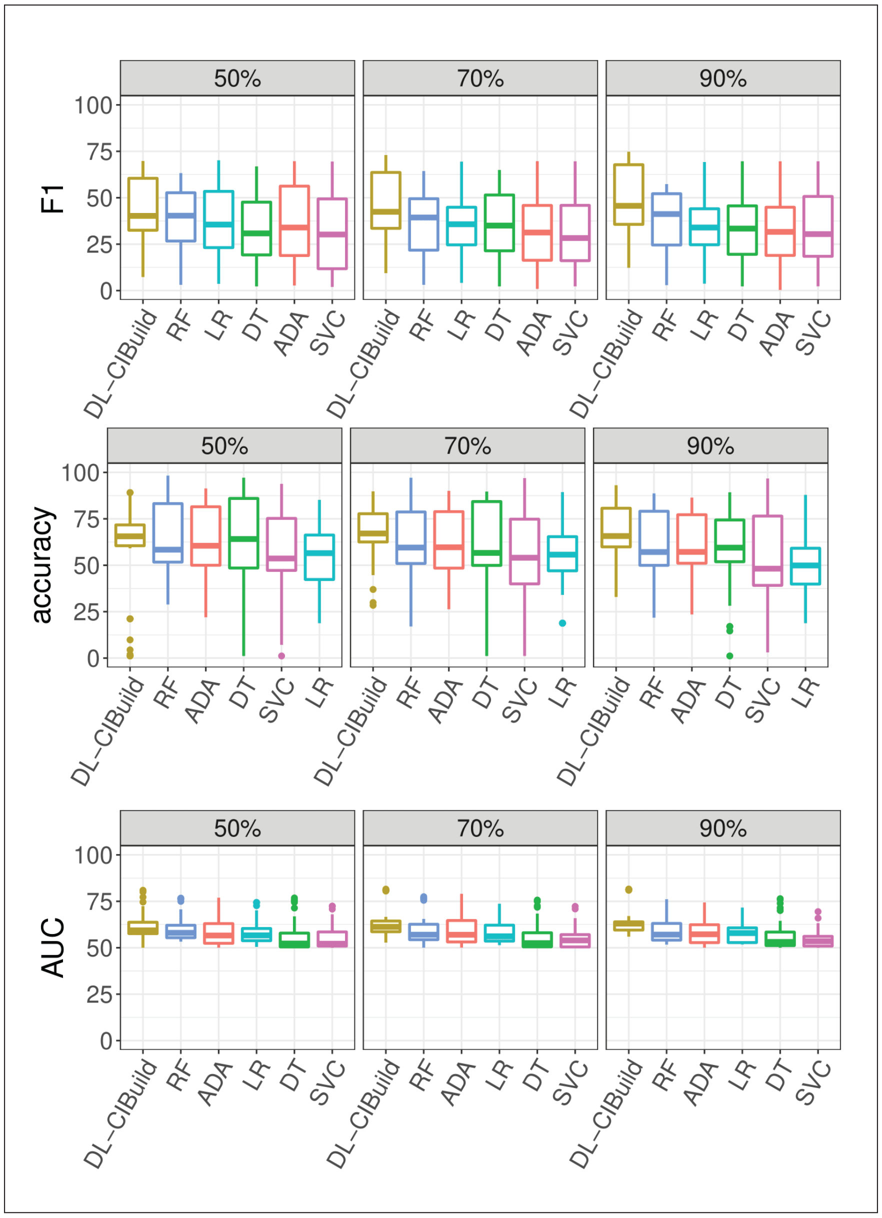


Figure 6.8 Comparison of the prediction performance with different training sets sizes

### 6.5.5 Results of the concept drift evaluation

In the last evaluation, we study the extent to which the approaches under evaluation suffer from concept drift (*i.e.,* the degradation in the predictive performance over time (Widmer & Kubat, 1996)). Figure 6.9 presents the performance (in terms of AUC, F1 and accuracy) on the test dataset, of the studied approaches, after training for 31 times using old (in red) and recent (in blue) training data. Additionally, Table 6.11 shows the ranks for all of the studied approaches when training in different time intervals.

As shown in Figure 6.9, we observe that, for all the computed measures, the performance of *DL-CIBuild* slightly increases when training the models on recent training data. For instance, we recorded a median improvement of 2% in terms of AUC when training on more recent data. However, when looking at Table 6.12, we see that the obtained results on both recent and old data seem to be comparable (same ranking group). These results indicate that while predicting the next builds is better when training on more recent build records, the data stream does not seem to be drifting for our approach. Thus, the models of *DL-CIBuild* do not need to be frequently retrained.

With regard to ML techniques, we found a significant drift in the performance of LR and ADA techniques while the RF, SVC and DT seem to be more robust to the concept drift as their results seem to be comparable using both old and recent data. But overall, we conjecture that, for different time intervals, our approach is more suitable than the ML techniques.

Table 6.12 The ranking of the approaches when varying the training time, divided into distinct groups that have a statistically significant difference in the average (Avg)

| **AUC** | | | **F1** | | | **Accuracy** | | |
|---|---|---|---|---|---|---|---|---|
| *Approach* | *Avg* | *Rank* | *Approach* | *Avg* | *Rank* | *Approach* | *Avg* | *Rank* |
| *DL-CIBuild*-recent | 67 | 1 | *DL-CIBuild*-recent | 48 | 1 | *DL-CIBuild*-recent | 70 | 1 |
| *DL-CIBuild*-old | 65 | | *DL-CIBuild*-old | 46 | | *DL-CIBuild*-old | 68 | |
| LR-recent | 58 | 2 | LR-recent | 39 | 2 | LR-recent | 60 | 2 |
| RF-recent | 56 | 3 | RF-old | 36 | 3 | ADA-recent | 59 | |
| RF-old | 56 | | SVC-recent | 35 | | RF-old | 57 | |
| LR-old | 56 | 4 | DT-recent | 35 | | ADA-old | 57 | 3 |
| ADA-recent | 56 | | LR-old | 35 | | RF-recent | 56 | |
| SVC-old | 56 | | SVC-old | 34 | | SVC-recent | 56 | |
| SVC-recent | 55 | | RF-recent | 34 | | LR-old | 54 | |
| ADA-old | 54 | 5 | DT-old | 34 | | SVC-old | 54 | |
| DT-recent | 53 | 6 | ADA-old | 34 | | DT-old | 52 | |
| DT-old | 53 | | ADA-recent | 33 | | DT-recent | 45 | 4 |

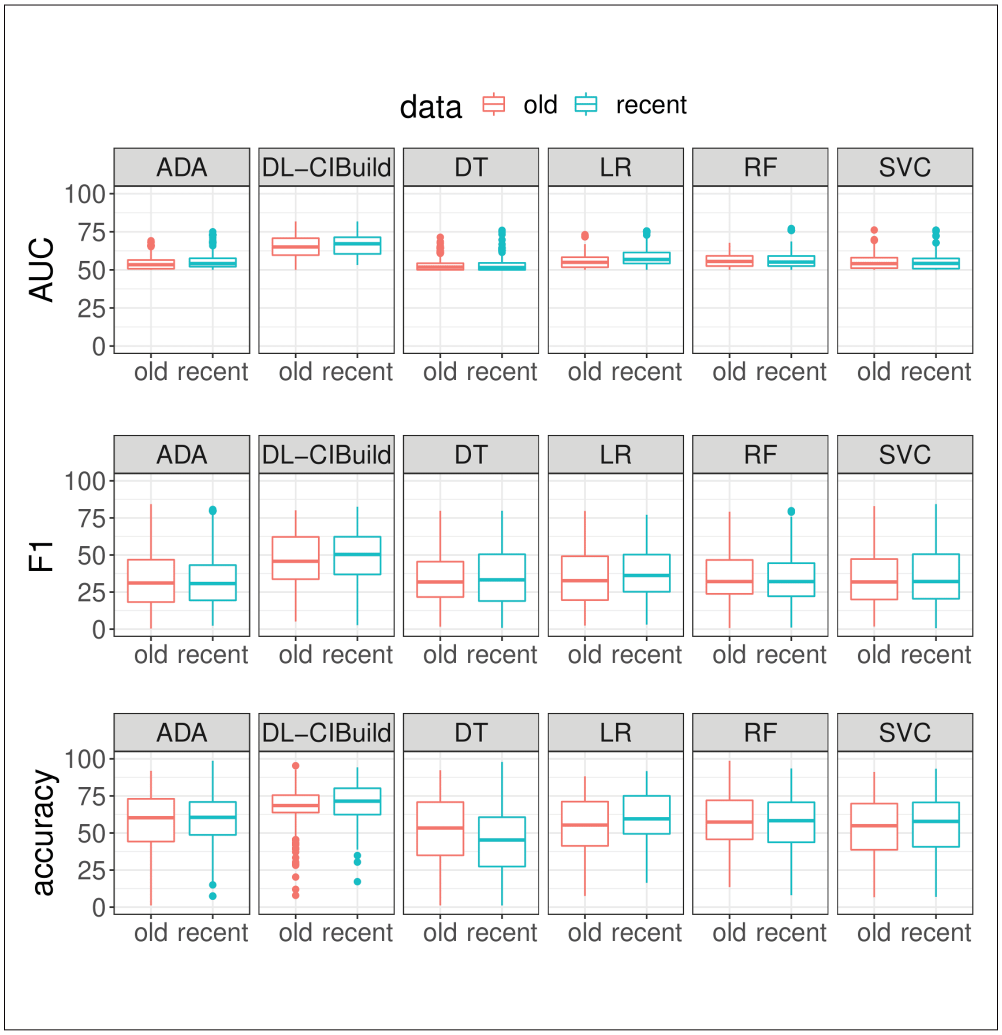


Figure 6.9 Comparison of the prediction performance of *DL-CIBuild* against ML techniques trained on old and recent data

**Summary for RQ5.** Unlike ADA and LR techniques, our approach showed an effective robustness to the concept drift which indicates that the latter do not need to be frequently retrained. Additionally, the results reveal that, again, *DL-CIBuild* is statically better than the baselines considering different training time intervals.

## 6.6 Discussions and Implications

In this section, we discuss our findings and their implications for CI developers, researchers and tool builders.

### 6.6.1 For CI Developers

**Usage scenarios, benefits and costs of using our tool.** We have shown that our approach is able to effectively predict the CI build results by achieving good results, reaching up to 80% in terms of AUC. The typical usage scenario of our tool is to provide suggestions on suspicious CI builds. Hence, *DL-CIBuild* allow teams to check their estimation of CI build results by providing accurate predictions on their builds that are likely to fail. In this way, developers can cut off the expenses of CI build process. Such accurate predictions can help save the build generation time and effort, especially when there are limited resources. However, the cost is that sometimes some few failed builds may be missed or result in a waste of effort on false positives.Additionally, developers cannot simply analyze the warnings made by our tool in isolation, but rather, they need the reasons behind the failure to easily localize it. Nonetheless, more details on the reasons for the failure are important, we plan to extend our approach with further support to software developers by providing sufficient details about what retro-actions needed to fix a failed build.

***DL-CIBuild* can run faster.** One of the acknowledged drawbacks of using our approach is that it is computationally expensive due to the massive training time of LSTM models as well as using Genetic Algorithm (GA) for Hyper-parameters Optimization (HPO). In order to mitigate this issue, we improve the efficiency of GA by enabling the parallel evaluation of the configurations in each generation (which includes the training of LSTM models using the candidate configurations). By integrating this parallelization mechanism, we significantly reduced the execution time of GA as can be shown in Figure 6.10. As we clearly see, the optimized version of GA can even run faster than BOHB technique that also supports the parallelization (Yang & Shami, 2020).

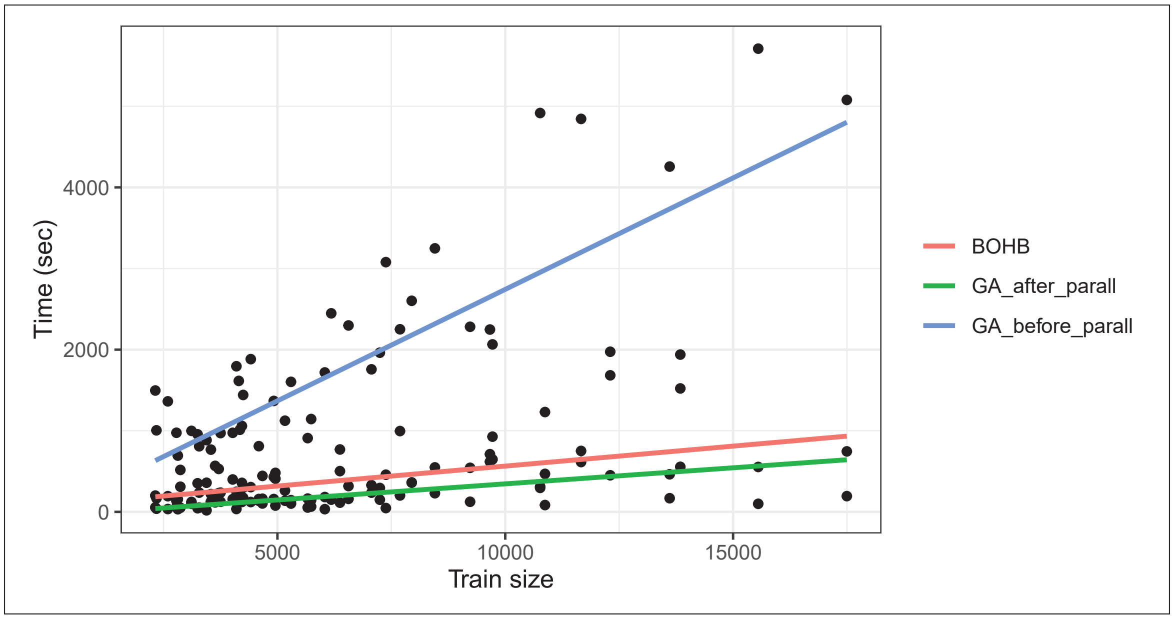


Figure 6.10 The impact of the training set size on the execution time to run GA before (blue) and after (green) parallelization compared to BOHB (in red)

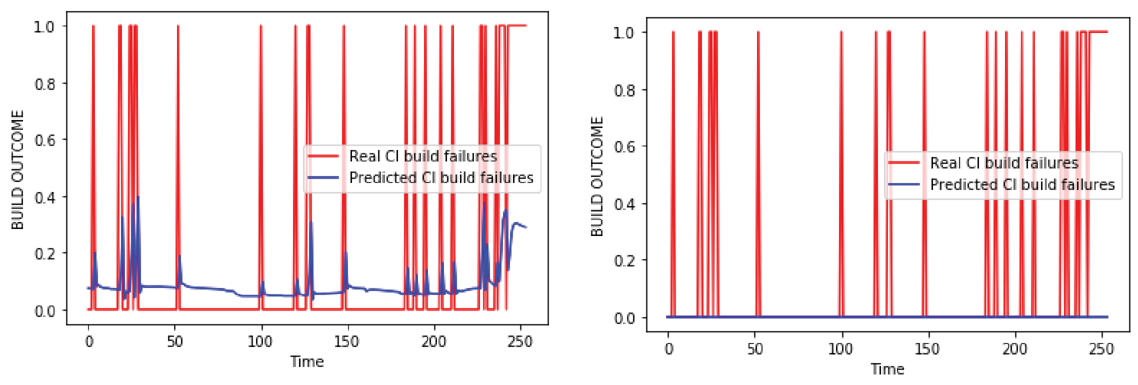
The time of GA can be optimized in other ways: We can reduce the size of the hyper-parameter search space to better value ranges or defining other termination conditions for example when there has been no improvement in the population for K iterations.

### 6.6.2 For Researchers

**Researchers could investigate periodicity in build failure.** Our study analysis lends support to previous research efforts (Rausch *et al.*, 2017) showing that many failed builds occurred consecutively which indicates that if the build failed, the next build is more likely to fail as well. This finding may encourage researchers to get insights into the periodic trends of build failure which would help researchers to enhance the CI practice. Researchers can further analyze the periodicity of CI build failures and investigate what software engineering activities may link with such failure periods, *e.g.,* feature requests, bug fixes, refactoring, release preparation, etc.

**Deep learning LSTM is a suitable modelling choice for software engineering problems for which the temporal dimension is important.** To the best of our knowledge, our work is the first attempt to use deep learning LSTM for the problem of learning CI build failures. The use of LSTM models has allowed to automatically learn the periodicity of build results and use this for predicting build failure. The evaluation results demonstrate the significant improvement that our DL approach has brought in terms of predictive performance especially with comparison to ML techniques. These results represent a significant improvement that can help researchers to mitigate the issues related to feature engineering which is a tedious and error-prone process that needs specific expertise with the domain knowledge to generate features for ML models. Moreover, *DL-CIBuild* has shown effectiveness in handling the lack of data considering cross-project validation while no existing solution has been demonstrated to work at this performance scale. Knowing that software engineering tasks are process-based where the temporal dimension is of crucial importance, our proposed approach can serve as the baseline for further research in the application of DL and LSTM models to time series problems in software engineering.

**Dynamic selection of the classification threshold.** Another possible direction to enhance the prediction accuracy of deep learning LSTM models is to accurately set the classification threshold (above which a build is considered failed) which can highly impact the prediction results. As illustrated in Figure 6.11, we can see an example highlighting the importance of threshold moving from the *Ruby* project. In this figure, the chart (a) plots the output of our LSTM model (which is following the real trend), while the chart (b) shows the prediction results when the classification threshold is set by default (=0.5) which results in classifying all the builds as succeeding (none of the failed builds can be detected). Based on these observations, an important research direction for CI researchers is to consider adaptive threshold selection over time when conceiving DL-based models. This selection can be performed dynamically over time, *i.e.,* adapted depending on the project's activity period such as major/minor releases, new features, library dependencies upgrade/migration, code reengineering, code optimization, etc. We conjecture that a dynamic selection can be an effective solution for deep learning LSTM based prediction.



a) Failure prediction probability

b) Failure prediction result when the threshold is set to 0.5

Figure 6.11 An example showing the impact of the threshold moving on the prediction accuracy extracted from the project *Ruby*

**Can the predictive performance be improved with re-sampling?** So far, we showed that *DL-CIBuild* provides an effective improvement over the ML techniques without re-sampling but instead using Threshold Moving (TM). Unlike sampling, TM does not rely on the manipulation of the training set but instead on manipulating the classifier output. However, one can argue that the use of resampling can further improve the identification of CI build failures for *DL-CIBuild*, even though the latter has shown less sensitivity to the class imbalance problem as pointed out in our previous research questions (RQ2+RQ3). Thus, we conduct a set of additional experiments to re-balance the input data prepared in Section 6.2(online and cross-project validations) using SMOTE (Chawla *et al.*, 2002) the standard oversampling approach; and re-run the LSTM-RNN learning process on the new balanced data. Similar to (Buda *et al.*, 2018), the combination of TM and SMOTE is also tested. To provide a comprehensive comparison, we compute the F1-score, AUC score and the overall accuracy. Figure 6.12 shows the obtained results using SMOTE (in red), TM (in blue) and by combining the two approaches (in green). Considering online validation, we observe that framework shows a better performance using TM than when applying SMOTE with a statistically significant (but small) improvement of 4% in terms of AUC and F1 respectively.

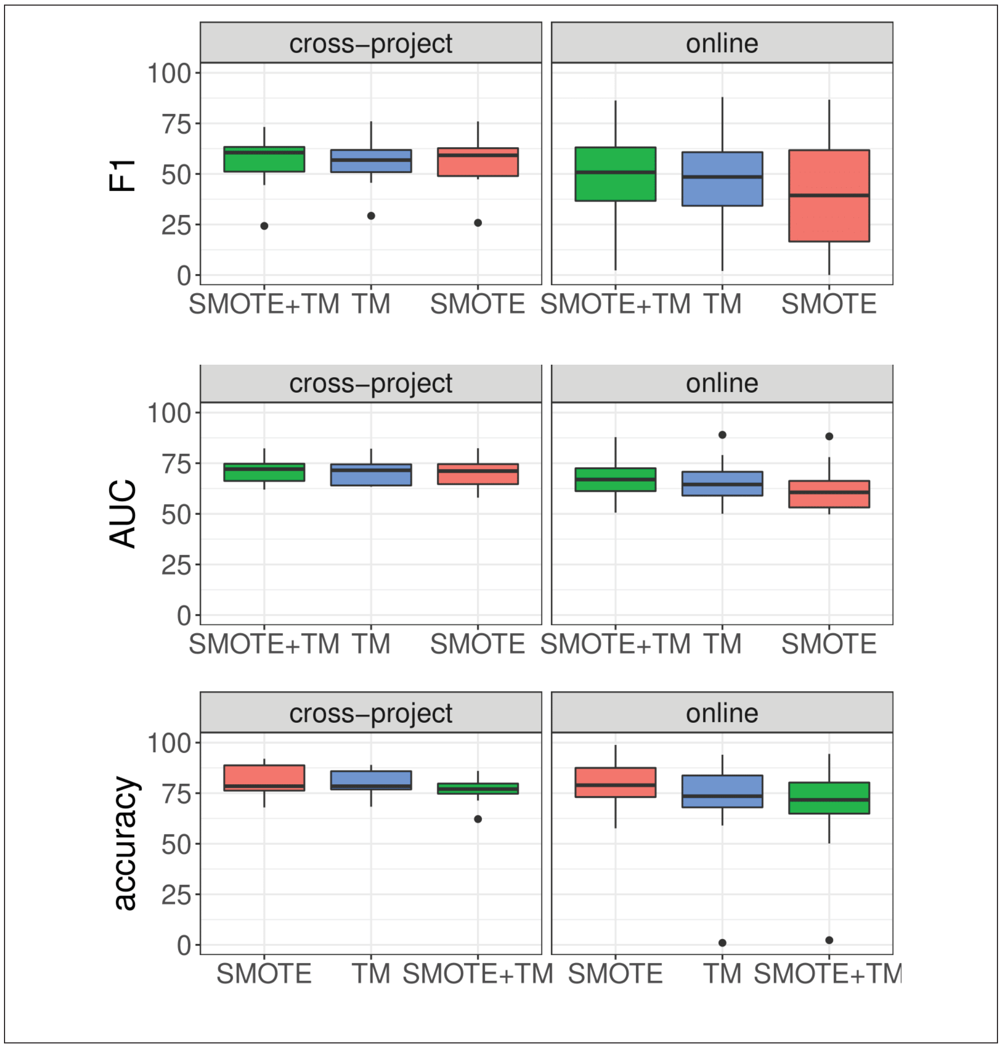


Figure 6.12 Comparison of *DL-CIBuild* results with SMOTE (red), Threshold Moving (blue) and by combining them (green)

Moreover, combining the two approaches can slightly enhance the TM results by 2% in terms of AUC and F1 respectively. However, the statistical test suggests that the difference between TM and the combination is negligible. Hence, using TM on a balanced dataset can provide comparable results to applying it to the original data. Additionally, the overall accuracy of SMOTE is slightly better due to the skewed data distribution in the testing set. When it comes to cross-project, the three strategies seem comparable with no significant differences for F1, AUC and the accuracy. This suggests that using although SMOTE is effective, it is not so good as threshold-moving which is in line with previous studies results(Zhou & Liu, 2005; Buda *et al.*, 2018). Additionally, taking into account the drawbacks of re-sampling such as over-fitting (Tantithamthavorn *et al.*, 2018a) and the computational expense (Bhowan, Johnston & Zhang, 2009), we advocate that threshold-moving alone can be successfully applied.

### 6.6.3 For tool builders

**Feedback mechanisms to predict build failures.** CI services such as Travis CI could provide mechanisms for developers to estimate the likelihood that their current build would fail. Information about the predicted build failures can help the software development team to avoid time overhead. Such information would provide decision support to avoid useless build runs or suggest running builds during project inactivity periods (*e.g.,* out of the working hours) in order to avoid the risk of reducing the team's productivity and release delays.

**Retro-actions to fix the build failure.** Besides failure prediction, tools are needed to help developers fixing build breakages. One possible direction is to define the delegated developers to fix the build which may result in a better management of the resources. We also encourage tool builders to go further by recommending the relevant actions and code changes needed to fix the failed build.

**Dealing with concept drift.** While in RQ5, we showed that *DL-CIBuild* is robust to concept drift, its performance can be further improved when training on more recent data. This result would encourage us and other tool builders to upgrade the prediction tools in a way to allow re-fitting the models periodically using the most recent historical data. However, the main difficulty remains in detecting the right moment when the model needs to be re-trained. One possible solution to this problem is to monitor the prediction performance and if it is degraded below a certain threshold (*i.e.,* a concept drift is detected), an alarm is triggered to re-train the model. This threshold can be configured by the tools users.

**Importance of hyper-parameters tuning.** In our Appendix 1, we provide all the optimal obtained by the Genetic Algorithm (GA) for each project and experiment considering different validations. We notice that the optimal parameters change over time and differ from one project to another. This highlights the importance of exploring the parameter space periodically in order to ensure the performance stability/improvement.

## 6.7 Threats to validity

This section describes the threats to the validity of our experiments.

***Internal validity*** is related to the relationship between treatment and outcome. In this paper, it concerns our selection of subject systems, methods and tools. A threat to internal validity could be related to the stream of the selected projects data. When most of the build failures occur early during the project growth phase, there is little added value in exploring their data later in the life-cycle (Shrikanth, Majumder & Menzies, 2020). To address this issue, we have double-checked each project data stream by computing the number of failed builds in each studied month. We have found that the build failure is well distributed among all the studied periods. We cannot also generalize our findings to other projects as they may have different temporal stream patterns. We also considered two validation scenarios: Online validation which is a realistic scenario as it considers the chronological order of CI builds and mimics what happens during the CI process. The second scenario we considered is cross-project which was used to assess the generalizability of our approach based on the Bellwether strategy. Future work is planned to validate our approach considering other scenarios/strategies. Another potential threat is related to the selected performance metrics. We basically used standard performance metrics namely F1-score, accuracy and AUC that are widely accepted in predictive models in software engineering (Hastie *et al.*, 2009). On the other hand, the variation of metrics also strengthens the generalization of our results as our findings are not based on one specific metric. Another potential threat could be related to the selection of the prediction techniques. We have investigated existing papers related to the prediction of CI builds, and we have adapted their algorithms in our comparative study (Xia *et al.*, 2017a; Ni & Li, 2017; Xia & Li, 2017; Luo *et al.*, 2017; Hassan & Wang, 2017). We replicated their models based on their descriptions, and we have used our dataset as a baseline to compare all approaches. It is important to point out that these models were tuned to use the set of features that are available with respect to the projects we use in our experiments. These ML techniques were used in previous CI and AI for software engineering research (Abdalkareem *et al.*, 2020; Ghotra *et al.*, 2015; Tantithamthavorn *et al.*, 2018b). Nevertheless, we plan as part of our future work to conduct a large-scale empirical study with other techniques. Another threat comes from our choice of HPO methods. We compared GA against methods that are often seen in literature and implemented in Python frameworks/libraries. But even with all that, we have not explored all the existing HPO methods. To some extent, that is because no single paper can explore all algorithms. But also, sometimes we choose not to explore certain algorithms since they are out-of-scope for this study.

It would be also interesting to compare the performance of the Threshold Moving against other sampling techniques like *MAHAKIL* (Bennin, Keung, Phannachitta, Monden & Mensah, 2017) or *SMOTUNED* (Agrawal & Menzies, 2018), which would be an interesting future work

***Construct validity*** refers to the extent to which the experiment setting reflects the theory. The first threat to construct validity is randomness that may introduce bias. To mitigate this threat, we performed 31 runs of each algorithm and considered the median value in each validation iteration and applied statistical tests to remove spurious distinctions. As for the used features to feed ML techniques, we used standard features from TravisTorrent dataset that commonly used in the literature (Xia *et al.*, 2017a; Ni & Li, 2018; Xia & Li, 2017; Luo *et al.*, 2017; Hassan & Wang, 2017; Ni & Li, 2018; Santolucito *et al.*, 2018). We plan to extend these features in an attempt to see their impact on the prediction performance. Additionally, the hyper-parameters search space could introduce some bias in our results as considering different ranges/parameters may yield different results. However, the exploration of the parameter space of automated HPO methods may require a considerable computational cost. Thus, future replication of this work should explore other ranges/parameters and their impacts on the predictive performance. Another threat to construct validity is related to the setting of RQ5 to detect the concept drift since defining another validation scenario could lead to different results. Further experiments are required to confirm/refute the existence of concept drift in CI builds. Another threat to construct validity could be related to the annotated set of builds as in our dataset, the build results are noisy (Ghaleb, da Costa, Zou & Hassan, 2019b; Gallaba, Macho, Pinzger & McIntosh, 2018). While according to our knowledge, TravisTorrent is the only available dataset of CI builds, a future work based on clean build breakage dataset is required.

***Conclusion validity*** affects the ability to draw correct conclusions about the relationship between treatment and outcome. We have carefully chosen non-parametric tests, namely Wilcoxon and Cliff's delta, in the study as they do not require data normality assumptions (Malhotra & Khanna, 2017). The suitability of the used statistical non-parametric methods with data ordinality, along with no assumption on their distribution raises our confidence about the significance of the analyzed statistical relationships. Moreover, to increase the confidence in the study results, we used three widely-acknowledged prediction performance measures, *i.e.,* F1-score, accuracy and AUC to evaluate the obtained results from the considered algorithms.

***External validity*** concerns the possibility to generalize our results. Our experimental results might have concerns of generalizability, since we performed the experiments with ten open source projects that use TravisTorrent as their CI host tool. While TravisTorrent is one most popular cloud-based platforms for providing CI services to software projects, our results could not be generalized to other CI tools and other open-source or industrial projects. As future work, we plan to extend our study on other open source and industrial projects as well as other CI tools. We also plan to provide our approach as bot to be integrated into code review and CI tools to help developers predicting their build failure risks.

***Reliability validity*** concerns the possibility of replicating this study. All the studies projects are publicly available. Moreover, the Python implementation of our approach is provided in our replication package (Saidani, 2020c).

## 6.8 Conclusion

In this paper, we introduced *DL-CIBuild* a two-phase framework for CI build failure prediction. In the first phase, we implement LSTM model based on the temporal information of build results. Then, we use Genetic Algorithm (GA) for tuning the model hyper-parameters. To evaluate the effectiveness of our approach, we conduct an empirical study on ten open-source projects that use the popular CI host system, Travis CI, with a total of 91,330 builds. In summary, the empirical study results show that (*i*) when compared to other methods for automated parameters tuning, GA can provide better configurations, (*ii*) under online-validation, our approach achieves a reasonable and better performance than the five Machine Learning techniques in terms of AUC, F1-score and accuracy (*iii*) when it comes to cross-project validation, *DL-CIBuild* has shown a good effectiveness to learn from cross-project training corpus which means that our approach is readily applicable to both within-project and cross-project predictions and (*iii*) the sensitivity check results reveal that our solution is more robust than ML techniques across varying the training set size and the predictive performance is estimated to be enhanced with larger base of CI build results.

*DL-CIBuild* represents an interesting case study on the effectiveness of deep learning LSTM for CI build failures prediction. As future works, we envision to improve the performance of our approach by considering other prominent aspects and perform the experiments on more projects. This can help developers and researcher get more insights on the CI build failures problem, as the next generation of software defects, and gain actionable information to improve the practice of CI in software projects. Moreover, we plan to implement a bot based on *DL-CIBuild* and conduct a user study with our industrial partner to better evaluate our approach in an industrial setting. Additionally, the tool can allow updating the trained model with more data when the performance degrades below a certain threshold; which could be configured by the tool users.

# CHAPTER 7

# DETECTING CONTINUOUS INTEGRATION SKIP COMMITS USING MULTI-OBJECTIVE EVOLUTIONARY SEARCH

Islem Saidani[a] , Ali Ouni[a] , Mohamed Wiem Mkaouer[b]

[a] Department of Software Engineering and IT, École de Technologie Supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[b] Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester, NY 14623, United States

Paper published in *IEEE Transactions on Software Engineering*, October 2021

## Abstract

Continuous Integration (CI) consists of integrating the changes introduced by different developers more frequently through the automation of build process. Nevertheless, the CI build process is seen as a major barrier that causes delays in the product release dates. One of the main reasons for such delays is that some simple changes (*i.e.,* can be skipped) trigger the build, which represents an unnecessary overhead and particularly painful for large projects. In order to cut off the expenses of CI build time, we propose in this paper, SkipCI, a novel search-based approach to automatically detect CI Skip commits based on the adaptation of Strength-Pareto Evolutionary Algorithm (SPEA2). Our approach aims to provide the optimal trade-off between two conflicting objectives to deal with both skipped and non-skipped commits. We evaluate our approach and investigate the performance of both within and cross-project validations on a benchmark of 14,294 CI commits from 15 projects that use Travis CI system. The statistical tests revealed that our approach shows a clear advantage over the baseline approaches with average scores of 92% and 84% in terms of AUC for cross-validation and cross-project validations respectively. Furthermore, the features analysis reveals that documentation changes, terms appearing in the commit message and the committer experience are the most prominent features in CI skip detection. When it comes to the cross-project scenario, the results reveal that besides the documentation changes, there is a strong link between current and previous commits results.

Moreover, we deployed and evaluated the usefulness of SkipCI with our industrial partner. Qualitative results demonstrate the effectiveness of SkipCI in providing relevant CI skip commit recommendations to developers for two large software projects from practitioner's point of view.

**Keywords.** Continuous Integration, Travis CI, Software Build, Search-Based Software Engineering, Genetic Programming.

## 7.1 Introduction

Continuous integration (CI) is a leading edge of modern software development practices that is widely adopted in industry and open-source environments (Vasilescu *et al.*, 2015). CI systems, such as Travis CI[1], advocate to continuously integrate code changes, by automating the process of building and testing (Fowler, 2006), which reduces the cost and risk of delivering defective changes. Nevertheless, the build process is seen as a critical problem that hinders CI adoption. In fact, in such context, the build process is typically time and resource-consuming and particularly painful for large projects, with various dependencies, in which their builds can take hours and even days (Ghaleb *et al.*, 2019a). The presence of such slow builds severely affects both the speed and cost of software development as well as the productivity of developers (Luo *et al.*, 2017). Such challenges motivated the research (Abdalkareem *et al.*, 2019, 2020) on developing techniques to speed up CI process and cut its expenses by detecting commits that do not require the system's build *e.g.,* commits affecting non-source files.

The first attempt in addressing this problem (Abdalkareem *et al.*, 2019), formulated the CI skip detection problem with rules that were manually defined through mining the historical commit changes. To raise the challenges related to making the manual procedure of defining the rules as well as the high false negative rate, Abdalkareem et al. (Abdalkareem *et al.*, 2020) proposed an approach based on Decision Tree (DT) that achieved satisfactory results within-project validation with 89% in terms of Area Under the ROC Curve (AUC). When it comes to cross-project validation, the prediction performance degraded to the accuracy of random guessing with 74%

[1] https://travis-ci.org/

in terms of AUC; which challenges the applicability of this approach on projects with little or no previous commit history. However, in practice, there is a lack of training data as the CI skip option (*e.g.,* adding the '*[ci skip]*' or '*[skip ci]*' in commit messages[2] in order to explicitly skip the commit by CI systems) is usually under-used. This suggests that the skip detection problem is not yet resolved.

Detecting a skip commit in practice is not a trivial task as the main difficulty lies in the large and complex search space to be explored due to exponential number of possible combinations of features and their associated threshold values. Hence, the CI skip commits detection problem is by nature a combinatorial optimization problem in order to find the optimal detection rules that should maximize as much as possible the detection accuracy. Additionally, taking into account the conflict between the minority (*i.e.,* skipped) and majority (*i.e.,* non-skipped) classes accuracies (Bhowan *et al.*, 2011), multi-objective formulation is well suited to search-based software engineering (SBSE) (Harman & Jones, 2001; Harman *et al.*, 2012). Recently, a number of researchers have highlighted the successful use of Multi-Objective Genetic Programming (MOGP) as an efficient method for developing prediction models (Harman *et al.*, 2012; Kessentini & Ouni, 2017; Ouni *et al.*, 2016). This technique is particularly effective with imbalanced problems as it allows the evolution of solutions with optimal balance between minority and majority classes, without need to rebalance the data (Bhowan *et al.*, 2012, 2013). This is crucial to keep the generated models accurate to the original data (Tantithamthavorn *et al.*, 2018a) and hence human interpretable.

Motivated by the need for help in efficiently identifying commit changes that could be skipped in the CI pipeline, we introduce in this paper, SkipCI, a novel approach that formulates the problem of CI skip detection as a search-based problem to (1) maximize the probability of detection (*i.e.,* skipped commits that are correctly classified) and (2) minimize the false alarms (*i.e.,* the commits incorrectly classified as skipped). In this approach, we adapt the Strength-Pareto Evolutionary Algorithm (SPEA2) (Zitzler *et al.*, 2001) with a tree-based solution representation,

[2] https://docs.travis-ci.com/user/customizing-the-build/#skipping-a-build

to generate the optimal detection rules that should cover as much as possible the accurately detected commits from the base of real world CI commits examples.

To evaluate our approach, we conducted an empirical study on a benchmark of 14,294 CI commits from 15 projects that use Travis CI system. First, we answer the following research question: **RQ1:** How effective is SkipCI compared to other existing search-based algorithms? Our multi-objective formulation has shown its effectiveness compared to other SBSE approaches. Then, we compare SkipCI to five ML techniques within-project validation in **RQ2:** How does our approach perform compared to ML techniques? and considering cross-project validation in **RQ3:** How effective is our approach when applied on cross-projects? Results show that SkipCI achieves a better performance over various baseline approaches with average AUC scores of 92% and 88% for cross-validation and cross-project validations, respectively. Next, we investigate the most important features by leveraging our generated rules in **RQ4:** What features are most important to detect skipped commits? The features analysis reveals that the number of previously skipped commits, the commit purpose, and the terms appearing in the commit message are the most influential features to indicate CI skip proneness. Moreover, we deployed SkipCI in an industrial setting in **RQ5:** Are the CI skip commit recommendations provided by SkipCI useful for developers who use CI in practice? The results of a qualitative evaluation of SkipCI with 14 developers indicate the relevance of SkipCI in practice as compared to baseline techniques.

### 7.1.1 Contributions

The main contributions of the paper can be summarized as follows:

1. A novel formulation of the CI skip detection as a multi-objective optimization problem to handle imbalance nature of CI skipped commits data.
2. An evaluation of SkipCI, on a benchmark of 14,294 Travis CI commits of 15 projects that use Travis CI, by (1) comparing our approach to search-based algorithms as well as ML techniques and (2) performing a qualitative analysis to discover which features are the most prominent to determine CI skipped commits using our proper rules.

3. An industrial evaluation of SkipCI. Specifically, we deployed our tool in the CI pipeline of two projects with our industrial partner and validated its relevance with 14 developers.

### 7.1.2 Replication Package

We provide our replication package containing all the materials to reproduce and extend our study (Saidani, 2020b). In particular, we provide (1) our SkipCI tool as a lightweight command line tool with the necessary documentation to run the tool, (2) the working data sets of our study and (3) the validation results along with (4) examples of the built models.

### 7.1.3 Paper Organization

The remainder of this paper is organized as follows. In Section 7.2, we motivate the formulation of CI skip detection with a real-world example. Then, we explain how we adapted SPEA2 to our problem in Section 7.3. Section 7.4 describes the experimental setup of our empirical study while Section 7.5 presents the results of this evaluation. In Section 7.6, we deploy SkipCI in an industrial setting and evaluate it from developers' perspectives. We discuss the implications of our findings in Section 7.7. Section 7.8 describes the threats to validity. Section 7.9 concludes the paper.

## 7.2 Motivating Example

Although the ML-based model (Abdalkareem *et al.*, 2020) was able to improve the CI skip detection compared to the rule-based approach (Abdalkareem *et al.*, 2019), it still misses cases of commits that should be skipped in CI. Figure 7.1 depicts a concrete example extracted from *SemanticMediaWiki*[3], a PHP framework, where the commit, despite not being a cosmetic change, it is worth to be skipped. Looking at the commit change, the developer has modified two files:

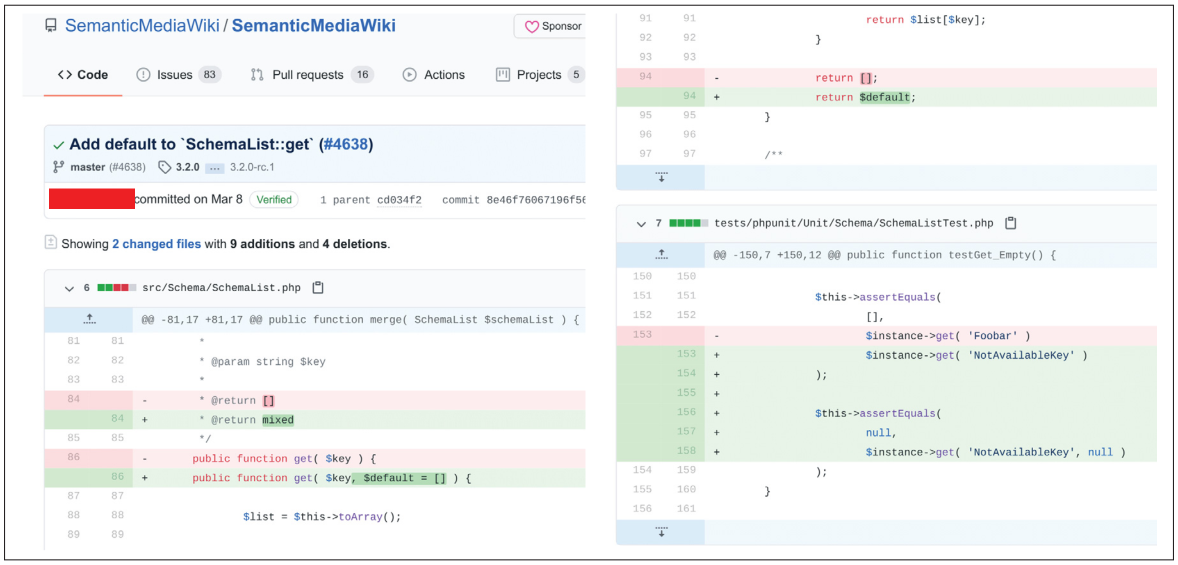
[3] https://github.com/SemanticMediaWiki/SemanticMediaWiki/commit/8e46f76067196f5677ee88ec667daefedf611b4d

- The first file is a source file (PHP) in which the developer added a default parameter (or optional) called *$default* to the signature of *get* function. This parameter is the result to be returned in case the key (a parameter called *$key*) is not in the list (of type *SchemaList*). We clearly see that this change neither alter the function behavior nor the behavior of the caller functions as it replaces the empty list with a parameter set by default to *[]*. This explains why this file was the only source file to be affected by this change, even though this function is called in other files.
- The second file is a test file, in which the developer (*i*) changed the value to be tested in order to check whether calling a non-existent key would return an empty list, and (*ii*) added another assertion to verify whether calling a non-existent key would return *null* in case *$default* is set to null. Looking at the content of this file, we see that the input list for these tests contains only one element set to *'Foo'*, which means these tests can be skipped.

However, using existing approaches (Abdalkareem *et al.*, 2019, 2020), we have found that they failed to detect this commit to be skipped. Indeed, the rule-based approach (Abdalkareem *et al.*, 2019) consists of five rules related mainly to non-source files (*e.g.,* documentation) and cosmetic changes (*e.g.,* . source code formatting), which explains why this approach cannot detect more sophisticated cases as this example shows. Hence, this approach is not suitable to the CI skip problem.

The machine learning approach (Abdalkareem *et al.*, 2020) has also failed to provide an adequate detection. Looking at the generated model, we found that the commit is classified as non-skipped based on three conditions including (*i*) having non-documentation files, (*ii*) a developer experience (*i.e.,* number of previously committed changes) greater than 1,700 and (*iii*) a number of changed files greater than one, which mismatches with the characteristics of the provided example.

This implies that solving this problem is not a trivial task and requires a more suitable approach to generate the adequate skip detection rules that learn from both classes (1) skipped commits and (2) non-skipped commits. In fact, when the changes are sophisticated, the problem resolution requires exploring a large search space composed of high number of possible combinations of features and their associated thresholds. Hence, the CI skip detection can be formulated as a search-based optimization problem to explore this large search space, in order to find the optimal detection rules. Additionally, a practical tool should also provide the developers with human explainable detection models to help them gaining insights into the CI commits to be skipped, especially when the changes are not trivial as shown in this example. This cannot be provided by ML techniques as re-sampling affects the interpretability of the generated models (Tantithamthavorn *et al.*, 2018a) since the original and the balanced training corpora have different characteristics. Furthermore, rebalancing can also be computationally expensive (Bhowan *et al.*, 2009). In this paper, we advocate that a MOGP technique is more suitable since it allows the evolution of solutions with optimal balance between minority and majority classes, without need to rebalance the data. In this way, the generated rules can provide useful explanations to the user.


SemanticMediaWiki / **SemanticMediaWiki**

**Add default to `SchemaList::get` (#4638)**

master (#4638) 3.2.0 ... 3.2.0-rc.1

[redacted] committed on Mar 8 Verified 1 parent cd034f2 commit 8e46f76067196f56

Showing **2 changed files** with **9 additions** and **4 deletions**.

6 src/Schema/SchemaList.php

```
@@ -81,17 +81,17 @@ public function merge( SchemaList $schemaList ) {
          *
          * @param string $key
          *
-         * @return []
+         * @return mixed
          */
-       public function get( $key ) {
+       public function get( $key, $default = [] ) {

                $list = $this->toArray();

...
                        return $list[$key];
                }

-               return [];
+               return $default;
        }

        /**
```

7 tests/phpunit/Unit/Schema/SchemaListTest.php

```
@@ -150,7 +150,12 @@ public function testGet_Empty() {

                $this->assertEquals(
                        [],
-                       $instance->get( 'Foobar' )
+                       $instance->get( 'NotAvailableKey' )
+               );
+
+               $this->assertEquals(
+                       null,
+                       $instance->get( 'NotAvailableKey', null )
                );
        }

```

Figure 7.1 A motivating example taken from the *SemanticMediaWiki* project

In the next section, we describe SkipCI and show how we formulated the CI skip detection problem as a multi-objective combinatorial optimization problem to address the above mentioned problems.

## 7.3 The SkipCI Approach

In this section, we describe our SkipCI approach to automate the detection of CI skipped commits by adapting Strength-Pareto Evolutionary Algorithm (SPEA2) (Zitzler *et al.*, 2001).

### 7.3.1 Approach Overview

Figure 7.2 depicts an overview of SkipCI. The first input is a set of collected examples of commits that were annotated by their original developers as skip commits. As output, SkipCI generate optimal rules using the SPEA2 algorithm. The search algorithm evolves toward finding the best trade-off between two objective functions to (1) maximize the true positive rate, and (2) minimize the false positive rate. Then, given a code change (*i.e.,* commit) which is composed of a number of changed files, SkipCI provides the user with an explained recommendation whether to skip or not the submitted change in the CI pipeline, *i.e.,* trigger the build process.

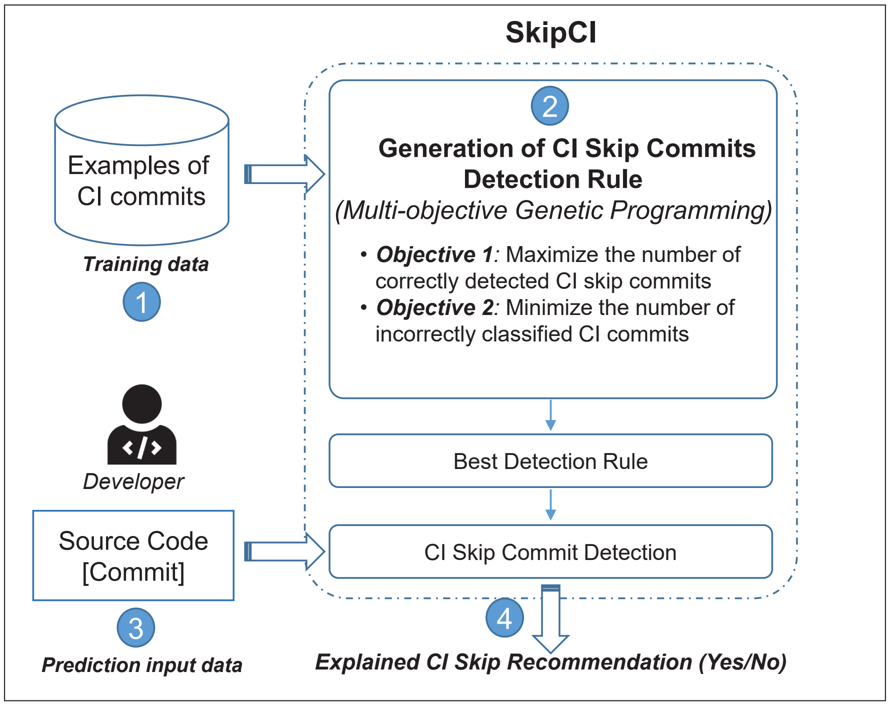

SkipCI
Examples of CI commits
Training data
1
2
Generation of CI Skip Commits Detection Rule
(Multi-objective Genetic Programming)
• Objective 1: Maximize the number of correctly detected CI skip commits
• Objective 2: Minimize the number of incorrectly classified CI commits
Developer
Best Detection Rule
Source Code [Commit]
CI Skip Commit Detection
3
Prediction input data
4
Explained CI Skip Recommendation (Yes/No)

Figure 7.2 Approach overview

### 7.3.2 Multi-objective Genetic Programming adaptation

This section shows how MOGP is adopted to CI skip commits detection problem using SPEA2. Then, we present the solution encoding, the objective functions to optimize, and the employed change operators.

#### 7.3.2.1 Adaptation of SPEA2

To adapt SPEA2 to our problem (*i.e.,* CI skip commit detection), we need to define (i) the initial population (line 1 in Algorithm 2.3), (ii) the fitness assignment (line 3), (iii) the variation operators (line 6) and (iv) the stopping criteria (line 2). The parameters setting, such as the population size, is described later in Section 7.4.4.

**i. Initialization:** Before defining the initial population generation, we need first to define the population individuals or *candidate solutions*. In MOGP, a candidate solution, *i.e.,* a detection rule, is represented as an IF – THEN rules with the following structure (Saidani *et al.*, 2020a; Ouni *et al.*, 2013; Kessentini & Ouni, 2017; Ouni, Kessentini, Inoue & Cinnéide, 2015):

**IF** *“Combination of features with their thresholds"* **THEN** *”RESULT”*.

The antecedent of the IF statement describes the conditions, *i.e.,* pairs of features and their threshold values connected with mathematical operators (*e.g.,* $=, >, \geq, <, \leq$), under which a CI commit is said to be skipped or not. These pairs are combined using logic operators (OR, AND in our formulation). Figure 7.3 provides an example of a solution.

This rule, represented by a binary tree, detects a CI skip commit if it fulfills the situation where (1) the number of added lines in the changed files (*LA*) is less or equals to 146, (2) the commit does not change source files (*IS_SRC*), and (3) the commit is not a bug fixing commit (*IS_FIX*).

**IF** *LA ≤146* AND *IS_SRC =0* AND *IS_FIX =0* **THEN** Skip commit.

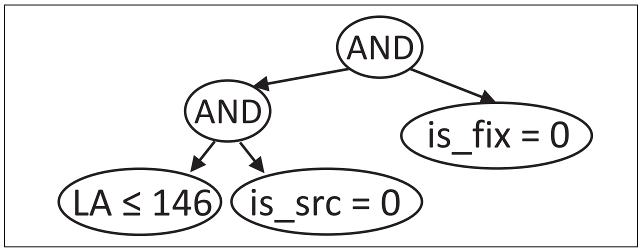


Figure 7.3 Example of solution representation

To generate the initial population, we start by randomly assigning a set of features and their thresholds to the different nodes of the trees. To control for complexity, each solution size, *i.e.,* the tree's length, should vary between a lower and upper-bound limits based on the total number of features to use within the detection rule. More precisely, for each solution, we assign:

- For each leaf node one feature and its corresponding threshold. The latter is generated randomly between lower and upper bounds according to the values ranging of the related feature.
- Each internal node (function) is randomly selected between AND and OR operators.

**ii. Fitness assignment:** Appropriate fitness function, also called objective function, should be defined to evaluate how good is a candidate solution. For the CI skip commit problem, we seek to optimize the two following objective functions:

1. Maximize the coverage of expected CI skipped commits over the actual list of detected skipped commits known as the True Positive Rate (TPR), or the False Probability Detection (PD).

$$TPR(S) = \frac{\{Detected\ Skipped\ Commits\} \cap \{Expected\ Skipped\ Commits\}}{\{Detected\ Skipped\ Commits\}}$$

2. Minimize the coverage of actual non-skipped commits that are incorrectly classified as skipped also known as False Positive Rate (FPR), or the probability of false alarm (FP).

$$FPR(S) = \frac{\{Detected\ Skipped\ Commits\} \cap \{Expected\ non\text{-}skipped\ Commits\}}{\{Detected\ Skipped\ Commits\}}$$

Additionally, since SPEA2 returns a set of optimal (*i.e.,* non-dominated) solutions in the *Pareto front* without ranking, we extract a single best solution which is the nearest to the ideal solution known as *True Pareto* in which TPR value equals to 1 and FPR equals to 0. Formally, the distance is computed in terms of Euclidean distance (Ouni *et al.*, 2016, 2013) as follows:

$$BestSol = \min_{i=1}^{n} \sqrt{(1 - TPR[i])^2 + FPR[i]^2}$$

where *n* represents the number of solutions generated by SPEA2 TPR[i] and FPR[i] compute the TPR and FPR of solution $i \in \{1..n\}$.

**iii. Variation:**

*Mutation:* In MOGP, this operator can be applied either to a terminal or a function node. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal, it is replaced by another terminal (other feature or other threshold value, or both); if it is a function (AND, OR operators) node, it is replaced by a new function. Then, the node and its sub-tree are replaced by the new randomly generated sub-tree. Figure 7.4 illustrates an example of a mutation process, in which the terminal is replaced containing *IS_FIX* feature, by another terminal composed of the condition $NS \leq 20$. Thus, we obtain the new rule:

**IF** *LA ≤146* AND *IS_SRC =0* AND *NS ≤20* AND *LT≤5* **THEN** Skip Commit.

*Crossover:* For MOGP, we use the standard single-point crossover operator where two parents are selected and a sub-tree is extracted from each one. Figure 7.5 depicts an example of the crossover process. In fact, rules P1 and P2 are combined to generate two new rules. For instance, the new rule C2 will be:

**IF** *NUC ≤ 2* OR *IS_SRC =0* **THEN** Skip Commit.

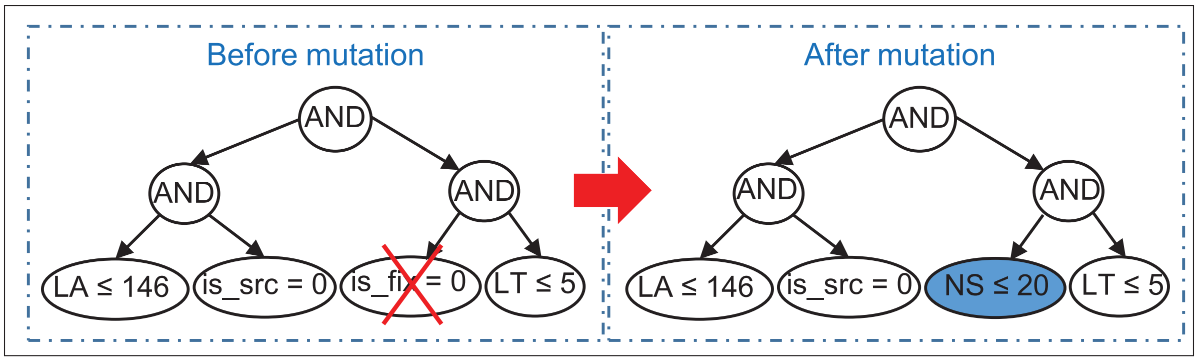


Figure 7.4 An example of mutation operation

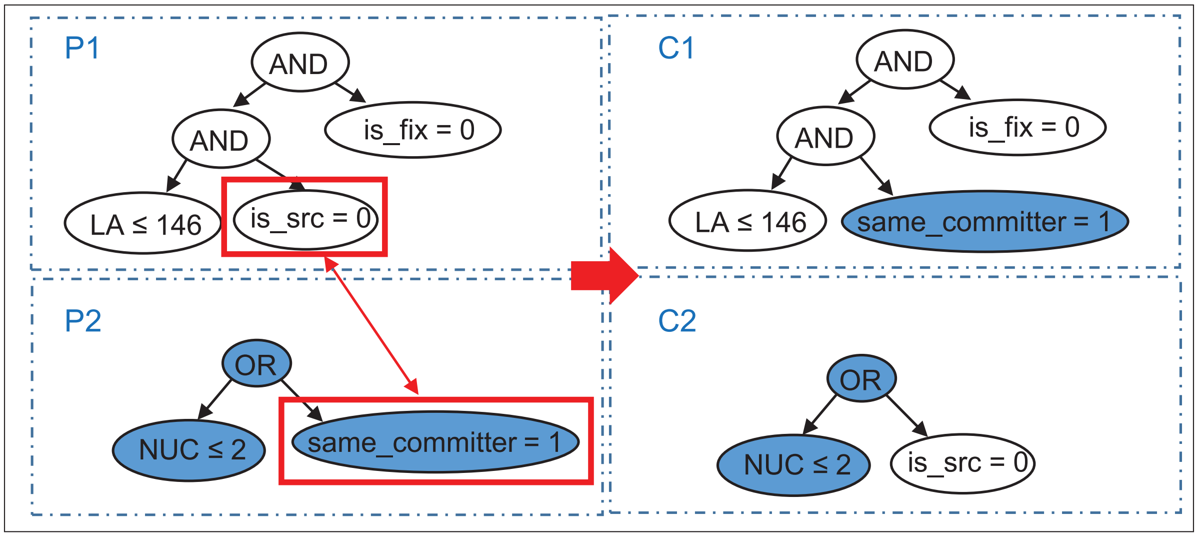


Figure 7.5 An example of crossover operator

**iv. Stopping criteria:** In our experiments, the algorithm stops when reaching a maximum number of generations.

### 7.3.3 Features for CI Skip commits detection

Tables 7.1 lists the features, used to learn and generate our detection rules. Besides the features used by Abdalkareem et al. (Abdalkareem *et al.*, 2020), we also generated other features related to the history of commits to better capture the salient characteristics of CI skip commits. Note that we wrote a Java script to compute the features based on their definition from the original paper. Please refer to our replication package for more details about the implemented script.

The selected features can be categorized as follows:

**Statistics about the current commit:** These features give a different information about the current commit change size (*e.g.,* number of lines added/deleted, entropy), the importance of terms in the commit message (*i.e.,* CM) that has proved its efficiency to predict CI Skip commits in the recent work of Abdalkareem et al. (Abdalkareem *et al.*, 2020) and other general statistics (*e.g.,* the day of the week).

**Commit purpose:** These features provide insights into the commit skip proneness. For instance, merge commits, commits containing only media or those changing documentation files are most likely to be skipped.

**Link to last to commit(s):** In addition to the previously used features that are linked to last commits (Abdalkareem *et al.*, 2020), such as the committer experience, we add and deduce other detailed features including the number of recently skipped commits (in the project and by the current committer), the feature indicating whether the last commit was skipped or not and whether the current commit is requested by the same committer of the previous one. These features are inspired from existing works of CI build failure detection (Ni & Li, 2017; Hassan & Wang, 2017; Xie & Li, 2018) in which the authors found that there is a strong link between the current build and the previous ones, and it was helpful to predict the failure in practice. In this paper, we hypothesize that likely to CI build failure, skipped commits may come consecutively *e.g.,* committing different changes of documentation or applying sequences of refactoring (*i.e.,* modifying the code structure without changing its function (Fowler, 2018)).

## 7.4 Experimental Study Design

In this section, we describe the design of our empirical study to evaluate our SkipCI approach.

Table 7.1 Features used form CI Skip detection extracted from literature

| Cat | Feature | Description | Rational |
|---|---|---|---|
| *Statistics about current commit* | Number of Sub-systems (NS) | The number of changed sub-systems. | Commits that affect many subsystems are not usually likely to be CI skipped. |
| | Number of Directories (ND) | The number of changed directories. | Commits that affect are based on many directories are not usually likely to be CI skipped. |
| | Number of Files (NF) | The number of changed files. | Usually non-trivial changes affect many files. |
| | ENTROPY | Measures the distribution of the change across the different files. | High entropy is an indication of complicated changes. |
| | Lines Added (LA) | The number of added lines. | Adding new lines means that the functionality of code has changed and hence should be tested. |
| | Lines Deleted (LD) | Number of deleted lines | Many deleted lines indicate that the change should not be CI skipped. |
| | Day of the Week (DAY_WEEK) | Day of the week when the commit was performed. | Routines of developers and the battle rhythms of projects can manifest themselves in CI skip activity. |
| | Commit Message (CM) | Measures the importance of terms appearing in the commit message using TF-IDF | Commit message may contain useful information about the type of commit. |
| | Types of Files Changed (TFC) | The number of the changed files' types identified by their extension. | The type of files indicate the type of changes. |
| *Commit Purpose* | Classification (CLASSIF) | Feature Addition (1), Corrective (2), Merge (3), Perfective (4), Preventative (5), Non-Functional (6), None (7) | The classification of the commit can be useful *e.g.,* commits of category (1) cannot generally be skipped). |
| | Fixing Commit (IS_FIX) | Whether the commit is a bug fixing commit | Fixing bugs means that the code need to be tested. |
| | Documentation Commit (IS_DOC) | Whether the commit is a documentation commit, *i.e.,* contains only doc files | Commits that change documentation files are likely to be skipped. |
| | Building Commit (IS_BUILD) | Tests whether the commits contains only build files | if the changes in a commit are mainly related to release version, the latter is likely to be skipped. |
| | Meta-files Commit (IS_META) | Whether the commit a meta commit (contains only meta files) | If a commit changes mainly meta files, it is likely to be CI skipped. |
| | Merging Commit (IS_MERGE) | The commit is a merge commit | If the commit is a merging one, it is likely to be CI skipped. |
| | Media Commit (IS_MEDIA) | Whether the commit a media commit, *i.e.,* contains only media files like images | Commits that contain only changed media files are likely to be CI skipped. |
| | Source Commit (IS_SRC) | Whether the commit a source commit *i.e.,* contains only source code files | Commits that contain only source files may need to be tested. |
| | Formatting Commit (FRM) | If the commit changes the formatting of the source code. | In this case, the commit is likely to be a CI skip commit |
| | Comments (COM) | If the commit modifies only source code comments | This type of changes indicates that the commits is likely to be skipped. |
| | Maintenance Commit (MC) | If the commit is a maintenance activity. | The type of maintenance activity may indicate the need to test the changes. |
| *Link to Last Commit(s)* | Project Recent Skip (PRS) | Number of recently commits that were skipped in the 5 past commits. | |
| | Committer Recent Skip (CRS) | Number of recently commits that were skipped in the 5 past commits by the current committer. | All previous skipped commits can impact the likelihood of the current commit to be skipped. |
| | Previous Commit Result (PCR) | Whether previous commit was skipped or not. | |
| | Same Committer (SC) | Whether it is the same committer of the last commit. | |
| | Number of Unique Changes (NUC) | The number of unique last commits of the modified files. | Larger NUC is an indication of the commit's complexity which requires testing the commit. |
| | AGE | The average time interval between the current and the last time these files were modified. | Faster changes can introduce more bugs and thus need to be tested. |
| | Number of Developers (NDEV) | The number of developers that previously changed the touched file(s) in the past. | The larger is NDEV, the more risky are the changes. |
| | Lines Transformed (LT) | Size of changed files before the commit. | Higher LT indicates the complexity of the changed files and thus the need to be tested. |
| | Developer Experience (EXP) | The number of commits made by the developer before the current commit. | Indication of how much is the developer familiar with the changed files and thus his/her knowledge about the need to skip the changes. |
| | Sub-system Experience (SEXP) | Subsystem experience measures the number of commits the developer made in the past to the subsystems that are modified by the current commit. | |
| | Recent Experience (REXP) | Recent experience is measured as the total experience of the developer in terms of commits, weighted by their age. | |

### 7.4.1 Research Questions

In this study, we define four Research Questions (RQs). In the first two RQs, we conduct a 10-fold cross validation by dividing the data of each project into 10 equal folds. We use 9 folds to train each algorithm, and use the remaining one fold to evaluate the predictive performance. Then, we perform cross-project validation in RQ3, and investigated the features influence in RQ4.

*RQ1. (SBSE validation) How effective is SkipCI compared to other existing search-based algorithms?*

**Motivation.** The aim in this question is to evaluate the SPEA2 formulation from an SBSE perspective as recommended by Harman and Jones (Harman & Jones, 2001). First, we compare SPEA2 against Random Search (RS) (Harman *et al.*, 2010; Karnopp, 1963), to evaluate the need for an intelligent method against the unguided search of solutions. Additionally, it is important to compare our multi-objective formulation mono-objective GP (GA) since if considering separate conflicting objectives fails to outperform aggregating them into a single objective function, then the proposed formulation is inadequate. Then, we evaluate the performance of SPEA2 with three widely-used multi-objective algorithms (Harman *et al.*, 2012; Harman, 2007; Mkaouer *et al.*, 2015) namely Non-dominated Sorting Genetic Algorithm (NSGA-II) (Deb *et al.*, 2002), NSGA-III (Deb & Jain, 2013) and Indicator-Based Evolutionary Algorithm (IBEA) (di Pierro *et al.*, 2007) in terms of the quality of non-dominated solutions known as *Pareto front* in the objective space (Hadka, 2014).

**Approach.** To answer **RQ1**, we first compute three widely used performance evaluation features namely F1-score, Area Under the ROC Curve (AUC) and the Accuracy measures. The first measure is defined as follows:

$$F1\text{-}score = 2 * \frac{Precision * Recall}{Precision + Recall} \in [0, 1] \tag{7.1}$$

In our study, the recall is the percentage of correctly classified CI skip commits over the total number of commits that are skipped, while the precision is the percentage of detected CI skip commits that are actually skipped. These features are computed as:

$$Recall = \frac{TP}{TP + FN} \in [0, 1] \tag{7.2}$$

$$Precision = \frac{TP}{TP + FP} \in [0, 1] \tag{7.3}$$

where TP is the number of the skipped commits that are correctly classified, TN is the number of non-skipped commits that are correctly classified while FP and FN represent the number of incorrectly classified commits for skipped and non-skipped commits respectively.

**AUC** measure assesses how well a model/rule performs on the minority and majority classes and is defined as follows (Cervantes *et al.*, 2013):

$$AUC = \frac{1 + \frac{TP}{TP+FN} - \frac{FP}{FP+TN}}{2} \in [0, 1] \tag{7.4}$$

The **Accuracy** refers to the proportion of correct predictions made by the model and defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \in [0, 1] \tag{7.5}$$

Moreover, since SPEA2, NSGA-II, NSGA-III and IBEA are Pareto-based search-based approaches, it is necessary to evaluate the quality of solution sets with respect to Pareto dominance (Chen, Li & Yao, 2020). This evaluation is generally based on quality indicators that are well-adopted in SBSE community. In this paper, we select three quality indicators based on previous practical guides (Wang, Ali, Yue, Li & Liaaen, 2016a; Riquelme *et al.*, 2015; Li & Yao, 2019).

- **Hyper-volume (HV):** is the most employed measure according to a previous study by Riquelme et al. (Riquelme *et al.*, 2015) which calculates the space covered by the non-dominated solutions. Note that a higher value of HV indicates a better performance of the algorithm.
- **Generational Distance (GD):** occupies the second ranking of the most used measures (Riquelme *et al.*, 2015). GD calculates how far are the Pareto front solutions from the true Pareto front (in our case, it is the optimal detection rule having $TPR = 1$ and $FPR = 0$). The smaller is GD, the better is the algorithm
- **Spacing (SP):** captures another important aspect of quality *i.e.,* uniformity of solutions (Meng, Zhang & Liu, 2005). In a nutshell, SP measures how evenly the members of a Pareto front are distributed. A value of 0 for SP means that all solutions are uniformly spaced *i.e.,* the algorithm possess an optimal quality.

It is worth to mention that all the search-based algorithms and quality indicators used in this study are implemented using MOEA Framework[4], an open-source framework for developing and experimenting with search-based algorithms (Hadka, 2013, 2014).

*RQ2. (Within-project validation) How does our approach perform compared to ML techniques?*

**Motivation.** After evaluating our approach in terms of SBSE performance, it is important to evaluate its efficiency in solving the problem against the state-of-art solution *i.e.,* the Machine Learning (ML) based techniques used in the previous work of Abdalkareem et al. (Abdalkareem *et al.*, 2020). This comparison is important to motivate the need for a search-based approach to improve the CI skip detection.

**Approach.** To answer **RQ2**, we evaluate the predictive performance of our MOGP formulation against five ML techniques, widely used software engineering research (Xia *et al.*, 2017a; Ni & Li, 2018; Xia & Li, 2017; Luo *et al.*, 2017; Hassan & Wang, 2017; Ni & Li, 2018; Santolucito *et al.*, 2018), namely Decision Tree (DT), Random Forest (RF), Naive Bayesian

[4] http://moeaframework.org/

(NB), Logistic Regression (LR) and Support Vector Classification (SVC). As ML models are sensitive to the scale of the inputs, we preprocess the raw data to scale the features using the Standard Scaler module [5].

In addition, to mitigate the issue related to the imbalanced nature of the dataset, we rely on Synthetic Minority Oversampling Technique (SMOTE) method (Chawla *et al.*, 2002), to resample the training data. It is worth to mention that we only resample the training data in order to assess these algorithms in a real-world situation. To compare the predictive performance of SkipCI with ML techniques, we use **AUC, F1-score and Accuracy** that were defined previously.

*RQ3.(Cross-project validation) How effective is our approach when applied on cross-projects?*

**Motivation.** In **RQ3**, we investigate the extent to which CI Skip commit identifications can be generalized through a cross-project prediction. In fact, many projects do not have sufficient historical labeled data to build a model (Abdalkareem *et al.*, 2020) (*e.g.,* small or new projects), which may prevent the project team from using a prediction tool. Cross-project validation is the-state-of-art technique to solve the lack of training data in software engineering (Xia *et al.*, 2017a).

**Approach.** To evaluate our approach on the cross-project validation, we train the studied approaches based on data from 14 projects and use the remaining one project to test the trained model/rule. This experiment is repeated for all the studied project. To gain better insights into the performance of our approach, we compare it with the ML techniques used in RQ2 based on F1-score, AUC and Accuracy in this RQ.

*RQ4. (Features analysis) What features are most important to detect skipped commits?*

**Motivation.** The goal of **RQ4** is to analyze the most influencing features to detect commits to be skipped. The perspective is for researchers interested in understanding how CI commits

[5] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

features can be related to building activities and for developers, who might want to identify the indications that can help them in their decisions to skip their committed changes.

**Approach.** We address **RQ4** by exploring and interpreting the valuable knowledge provided by our generated models, *i.e.,* detection rules. Since we use 10-fold cross-validation and cross-project validation, the analysis produces 11 optimal rules for each project. Additionally, the same feature may occur multiple times in the obtained rules. Thus, to analyze the features importance, we consider that the higher the number of occurrences of a feature in the optimal rules, the more important is the feature in identifying CI skip commits. In addition, to give a more general view, we aggregate the results of features occurrences for each project and feature category (cf. Section 7.3.3). Note that as our approach produces 31 rules at each experimentation, we choose the best rule across these repetitions based on the obtained AUC scores.

### 7.4.2 Studied Projects

Our experiments are mainly based on the dataset provided by Abdalkareem et al. (Abdalkareem *et al.*, 2020) which comprises ten open-source Java projects using Travis CI. Moreover, we extended this dataset based on the same filter by Abdalkareem et al. (Abdalkareem *et al.*, 2020) while considering different programming languages, *i.e.,* projects that (*i*) use Travis CI system, a popular CI service on GitHub (Hilton *et al.*, 2016), and (*ii*) have at least 10% of skipped commits on the master branch, which is required to feed our tool with sufficient historical CI skip records. The projects were gathered using the Big Query Google[6] to query on the GitHub data [7]. Our filters result in a total of 15 projects including the 10 projects studied in Abdalkareem et al. (Abdalkareem *et al.*, 2020). Note that we only consider commits from the date a given project starts using Travis CI. After this filtering process, we end up with 16,334 commits in total.

[6] https://bigquery.cloud.google.com

[7] https://console.cloud.google.com/bigquery?p=bigquery-public-data&d=github_repos&page=dataset

**Noise Detection and Removal.** Since the CI skip option is under-used in practice and that developers may skip commits that cause build failure if not skipped, it is inevitable that the collected data have noise. However, the noise can cause the deterioration of the classifier performance (Gupta & Gupta, 2019; Frénay & Verleysen, 2013) and difficulty in identifying the relevant features (Ekambaram, Goldgof & Hall, 2017). Therefore, we must proceed with a data cleaning process to find and remove the mislabeled instances. To deal with the noise in skipped commits class, we search and find the unit tests for the skipped commits to check whether those changes can cause test failure. To identify the linked tests, we verify, for each commit, whether the names of the changed source files appear in any of the test files (*e.g.,* in Java, we use import or new to call the class). Fortunately, the tested files have passed so we can consider that the problem is handled for class "skipped" (*i.e.,* = 1). Then, we employ the Python package *cleanlab* [8] which supports PU learning (Learning from Positive and Unlabeled examples). PU learning is special case when one of the classes has no error. In fact, P stands for the positive class and is assumed to have zero label errors (the skipped commits in our case) and U stands for unlabeled data which we assume contains some positive examples (the non skipped commits). The goal of PU learning is to (1) estimate the proportion of positives in the negative class, (2) find the errors, and (3) train on clean data. Using this approach, we removed around 12% of the data which left us with 14,294 commits. Table 7.2 provides some statistics about the studied projects and provides information about the level of noise in each project.

### 7.4.3 Statistical Test methods Used

Due to the stochastic nature of search-based algorithms, DT and RF techniques, we compare the performance of these algorithms by running them 31 independent runs for each experimentation then we choose the rule/model with the median value as suggested in (Arcuri & Briand, 2011). Additionally, in order to provide support for the conclusions derived from the obtained results, we use Wilcoxon signed rank test (Wilcoxon *et al.*, 1970), a non-parametric statistical test method to detect significant performance differences with a 95% confidence level ($\alpha$ is set at

[8] https://github.com/cgnorthcutt/cleanlab

Table 7.2 Statistics of the studied projects

| **Project** | **Language** | **Study period** | **the number of commits (filtered)** | **CI SKIP percentage(%)** | **Noise level(%)** |
|---|---|---|---|---|---|
| contextlogger | Java | 2014-12-12 - 2017-03-14 | 211 | 31 | 2 |
| SAX | Java | 2015-06-20 - 2018-02-20 | 403 | 22 | 5 |
| future | Java | 2017-02-05 - 2018-10-02 | 241 | 25 | 14 |
| solr-iso639-filter | Java | 2013-08-20 - 2015-06-16 | 384 | 45 | 9 |
| GI | Java | 2015-06-20 - 2018-05-27 | 338 | 12 | 3 |
| grammarviz2_src | Java | 2014-08-22 - 2018-03-06 | 403 | 14 | 5 |
| parallec | Java | 2015-10-28 - 2018-01-13 | 124 | 60 | 14 |
| candybar-library | Java | 2017-02-22 - 2018-02-05 | 250 | 80 | 13 |
| steve | Java | 2015-10-07 - 2020-04-01 | 770 | 10 | 5 |
| mtsar | Java | 2015-05-06 - 2019-07-17 | 338 | 40 | 13 |
| searchkick | Ruby | 2013-08-12 - 2020-03-31 | 1,509 | 30 | 18 |
| groupdate | Ruby | 2013-04-25 - 2020-03-18 | 535 | 25 | 17 |
| SemanticMediaWiki | PHP | 2013-06-17 - 2020-04-19 | 6,861 | 19 | 14 |
| ransack | Ruby | 2011-07-17 - 2020-04-04 | 1,371 | 20 | 8 |
| pghero | Ruby | 2016-02-19 - 2020-04-06 | 556 | 25 | 8 |

0.05). Vargha-Delaney A (VDA) (Vargha & Delaney, 2000) is also used to measure the effect size. This non-parametric method is widely recommended in SBSE context (Nejati & Gay, 2019). It indicates the probability that one technique will achieve better performance than another technique. When the A measure is 0.5, the two techniques are equal. When the A measure is above or below 0.5, one of the techniques outperforms the other (Thomas *et al.*, 2014). Vargha-Delaney statistic also classifies the magnitude of the obtained effect size value into four different levels: *negligible, small, medium*, and *large* (Scalabrino *et al.*, 2016; Saidani *et al.*, 2020a).

### 7.4.4 Parameter Tuning and Setting

One of the most important aspects of research on SBSE is *parameters tuning* which has a critical impact on the algorithm's performance (Arcuri & Fraser, 2011). This is also compulsory when using ML techniques (Tantithamthavorn *et al.*, 2018a). There is no optimal parameters' setting to solve all problems, therefore, we used a trial-and-error method to select the hyper-parameters (Harman *et al.*, 2012) to handle parameter tuning for search-based algorithms which is a common practice in SBSE (Harman *et al.*, 2012). These parameters are fixed as follows: population size

= 100; maximum the number of generations = 500; crossover probability =0.7; and mutation probability = 0.1.

As for ML techniques, we employed *Grid Search* (Scikit-learn.org, 2006), an exhaustive search-based tuning method widely used in practice. We report in Table 7.3, the parameters' setting of the ML techniques used in this study.

Table 7.3 Parameters' settings for ML techniques under comparison

| **Algorithm** | **Parameters** |
|---|---|
| RF | Max depth of the tree =10<br>the number of estimators = 400 |
| DT | Max depth of the tree=10 |
| NB | Used NB model= Bernoulli NB |
| LR | Max iterations= 200<br>penalty = 'l2' |
| SVC | C=1<br>kernel='rbf'<br>Max iterations= 2000 |

## 7.5 Experimental Study Results

This section presents and discusses the experimental results to answer our research questions.

### 7.5.1 Results of SBSE validation (RQ1)

Figure 7.6 plots the predictive performance of the search-based algorithms under comparison over 4,650 experiment instances (31 runs × 10 folds × 15 projects).

As shown in Figure 7.6, GA achieved in median 85%, 84% and 91% in terms of AUC, F1-score and accuracy, respectively, while RS has shown a lower predictive performance of 80%, 78% and 86% in terms of AUC, F1-score and accuracy, respectively. In comparison with the multi-objective formulation, we clearly see that MOGP techniques (IBEA, SPEA2, NSGA-II, and NSGA-III) outperform the mono-objective algorithm (GA) by achieving, at least, a difference of

6%, 5% and 4% in terms of AUC, F1-score and accuracy respectively. Moreover, the statistical tests' results (Table 7.4) reveal that over 31 runs (of each project and each validation iteration), MOGP techniques show significant improvement over GA and RS with large VDA effect sizes. These findings confirm the suitability of multi-objective formulation for the detection problem as it can provide a better compromise between TPR and FPR. Therefore, our problem formulation passes the SBSE validation.

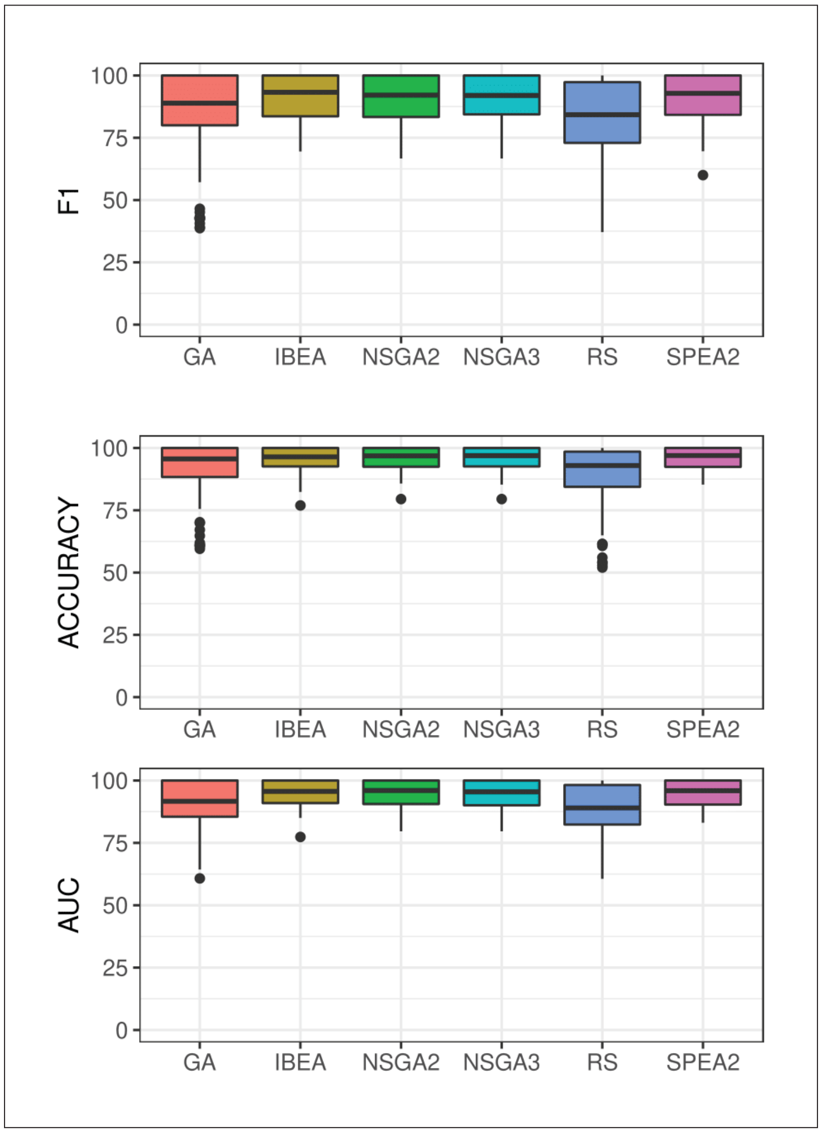


Figure 7.6 Results of the search algorithms for the 4,650 experiment instances (31 runs, 10 validation iterations, 15 projects)

Table 7.4 Statistical tests results of SPEA2 compared to other search-based techniques under cross-validation

| SkipCI | | vs. IBEA | vs. NSGA-II | vs. NSGA-III | vs. GA | vs. RS |
|---|---|---|---|---|---|---|
| AUC | *p-value* | 1 | 1 | 1 | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.52 | 0.52 | 0.53 | 0.76 | 0.79 |
| | *Magnitude* | N | N | N | L | L |
| F1 | *p-value* | 1 | 1 | 1 | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.52 | 0.51 | 0.51 | 0.68 | 0.71 |
| | *Magnitude* | N | N | N | L | L |
| Accuracy | *p-value* | 1 | 1 | 1 | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.53 | 0.52 | 0.52 | 0.71 | 0.75 |
| | *Magnitude* | N | N | N | L | L |

L: Large, M: Medium, S: Small, N: Negligible

With regards to MOGP algorithms, Table 7.5 shows the results of comparison based on the Hyper-Volume (HV), Generational Distance (GD) and Spacing (SP) as described in Section 7.4.1. As reported in the table, the best scores of HV, GD and SP were obtained by SPEA2. In fact, SPEA2 obtained in median a HV score of 0.97, compared to 0.95 achieved by NSGA-II and 0.94 achieved by NSGA-III and IBEA respectively. In terms of GD, SPEA2 achieved a better distance between its generated solutions and the optimal one ( *i.e.,* Pareto front) by reaching 0.08 compared to 0.10, 0.14 and 0.13 for NSGA-II, NSGA-III and IBEA, respectively. As for the SP, the median scores are barely distinguishable between all the algorithms, so they achieve a similar spacing between the generated solutions, even though SPEA2 is slightly better with a median of 0.06 as compared to 0.05 for the other algorithms under comparison.

Overall, we observe that SPEA2 provides the highest median performance among the compared MOGP algorithms, which motivates our choice to adopt it as a search method.

Table 7.5 Results of Pareto-based comparison in terms of hyper-volume (HV), Generational Distance (GD), and SPacing (SP)

| | **SPEA2** | **NSGA-II** | **NSGA-III** | **IBEA** |
|---|---|---|---|---|
| *HV* | **0.97** | 0.95 | 0.94 | 0.94 |
| *GD* | **0.08** | 0.10 | 0.14 | 0.13 |
| *SP* | **0.07** | 0.05 | 0.05 | 0.05 |

**Summary for RQ1.** Our multi-objective formulation has shown its effectiveness compared to mono-objective and random search algorithms by reaching 93% of AUC, 88% of F1-score and 95% of accuracy in median. SPEA2 achieved also higher optimization performance among the studied MOGP algorithms, which motivates our choice to use it a search-based approach.

### 7.5.2 Results of within project validation (RQ2)

Table 7.6 (Appendix A) reports the average (of 10 cross-validation iterations) AUC, F1-score and accuracy scores achieved by each of the studied approaches; while Table 7.7 shows the statistical tests' comparison using the Wilcoxon signed rank test and Vargha-Delaney A effect size.

With regards to AUC, we clearly see that, for all the studied projects, the best scores were obtained by SPEA2 achieving on average 93% with an improvement of at least 4% over the best ML algorithm (SVC). Additionally, the statistical analysis underlines the significant difference with *small* to *large* VDA effect sizes (cf. Table 7.7). For instance, in the *mtsar* project, our approach obtained an AUC score of 91% while we recorded scores of 79%, 78%, 75%, 73% and 67% for SVC, RF, LR, NB and DT respectively. Overall, the results for AUC reveal that SPEA2 can reach the best balance between both minority (skipped commit) and majority (non-skipped) class accuracies, than all the ML techniques. It is worth noting that all ML techniques are trained using re-sampled training sets unlike in SPEA2, which confirms that multi-objective formulation is more suited to handle the imbalance problem (Bhowan *et al.*, 2010; Saidani *et al.*, 2020a).

Looking at F1-score, we also observe that SPEA2 achieved the best results for 14 out of 15 projects with an average score of 87% compared to 79% achieved by RF the best performer among ML techniques. The statistical tests' results reveal that SPEA2 outperforms ML with small (compared to SVC, LR, RF), medium (compared to NB) and large (with DT) effect sizes. Hence, F1-score results demonstrate a compelling superiority of SPEA2 to identify more CI skip commits.

Table 7.6 Performance of SPEA2 vs. ML techniques under cross-validation

| **Project** | **AUC** | | | | | | **F1** | | | | | | **Accuracy** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *SPEA2* | *SVC* | *RF* | *LR* | *NB* | *DT* | *SPEA2* | *SVC* | *RF* | *LR* | *NB* | *DT* | *SPEA2* | *SVC* | *RF* | *LR* | *NB* | *DT* |
| candybar-library | **99** | 91 | 87 | 87 | 85 | 79 | **99** | 94 | 93 | 93 | 90 | 91 | **98** | 91 | 90 | 89 | 85 | 86 |
| contextlogger | **100** | 97 | 100 | 97 | 99 | 99 | **100** | 97 | 100 | 96 | 98 | 98 | **100** | 98 | **100** | 98 | 99 | 99 |
| future | **100** | 96 | 98 | 95 | 93 | 82 | **100** | 88 | 98 | 91 | 87 | 71 | **100** | 94 | 99 | 95 | 92 | 79 |
| GI | **95** | 90 | 91 | 93 | 86 | 89 | **92** | 85 | 88 | 85 | 77 | 71 | **98** | 97 | 97 | 96 | 95 | 93 |
| grammarviz2 | **93** | 89 | 91 | **93** | 86 | 85 | **92** | 85 | 87 | 87 | 73 | 62 | **98** | 97 | 97 | 97 | 93 | 85 |
| groupdate | **93** | 85 | 83 | 85 | 83 | 73 | **86** | 77 | 72 | 76 | 73 | 59 | **94** | 88 | 85 | 87 | 85 | 71 |
| mtsar | **91** | 79 | 78 | 75 | 73 | 67 | **88** | 73 | 72 | 68 | 67 | 62 | **92** | 79 | 79 | 75 | 74 | 66 |
| parallec | **96** | 95 | 93 | 95 | **96** | 94 | **97** | 94 | 91 | 94 | 95 | 93 | **96** | 94 | 90 | 93 | 94 | 92 |
| pghero | **92** | 85 | 86 | 84 | 83 | 72 | **85** | 75 | 75 | 75 | 72 | 57 | **93** | 88 | 86 | 88 | 86 | 68 |
| ransack | **89** | 88 | 85 | 88 | 86 | 67 | **85** | 83 | 72 | 77 | 71 | 47 | **95** | 94 | 88 | 91 | 86 | 56 |
| SAX | **99** | 93 | 94 | 92 | 92 | 93 | **99** | 91 | 93 | 88 | 87 | 86 | **99** | 97 | 97 | 95 | 94 | 94 |
| searchkick | **90** | **90** | **90** | 89 | 87 | 79 | 83 | **85** | 83 | 83 | 81 | 68 | 90 | **91** | 89 | 89 | 89 | 77 |
| SemanticMediaWiki | **69** | 52 | 56 | 66 | 65 | 53 | **45** | 33 | 35 | 43 | 42 | 32 | **70** | 25 | 46 | **70** | 67 | 38 |
| solr-iso639-filter | **90** | 76 | 84 | 78 | 72 | 73 | **88** | 69 | 80 | 75 | 68 | 71 | **90** | 77 | 85 | 78 | 72 | 73 |
| steve | **82** | 80 | 75 | 77 | 79 | 61 | **62** | 46 | 50 | 39 | 47 | 25 | **94** | 86 | 91 | 82 | 87 | 73 |
| **Median** | **93** | 89 | 87 | 88 | 86 | 79 | **88** | 85 | 83 | 83 | 73 | 68 | **95** | 91 | 90 | 89 | 87 | 77 |
| **Average** | **92** | 86 | 86 | 86 | 84 | 78 | **87** | 78 | 79 | 78 | 75 | 66 | **94** | 86 | 88 | 88 | 87 | 77 |

As for the accuracy scores, the obtained results also show that SPEA2 is a better performer than the five considered ML techniques with a significant improvement of 5% on average, and medium to large effect sizes as shown in Table 7.7. Additionally, the accuracy scores of our approach range from 86% to 100% while achieving in median a high score of 92%. For all the studied projects, the accuracy values of SPEA2 exceed those of ML techniques.

However, in the *searchkick* project, SVC is slightly better by achieving a F1-score of 85% compared to 83% for SPEA2. But due to the black-box nature of SVC (it returns probabilities for belonging to a certain class), we cannot have an comprehensible explanation for this result.

Table 7.7 Statistical tests' results of SPEA2 compared to ML techniques under cross-validation

| SkipCI | | **vs. DT** | **vs. RF** | **vs. LR** | **vs. NB** | **vs. SVC** |
|---|---|---|---|---|---|---|
| **AUC** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.77 | 0.62 | 0.65 | 0.71 | 0.64 |
| | *Magnitude* | L | S | S | M | S |
| **F1** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.77 | 0.60 | 0.66 | 0.70 | 0.64 |
| | *Magnitude* | L | S | S | M | S |
| **Accuracy** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.82 | 0.63 | 0.70 | 0.74 | 0.67 |
| | *Magnitude* | L | S | M | L | M |

L: Large, M: Medium, S: Small, N: Negligible

**Summary for RQ2.** SkipCI can achieve higher predictive performance than the studied ML techniques with a statistically significant difference within-validation, with an average of 92% and 87% in terms of AUC and F1-score, respectively, while reaching 94% of the overall classification accuracy.

### 7.5.3 Results of cross-project validation (RQ3)

In this RQ, we compare SkipCI with ML techniques under cross-project validation, using our evaluation measures, the Area Under the ROC Curve (AUC), F1-score, and accuracy, to measure the performance of our approach. Table 7.8 (Appendix A) presents the effectiveness of cross-project modeling compared to ML techniques while Table 7.9 reports the statistical tests' results. Note that we do not include the deterministic ML algorithms (*i.e.,* LR, NB and SVC) for the statistical tests' comparisons as we only recorded 15 observations for each of them, under the cross-project validation. Recall that in cross-project validation, we train the studied approaches based on data from 14 projects and use the remaining project's data as a testing set and this experiment is repeated for the 15 projects.

First, the results show that SPEA2 achieves high scores of AUC with a median of 86% while the values range from 63-97%. We observe that 10 out of 15 projects showed high performance results ($\geq$ 80%). In particular, *contextlogger* achieves a significant AUC score of 98%. Additionally, we observe an improvement of 2% on average over LR the best performer. Moreover, the statistical tests' results show that the difference is significant with *large* effect sizes compared to RF and DT.

The same observations for AUC are also applied to F1-score for which we see that SPEA2 is the best technique 11 out of 15 projects by achieving 84% on median compared to ML techniques that achieved F1-scores of 72% for LR, 69% for NB, 43% for RF as well as SVC and 38% for DT. The statistical analysis confirms the significant difference with RF and DT with large effect sizes, as reported in Table 7.9.

Looking at the accuracy results, we see that the scores are significantly improved compared to ML results, for 9 projects out of 15 by achieving in median 91% (and 87% on average) and the accuracy values range from 61-98%. Similarly to within validation, SPEA2 obtained better accuracy results compared to ML with significant differences compared to DT and RF and large effect sizes.

Table 7.8 Performance of SPEA2 vs. ML techniques under cross-project validation

| **Project** | **AUC** | | | | | | **F1** | | | | | | **Accuracy** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *SPEA2* | *SVC* | *RF* | *LR* | *NB* | *DT* | *SPEA2* | *SVC* | *RF* | *LR* | *NB* | *DT* | *SPEA2* | *SVC* | *RF* | *LR* | *NB* | *DT* |
| candybar-library | **78** | **78** | **78** | 71 | 60 | 59 | **92** | 86 | 72 | 84 | 85 | 81 | **86** | 79 | 65 | 75 | 76 | 70 |
| contextlogger | **98** | **98** | **98** | 53 | 60 | 57 | **97** | 96 | 96 | 49 | 53 | 51 | **98** | 97 | 98 | 36 | 45 | 41 |
| future | **91** | 89 | 71 | 38 | 48 | 45 | **86** | 78 | 55 | 32 | 38 | 37 | **93** | 87 | 70 | 21 | 28 | 23 |
| GI | **89** | 88 | 86 | 54 | 53 | 53 | **87** | 62 | 56 | 24 | 23 | 23 | **97** | 86 | 82 | 20 | 18 | 17 |
| grammarviz2 | **93** | 85 | 87 | 55 | 54 | 52 | **92** | 57 | 65 | 27 | 26 | 26 | **98** | 80 | 87 | 25 | 22 | 20 |
| groupdate | **84** | **84** | 83 | 68 | 69 | 48 | 72 | **74** | **74** | 51 | 52 | 37 | 85 | **86** | **86** | 58 | 60 | 32 |
| mtsar | **72** | 68 | 68 | 66 | 56 | 53 | **68** | 64 | 60 | 63 | 59 | 54 | **71** | 67 | **71** | 63 | 49 | 47 |
| parallec | 95 | 94 | **97** | 85 | 58 | 76 | 96 | 96 | **97** | 91 | 78 | 86 | 95 | 95 | **97** | 88 | 66 | 81 |
| pghero | 80 | **85** | 82 | 57 | 55 | 50 | 68 | **75** | 72 | 43 | 43 | 40 | 82 | **86** | 85 | 46 | 37 | 29 |
| ransack | **86** | 83 | 84 | 59 | 54 | 54 | **84** | 61 | 67 | 37 | 35 | 34 | **95** | 78 | 84 | 37 | 26 | 27 |
| SAX | **95** | 86 | 88 | 55 | 53 | 51 | **95** | 69 | 74 | 38 | 38 | 36 | **98** | 82 | 86 | 32 | 27 | 24 |
| searchkick | 86 | **88** | **88** | 70 | 50 | 48 | 79 | **82** | **82** | 58 | 45 | 44 | 87 | **89** | **89** | 64 | 31 | 31 |
| SemanticMediaWiki | **64** | 61 | 62 | 63 | 63 | 62 | **41** | 37 | 39 | 39 | 40 | 38 | 69 | 73 | **76** | 68 | 76 | 68 |
| solr-iso639-filter | 75 | 65 | **77** | 71 | 61 | 45 | **75** | 65 | 72 | 73 | 67 | 44 | 73 | 64 | **79** | 68 | 57 | 45 |
| steve | 75 | 68 | **77** | 41 | 43 | 45 | **52** | 38 | 47 | 13 | 14 | 14 | **91** | 88 | 87 | 12 | 14 | 14 |
| **Median** | **86** | 85 | 83 | 59 | 55 | 52 | **84** | 69 | 72 | 43 | 43 | 38 | **91** | 86 | 85 | 46 | 37 | 31 |
| **Average** | **84** | 81 | 82 | 60 | 56 | 53 | **79** | 69 | 69 | 48 | 46 | 43 | **87** | 82 | 83 | 47 | 42 | 38 |

In some projects such as *parallec* and *pghero*, NB and LR showed a slightly better performance than SPEA2 but again due to the black-box nature of these algorithms, it is not possible to have an explanation for this result. This recalls the need for explainable classification in machine learning.

However, compared to the within-project validation (RQ2), the results indicate that our approach can achieve a less significant performance with small to medium effect sizes, which may be explained by the fact under cross-project, the target project may have a low collinearity with the source project features thresholds. Overall, SkipCI still be a very promising solution to mitigate the lack of data, especially for new software projects having no enough history.

Table 7.9 Statistical tests' results of SPEA2 under cross-projects compared to its achieved within-project results, RF and DT (*)

| SkipCI | | **vs. Cross-Validation** | **vs. DT** | **vs. RF** |
|---|---|---|---|---|
| **AUC** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.27 | 0.98 | 0.94 |
| | *Magnitude* | M | L | L |
| **F1** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.34 | 0.91 | 0.87 |
| | *Magnitude* | S | L | L |
| **Accuracy** | *p-value* | $< 10^{-16}$ | $< 10^{-16}$ | $< 10^{-16}$ |
| | *A estimate* | 0.31 | 0.97 | 0.94 |
| | *Magnitude* | M | L | L |

L: Large, M: Medium, S: Small, N: Negligible
(*) LR, NB and SVC were executed once for each experiment instance since they are deterministic techniques. Hence we cannot compare the statistical differences with them as we only recorded 15 observation for each of them.

**Summary for RQ3.** *Under cross-project validation, our SkipCI still outperforms state-of-the-art ML techniques by achieving an average of 84%, 79% and 87%* in terms of AUC, F1-score and accuracy, respectively in identifying CI skip commits. While cross-project results are lower than the within-project results (RQ2), our approach is still a promising solution that can be practically used when little/no data is available for new projects.

### 7.5.4 Results of Features analysis (RQ4)

In the following, we report the results of features analysis within and cross-project validations.

#### 7.5.4.1 Within-project results

Tables 7.10 summarizes the ranking of the most occurring features among all the studied projects considering the cross-validation scenario.

**Link to Last Commits** category has shown to be a strong indicator of the commit likelihood to be CI skip as six out of the 12 top-features belong to this category. Additionally, theses features appear the most in 72 out of 150 rules. For example, the committer experience *i.e., EXP*, the top three feature, appears the most in 23 rules out of 150. Similarly to *EXP*, *SEXP* and *REXP* appear also in the top-10 features which indicates that the experienced developers are more familiar with CI features. We observe also that the feature *CRS*, *i.e.,* number of recently skipped commits by the committer, is the most appearing feature in 21 out of 150. A closer examination reveals that this feature has a clear indication of whether a commit should be CI skip or not. For example, in *candybar-library* project, our tool suggests that to a label a commit as skipped, it should have at least 3 recently skipped commits. This condition covers alone 85% of the commits in this project. A similar observation can be applied in *searchkick* project, in which we also observed that having at least 2 recently commits alone would detect the CI skip commits in this project with a F1-score of 78%. Similarly to *CRS* feature, *PRS* also seems to be relevant and appears the most in 17 out of 150 rules. Thus, it is clear that, similar to build failures (Ni & Li, 2017), developers tend to skip commits consecutively maybe because simple changes are generally performed during a specific period of the development *e.g.,* after a release. *NDEV*, *i.e.,* the number of developers that previously touched the changed files, is also a prominent feature and appears the most in three out of 150 rules. For example, in *candybar-library* project, the condition of having $NDEV \leq 3$ would detect alone the CI skip commits in this project with a F1-score of 89%. This indicates that less is *NDEV*, the less risky are the changes and thus the commit can be safely skipped.

**Statistics about the current commit** have a strong probability to indicate whether this commit should be skipped and four of these features appear in the top-1 list and the most in 43 out of 150 rules. First, terms appearing in the commit message can be useful as *CM* is the most occurring feature in 27 out of 150 rules. For example, in *contextlogger* 90% of skipped commit messages contain "Update Readme.md". This finding was previously stated by Abdalkareem et al. (Abdalkareem *et al.*, 2020). In addition, features providing statistics about the current commit change size can also be useful to detect CI skip commits. For example, in the project *mtsar* for which the number of added lines $LA$ is the most important feature, 60% of skipped commits have $LD \leq 23$ which indicates that small changes are more likely to be skipped. The same observation is applied to *ENTROPY* and *ND*.

**Commit Purpose** are also important in detecting CI Skip commits as they appear the most in 35 out of 150 rules. Additionally, the top-1 feature, *i.e., IS_DOC* belongs to this category. For instance, in *parallec* project, the condition $IS\_DOC = 1$ can detect alone the CI skip commits with 95% of F1-score, and in *candybar-library* project 90% of the non-fixing commits, *i.e.,* $IS\_FIX = 0$, are skipped, which is consistent with the real world situation as fixing bugs/problems should be tested. Abdalkareem et al. (Abdalkareem *et al.*, 2020) also pointed out that these features (*i.e.,* CI skip rules as mentioned in the paper) can be strong indicators for CI skip detection.

#### 7.5.4.2 Cross-project results

The top-features analysis under cross-project validation is presented in Table 7.11. This table clearly indicates that, similarly to within-project results, *IS_DOC* is the most important feature across the studied projects. This result is in line with the observations of Abdalkareem et al. (Abdalkareem *et al.*, 2019) as they found that 52% of skipped commits are those that touch documentation or non-source code files. Additionally, statistics linked to last commits are prominent features in cross-project context. Specifically, recent skipped commits from all developers in the project *PRS* are dominant in 4 out of 15 rules while the number of recently skipped commits from the current committer *CRS* is dominant in 3 out of the 15 cross-project

Table 7.10 Top-Features Analysis for all projects (within-project validation)

| **TOP-1** | **# of rules** |
|---|---|
| IS_DOC | 34 |
| CM | 27 |
| EXP | 23 |
| CRS | 21 |
| PRS | 17 |
| ENTROPY | 8 |
| LA | 7 |
| SEXP | 6 |
| NDEV | 3 |
| REXP | 2 |
| IS_FIX | 1 |
| ND | 1 |
| **Total** | 150 |

rules. This strengthens our previous findings claiming that if the commit is skipped, the next commit is more likely to be skipped as well.

Table 7.11 Top-Features Analysis for all projects (cross-project validation)

| **TOP-1** | **the number of rules** |
|---|---|
| IS_DOC | 8 |
| PRS | 4 |
| CRS | 3 |
| **Total** | 15 |

Another important observation to consider is that the statistics of the current commit is not present in the top-features list, which indicates that these features are less likely to be generalized, as CI skip commits depend mainly on the specific context of the project (*e.g.,* the day of the week when developers usually skip commits can differ from a project to another).

**Summary for RQ4.** *Within-project validation, the feature analysis reveals that (1) whether the commit changes only documentation files i.e., IS_DOC, (2) terms appearing in the commit message and (3) the committer experience are the most prominent features in CI skip detection. When it comes to the cross-project rules, the results confirm that IS_DOC is a dominant feature and that there is a strong link between current and previous commits results*

## 7.6 Industrial Case Study

While in RQ1-RQ3, we have shown the efficiency of SkipCI to detect CI Skip commits, we aim in this section to assess the applicability of our approach in practice *i.e.,* from developers' perspectives. We first present the case study design then we report the obtained results.

### 7.6.1 Case Study Design

We designed our industrial case study to address the following research question:

*RQ5.Are the CI skip commit recommendations provided by SkipCI useful for developers who use CI in practice?*

#### 7.6.1.1 Case selection

We conducted a user study with our industrial partner, a large company producing digital document products, services and printers. We evaluate SkipCI during a period of two weeks, *i.e.,* 10 working days on two large and long-lived software systems developed by our industrial partner. We denote both projects as *Project-1* and *Project-2* in this paper for confidentiality reasons. Both projects use a proprietary customized CI system that supports high configurability. The CI system provides the CI skip option, similar to Travis CI. Both projects had 24% and 31% of commits are skipped in project-1 and project-2, respectively, during the last 3 years. Usually, developers working on both projects, integrate their code changes more than once per day. As for case study participants, we have reached out to 36 developers working full time on both projects and invited them to participate in our experiments. A total number of 14 developers volunteered to participate in the experiments, where 8 developers are from project-1, and 6 developers are from project-2.

Participants were first asked to fill out a pre-study questionnaire that consists of seven control questions. The questionnaire helped us to collect background and demographic information such as their role within the company and within the project, their experience with the studied project, their academic degree, their proficiency in CI practice, their familiarity with the CI skip commit feature, and their experience with software quality assurance. The list of questionnaires and the obtained results can be found in our online replication package. All the participants had a minimum of 6 years experience post-graduation and were working as active programmers with strong backgrounds in CI practices, and software quality assurance. All participants are familiar with the studied projects (71% have over 3 years, and 29% have over 2 years experience with the concerned projects). Moreover, all of our participants hold an academic degree related to computer science and/or software engineering (50% Bachelor, 35.7% Masters, 14.3% Ph.D.).

#### 7.6.1.2 Study setup and analysis method

As a first step to prepare and integrate the SkipCI tool into the CI pipeline, we collected data about the history of commits and builds for both projects, from the last 3 years from the version control system and the CI tool. Such data allowed us to generate the set of required features to train our model to generate the CI skip detection rules for both studied projects. Thereafter, we deployed SkipCI and integrated it into the CI service pipeline for both projects. The SkipCI tool is triggered whenever a new commit is pushed and shows a pop-up notification message recommending whether the current commit could be (1) skipped or (2) not skipped. To keep track of the developers' decisions for our evaluation, we integrated a routine in our SkipCI tool to record the commit ID, the recommended action, the developer decision, and her/his comments (if any).

With the sake of comparing the results of our tool with a baseline approach, and better understanding the participants behavior, we also considered a random recommender tool (that we refer to as RANDOM in this paper) that generates CI skip commits at a random way. Then, for each project, we split the concerned participants into two groups (A and B) of equal sizes (*i.e.,* 7 developers each), in such a way that each group will use one of the tools, SKIPCI or RANDOM. Hence, in total, we had four groups (2 groups per project) as shown in Table 7.12.

Table 7.12 Participants partition statistics

| **Project** | **Tool** | **Group** | **the number of developers** | **the number of commits** |
|---|---|---|---|---|
| Project-1 | SKIPCI | A1 | 4 | 61 |
| | RANDOM | B1 | 4 | 52 |
| Project-2 | SKIPCI | A2 | 3 | 43 |
| | RANDOM | B2 | 3 | 48 |

During the study period, for each commit, the developer receives a recommendation from either SKIPCI or RANDOM (depending on her/his group) indicating to skip the commit or not. The developer can either *“accept"* or *“decline"* the skip recommendations. Moreover, we configured both tools to show a pop-up notification asking the developer to optionally leave her/his justification about his accept/decline decision, immediately after she/he makes a decision. To avoid potential biases in our experiments, the individual developers were not aware of the specific tools being compared (*i.e.,* SKIPCI and RANDOM).

In total, the 14 participants performed 204 commits (113 for Project-1 and 91 commits for Project-2) during the study period of 10 business days. The number of performed commits by each group for each project are reported in Table 7.12. After the study period, we collected the experiments records of the accepted and declined recommendations for each developer for both tools. To analyze the collected data and answer *RQ5*, we compute for each project, and each group, the ratio of both *accepted* and *declined* recommendations by the developers, with respect to the total number of recommendations provided from each tool.

Moreover, to further understand how to improve SkipCI, we performed an online interview with four developers from both groups A1 and A2 who assessed the SkipCI bot (2 developers from each project). The interview consists of a structured discussion guided by three main questions:

- **Q1:** *How important is the CI Skip option in CI practice?*
- **Q2:** *How efficient is our CI skip recommendation bot in the context of your project?*
- **Q3:** *What additional features or improvements do you recommend to further improve SkipCI?*

In the next subsection, we present and discuss the obtained results for our user case study.

### 7.6.2 Case study results and discussion

Table 7.13 summarizes the results of deploying SkipCI during 10 working days with our industrial partner on both projects. We observe that developers accepted most of the *"Skip"* recommendations provided by SkipCI in both projects (*i.e.,* groups A1 and A2), with 88.9% and 90.9% in project-1 and project-2, respectively. We also observe that SkipCI recommended to skip 18 out of the 61 commits from project 1 (29.5%), and 11 out of the 48 commits from project-2 (25.6%). On the other side, developers tend to accept only a small number of *"Skip"* recommendations provided by Random, with 17.9% and 13.6% of the total skip recommendations from project-1 and project-2, respectively.

As for the *"Non-skip"* recommendations, we also observe SkipCI recommended not to skip 43 out of 61 commits (70.4%) for project-1, and not to skip 32 out 43 commits (74.4%) for project-2, that were accepted by developers with over 90% for both projects. Looking at the Random recommendations, we found that developers accepted 66.7% and 65.4% of non-skip recommendations from project-1 and project-2, respectively. While the results of random recommendations may seem quite high, their results could be justified by the fact that developers tend to generally run the build after each commit in the context of CI.

By looking at the comments left by the SkipCI participants when justifying their decisions, developers of both projects highlighted in their comments that they found the CI skip commit relevant because it can save time for trivial changes that do not need to build the entire system. For instance, one developer wrote in a comment in response to our recommendation:

💬 *"I agree! So I actually do not see benefit of starting the CI build immediately, there are only non-code changes were made in my last commit"*

Another developer mentioned in his comment:

💬 *"[...] of course this commit and my previous one should be skipped, my changes are only on markdown and JSON files. It's bad for this commit to wastefully use server time"*

Furthermore, another developer who declined a CI skip recommendation from SkipCI commented:

💬 *"I am fine with skipping this commit, but this time I want to merge my changes to the master branch. My changes can only be merged when the CI build has successfully run [...] this will never happen if I skip the CI build".*

In another declined CI skip recommendation, the developer left the following comment:

💬 *"Well I don't agree, even very few lines have been changed related to my function call redirection and my variable rename changes, I am inclined to run the build to be on the safe side anyway"*

Furthermore, to gain insights and better understand how to improve SkipCI, we interviewed four developers (2 developers from each project) who assessed SkipCI within the two-week study period, as described in Section 7.6.1.2. The four developers highly appreciated the CI skip feature provided by the CI systems given the waste of time of resources for unnecessary builds. The four developers were also very positive on the relevance of our SkipCI bot in the context of their projects. In response to possible improvement of SkipCI, one of the interviewees indicated that:

💬 *"[...] I found the bot very useful to me, basically what I used to do is to skip the entire build pipeline if certain condition is met based on my "safe list". I manually defined a basic, yet simple, script to check my conditions like "if the changed files are in a given directory, then trigger the CI build, else skip it". However, I have to manually go through the change diff to* *look at each individual file [...] and I often ignore or change my defined conditions. I wish to have more control over the proposed CI skip recommendation tool, based on what I'm doing and based on how the project evolves"*

Two other developers pointed out the importance of providing more details by SkipCI to justify the recommended decisions along with a summary of the changes in either a textual manner or in a user-friendly visualization to help them taking the right decision while giving more trust and transparency to our tool. Another developer also recommended to add a user-friendly configuration layer on top of the tool that allows the developer to customize the current conditions and add her/his own conditions to the tool.

Table 7.13 Case study results

| **Project** | **Group** | **Tool** | **Total commits** | **Recommendation** | **# commits** | **Results*** | |
|---|---|---|---|---|---|---|---|
| Project-1 | A1 | SkipCI | 61 | Skip | 18 | 11.1% 🟥 | 🟦 88.9% |
| | | | | Non-skip | 43 | 9.3% 🟥 | 🟦 90.7% |
| | B1 | Random | 52 | Skip | 28 | 82.1% 🟥 | 🟦 17.9% |
| | | | | Non-skip | 24 | 33.3% 🟥 | 🟦 66.7% |
| Project-2 | A2 | SkipCI | 43 | Skip | 11 | 9.1% 🟥 | 🟦 90.9% |
| | | | | Non-skip | 32 | 9.4% 🟥 | 🟦 90.6% |
| | B2 | Random | 48 | Skip | 22 | 86.4% 🟥 | 🟦 13.6% |
| | | | | Non-skip | 26 | 34.6% 🟥 | 🟦 65.4% |

* 🟦 *Accepted*, 🟥 *Declined*

Overall, the outcomes of this survey are aligned with the motivations of this paper advocating for using interpretable rule-based recommendations based on various features that could be learned from the CI build and project history considering both (1) skipped and (2) non-skipped CI commits (*i.e.,* the minority and majority classes).

From this user study, we learn that to improve SkipCI, we have to include three aspects. First, we need to deploy along with the tool, a set of interpretable and configurable rules so that developers can understand and customize the tool based on their preferences/experience on what should or should not be skipped. Second, the tool needs to provide more details along with the recommended skip decisions to justify whether the commit should be skipped or not. Third, the rules should be re-trained and updated regularly as the project evolves through learning from the recent decisions taken by the developers. Moreover, based on our interactions with developers in this industrial case study, we advocate that the CI skip feature should be used *wisely* and with *caution*. The developer needs to make the necessary verification based on her/his current code changes in the commit, before taking the decision to skip or not. This is indeed an important aspect, as it remains to some extent hard to understand the semantics of source code changes.

## 7.7 Discussion and Implications

### 7.7.1 For CI developers

**Developers can efficiently identify the CI Skip commits.** The usefulness of our SkipCI approach has been shown through its achieved results within and cross-project validations as well as our industrial case study. Nevertheless, we believe that the key innovation of our approach is its ability to provide the user with a comprehensible justification for the detection of CI skip commits especially when the changes made in the commit are non-trivial. For instance, Figure 7.7 illustrates an example of a detection rule generated by SkipCI to detect CI Skip commits in the *Semantic MediaWiki*[9] project with a high AUC score of 80%. Using this rule, we can detect the CI commit described early in Section 7.2 as CI skip since its characteristics satisfy the conditions of the rule, *e.g.,* having a number of changed files $NS \leq 2$, $LD \leq 93$ and $IS\_FIX = 0$. Moreover, it is worth noting that, thanks to the flexibility of MOGP techniques, it may be possible to reduce the complexity of the generated detection rules (*e.g.,* tree size and/or depth) in order to generate more comprehensible justification by considering this objective in the fitness function (or as a constraint in the solution encoding), but at the cost of scarifying the accuracy as these objectives are in conflict (Saidani *et al.*, 2020a, 2021a).

[9] https://github.com/SemanticMediaWiki/SemanticMediaWiki

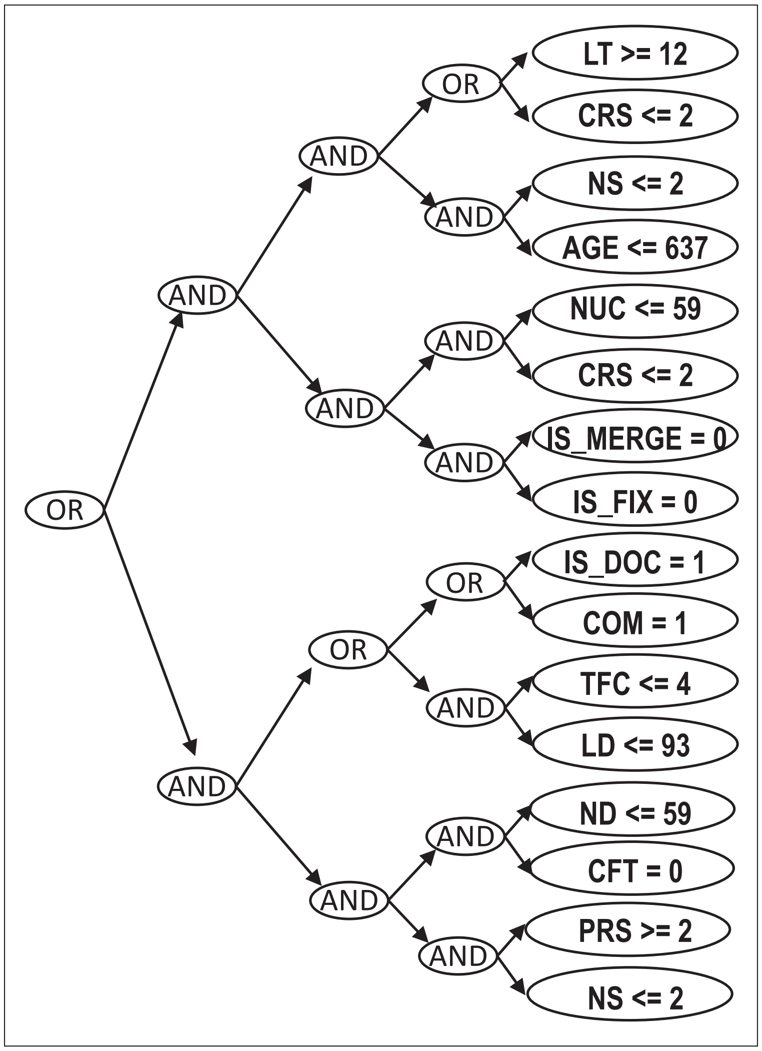


Figure 7.7 An illustrative example of CI skip detection rule for the *Semantic MediaWiki* project

### 7.7.2 For researchers

**Can the predictive performance be improved with the use of SMOTE?** So far, we showed that SkipCI provides an effective improvement over the state-of-the-art without rebalancing. However, one can argue that the use of resampling can further improve the identification of CI Skip commits of SkipCI, even though the latter has shown less sensitivity to the class imbalance problem as pointed out in prior research (Bhowan *et al.*, 2013, 2012, 2010, 2009). Thus, we conduct a set of additional experiments to rebalance the input data prepared in Section 7.4.2(10 cross-validation) using SMOTE and re-run the MOGP learning process on the new balanced data. We use the same approach described in Section 7.3 to generate our detection rules. To provide a comprehensive comparison, we compute the F1-score, AUC score and the overall accuracy. Figure 7.8 shows the obtained results with (in red) and without (in blue) using SMOTE.

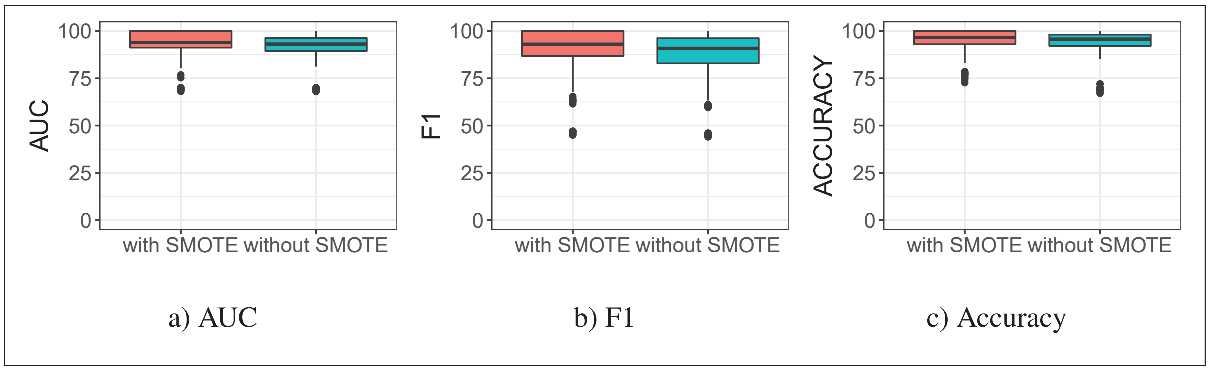


Figure 7.8  Comparison of SkipCI results with (red) and without (blue) using SMOTE

We observe that by using SMOTE, SkipCI can achieve in median 94%, 93% and 96% in terms of AUC, F1-score and accuracy respectively, which represents an improvement of 1%, 5% and 1%, respectively with negligible (for AUC and accuracy) to small (for F1-score) effect sizes. This suggests that using an external approach to artificially rebalance the dataset can slightly improve the classification performance of SkipCI. However, the sampling techniques have their own drawbacks. In fact, sampling can lead to over-fitting and affect the interpretability of the generated rules (Tantithamthavorn *et al.*, 2018a) as the original and the balanced training corpora have different characteristics. Furthermore, rebalancing can also be computationally expensive (Bhowan *et al.*, 2009). For these reasons, in this paper, we advocate that using MOGP alone is a better classification strategy when the data is imbalanced.

**Researchers can investigate periodicity in CI skipped commits.** Our features analysis results (RQ4) reveal that many CI skip commits occurred consecutively and features related to historical statistics about the commits are strong indicators of the current commit outcome. Hence, we encourage researchers to investigate what software engineering activities may link with such skip periods, *e.g.,* automated refactoring etc.

### 7.7.3 For tool builders

**Further tool support in CI.** Our industrial case study reveals the importance of the SkipCI tool from developers' perspectives. As discussed in Section 7.6.2, one of the developers indicates that he is using his own simplified defined conditions to help him decide whether to run or skip the build for a given commit. This recalls the importance of providing efficient, lightweight and practical tool support to further improve the CI pipeline. Indeed, the current CI practice is still in its infancy, as CI technologies are experiencing an exponential growth in both open-source and commercial projects. Further support from tool builders is needed to adequately respond to the developers' needs and cut with the expenses of CI build in modern software engineering.

## 7.8 Threats to validity

**Threats to internal validity** are concerned with the factors that could have affected the validity of our results. The main concern could be related to the stochastic nature of search-based algorithms, RF and DT. To address this issue, we repeated each experimentation 31 times and considered the median scores values used to evaluate the predictive performance. Threats to internal validity could also be related to our industrial case study that aimed at deploying and evaluating our SkipCI approach in practice. The opinions/decisions of the practitioners involved in our study may be divergent or influenced by the used tool when it comes to accepting or declining a CI skip commit recommendation, which can impact our results. To mitigate this threat, we compared the results of our tool to a baseline tool with random recommendations.

**Threats to construct validity** are mainly related to the rigor of the study design. First, one possible threat is related to the selected performance metrics as there exist many other metrics. We basically used standard performance metrics namely F1-score, AUC and accuracy that are widely employed in predictive models comparison (Hastie *et al.*, 2009) and also considered three performance measures, *i.e.,* hyper-volume (HV), generational distance (GD) and spacing (SP) in order to compare multi-objective algorithms from SBSE perspective. Second, although we used different search-based and ML algorithms, there exist other techniques. As a future work, we plan to extend our empirical study with other baseline techniques. Another threat to construct validity is related to our industrial validation since we considered the random search algorithm as baseline. We plan as a future work to extend this validation with other baseline approaches.

Third, a threat to construct validity could be related to the annotated set of skipped commits used in our dataset. To mitigate this issue, we ran unit tests that are linked to the skipped commits and checked that none of those tests would fail. Second, to deal with the noise of the negative class, we employed the python package *cleanlab* which helped us to detect about 12% of the data as noise. We also checked manually the precision of the detected labels by verifying that at least those non skipped commits are not correlated with a build failure. We found that the precision of this tool ranges from 80% to 100%. Nevertheless, it may misses other commits that should have been skipped (*i.e.,* the recall). Another threat to construct validity could be related to parameters' tuning as setting different parameters can lead to different results for search-based as well as ML techniques. We mitigated this issue by applying several trial and error iterations to tune search-based algorithms and relied on Grid Search (Scikit-learn.org, 2006) method to find the optimal settings of ML techniques.

**Threats to external validity** are concerned with the generalizability of results since the experiments were based on 15 open-source projects that use Travis CI, and two projects from our industrial partner. However, it is worth to recall that CI skip option is generally under-used in practice and hence labeled training data is not always available. Additionally, while in our approach we focus on Travis CI, a popular CI system (Hilton *et al.*, 2016), our approach can be applied to other CI systems. Future replications of this study are necessary to confirm our findings.

## 7.9 Conclusions and Future Work

This paper proposed a novel search-based approach for CI Skip detection, SkipCI, in which we adapted SPEA2 to generate optimal detection rules with a tree-like representation in order to find the best trade-off between two conflicting objective functions to (1) maximize the true positive rate, and (2) minimize the false positive rate.

An empirical study conducted on a benchmark of 14,294 Travis CI commits of 15 projects that use Travis CI shows that SkipCI outperforms Random Search, mono-objective Genetic Algorithm and three other multi-objective algorithms which indicates that our adaptation is more suited than other search-based techniques. Considering two validations namely cross-validation and cross-project validation, the statistical analysis of the obtained results reveals that SkipCI is advantageous over five Machine Learning (ML) techniques confirming that our formulation is better to solve the problem. Moreover, our experience with the industrial partner demonstrates the effectiveness of SkipCI in providing relevant recommendations to developers from two different projects. Regarding the features analysis, we found that documentation changes,terms appearing in the commit message and the committer experience are the most prominent features in CI skip detection. When it comes to the cross-project scenario, the results reveal that besides the documentation changes, there is a strong link between current and previous commits results.

Our future research agenda includes performing a larger empirical study with other open-source projects while considering other features. For instance, one can measure the code similarity before and after the commit to identify whether the changes do probability preserve the code functionality and hence can be skipped. The tool can also detect whether the developer performed a code refactoring (Fowler, 2018) such as renaming. This would allow the tool to detect other CI skip opportunities that may have not been detected yet using the current features.

# CONCLUSION AND RECOMMENDATIONS

The core goal of this thesis is to support the adoption of Continuous Integration (CI), the leading edge of modern software engineering. As a first step towards this support, this thesis is focused on gaining insights into CI adoption by conducting empirical studies to understand its impacts and challenges. Then, we proposed novel approaches to address its major issue: the build process.

In Chapter 3, we presented the first empirical study on the challenges of CI based on the discussion of Stack Overflow (S0), a popular Q&A website leveraged by developers to seek help with development issues. We analyzed SO posts related to CI through a mixed-method with quantitative and qualitative analyzes. To study the trends of CI discussions, we investigated the metadata of CI questions, users and tags. Then, we extracted the CI main topics using Latent Dirichlet Allocation (LDA) tuned with Genetic Algorithm (GA). Finally, we investigated the most popular and difficult topics faced by developers and perform a qualitative analysis based on a statistical sample of unanswered questions to get further insights into CI challenges. The LDA clustering revealed that developers face challenges with six main topics namely Build, Testing, Version Control, Configuration, Deployment and CI Culture. Particularly, we found that the build topic is the most popular among the studied topics and that version control and testing topics are the most difficult for SO community.

In Chapter 4, we conducted the first empirical study to emphasize the role of CI process in changing the way code Refactoring (software changes that aim to ensure/improve the code quality) is applied. Indeed, we investigated the evolution of Refactoring practices, in terms of frequency, size and involved developers, after the switch to CI. We collected a corpus of 99,545 commits and 89,926 Refactoring operations extracted from 39 open-source GitHub projects and analyzed the changes using Multiple Regression Analysis (MRA). Our study revealed that the adoption of CI is associated with a drop in the Refactoring size as recommended, while

Refactoring frequency as well as the number (and its related rate) of developers that perform Refactoring are estimated to decrease after the shift to CI, indicating that Refactoring is less likely to be applied in CI context.

Motivated by the fact that CI builds discussions represent more than 40% of developers discussions on SO (as revealed in Chapter 3). We proposed in this thesis, two novel approaches to address the build failure detection:

- First in Chapter 5, we proposed an approach based on Multi-Objective Genetic Programming (MOGP) to automatically generate the prediction rules. Our approach aims at finding the best combination of CI built features and their appropriate threshold values, based on two conflicting objective functions to deal with both failed and passed builds, which allows to tackle the imbalanced nature of CI builds. This approach supports also an explanation mechanism that helps the developers in the resolution process. This approach was empirically validated and implemented as a standalone Java tool called BF-Detector (Saidani *et al.*, 2021a). Additionally, we have improved the predictive performance of our tool by considering the code smells related information (Saidani & Ouni, 2021).
- In 6, we introduced *DL-CIBuild* a novel approach that uses Long Short-Term Memory (LSTM)-based Recurrent Neural Networks (RNN) to construct prediction models for CI build outcome prediction. The problem was comprised of a single series of CI build outcomes and a model is required to learn from the series of past observations to predict the next CI build outcome in the sequence. In addition, we tailored Genetic Algorithm (GA) to tune the hyper-parameters for our LSTM model. Through an empirical validation, we showed that *DL-CIBuild* outperforms traditional approaches, has less sensitivity to the training set size and an effective robustness to the concept drift.

Build duration is also a potential concern for CI developers. To address this problem, we proposed in Chapter 7, a novel search-based approach to automatically detect CI Skip commits based on the adaptation of Strength-Pareto Evolutionary Algorithm (SPEA2). Our approach aims to provide the optimal trade-off between two conflicting objectives to deal with both skipped and non-skipped commits. We evaluated our approach on a benchmark of 14,294 CI commits from 15 projects. The statistical tests revealed that our approach shows a clear advantage over the baseline approaches. Furthermore, we deployed and evaluated the usefulness of our approach as standalone tool called SkipCI, with our industrial partner. The qualitative results demonstrated the effectiveness of SkipCI in providing relevant CI skip commit recommendations to developers for two large software projects from practitioner's point of view.

## Perspectives

Although this Ph.D. work has made many significant contributions towards supporting the adoption of CI, many different avenues for future work remain unexplored. We summarize some of the main directions for future research on CI.

### Short-Term

**Improve SkipCI.** We plan to improve SkipCI by considering other features *e.g.,* the semantics of source code changes. Such features would allow detecting the changes that do alter the behaviour of the code *i.e.,* refactoring.

**Test *DL-CIBuild* and BF-Detector in an industrial environment.** Although we showed through empirical validation that our tools are effective in detecting CI build failure, we believe that an industrial validation is needed in order to confirm their effectiveness from developers' perspective.

**Compare *DL-CIBuild* vs BF-Detector.** We plan in the near future to conduct an additional empirical study in order to compare the predictive performance of these two tools using the same dataset and set of evaluation metrics.

**Collect data of other CI services.** First, it is essential to mention that our empirical study in Chapter 4 and validations in Chapters 5, 6 and 7 are mainly based on TravisTorrent (Beller *et al.*, 2017), the only available dataset that contains information of builds considering Travis CI service. Hence, there still many other CI services, such as Jenkins (Jenkins, 2019), are not yet studied. Hence, future work is required to first collect data of other CI services and then investigate the generalizability of our results based on that data.

**Provide a generic GUI-based tool.** As a way to combine all the proposed approaches, we plan to develop a GUI-based tool in which we allow users to interact and provide feedback about our recommendations as this would help improving the tools.

**Long-Term**

**Automated tools to support quality assurance efforts.** We have shown through our study in Chapter 4, that the application of code Refactoring is even harder after the switch to CI. This finding would encourage us to propose novel approaches to support developers on this matter. We therefore plan to develop a new approach to recommend the appropriate refactorings dedicated to CI context *i.e.,* these recommendation should be simple (as the guidelines propose to commit small changes) and frequent to cope with the agile nature of CI.

**Tools for resolving CI build failures.** Once the build failure is detected, developers may follow a tedious process to localize the cause of the failure and resolve it. Hence, a future work is needed to supply development teams with tools to identify potential files in order to accelerate the build fixing process. We believe that the explainable prediction of build failures proposed in Chapter 5 can provide a valuable support on how to proceed with failure localization and resolution based on the violated rules.

**Need for further effort to speed-up CI process.** In Chapter 7, we proposed an effective solution to partially address the problem of CI long process, by detecting the changes that can be skipped during the build. Nevertheless, we believe that more effort is need to speed up the time need for changes that cannot be skipped.

# APPENDIX I

# OPTIMAL PARAMETERS FOR EACH RQ (*DL-CIBUILD* PAPER)

## 1. Optimal Parameters for online validation (RQ1 and RQ2)

Table-A I-1 Optimal Parameters for RQ1 and RQ2

| Project | Experiment | nb_epochs | nb_batch | time_step | drop_proba | nb_layers | nb_units | optimizer |
|---|---|---|---|---|---|---|---|---|
| cloudify | 1 | 7 | 38 | 38 | 0.11 | 3 | 72 | adam |
| | 2 | 6 | 26 | 36 | 0.07 | 3 | 50 | rmsprop |
| | 3 | 7 | 14 | 36 | 0.02 | 2 | 94 | adam |
| | 4 | 7 | 22 | 44 | 0.12 | 3 | 76 | adam |
| | 5 | 7 | 23 | 60 | 0.03 | 3 | 64 | adam |
| graylog2-server | 1 | 7 | 27 | 31 | 0.06 | 2 | 44 | adam |
| | 2 | 6 | 16 | 54 | 0.19 | 4 | 76 | adam |
| | 3 | 7 | 64 | 39 | 0.05 | 2 | 52 | rmsprop |
| | 4 | 4 | 22 | 33 | 0.16 | 2 | 54 | adam |
| | 5 | 4 | 24 | 39 | 0.05 | 4 | 63 | rmsprop |
| jackrabbit-oak | 1 | 7 | 14 | 34 | 0.12 | 4 | 60 | adam |
| | 2 | 7 | 45 | 36 | 0.14 | 2 | 92 | adam |
| | 3 | 7 | 11 | 30 | 0.16 | 4 | 72 | adam |
| | 4 | 7 | 14 | 45 | 0.18 | 2 | 57 | rmsprop |
| | 5 | 7 | 15 | 49 | 0.15 | 3 | 94 | adam |
| jruby | 1 | 7 | 21 | 38 | 0.12 | 2 | 47 | adam |
| | 2 | 6 | 22 | 48 | 0.08 | 2 | 52 | adam |
| | 3 | 6 | 5 | 30 | 0.05 | 4 | 49 | adam |
| | 4 | 5 | 12 | 31 | 0.05 | 2 | 82 | adam |
| | 5 | 6 | 16 | 40 | 0.06 | 4 | 39 | rmsprop |
| metasploit-framework | 1 | 5 | 52 | 56 | 0.19 | 2 | 96 | rmsprop |
| | 2 | 6 | 24 | 50 | 0.15 | 2 | 45 | adam |
| | 3 | 4 | 16 | 45 | 0.19 | 2 | 61 | rmsprop |
| | 4 | 5 | 16 | 49 | 0.04 | 2 | 79 | rmsprop |
| | 5 | 6 | 56 | 34 | 0.07 | 3 | 53 | adam |

Table-A I-2 Optimal Parameters for RQ1 and RQ2 (continued)

| Project | Experiment | nb_epochs | nb_batch | time_step | drop_proba | nb_layers | nb_units | optimizer |
|---|---|---|---|---|---|---|---|---|
| open-build-service | 1 | 6 | 28 | 32 | 0.07 | 2 | 68 | adam |
| | 2 | 5 | 52 | 33 | 0.03 | 3 | 78 | rmsprop |
| | 3 | 5 | 63 | 42 | 0.08 | 2 | 92 | adam |
| | 4 | 6 | 13 | 39 | 0.07 | 3 | 95 | adam |
| | 5 | 6 | 26 | 31 | 0.10 | 2 | 40 | adam |
| openproject | 1 | 7 | 12 | 37 | 0.14 | 2 | 94 | adam |
| | 2 | 7 | 30 | 40 | 0.16 | 2 | 65 | rmsprop |
| | 3 | 5 | 20 | 60 | 0.13 | 4 | 46 | rmsprop |
| | 4 | 5 | 18 | 35 | 0.13 | 2 | 45 | adam |
| | 5 | 7 | 28 | 38 | 0.11 | 3 | 46 | adam |
| rails | 1 | 5 | 58 | 36 | 0.05 | 2 | 45 | adam |
| | 2 | 7 | 49 | 33 | 0.20 | 3 | 67 | rmsprop |
| | 3 | 5 | 43 | 47 | 0.11 | 3 | 68 | rmsprop |
| | 4 | 6 | 63 | 50 | 0.14 | 3 | 68 | adam |
| | 5 | 4 | 62 | 55 | 0.11 | 3 | 55 | adam |
| ruby | 1 | 6 | 22 | 38 | 0.08 | 2 | 82 | adam |
| | 2 | 7 | 10 | 48 | 0.05 | 4 | 83 | adam |
| | 3 | 7 | 26 | 58 | 0.18 | 3 | 67 | adam |
| | 4 | 6 | 55 | 50 | 0.06 | 3 | 73 | adam |
| | 5 | 4 | 4 | 30 | 0.06 | 4 | 48 | adam |
| sonarqube | 1 | 6 | 7 | 55 | 0.06 | 4 | 79 | rmsprop |
| | 2 | 6 | 8 | 51 | 0.18 | 4 | 87 | rmsprop |
| | 3 | 5 | 4 | 31 | 0.01 | 2 | 58 | rmsprop |
| | 4 | 6 | 16 | 51 | 0.02 | 2 | 86 | rmsprop |
| | 5 | 5 | 43 | 43 | 0.12 | 2 | 90 | adam |

## 2. Optimal Parameters for cross-project validation (RQ3)

Table-A I-3 Optimal Parameters for cross-project validation (Jruby is the training project)

| **Parameter** | **Optimal value** |
|---|---|
| nb_epochs | 5 |
| nb_batch | 16 |
| time_step | 60 |
| drop_proba | 0.2 |
| nb_layers | 4 |
| nb_units | 32 |
| optimizer | "adam" |

## 3. Optimal Parameters for RQ4

Table-A I-4 Optimal Parameters for RQ4

| Project | Experiment | nb_epochs | nb_batch | time_step | drop_proba | nb_layers | nb_units | optimizer |
|---|---|---|---|---|---|---|---|---|
| | 1 | 4 | 42 | 58 | 0.11 | 2 | 81 | adam |
| cloudify | 2 | 6 | 10 | 50 | 0.11 | 3 | 76 | adam |
| | 3 | 3 | 5 | 46 | 0.02 | 3 | 72 | adam |
| | 1 | 5 | 45 | 46 | 0.16 | 2 | 60 | adam |
| graylog2-server | 2 | 5 | 45 | 49 | 0.06 | 2 | 91 | adam |
| | 3 | 4 | 17 | 35 | 0.08 | 2 | 86 | adam |
| | 1 | 6 | 25 | 59 | 0.20 | 2 | 76 | adam |
| jackrabbit-oak | 2 | 2 | 29 | 52 | 0.02 | 3 | 90 | adam |
| | 3 | 5 | 62 | 55 | 0.09 | 2 | 73 | adam |
| | 1 | 3 | 44 | 45 | 0.07 | 2 | 74 | adam |
| jruby | 2 | 5 | 20 | 53 | 0.14 | 2 | 69 | adam |
| | 3 | 5 | 23 | 42 | 0.04 | 2 | 81 | adam |
| | 1 | 4 | 5 | 55 | 0.06 | 3 | 33 | adam |
| metasploit-framework | 2 | 5 | 24 | 46 | 0.09 | 3 | 34 | adam |
| | 3 | 5 | 34 | 31 | 0.01 | 2 | 40 | adam |
| | 1 | 4 | 34 | 30 | 0.16 | 2 | 37 | adam |
| open-build-service | 2 | 6 | 54 | 56 | 0.14 | 2 | 73 | adam |
| | 3 | 6 | 54 | 54 | 0.19 | 3 | 78 | adam |
| | 1 | 2 | 17 | 50 | 0.07 | 2 | 70 | adam |
| openproject | 2 | 6 | 36 | 37 | 0.06 | 2 | 84 | adam |
| | 3 | 4 | 57 | 53 | 0.02 | 2 | 42 | adam |
| | 1 | 4 | 5 | 59 | 0.03 | 2 | 38 | adam |
| rails | 2 | 3 | 11 | 53 | 0.14 | 2 | 51 | adam |
| | 3 | 5 | 49 | 38 | 0.11 | 3 | 61 | adam |
| | 1 | 3 | 55 | 50 | 0.09 | 3 | 88 | adam |
| ruby | 2 | 3 | 35 | 37 | 0.15 | 2 | 66 | adam |
| | 3 | 5 | 32 | 41 | 0.03 | 3 | 86 | adam |
| | 1 | 5 | 18 | 48 | 0.07 | 3 | 62 | adam |
| sonarqube | 2 | 4 | 15 | 52 | 0.19 | 3 | 48 | adam |
| | 3 | 3 | 45 | 59 | 0.18 | 3 | 74 | adam |

## 4. Optimal Parameters for RQ5

Table-A I-5 Optimal Parameters for RQ5 (old data)

| Project | Experiment | nb_epochs | nb_batch | time_step | nb_units | optimizer | drop_proba | nb_layers |
|---|---|---|---|---|---|---|---|---|
| cloudify | 1 | 6 | 4 | 41 | 32 | rmsprop | 0.15 | 4 |
|  | 2 | 5 | 4 | 37 | 32 | rmsprop | 0.17 | 3 |
|  | 3 | 5 | 16 | 44 | 32 | adam | 0.07 | 1 |
|  | 4 | 5 | 16 | 33 | 32 | adam | 0.03 | 2 |
| graylog2-server | 1 | 4 | 32 | 31 | 32 | rmsprop | 0.03 | 1 |
|  | 2 | 5 | 32 | 48 | 32 | adam | 0.19 | 1 |
|  | 3 | 6 | 8 | 59 | 32 | rmsprop | 0.09 | 1 |
|  | 4 | 6 | 16 | 50 | 32 | rmsprop | 0.03 | 3 |
| jackrabbit-oak | 1 | 4 | 4 | 32 | 32 | rmsprop | 0.03 | 4 |
|  | 2 | 5 | 32 | 55 | 32 | rmsprop | 0.1 | 4 |
|  | 3 | 6 | 8 | 44 | 32 | adam | 0.12 | 4 |
|  | 4 | 4 | 8 | 57 | 32 | adam | 0.19 | 3 |
| jruby | 1 | 6 | 4 | 44 | 32 | adam | 0.05 | 2 |
|  | 2 | 5 | 32 | 42 | 32 | adam | 0.06 | 4 |
|  | 3 | 5 | 4 | 49 | 32 | adam | 0.07 | 4 |
|  | 4 | 6 | 32 | 45 | 32 | adam | 0.2 | 3 |
| metasploit-framework | 1 | 5 | 8 | 59 | 32 | adam | 0.1 | 1 |
|  | 2 | 4 | 4 | 30 | 32 | adam | 0.04 | 1 |
|  | 3 | 6 | 16 | 60 | 32 | rmsprop | 0.07 | 1 |
|  | 4 | 6 | 32 | 46 | 32 | rmsprop | 0.17 | 1 |
| open-build-service | 1 | 4 | 8 | 54 | 32 | adam | 0.02 | 1 |
|  | 2 | 6 | 4 | 33 | 32 | rmsprop | 0.17 | 3 |
|  | 3 | 4 | 8 | 37 | 32 | rmsprop | 0.07 | 1 |
|  | 4 | 6 | 8 | 42 | 32 | rmsprop | 0.06 | 3 |
| openproject | 1 | 6 | 8 | 54 | 32 | rmsprop | 0.08 | 1 |
|  | 2 | 4 | 16 | 35 | 32 | rmsprop | 0.1 | 2 |
|  | 3 | 6 | 16 | 36 | 32 | rmsprop | 0.07 | 1 |
|  | 4 | 6 | 8 | 57 | 32 | rmsprop | 0.11 | 4 |
| rails | 1 | 6 | 16 | 59 | 32 | adam | 0.14 | 4 |
|  | 2 | 6 | 4 | 51 | 32 | adam | 0.16 | 4 |
|  | 3 | 5 | 8 | 35 | 32 | adam | 0.01 | 4 |
|  | 4 | 6 | 8 | 38 | 32 | adam | 0.01 | 4 |
| ruby | 1 | 5 | 4 | 51 | 32 | adam | 0.19 | 1 |
|  | 2 | 5 | 4 | 39 | 32 | rmsprop | 0.09 | 4 |
|  | 3 | 5 | 4 | 42 | 32 | rmsprop | 0.2 | 3 |
|  | 4 | 5 | 8 | 53 | 32 | adam | 0.14 | 2 |
| sonarqube | 1 | 6 | 32 | 39 | 32 | rmsprop | 0.1 | 1 |
|  | 2 | 6 | 16 | 32 | 32 | adam | 0.14 | 1 |
|  | 3 | 6 | 32 | 48 | 32 | rmsprop | 0.04 | 3 |
|  | 4 | 6 | 4 | 56 | 32 | rmsprop | 0.1 | 1 |

Table-A I-6 Optimal Parameters for RQ5 (recent data)

| Project | Experiment | nb_epochs | nb_batch | time_step | nb_units | optimizer | drop_proba | nb_layers |
|---|---|---|---|---|---|---|---|---|
| cloudify | 1 | 5 | 8 | 52 | 32 | rmsprop | 0.1 | 1 |
| | 2 | 4 | 8 | 38 | 32 | adam | 0.19 | 4 |
| | 3 | 5 | 32 | 45 | 32 | rmsprop | 0.04 | 1 |
| | 4 | 4 | 4 | 41 | 32 | rmsprop | 0.17 | 4 |
| graylog2-server | 1 | 5 | 8 | 60 | 32 | adam | 0.05 | 2 |
| | 2 | 6 | 4 | 39 | 32 | adam | 0.09 | 2 |
| | 3 | 5 | 16 | 32 | 32 | adam | 0.11 | 1 |
| | 4 | 5 | 8 | 60 | 32 | adam | 0.05 | 2 |
| jackrabbit-oak | 1 | 4 | 32 | 43 | 32 | rmsprop | 0.2 | 1 |
| | 2 | 4 | 16 | 51 | 32 | adam | 0.02 | 4 |
| | 3 | 4 | 32 | 43 | 32 | rmsprop | 0.2 | 1 |
| | 4 | 4 | 16 | 51 | 32 | adam | 0.02 | 4 |
| jruby | 1 | 4 | 8 | 43 | 32 | adam | 0.1 | 2 |
| | 2 | 6 | 8 | 59 | 32 | adam | 0.17 | 4 |
| | 3 | 5 | 4 | 30 | 32 | adam | 0.04 | 2 |
| | 4 | 4 | 8 | 43 | 32 | adam | 0.1 | 2 |
| metasploit-framework | 1 | 6 | 8 | 52 | 32 | rmsprop | 0.1 | 2 |
| | 2 | 4 | 4 | 37 | 32 | rmsprop | 0.02 | 1 |
| | 3 | 6 | 4 | 41 | 32 | rmsprop | 0.14 | 3 |
| | 4 | 4 | 32 | 54 | 32 | rmsprop | 0.19 | 2 |
| open-build-service | 1 | 4 | 4 | 32 | 32 | rmsprop | 0.13 | 3 |
| | 2 | 6 | 32 | 35 | 32 | adam | 0.15 | 4 |
| | 3 | 4 | 4 | 32 | 32 | rmsprop | 0.13 | 3 |
| | 4 | 6 | 64 | 38 | 32 | rmsprop | 0.19 | 2 |
| openproject | 1 | 6 | 8 | 45 | 32 | adam | 0.1 | 1 |
| | 2 | 6 | 16 | 47 | 32 | rmsprop | 0.03 | 3 |
| | 3 | 6 | 32 | 37 | 32 | adam | 0.04 | 2 |
| | 4 | 6 | 32 | 37 | 32 | adam | 0.04 | 2 |
| rails | 1 | 4 | 16 | 32 | 32 | adam | 0.03 | 2 |
| | 2 | 6 | 32 | 31 | 32 | rmsprop | 0.02 | 4 |
| | 3 | 6 | 32 | 31 | 32 | rmsprop | 0.02 | 4 |
| | 4 | 4 | 16 | 32 | 32 | adam | 0.03 | 2 |
| ruby | 1 | 4 | 8 | 51 | 32 | rmsprop | 0.2 | 3 |
| | 2 | 6 | 32 | 33 | 32 | rmsprop | 0.14 | 3 |
| | 3 | 5 | 4 | 46 | 32 | rmsprop | 0.02 | 1 |
| | 4 | 4 | 8 | 51 | 32 | rmsprop | 0.2 | 3 |
| sonarqube | 1 | 5 | 16 | 42 | 32 | rmsprop | 0.03 | 1 |
| | 2 | 6 | 8 | 30 | 32 | rmsprop | 0.18 | 4 |
| | 3 | 5 | 8 | 42 | 32 | adam | 0.18 | 4 |
| | 4 | 5 | 16 | 42 | 32 | rmsprop | 0.03 | 1 |

# BIBLIOGRAPHY

Abdalkareem, R., Mujahid, S., Shihab, E. & Rilling, J. (2019). Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering*.

Abdalkareem, R., Mujahid, S. & Shihab, E. (2020). A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering*.

Abdellatif, A., Costa, D., Badran, K., Abdalkareem, R. & Shihab, E. (2020). Challenges in chatbot development: A study of stack overflow posts. *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 174–185.

Agrawal, A. & Menzies, T. (2018). Is" Better Data" Better Than" Better Data Miners"? *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1050–1061.

Alizadeh, V., Ouali, M. A., Kessentini, M. & Chater, M. (2019). RefBot: intelligent software refactoring bot. *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 823–834.

Almarimi, N., Ouni, A., Bouktif, S., Mkaouer, M. W., Kula, R. G. & Saied, M. A. (2019). Web service API recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing*, 85, 105830.

AlOmar, E. A., Mkaouer, M. W., Ouni, A. & Kessentini, M. (2019). On the impact of refactoring on the relationship between quality attributes and design metrics. *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11.

AlOmar, E. A., Barinas, D., Liu, J., Mkaouer, M. W., Ouni, A. & Newman, C. (2020). An exploratory study on how software reuse is discussed in stack overflow. *International Conference on Software and Software Reuse*, pp. 292–303.

Arcuri, A. & Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. *33rd International Conference on Software Engineering (ICSE)*, pp. 1–10.

Arcuri, A. & Fraser, G. (2011). On parameter tuning in search based software engineering. *International Symposium on Search Based Software Engineering*, pp. 33–47.

Atchison, A., Berardi, C., Best, N., Stevens, E. & Linstead, E. (2017). A time series analysis of TravisTorrent builds: to everything there is a season. *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 463–466.

Athiwaratkun, B. & Stokes, J. W. (2017). Malware classification with LSTM and GRU language models and a character-level CNN. *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2482–2486.

Baltes, S., Dumani, L., Treude, C. & Diehl, S. (2018). Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts. *Proceedings of the 15th international conference on mining software repositories*, pp. 319–330.

Barua, A., Thomas, S. W. & Hassan, A. E. (2014). What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3), 619–654.

Bavota, G., De Lucia, A., Marcus, A., Oliveto, R. & Palomba, F. (2012). Supporting extract class refactoring in eclipse: The aries project. *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1419–1422.

Beller, M., Gousios, G. & Zaidman, A. (2017). TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 447-450.

Beller, M., Gousios, G. & Zaidman, A. (2017). Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. *IEEE/ACM International Conference on Mining Software Repositories*, pp. 356–367.

Bennin, K. E., Keung, J., Phannachitta, P., Monden, A. & Mensah, S. (2017). Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. *IEEE Transactions on Software Engineering*, 44(6), 534–550.

Bergstra, J. & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb), 281–305.

Bernardo, J. H., da Costa, D. A. & Kulesza, U. (2018). Studying the impact of adopting continuous integration on the delivery time of pull requests. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 131–141.

Bhowan, U., Johnston, M. & Zhang, M. (2009). Differentiating between individual class performance in genetic programming fitness for classification with unbalanced data. *2009 IEEE Congress on Evolutionary Computation*, pp. 2802–2809.

Bhowan, U., Zhang, M. & Johnston, M. (2010). Genetic programming for classification with unbalanced data. *European Conference on Genetic Programming*, pp. 1–13.

Bhowan, U., Johnston, M. & Zhang, M. (2011). Evolving ensembles in multi-objective genetic programming for classification with unbalanced data. *Annual conference on Genetic and evolutionary computation (GECCO)*, pp. 1331–1338.

Bhowan, U., Johnston, M., Zhang, M. & Yao, X. (2012). Evolving diverse ensembles using genetic programming for classification with unbalanced data. *IEEE Transactions on Evolutionary Computation*, 17(3), 368–386.

Bhowan, U., Johnston, M., Zhang, M. & Yao, X. (2013). Reusing genetic programming for ensemble selection in classification of unbalanced data. *IEEE Transactions on Evolutionary Computation*, 18(6), 893–908.

Bishop, C. Pattern Recognition and Machine Learning (Information Science and Statistics)(Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006).

Blei, D. M., Ng, A. Y. & Jordan, M. I. (2003). Latent dirichlet allocation. *the Journal of machine Learning research*, 3, 993–1022.

Boukharata, S., Ouni, A., Kessentini, M., Bouktif, S. & Wang, H. (2019). Improving web service interfaces modularity using multi-objective optimization. *Automated Software Engineering*, 26(2), 275–312.

Bouktif, S., Fiaz, A., Ouni, A. & Serhani, M. A. (2018). Optimal deep learning lstm model for electric load forecasting using feature selection and genetic algorithm: Comparison with machine learning approaches. *Energies*, 11(7), 1636.

Bouktif, S., Fiaz, A., Ouni, A. & Serhani, M. A. (2020). Multi-Sequence LSTM-RNN Deep Learning and Metaheuristics for Electric Load Forecasting. *Energies*, 13(2), 391.

Brandtner, M., Giger, E. & Gall, H. (2014). Supporting continuous integration by mashing-up software quality information. *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 184–193.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1), 5–32.

Buda, M., Maki, A. & Mazurowski, M. A. (2018). A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks*, 106, 249–259.

Cervantes, J., Li, X. & Yu, W. (2013). Using genetic algorithm to improve classification accuracy on imbalanced data. *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 2659–2664.

Chawla, N. V., Bowyer, K. W., Hall, L. O. & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16, 321–357.

Chen, J., Nair, V., Krishna, R. & Menzies, T. (2018). "Sampling" as a baseline optimizer for search-based software engineering. *IEEE Transactions on Software Engineering*, 45(6), 597–614.

Chen, L. & Babar, M. A. (2014). Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development. *2014 IEEE/IFIP Conference on Software Architecture*, pp. 195–204.

Chen, T., Li, M. & Yao, X. (2020). How to Evaluate Solutions in Pareto-based Search-Based Software Engineering? A Critical Review and Methodological Guidance. *arXiv preprint arXiv:2002.09040*.

Choetkiertikul, M., Dam, H. K., Tran, T., Pham, T., Ghose, A. & Menzies, T. (2018). A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, 45(7), 637–656.

CI, T. [Accessed: 2019-03-01]. (2021). Travis CI Web site [Format]. Retrieved from: https://travis-ci.org/.

Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3), 494.

Cohen, P., West, S. G. & Aiken, L. S. (2014). *Applied multiple regression/correlation analysis for the behavioral sciences*. Psychology Press.

colah's blog. [Accessed: 2020-03-01]. (2020). Understanding LSTM Networks. Retrieved from: url={http://colah.github.io/posts/2015-08-Understanding-LSTMs/},.

Collell, G., Prelec, D. & Patil, K. R. (2018). A simple plug-in bagging ensemble based on threshold-moving for classifying binary and multiclass imbalanced data. *Neurocomputing*, 275, 330–340.

Colomo-Palacios, R., Fernandes, E., Soto-Acosta, P. & Larrucea, X. (2018). A case analysis of enabling continuous software deployment through knowledge management. *International Journal of Information Management*, 40, 186–189.

Cook, T. D. (81). Campbell, D, T.(1979). Quasi-experimentation: Design and Analysis Issues for Field Settings. *Boston Houghtori Mitfliri*.

Cui, Y., Wang, S. & Li, J. (2015). LSTM neural reordering feature for statistical machine translation. *arXiv preprint arXiv:1512.00177*.

Davis, L. (1991). Handbook of genetic algorithms.

Deb, K. & Jain, H. (2013). An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4), 577–601.

Deb, K., Pratap, A., Agarwal, S. & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. 6(2), 182–197.

Debbiche, A., Dienér, M. & Svensson, R. B. (2014). Challenges when adopting continuous integration: A case study. *International Conference on Product-Focused Software Process Improvement*, pp. 17–32.

di Pierro, F., Khu, S.-T. & Savic, D. A. (2007). An investigation on preference order ranking scheme for multiobjective evolutionary optimization. *IEEE Transactions on Evolutionary Computation*, 11(1), 17–45.

Duvall, P., Matyas, S. M. & Glover, A. (2007a). *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional.

Duvall, P. M., Matyas, S. & Glover, A. (2007b). *Continuous integration: improving software quality and reducing risk*. Pearson Education.

Eckart, Z., Marco, L. & Lothar, T. (2001). Improving the strength Pareto evolutionary algorithm for multiobjective optimi-zation. *EUROGEN, Evol. Method Des. Optim. Control Ind. Problem*, 1–21.

Edwards, A. L. (1985). *Multiple regression and the analysis of variance and covariance*. WH Freeman/Times Books/Henry Holt & Co.

Ekambaram, R., Goldgof, D. B. & Hall, L. O. (2017). Finding label noise examples in large scale datasets. *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2420–2424.

Ekanayake, J., Tappolet, J., Gall, H. C. & Bernstein, A. (2009). Tracking concept drift of software projects using defect prediction quality. *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 51–60.

Elazhary, O., Werner, C., Li, Z. S., Lowlind, D., Ernst, N. A. & Storey, M.-A. (2021). Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering*.

Falkner, S., Klein, A. & Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. *International Conference on Machine Learning*, pp. 1437–1446.

Ferrucci, F., Harman, M., Ren, J. & Sarro, F. (2013). Not going to take this anymore: multi-objective overtime planning for software engineering projects. *2013 35th International Conference on Software Engineering (ICSE)*, pp. 462–471.

Fokaefs, M., Tsantalis, N., Stroulia, E. & Chatzigeorgiou, A. (2011). JDeodorant: identification and application of extract class refactorings. *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1037–1039.

Fowler, M. [Accessed: 2020-01-01]. (2006). Continuous Integration. Retrieved from: url={https://www.martinfowler.com/articles/continuousIntegration.html},.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, d. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Frénay, B. & Verleysen, M. (2013). Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5), 845–869.

Gallaba, K., Macho, C., Pinzger, M. & McIntosh, S. (2018). Noise and heterogeneity in historical build data: an empirical study of Travis CI. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 87–97.

Garcia, S. & Trinh, C. T. (2019). Comparison of Multi-objective Evolutionary Algorithms to Solve the Modular Cell Design Problem for Novel Biocatalysis. *Processes*, 7(6), 361.

Ghaleb, T. A., da Costa, D. A. & Zou, Y. (2019a). An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 1–38.

Ghaleb, T. A., da Costa, D. A., Zou, Y. & Hassan, A. E. (2019b). Studying the Impact of Noises in Build Breakage Data. *IEEE Transactions on Software Engineering*.

Ghotra, B., McIntosh, S. & Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1, 789–800.

Goldberg, D. E. (1989). Genetic algorithms in search. *Optimization, and MachineLearning*.

Graves, A. & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural networks*, 18(5-6), 602–610.

Graves, A., Jaitly, N. & Mohamed, A.-r. (2013). Hybrid speech recognition with deep bidirectional LSTM. *2013 IEEE workshop on automatic speech recognition and understanding*, pp. 273–278.

Gruhn, V., Hannebauer, C. & John, C. (2013). Security of public continuous integration services. *Proceedings of the 9th International Symposium on open collaboration*, pp. 1–10.

Guo, B., Hu, J., Wu, W., Peng, Q. & Wu, F. (2019). The tabu_genetic algorithm: a novel method for hyper-parameter optimization of learning algorithms. *Electronics*, 8(5), 579.

Gupta, S. & Gupta, A. (2019). Dealing with noise problem in machine learning data-sets: A systematic review. *Procedia Computer Science*, 161, 466–474.

Gupta, Y., Khan, Y., Gallaba, K. & McIntosh, S. (2017). The impact of the adoption of continuous integration on developer attraction and retention. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 491–494.

Habchi, S., Rouvoy, R. & Moha, N. (2019). On the Survival of Android Code Smells in the Wild.

Hadka, D. [Accessed: 2020-01-01]. (2013). MOEA Framework. Retrieved from: url={http://moeaframework.org/},.

Hadka, D. (2014). MOEA framework user guide.

Harman, M. (2007). The current state and future of search based software engineering. 342–357.

Harman, M. & Clark, J. (2004). Metrics are fitness functions too. *10th International Symposium on Software Metrics*, pp. 58–69.

Harman, M. & Jones, B. F. (2001). Search-based software engineering. *Information and software Technology*, 43(14), 833–839.

Harman, M., McMinn, P., De Souza, J. T. & Yoo, S. (2010). Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification* (pp. 1–59).

Harman, M., Mansouri, S. A. & Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1), 11.

Hashimi, S. Y. & Hashimi, S. I. (2006). The Unified Build Engine: MSBuild. *Deploying. NET Applications: Learning MSBuild and ClickOnce*, 21–43.

Hassan, F. & Wang, X. (2017). Change-aware build prediction model for stall avoidance in continuous integration. *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 157–162.

Hastie, T., Tibshirani, R. & Friedman, J. (2009). *The elements of statistical learning: data mining, inference, and prediction.* Springer Science & Business Media.

Hilton, M., Tunnell, T., Huang, K., Marinov, D. & Dig, D. (2016). Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. *31st IEEE/ACM International Conference on Automated Software Engineering*, (ASE 2016), 426–437.

Hilton, M., Nelson, N., Tunnell, T., Marinov, D. & Dig, D. (2017). Trade-offs in continuous integration: assurance, security, and flexibility. *11th Joint Meeting on Foundations of Software Engineering*, pp. 197–207.

Hochreiter, S. & Schmidhuber, J. (1997a). Long short-term memory. *Neural computation*, 9(8), 1735–1780.

Hochreiter, S. & Schmidhuber, J. (1997b). LSTM can solve hard long time lag problems. *Advances in neural information processing systems*, pp. 473–479.

HpBandSter. [Accessed: 2021-12-16]. (2021a). HpBandSter Web site [Format]. Retrieved from: https://automl.github.io/HpBandSter/build/html/index.html.

HpBandSter. [Accessed: 2021-12-16]. (2021b). Hyperopt Web site [Format]. Retrieved from: http://hyperopt.github.io/hyperopt/.

Hsu, C.-W., Chang, C.-C., Lin, C.-J. et al. (2003). A practical guide to support vector classification. Taipei.

Imbens, G. W. & Lemieux, T. (2008). Regression discontinuity designs: A guide to practice. *Journal of econometrics*, 142(2), 615–635.

Janssen, K., Moons, K., Kalkman, C., Grobbee, D. & Vergouwe, Y. (2008). Updating methods improved the performance of a clinical prediction model in new patients. *Journal of clinical epidemiology*, 61(1), 76–86.

Jebnoun, H., Braiek, H. B., Rahman, M. M. & Khomh, F. (2020). The Scent of Deep Learning Code: An Empirical Study.

Jelihovschi, E. G., Faria, J. C. & Allaman, I. B. (2014). ScottKnott: a package for performing the Scott-Knott clustering algorithm in R. *TEMA (São Carlos)*, 15(1), 3–17.

Jenkins. (2019). Jenkins Web site [Format]. Retrieved from: https://jenkins.io/.

Jin, Y. & Sendhoff, B. (2008). Pareto-based multiobjective machine learning: An overview and case studies. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(3), 397–415.

John, G. H. & Langley, P. (2013). Estimating continuous distributions in Bayesian classifiers. *arXiv preprint arXiv:1302.4964*.

Jruby. [Accessed: 2021-12-16]. (2019). Jruby project [Format]. Retrieved from: https://github.com/jruby/jruby/.

Karnopp, D. C. (1963). Random search techniques for optimization problems. *Automatica*, 1(2-3), 111–121.

Karpathy, A., Johnson, J. & Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*.

Kessentini, M. & Ouni, A. (2017). Detecting android smells using multi-objective genetic programming. *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pp. 122–132.

Koza, J. R. & Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. MIT press.

Krawczyk, B. & Woźniak, M. (2015). Cost-sensitive neural network with roc-based moving threshold for imbalanced classification. *International Conference on Intelligent Data Engineering and Automated Learning*, pp. 45–52.

Krishna, R., Menzies, T. & Fu, W. (2016). Too much automation? The bellwether effect and its implications for transfer learning. *31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 122–131.

Lambiase, S., Cupito, A., Pecorelli, F., De Lucia, A. & Palomba, F. (2020). Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. *28th International Conference on Program Comprehension*, pp. 441–445.

Längkvist, M., Karlsson, L. & Loutfi, A. (2014). A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42, 11–24.

Laukkanen, E., Paasivaara, M. & Arvonen, T. (2015). Stakeholder perceptions of the adoption of continuous integration–a case study. *2015 agile conference*, pp. 11–20.

Laukkanen, E., Itkonen, J. & Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, 82, 55–79.

Lehman, M. M. (1996). Laws of software evolution revisited. *European Workshop on Software Process Technology*, pp. 108–124.

Li, H., Gong, X.-J., Yu, H. & Zhou, C. (2018). Deep neural network based predictions of protein interactions using primary sequences. *Molecules*, 23(8), 1923.

Li, M., Zhang, H., Wu, R. & Zhou, Z.-H. (2012). Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2), 201–230.

Li, M. & Yao, X. (2019). Quality evaluation of solution sets in multiobjective optimisation: A survey. *ACM Computing Surveys (CSUR)*, 52(2), 1–38.

Li, Y., Dong, M., Wang, Y. & Xu, C. (2020). Neural architecture search in a proxy validation loss landscape. *International Conference on Machine Learning*, pp. 5853–5862.

Lorenzo, P. R., Nalepa, J., Kawulok, M., Ramos, L. S. & Pastor, J. R. (2017). Particle swarm optimization for hyper-parameter selection in deep neural networks. *Proceedings of the genetic and evolutionary computation conference*, pp. 481–488.

Luo, Y., Zhao, Y., Ma, W. & Chen, L. (2017). What are the Factors Impacting Build Breakage? *14th Web Information Systems and Applications Conference (WISA)*, pp. 139–142.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27, 504–518.

Malhotra, R. & Khanna, M. (2017). An exploratory study for software change prediction in object-oriented systems using hybridized techniques. *Automated Software Engineering*, 24(3), 673–717.

Meng, H.-y., Zhang, X.-h. & Liu, S.-y. (2005). New quality measures for multiobjective programming. *International Conference on Natural Computation*, pp. 1044–1048.

Mens, T. & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2), 126–139.

Menzies, T., Greenwald, J. & Frank, A. (2006). Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1), 2–13.

Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K. & Ouni, A. (2015). Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3), 17.

Murphy-Hill, E., Parnin, C. & Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering (TSE)*, 38(1), 5–18.

Nam, J., Fu, W., Kim, S., Menzies, T. & Tan, L. (2017). Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9), 874–896.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E. & Dig, D. (2012). *Using Continuous Code Change Analysis to Understand the Practice of Refactoring*.

Nejati, S. & Gay, G. (2019). *11th International Symposium Search-Based Software Engineering*.

Ni, A. & Li, M. (2017). Cost-effective build outcome prediction using cascaded classifiers. *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 455–458.

Ni, A. & Li, M. (2018). Poster: ACONA: Active Online Model Adaptation for Predicting Continuous Integration Build Failures. *IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 366–367.

Openja, M., Adams, B. & Khomh, F. (2020). Analysis of Modern Release Engineering Topics:– A Large-Scale Study using StackOverflow–. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 104–114.

Optunity. [Accessed: 2021-12-16]. (2019). Optunity framework [Format]. Retrieved from: https://optunity.readthedocs.io/en/latest/.

Ordóñez, F. J. & Roggen, D. (2016). Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors*, 16(1), 115.

Ouni, A. (2020). Search-based Software Engineering: Challenges, Opportunities and Recent Applications. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1114–1146.

Ouni, A., Kessentini, M., Sahraoui, H. & Hamdi, M. S. (2012). Search-based refactoring: Towards semantics preservation. *IEEE International Conference on Software Maintenance (ICSM)*, pp. 347–356.

Ouni, A., Kessentini, M., Sahraoui, H. & Boukadoum, M. (2013). Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1), 47–79.

Ouni, A., Kessentini, M., Inoue, K. & Cinnéide, M. O. (2015). Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, 10(4), 603–617.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K. & Deb, K. (2016). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3), 23.

Ouni, A., Kessentini, M., Inoue, K. & Cinnéide, M. O. (2017). Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, 10(4), 603–617.

Paixao, M., Harman, M., Zhang, Y. & Yu, Y. (2017). An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Transactions on Evolutionary Computation*, 22(3), 394–414.

Palomba, F., Panichella, A., Zaidman, A., Oliveto, R. & De Lucia, A. (2017). The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, 44(10), 977–1000.

Pantiuchina, J., Bavota, G., Tufano, M. & Poshyvanyk, D. (2018). Towards just-in-time refactoring recommenders. *IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 312–3123.

Pascanu, R., Mikolov, T. & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. *International conference on machine learning*, pp. 1310–1318.

Peruma, A., Simmons, S., AlOmar, E. A., Newman, C. D., Mkaouer, M. W. & Ouni, A. (2022). How do i refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empirical Software Engineering*, 27(1), 1–43.

Pinto, G., Castor, F., Bonifacio, R. & Rebouças, M. (2018). Work practices and challenges in continuous integration: A survey with Travis CI users. *Software: Practice and Experience*, 48(12), 2223–2236.

Provost, F. (2000). Machine learning from imbalanced data sets 101. *Proceedings of the AAAI'2000 workshop on imbalanced data sets*, 68(2000), 1–3.

Quinlan, J. R. (2014). *C4. 5: programs for machine learning*. Elsevier.

Rahman, A., Agrawal, A., Krishna, R. & Sobran, A. (2018). Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects. *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, pp. 8–14.

Rausch, T., Hummer, W., Leitner, P. & Schulte, S. (2017). An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. *14th international conference on mining software repositories*, pp. 345–355.

Riquelme, N., Von Lücken, C. & Baran, B. (2015). Performance metrics in multi-objective optimization. *Latin American Computing Conference (CLEI)*, pp. 1–11.

Röder, M., Both, A. & Hinneburg, A. (2015). Exploring the space of topic coherence measures. *Proceedings of the eighth ACM international conference on Web search and data mining*, pp. 399–408.

Romano, J., Kromrey, J. D., Coraggio, J. & Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. *Annual meeting of the Florida Association of Institutional Research*, pp. 1–33.

Saidani, I. (2020a). Replication Package of the paper "On the Impact of Continuous Integration on Refactoring Practice: An Exploratory Study on TravisTorrent". Retrieved from: Availableat:url={https://github.com/ci-ref/replication-package}.

Saidani, I. (2020b). Replication Package for CI build prediction. Retrieved from: url={https://github.com/stilab-ets/DL-CIBuild}.

Saidani, I. (2020c). Replication Package for the paper "Improving the Prediction of Continuous Integration Build Failures Using Deep Learning". Retrieved from: Availableat:url={https://github.com/stilab-ets/DL-CIBuild}.

Saidani, I. (2020d). Dataset for the paper "Predicting Continuous Integration Build Failures Using Evolutionary Search". Retrieved from: Availableat:url={https://github.com/GP-CI-Build-Fail/replication-package}.

Saidani, I. (2021). Replication package for the paper "An Empirical Study on Continuous Integration Trends, Topics and Challenges in Stack Overflow". Retrieved from: availableaturl= {https://github.com/stilab-ets/CISO}.

Saidani, I. & Ouni, A. (2021). Toward a Smell-aware Prediction Model for CI Build Failures. *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pp. 18–25.

Saidani, I., Ouni, A., Chouchen, M. & Mkaouer, M. W. (2020a). Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128, 106392.

Saidani, I., Ouni, A., Chouchen, M. & Mkaouer, M. W. (2020b). Predicting Continuous Integration Build Failures Using Evolutionary Search. *Journal of Information and Software Technology*, 128.

Saidani, I., Ouni, A., Chouchen, M. & Mkaouer, M. W. (2020c). On the prediction of continuous integration build failures using search-based software engineering. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 313–314.

Saidani, I., Ouni, A., Chouchen, M. & Mkaouer, M. W. (2021a). BF-detector: an automated tool for CI build failure detection. *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1530–1534.

Saidani, I., Ouni, A., Mkaouer, M. W. & Palomba, F. (2021b). On the impact of Continuous Integration on refactoring practice: An exploratory study on TravisTorrent. *Information and Software Technology*, 138, 106618.

Saidani, I., Ouni, A. & Mkaouer, W. (2021c). Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*.

Saidani, I., Ouni, A. & Mkaouer, M. W. (2022). Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1), 1–61.

Santolucito, M., Zhang, J., Zhai, E. & Piskac, R. (2018). Statically Verifying Continuous Integration Configurations. *Technical Report*.

Scalabrino, S., Grano, G., Di Nucci, D., Oliveto, R. & De Lucia, A. (2016). Search-based testing of procedural programs: Iterative single-target or multi-target approach? *International Symposium on Search Based Software Engineering*, pp. 64–79.

Schapire, R. E. (2013). Explaining adaboost. In *Empirical inference* (pp. 37–52). Springer.

Schermann, G., Cito, J., Leitner, P. & Gall, H. C. (2016). Towards quality gates in continuous delivery and deployment. *24th international conference on program comprehension (ICPC)*, pp. 1–4.

Scikit-learn.org. [Accessed: 2020-04-01]. (2006). Parameter Estimation Using Grid Search with Scikit-Learn. Available online:. Retrieved from: url={https://scikit-learn.org/stable/modules/grid_search.html},.

Shahin, M., Babar, M. A. & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5, 3909–3943.

Shan, Y., Hoens, T. R., Jiao, J., Wang, H., Yu, D. & Mao, J. (2016). Deep crossing: Web-scale modeling without manually crafted combinatorial features. *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 255–262.

Shi, Y. & Eberhart, R. C. (1998). Parameter selection in particle swarm optimization. *International conference on evolutionary programming*, pp. 591–600.

Shrikanth, N., Majumder, S. & Menzies, T. (2020). Early Life Cycle Software Defect Prediction. Why? How? *arXiv preprint arXiv:2011.13071*.

Silva, D. & Valente, M. T. (2017). RefDiff: detecting refactorings in version histories. *14th International Conference on Mining Software Repositories*, pp. 269–279.

Silva, D., Tsantalis, N. & Valente, M. T. (2016). Why We Refactor? Confessions of GitHub Contributors. *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (FSE 2016), 858–870.

Singh, A., Walenstein, A. & Lakhotia, A. (2012). Tracking concept drift in malware families. *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pp. 81–92.

Sofianopoulos, S. & Tambouratzis, G. (2011). Studying the spea2 algorithm for optimising a pattern-recognition based machine translation system. *IEEE Symposium on Computational Intelligence in Multicriteria Decision-Making (MDCM)*, pp. 97–104.

Stamelos, I. G. & Sfetsos, P. (2007). *Agile software development quality assurance*. Igi Global.

Steinmacher, I., Wiese, I., Chaves, A. P. & Gerosa, M. A. (2013). Why do newcomers abandon open source software projects? *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 25–32.

Steinmacher, I., Conte, T., Gerosa, M. A. & Redmiles, D. (2015). Social barriers faced by newcomers placing their first contribution in open source software projects. *18th ACM conference on Computer supported cooperative work & social computing*, pp. 1379–1392.

Ståhl, D., Mårtensson, T. & Bosch, J. (2017). The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software*, 127, 150 - 167.

Sundermeyer, M., Schlüter, R. & Ney, H. (2012). LSTM neural networks for language modeling. *Thirteenth annual conference of the international speech communication association*.

Sundsøy, P., Bjelland, J., Reme, B.-A., Iqbal, A. M. & Jahani, E. (2016). Deep learning applied to mobile phone data for individual income classification. *Proceedings of the 2016 International Conference on Artificial Intelligence: Technologies and Applications, Bangkok, Thailand*, pp. 24–25.

Szóke, G., Antal, G., Nagy, C., Ferenc, R. & Gyimóthy, T. (2014). Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 95–104.

Tan, L. & Bockuisch, C. A Survey of Refactoring Detection Tools.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2017). An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. (1).

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2018a). The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7), 683–711.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2018b). The Impact of Automated Parameter Optimization for Defect Prediction Models.

Tharwat, A. & Hassanien, A. E. (2019). Quantum-behaved particle swarm optimization for parameter optimization of support vector machine. *Journal of Classification*, 36(3), 576–598.

Thomas, S. W., Hemmati, H., Hassan, A. E. & Blostein, D. (2014). Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1), 182–212.

Tsantalis, N., Guana, V., Stroulia, E. & Hindle, A. (2013). A multidimensional empirical study on refactoring activity. *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 132–146.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D. & Dig, D. (2018a). Accurate and Efficient Refactoring Detection in Commit History. *40th International Conference on Software Engineering*, (ICSE '18), 483–494. doi: 10.1145/3180155.3180206.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D. & Dig, D. (2018b). Accurate and efficient refactoring detection in commit history. *40th International Conference on Software Engineering*, pp. 483–494.

Tsymbal, A. (2004). The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2), 58.

Ujihara, N., Ouni, A., Ishio, T. & Inoue, K. (2017). c-JRefRec: Change-based identification of Move Method refactoring opportunities. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 482–486.

Vargha, A. & Delaney, H. D. (2000). A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2), 101–132.

Vasilescu, B., Yu, Y., Wang, H., Devanbu, P. & Filkov, V. (2015). Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. *10th Joint Meeting on Foundations of Software Engineering*, (ESEC/FSE 2015), 805–816.

Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Di Penta, M. & Panichella, S. (2017). A tale of CI build failures: An open source and a financial organization perspective. *IEEE international conference on software maintenance and evolution (ICSME)*, pp. 183–193.

Vassallo, C., Palomba, F., Bacchelli, A. & Gall, H. C. (2018a). Continuous Code Quality: Are We (Really) Doing That? *33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 790–795.

Vassallo, C., Palomba, F. & Gall, H. C. (2018b). Continuous Refactoring in CI: A Preliminary Study on the Perceived Advantages and Barriers. *IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pp. 564–568. doi: 10.1109/ICSME.2018.00068.

Vassallo, C., Grano, G., Palomba, F., Gall, H. C. & Bacchelli, A. (2019a). A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180, 1–15.

Vassallo, C., Proksch, S., Gall, H. C. & Di Penta, M. (2019b). Automated reporting of anti-patterns and decay in continuous integration. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 105–115.

Wang, S., Ali, S., Yue, T., Li, Y. & Liaaen, M. (2016a). A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. *38th International Conference on Software Engineering*, pp. 631–642.

Wang, Y., Huang, M., Zhu, X. & Zhao, L. (2016b). Attention-based LSTM for aspect-level sentiment classification. *Proceedings of the 2016 conference on empirical methods in natural language processing*, pp. 606–615.

Wang, Y. (2009). What motivate software engineers to refactor source code? evidences from professional developers. *2009 ieee international conference on software maintenance*, pp. 413–416.

Wedyan, F., Alrmuny, D. & Bieman, J. M. (2009). The effectiveness of automated static analysis tools for fault detection and refactoring prediction. *International Conference on Software Testing Verification and Validation*, pp. 141–150.

Wicaksono, A. S. & Supianto, A. A. (2018). Hyper parameter optimization using genetic algorithm on machine learning methods for online news popularity prediction. *Int. J. Adv. Comput. Sci. Appl*, 9(12), 263–267.

Widder, D., Vasilescu, B., Hilton, M. & Kästner, C. (2018). I'm leaving you, Travis: a continuous integration breakup story. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 165–169.

Widder, D. G., Hilton, M., Kästner, C. & Vasilescu, B. (2019). A conceptual replication of continuous integration pain points in the context of Travis CI. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 647–658.

Widmer, G. & Kubat, M. (1996). Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1), 69–101.

Wilcoxon, F., Katti, S. & Wilcox, R. A. (1970). Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Selected tables in mathematical statistics*, 1, 171–259.

Wyrich, M. & Bogner, J. (2019). Towards an autonomous bot for automatic source code refactoring. *IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pp. 24–28.

Xia, J. & Li, Y. (2017). Could we predict the result of a continuous integration build? An empirical study. *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 311–315.

Xia, J., Li, Y. & Wang, C. (2017a). An Empirical Study on the Cross-Project Predictability of Continuous Integration Outcomes. *14th Web Information Systems and Applications Conference (WISA)*, pp. 234–239.

Xia, Y., Liu, C., Li, Y. & Liu, N. (2017b). A boosted decision tree approach using Bayesian hyper-parameter optimization for credit scoring. *Expert Systems with Applications*, 78, 225–241.

Xie, Z. & Li, M. (2018). Cutting the Software Building Efforts in Continuous Integration by Semi-Supervised Online AUC Optimization. *IJCAI*, pp. 2875–2881.

Xing, Z. & Stroulia, E. (2005). UMLDiff: an algorithm for object-oriented design differencing. *IEEE/ACM international Conference on Automated software engineering*, pp. 54–65.

Yang, L. & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415, 295–316.

Yang, X.-L., Lo, D., Xia, X., Wan, Z.-Y. & Sun, J.-L. (2016). What security questions do developers ask? a large-scale study of stack overflow posts. *Journal of Computer Science and Technology*, 31(5), 910–924.

Yu, Y., Vasilescu, B., Wang, H., Filkov, V. & Devanbu, P. (2016a). Initial and eventual software quality relating to continuous integration in GitHub. *arXiv preprint arXiv:1606.00521*.

Yu, Y., Yin, G., Wang, T., Yang, C. & Wang, H. (2016b). Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences*, 59(8), 080104.

Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H. & Di Penta, M. (2020). An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25(2), 1095–1135.

Zenisek, J., Holzinger, F. & Affenzeller, M. (2019). Machine learning based concept drift detection for predictive maintenance. *Computers & Industrial Engineering*, 137, 106031.

Zhang, C., Chen, B., Chen, L., Peng, X. & Zhao, W. (2019). A large-scale empirical study of compiler errors in continuous integration. *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 176–187.

Zhang, F., Zheng, Q., Zou, Y. & Hassan, A. E. (2016). Cross-project defect prediction using a connectivity-based unsupervised classifier. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 309–320.

Zhang, Y., Harman, M., Ochoa, G., Ruhe, G. & Brinkkemper, S. (2018). An empirical study of meta-and hyper-heuristic search for multi-objective release planning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(1), 1–32.

Zhao, F., Lei, W., Ma, W., Liu, Y. & Zhang, C. (2016). An improved SPEA2 algorithm with adaptive selection of evolutionary operators scheme for multiobjective optimization problems. *Mathematical Problems in Engineering*, 2016.

Zhao, H. (2007). A multi-objective genetic programming approach to developing Pareto optimal decision trees. *Decision Support Systems*, 43(3), 809–826.

Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V. & Vasilescu, B. (2017). The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. *32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 60–71.

Zheng, J. (2010). Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications*, 37(6), 4537–4543.

Zhou, Z.-H. & Liu, X.-Y. (2005). Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on knowledge and data engineering*, 18(1), 63–77.

Zitzler, E., Laumanns, M. & Thiele, L. (2001). SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK-report*, 103.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M. & Da Fonseca, V. G. (2003). Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on evolutionary computation*, 7(2), 117–132.