BF-Detector: An Automated Tool for CI Build Failure Detection

Islem Saidani
islem.saidani.1@ens.etsmtl.ca
Ecole de Technologie Superieure
Montreal, QC, Canada

Ali Ouni
ali.ouni@etsmtl.ca
Ecole de Technologie Superieure
Montreal, QC, Canada

Moataz Chouchen
moataz.chouchen.1@ens.etsmtl.ca
Ecole de Technologie Superieure
Montreal, QC, Canada

Mohamed Wiem Mkaouer
mwm@se.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

ABSTRACT

Continuous Integration (CI) aims at supporting developers in integrating code changes quickly through automated building. However, there is a consensus that CI build failure is a major barrier that developers face, which prevents them from proceeding further with development. In this paper, we introduce BF-Detector, an automated tool to detect CI build failure. Based on the adaptation of Non-dominated Sorting Genetic Algorithm (NSGA-II), our tool aims at finding the best prediction rules based on two conflicting objective functions to deal with both minority and majority classes. We evaluated the effectiveness of our tool on a benchmark of 56,019 CI builds. The results reveal that our technique outperforms state-of-the-art approaches by providing a better balance between both failed and passed builds. BF-Detector tool is publicly available, with a demo video, at: https://github.com/stilab-ets/BF-Detector

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

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

ACM Reference Format:
Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2021. BF-Detector: An Automated Tool for CI Build Failure Detection. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473115

1 INTRODUCTION

Continuous integration (CI) [10] is a set of software development practices that are widely adopted in industry and open source environments [32]. It consists of continuously integrating code changes into a shared repository branch, by automating the process of building and testing [12], which improves the productivity [17] and reduces the cost and risk of delivering defective changes [32]. However, despite its valuable benefits, CI brings its own challenges. Hilton et al. [16] revealed that build failure is a major barrier that developers face when using CI. In fact, CI build failure 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 [2]. Such challenges motivated researchers and practitioners to develop Machine Learning (ML) based techniques for preemptively detecting when a software state is most likely to trigger a failure when built. 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 [5].

As an alternative to ML techniques, Multi-Objective Genetic Programming (MOGP) [22, 35] was proposed to promote the diversity between solutions equally on both minority and majority classes and thus allowing the imbalanced training data to be used directly in the learning process without relying on sampling techniques to re-balance the data [6, 7]. This advocates that MOGP approaches are more suitable for binary classification tasks with imbalanced data [5]. Recently, in [28, 29] we proposed a MOGP technique to predict CI build failure based on the adaption of the Non-dominated Sorting Genetic Algorithm (NSGA-II) [9] with a tree-based solution representation. The idea is to generate prediction rules from historical data of CI builds using two competing objectives in the learning process to handle both passed and failed builds.

In this paper, we present a tool, called BF-Detector (Build Failure Detector), that applies our MOGP technique based on our previous work [29]. The tool is very lightweight and can be easily integrated with Travis CI [8]. Our tool was built to work with this system, since it is one of the most popular CI services [31]. In a nutshell, our tool takes as input, a given build, calculates a set of metrics that are fed into the prediction rule, previously generated using the history of builds, and returns the prediction outcome (*i.e.*whether the build is likely to pass or fail) as well as a justification for the prediction.

To evaluate the performance of BF-Detector, 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. We compare our predictive performance to three widely-used ML techniques namely Random Forest, Decision Tree and Naive Bayes. The statistical results reveal that our tool advances the state-of-the-art by achieving median score of 68% in terms of AUC (Area Under The Curve) compared to 61% achieved by existing ML techniques for which we applied re-sampling.

2 RELATED WORK

The analysis of CI build failure is growing as an active and challenging topic for software engineering research. Rausch et al. [27] investigated the impacts that can affect build failure on Travis CI. They observed by analyzing build logs that the most common reasons for build failures are failing integration tests, code quality measures being below a required threshold, and compilation errors. Luo et al. [20] found that the total number of commits in a build is the main influence feature that causes build failure. The number of files changed and the density of tests also impact a lot.

Recently, various research efforts have focused on studying CI build failure in order to cut off the expenses of CI build failure. Hassan and Wang [15] proposed a prediction model of CI build outcome on three build systems, namely Ant, Maven and Gradle, under the cross-project prediction and cross-validation scenarios by adopting Random Forest (RF). Xia et al. [34] conducted an empirical study to evaluate the predictive performance of six common classifiers including RF, NB and DT under cross-project validation. According to the results of their experiments, they found that DT classifier performs the best. In another paper, Xia and Li [33] compared nine ML models to construct CI prediction models. 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 DT and RF achieved the best performance scores. Recently, Saidani et al. [28, 29] proposed a customized MOGP-based approach to predict CI build failure. The proposed approach learns rule-based models from world instances of failed and succeeded builds while handling both minority and majority classes. Results show that MOGP outperforms various baseline ML techniques in terms accuracy.

Related efforts [1, 2] were proposed to speed up CI build process by detecting which commits are not necessary to trigger the build process. In this paper, we believe that BF-Detector could be used jointly with these techniques to provide better support to CI developers while reducing CI build costs and enhancing developer's experience in CI environments.

3 BF-DETECTOR ARCHITECTURE

Figure 1 provides an overview of our tool to predict the CI build outcome. We start from the observation that it is more beneficial for CI developers to identify good practices to follow in order to avoid build failure rather than simply detecting whether the build will succeed or fail. Our BF-Detector tool takes as input (1) an URL which can be the path to the GIT folder if the project is locally available, or the GitHub repository URL; the list of commits to be built (separated by #); and an option to launch the cross-project prediction. Next, the build information is extracted (2). First, the tool extracts the history of builds (`history_builds.csv` file). Our tool generates also the features of the current build to be saved in an `input_build_inf.csv` file. In case the user opts for within-project prediction (3.a) (*i.e.*, using the build information of the input project), then BF-Detector uses the Search-based Training module to generate the predictive rule that should provide a high accuracy for both failed and passed builds.

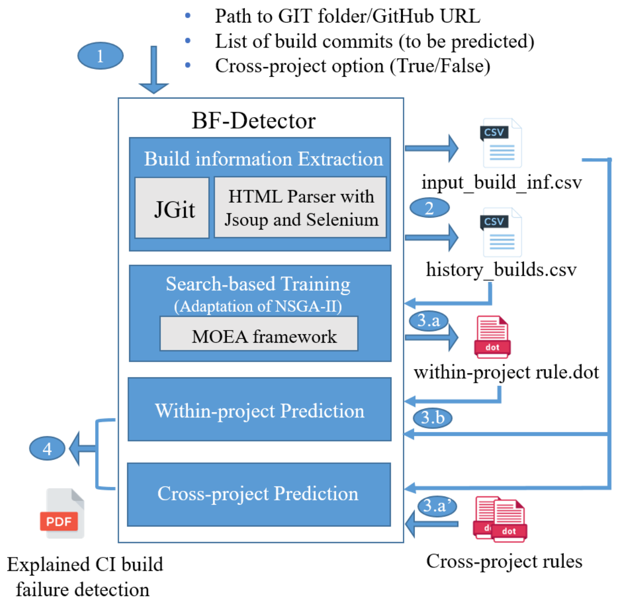
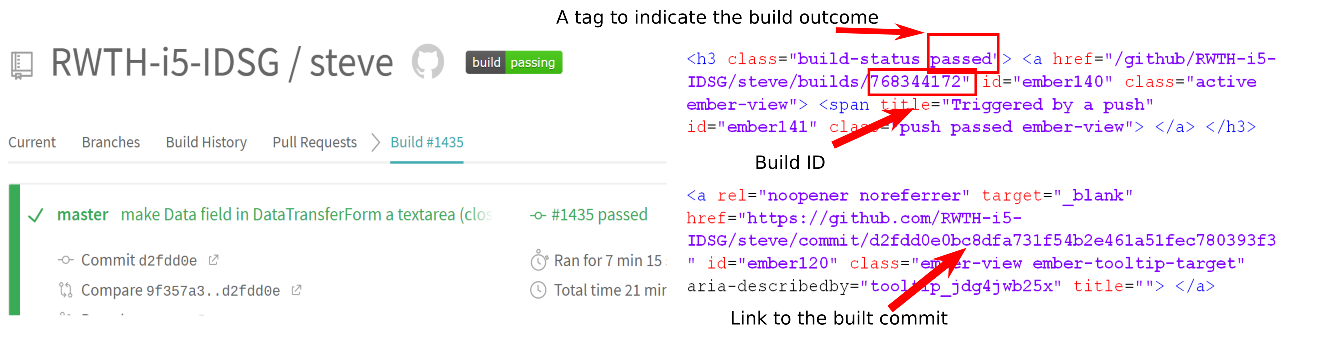

Path to GIT folder/GitHub URL
List of build commits (to be predicted)
Cross-project option (True/False)
1
BF-Detector
Build information Extraction
JGit
HTML Parser with Jsoup and Selenium
Search-based Training
(Adaptation of NSGA-II)
MOEA framework
Within-project Prediction
Cross-project Prediction
input_build_inf.csv
2
history_builds.csv
3.a
within-project rule.dot
3.b
3.a'
Cross-project rules
4
Explained CI build failure detection

Figure 1: Architecture and workflow of BF-Detector.

This rule is saved as the file `within-project rule.dot`. Once this file is created, it can be loaded each time the tool is triggered; which allows to save time. If the cross-project option is selected (3.b), the tool loads a set of rules (*i.e.*, `Cross-project rules`) generated previously based on real examples of build information extracted from TravisTorrent dataset [4]. At the prediction level and depending on the selected option, the system uses the prediction rule(s) and checks whether the features of the input commits respond to the conditions of the rule. Finally, the recommendation is returned to the user along with an explanation to justify the prediction (4). In the following subsections, we provide more details of each module.

3.1 Build Information Extraction

Before generating the features of the project, BF-Detector first extracts the build records. Since no tool is available to get the CI builds, we parse the corresponding web-page of the CI system to get such information. Indeed, CI systems such as Travis CI [8] provide development teams with detailed information about the builds ID, outcome, built commits and the date. In our tool, the build data crawling and extraction is based on the Java HTML Parser Jsoup [18] which is a Java library that parses HTML pages and collects the elements of interest (*e.g.*, CSS selectors).



```
<h3 class="build-status passed"> <a href="/github/RWTH-i5-
IDSG/steve/builds/768344172" id="ember140" class="active
ember-view"> <span title="Triggered by a push"
id="ember141" class="push passed ember-view"> </a> </h3>

<a rel="noopener noreferrer" target="_blank"
href="https://github.com/RWTH-i5-
IDSG/steve/commit/d2fdd0e0bc8dfa731f54b2e461a51fec780393f3
" id="ember120" class="ember-view ember-tooltip-target"
aria-describedby="tooltip_jdg4jwb25x" title=""> </a>
```

Figure 2: Example showing how to extract information about builds, using Jsoup, from `RWTH-i5-IDSG/steve` GitHub repository.

Taking the project `RWTH-i5-IDSG/steve` as an example, Figure 2 shows how BF-Detector can extract information about the build with *ID* = 768344172 [1] using Jsoup.

To simulate the interaction with the CI system web site, we use Selenium Web driver [30]. This toolkit allows to create a automated browser instance (*e.g.*, Google Chrome) to simulate human actions such as clicking or page scrolling.

Once the build information is extracted, BF-Detector mines the CI related features to be used for the build failure detection. Such information is gathered by mining the Git repository. To interact with Git, we rely on the Java library JGit [11].

The full list of the used metrics can be found in the tool's webpage https://github.com/stilab-ets/BF-Detector.

3.2 Search-Based Training

Our adaptation to the NSGA-II algorithm [9] is to adopt it following a tree-like representation instead of fixed length linear string formed from a limited alphabet of symbols. In our problem formulation, a candidate solution, *i.e.*, a prediction rule, is represented as an IF-THEN clause which describes the conditions under which a build is said to be succeeded or failed [3, 19, 23–26]. 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 predefined CI related metrics (extracted from literature) and their corresponding thresholds are generated randomly. Each internal-node belongs to the connective set C = {AND, OR}. All the generated solutions are evaluated using two objectives to (1) maximize the true positive rate (TPR), and (2) minimize the false positive rate (FPR). Change operators are applied, at every iteration, to generate new solutions. After repeating this process until reaching a stop criteria, the best solution *i.e.*prediction rule is returned by the algorithm.

It is worth to mention that in our BF-Detector tool, NSGA-II is implemented using the MOEA framework [13], an open source framework for developing and experimenting with Multi-Objective Evolutionary Algorithms (MOEAs) [14].

[1] https://travis-ci.org/github/RWTH-i5-IDSG/steve/builds/768344172

3.3 Within-Project Prediction

This option is generally selected when the project contains enough data to feed the search-based training module. After the generation/loading of the detection rule, our tool analyzes the changes made in the input commits to determine whether the build is likely to fail or pass.

3.4 Cross-Project Prediction

When the project does not have enough training data (*e.g.*, small or new project), this does not prevent it from using our tool. Indeed, BF-Detector generates, based on TravisTorrent dataset [4], a set of optimal detection rules that can be loaded to predict the current CI build outcome. Thereafter, all the outputs of these rules are combined to a single detection model using majority voting. If the majority of rules predict the build as failed, then it is predicted as failed and the best rule will be displayed.

3.5 Prediction Justification

The final step in BF-Detector's workflow is to provide an explained recommendation to the user in order to guide him in his decision to (not) build the committed changes. In Figure 3, we provide an example of a prediction justification. The red circle indicates a non-satisfied condition while the green one indicates that the input commit(s) properties satisfy the condition. When the root node (in this example it is OR) is green, this means that the committed changes are predicted to fail.

3.6 BF-Detector Usage

BF-Detector is implemented as a command line-based tool that is available as a standalone Java `jar` file in order to facilitate its integration within CI frameworks. BF-Detector can be executed via the following command:

```
java -jar BF-Detector.jar <Path to Git repository>
  <list of commits hashes separated by # > <T/F>
```

The jar file can be downloaded from https://github.com/stilab-ets/BF-Detector.

4 PERFORMANCE EVALUATION

We conducted an empirical study on the effectiveness of BF-Detector based on TravisTorrent dataset [31], from which

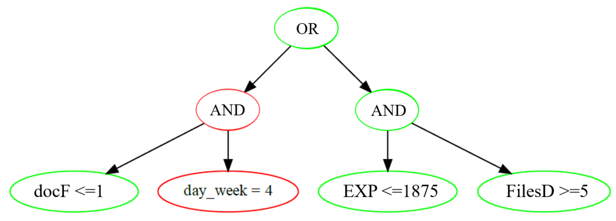


Figure 3: An simplified example of justification.

we selected 10 Java projects (after removing builds with "error"/ "canceled" status) as reported in Table 1.

Table 1: Studies projects statistics.

Project Name	# of Builds	Failure Rate (%)	Age at CI (days)
`CloudifySource/cloudify`	4,568	25	220
`gradle/gradle`	3,822	8	1,833
`Graylog2/graylog2-server`	3,341	12	470
`mitchellh/vagrant`	3,569	14	765
`openMF/mifosx`	2,252	7	2
`opf/openproject`	5,913	35	287
`rails/rails`	11,732	30	2,354
`rapid7/metasploit-framework`	6,391	7	2,571
`ruby/ruby`	11,814	21	5,099
`SonarSource/sonarqube`	2,317	24	1,013
Average	5,602	19	1,461

We evaluate our predictive performance against ML techniques namely Random Forest (RF), Decision Tree (DT) and Naive Bayesian (NB). To validate the predictive performance, we consider online validation in which builds are ordered and predicted chronologically [33]. Table 2 reports the detection results for balance [21] and Area Under the ROC Curve (AUC) scores. These metrics assess how well a model/rule performs on the minority and majority classes. Note that NSGA-II, RF and DT were executed 1,000 times for each experimentation instance to deal with their stochastic nature.

Overall, BF-Detector achieves an average AUC of 69% and an average balance of 66%. Moreover, we observe from Table 2 that for the 10 studied projects, the best AUC and balance values were achieved by the BF-Detector. On the other hand, for the different projects, the statistical analysis provides evidence that our approach performs better than the ML techniques. 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 our tool 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.

Table 2: Performance of BF-Detector vs ML techniques.

Project	AUC				Balance			
	BF-DETECTOR	DT	RF	NB	BF-DETECTOR	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

5 APPLICABILITY OF BF-DETECTOR

For CI Practitioners. We envisage our tool being used by developers, in their daily CI workflow to check whether their changes will break the build. The key innovation of our tool is that it is able to provide an explainable prediction model, and also some modalities to be respected in order to avoid build failure. Another benefit of using BF-Detector is that it can save the learning rule to be used for the prediction or updated later when more data is available over time as the project evolves.

For Researchers. Our explainable prediction of build outcome can be used as a starting point to prepare or recommend retroaction 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.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced BF-Detector a new command-line based tool for CI build failure detection. We described the architecture of BF-Detector, its usage scenarios, and its applicability by developers and researchers. We evaluated our tool based on a benchmark of 56,019 CI builds of ten projects that use Travis CI. Considering online validation, the statistical analysis of the obtained results provides evidence that our approach outperforms three Machine Learning (ML) techniques.

Our future work includes upgrading our tool to support other CI systems. We also plan to integrate another module to allow dynamic update of the generated rules when more data is available over time, *i.e.*, after a given number of builds, or generalized among other projects.

ACKNOWLEDGMENTS

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) discovery grant RGPIN-2018-05960.

REFERENCES

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2020. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering* (2020).

[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2019. Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering* (2019).

[3] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. 2020. On the detection of community smells using genetic programming-based ensemble classifier chain. In *15th International Conference on Global Software Engineering (ICGSE)*. 43–54.

[4] M. Beller, G. Gousios, and A. Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 447–450.

[5] Urvesh Bhowan, Mark Johnston, and Mengjie Zhang. 2011. Evolving ensembles in multi-objective genetic programming for classification with unbalanced data. In *Annual conference on Genetic and evolutionary computation (GECCO)*. 1331–1338.

[6] Urvesh Bhowan, Mark Johnston, Mengjie Zhang, and Xin Yao. 2012. Evolving diverse ensembles using genetic programming for classification with unbalanced data. *IEEE Transactions on Evolutionary Computation* 17, 3 (2012), 368–386.

[7] Urvesh Bhowan, Mark Johnston, Mengjie Zhang, and Xin Yao. 2013. Reusing genetic programming for ensemble selection in classification of unbalanced data. *IEEE Transactions on Evolutionary Computation* 18, 6 (2013), 893–908.

[8] Travis CI. [n.d.]. Travis CI. https://travis-ci.org/ , note = Accessed: 2021-04-01.

[9] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2, 182–197.

[10] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[11] Eclipse. [n.d.]. Eclipse Java Git library. https://www.eclipse.org/jgit//, note = Accessed: 2021-04-01.

[12] Martin Fowler. 2006. Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html,. Accessed: 2020-01-01.

[13] David Hadka. [n.d.]. MOEA Framework. http://moeaframework.org/,. Accessed: 2020-01-01.

[14] David Hadka. 2014. MOEA framework user guide. (2014).

[15] Foyzul Hassan and Xiaoyin Wang. 2017. Change-aware build prediction model for stall avoidance in continuous integration. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 157–162.

[16] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *11th Joint Meeting on Foundations of Software Engineering*. 197–207.

[17] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. 426–437.

[18] Jsoup. [n.d.]. Jsoup. https://jsoup.org/ , note = Accessed: 2021-04-01.

[19] Marouane Kessentini and Ali Ouni. 2017. Detecting android smells using multi-objective genetic programming. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. 122–132.

[20] Yang Luo, Yangyang Zhao, Wanwangying Ma, and Lin Chen. 2017. What are the Factors Impacting Build Breakage?. In *14th Web Information Systems and Applications Conference (WISA)*. 139–142.

[21] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[22] Ali Ouni. 2020. Search-based Software Engineering: Challenges, Opportunities and Recent Applications. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. 1114–1146.

[23] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. 2015. Web service antipatterns detection using genetic programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1351–1358.

[24] Ali Ouni, Marouane Kessentini, Katsuro Inoue, and Mel O Cinnéide. 2017. Search-based web service antipatterns detection. *IEEE Transactions on Services Computing* 10, 4 (2017), 603–617.

[25] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2013. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering* 20, 1 (2013), 47–79.

[26] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 23.

[27] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *14th international conference on mining software repositories*. 345–355.

[28] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. On the prediction of continuous integration build failures using search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. 313–314.

[29] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology* 128 (2020), 106392.

[30] Selenium. [n.d.]. Selenium. https://www.selenium.dev/ , note = Accessed: 2021-04-01.

[31] TravisTorrent. [n.d.]. TravisTorrent Dataset. https://travistorrent.testroots.org/ , note = Accessed: 2021-04-01.

[32] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. 805–816.

[33] Jing Xia and Yanhui Li. 2017. Could we predict the result of a continuous integration build? An empirical study. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 311–315.

[34] Jing Xia, Yanhui Li, and Chuanqi Wang. 2017. An Empirical Study on the Cross-Project Predictability of Continuous Integration Outcomes. In *14th Web Information Systems and Applications Conference (WISA)*. 234–239.

[35] Huimin Zhao. 2007. A multi-objective genetic programming approach to developing Pareto optimal decision trees. *Decision Support Systems* 43, 3 (2007), 809–826.