Predicting continuous integration build failures using evolutionary search

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

[a] *ETS Montreal, University of Quebec, Montreal, QC, Canada*
[b] *Rochester Institute of Technology, Rochester, NY, USA*

ARTICLE INFO

ABSTRACT

Context: 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.

Objective: 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.

Method: 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.

Results: 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.

Conclusion: 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.

1. Introduction

Continuous integration (CI) [1] is a set of software development practices that are widely adopted in industry and open source environments [2]. A typical CI system, such as Travis CI[1], advocates to continuously integrate code changes, introduced by different developers, into a shared repository branch. The key to making this possible, according to Fowler [3], 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 [4], helps to maintain code quality [2] and allows for a higher release frequency [5].

However, despite its valuable benefits, CI brings its own challenges. Hilton et al. [6]. 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,

---

* Corresponding author.
E-mail addresses: islem.saidani.1@ens.etsmlt.ca (I. Saidani), ali.ouni@etsmtl.ca (A. Ouni), moataz.chouchen.1@ens.etsmtl.ca (M. Chouchen), mwmvse@rit.edu (M.W. Mkaouer).
[1] https://travis-ci.org/

the speed of software development and the productivity of developers [7]. 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 [8] used Random Forest (RF), for the binary classification of build outcome, and Ni and Li [9] 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 [10–13]. 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 [10,14].

To deal with the above mentioned challenges, Evolutionary Multi-Objective Optimisation (EMO) [15–19] have been found useful for developing software engineering predictive models [20,21]. 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 [22], 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 [12,23] which advocates that MOGP approaches are more suitable for binary classification tasks with imbalanced data [10].

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) [24] 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 [25]. 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 [26] 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 [26] 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. Section 2 provides an overview of the CI build process and the related work. We present our approach in Section 3. Section 4 shows the experimental setup of our empirical study. Section 5 presents the results and findings of our studied research questions. Section 6 discusses the implications of our findings for developers, researchers and tool builders. Section 7 reviews the threats to the validity of our results. Finally, Section 8 concludes the paper and outlines avenues for future work.

2. Background and related work

In this section, we provide an overview of CI and the related work.

2.1. CI Build 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 Fowler [3] 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 defined as follows. 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. When the work is done, the contributor submits the changes to the original repository. At this point, the CI service carries out a series of tasks to build and test these changes. Then, it provides immediate feedback on the outcome of the test to the core team, *i.e.*, developers who dispose of write access to a project's code repository [2]. 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.

2.2. Related work

This section presents the related research about CI builds while highlighting the contributions of our work.

Prediction of CI builds: Many research works have introduced prediction models to predict the CI build status. Xia and Li [27] compared nine ML classifiers 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. In [8], Foyzul and Wang proposed the prediction model of CI build outcome on three build systems, namely Ant, Maven and Gradle, under the cross-project prediction and cross-validation scenarios. Using random forest, 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 amount of redundant lines which may influence the validity of the reported results. We also found that the dataset is perfectly balanced (45% of failed builds) which is not in compliance with the real world situation as it is generally known that failed builds are much less to occur than passed ones [28]. In this paper, we found that when applying RF to our generated dataset, our approach can achieve better results. Xia et al. [29] conducted an empirical study to evaluate the predictive performance of six common classifiers including RF, NB and DT under 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 performs better than the two other methods. And among the used classifiers, they found that Decision Tree (DT) classifier performs the best achieving a score of 17% for F1-measure on average.

Although most of the existing approach achieved good results by using variety of domain knowledge and historical information the of CI builds, none of these works actually construct the prediction model that perfectly fits the imbalance in build outcomes characteristics of the CI build outcome which challenges their applicability. Additionally, the predictive performance of the used techniques like RF, depends highly on the used features, the dataset representativeness and the failure rate which may explain the differences in the obtained results.

Insights into CI builds: The analysis of CI build failures is growing as an active and challenging topic for software engineering research. Rausch et al. [30]. investigated the impacts that can affect build failures 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. Beller et al. [31]. focused on testing with an in-depth analysis of CI builds. The main finding of their study is that 59% of build failures occur during test phase for Java projects. Luo et al. [32] proposed a case study to investigate what features have greater impact on the build result. Conducting a case study on the TravisTorrent dataset, they 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. In this paper, we conduct a deep analysis to investigate the most influencing factors of build outcome using our proper generated rules.

Other Studies About CI Builds Atchison et al. [33]. conducted a time-series analysis of the history of CI builds to identify temporal patterns in build volume within TravisTorrent dataset [34]. By observing a clear seasonality in build activity, their approach was able to estimate the number of builds to be generated in the future, with an average accuracy of 86%. Another interesting study was conducted by Ghaleb et al. [35]. to analyse the long duration of builds over 67 GitHub projects that are linked with Travis CI. The main finding of their study is that about 40% of builds take over 30 minutes to run which points to the high energy cost of CI builds that increases as the build duration increases.

3. 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.

3.1. Approach overview

Fig. 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 Fig. 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 3.3). 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 4.5.

3.2. NSGA-II adaptation

In this section, we describe in details our search-based approach. We first provide an overview of NSGA-II and then we define how we adapt it to our build failure prediction problem.

3.2.1. NSGA-II overview

We employed a widely used computational search technique, NSGA-II [36] that has proven good performance in solving many software engineering problems [15,37–39]. As described in Algorithm 1, 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* [24] is the technique used by MOGP to classify individual solutions into different dominance levels (line 6) [24]. 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 front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [24] to make
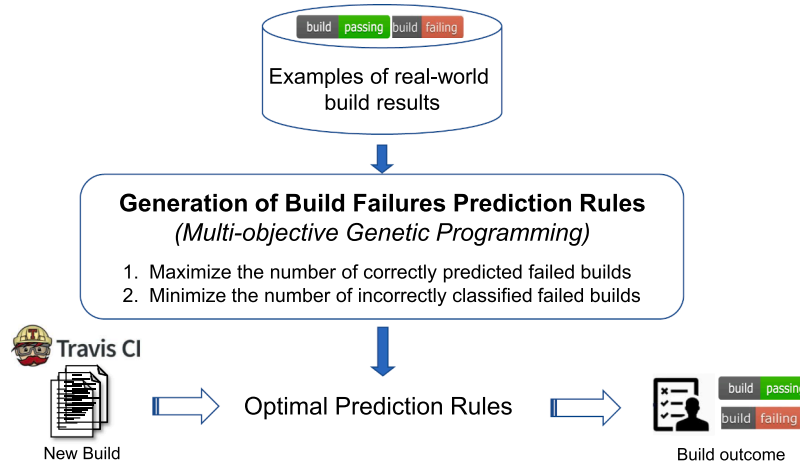
Fig. 1. An overview of our approach.

1: Create an initial population P_0
2: Create an offspring population Q_0
3: $t = 0$
4: **while** stopping criteria 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** $| P_{t+1} | + | F_i | \leqslant N$ **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:      $P_{t+1} = P_{t+1} \cup F_i[N - | P_{t+1} |]$
15:      $Q_{t+1} = $ create-new-pop(P_{t+1})
16:      t = t+1
17: **end while**

Algorithm 1. High level pseudo code of NSGA-II.

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 criteria (line 4).

3.2.2. Adaptation

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 [40]. 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 1 and their corresponding thresholds are generated randomly. Each internal-node belongs to the connective set C = {AND, OR}. Fig. 2 shows an illustrative example of a solution. 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 $\geq$ 20 AND gh_diff_files_added $\geq$ 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 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 ($\geq, \leq, =$) that depends on the metric category. Note that "=" is only used for categorical metrics (e.g., gh_is_pr), $\geq$ and $\leq$ 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. Fig. 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. For example, after applying the crossover operator the new rule C2 to predict build failure

Table 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	[9,27–29, 32]
	gh_num_commits_on_files_touched	T	# of unique commits on the files touched in the built commits	[27,32]
	git_diff_src_churn	T	# of lines of code changed in all built commits	[8,27,28]
Files change	gh_diff_files_added	T	# of files added in all built commits	[9,27,29,32]
	gh_diff_files_deleted	T	# of files deleted by all built commits	[27,29,32]
	gh_diff_files_modified	T	# of files modified by all built commits	[9,27,29,32]
Cooperation	gh_num_commit_comments	T	# of comments of all built commits	[27,29,32]
	num_of_distinct_authors	T	# of distinct authors in all built commits	[28,29]
	Total_Number_Of_Revisions	G	# of revisions on all the files touched by the current build	[28]
Triggering commit	gh_by_core_team_member	T	Whether the commit that has triggered the build was authored by a core team member	[27,32]
	gh_is_pr	T	Whether this build was triggered as part of a pull request on GitHub.	[32]
	day_week	G	Day of week of the first commit for the build	[8]
Change type	gh_diff_src_files	T	# of src files changed by all built commits	[27,29]
	gh_diff_doc_files	T	# of documentation files changed by all built commits	[27,29,32]
	gh_diff_other_files	T	# of files which are neither source code nor documentation.	[27,29,32]
	num_config_files	G	# of configuration files (*.xml, *.yml, etc) edited in this commit.	[8,9,44]
Test Change	git_diff_test_churn	T	# of lines of test code changed in all built commits	[8,27,29,32]
	gh_diff_tests_added	T	# of test cases added in all built commits	[27,32]
	gh_diff_tests_deleted	T	# of test cases deleted in all built commits	[27,32]
Link to last build	prev_built_result	G	Result of last build	[8,9,28]
	same_committer	T	Indicates whether the committer is the same as last build	[9]
	elapsed_days_last_build	T	Counts the days since last build	[9]
	git_prev_commit_resolution_status	T	= it could be "build found", "merge found" or "no previous build"	[27,32]
Committer Experience	committer_fail_history	G	The fail rate of the builds by the current committer in the past	[9]
	committer_fail_recent	G	Similar to committer history, but measuring only his last five builds	[9]
	committers_avg_exp	G	The average number of builds the committers made in the project before this build	[9]
Project History	project_fail_history	G	The fail rate of the all the projectâs previous build	[9]
	project_fail_recent	G	Similar to project fail history but using only last five builds	[9]
	gh_team_size	T	# of developers that committed from the moment the build was triggered and 3 months back.	[8,27,32]
	gh_sloc	T	# of source lines of code, in the entire repository at the time of this build.	[27–29,32]
Test Density	gh_test_lines_per_kloc	T	# of lines in test cases per 1000 gh_sloc.	[27,32]
	gh_test_cases_per_kloc	T	# of test cases per 1000 gh_sloc.	[27,32]
	gh_asserts_cases_per_kloc	T	# of assertions per 1000 gh_sloc.	[27,29,32]

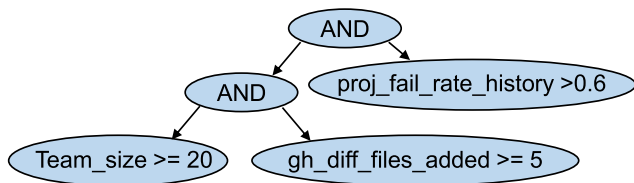T: TravisTorrent, G: Generated

Fig. 2. A simplified example of solution encoding for CI build failure prediction.

will be:

IF gh_is_pr = 1 OR gh_diff_files_added $\geq$ 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. Fig. 4 illustrates the effect of a mutation that deletes the note containing proj_fail_rate_history feature,
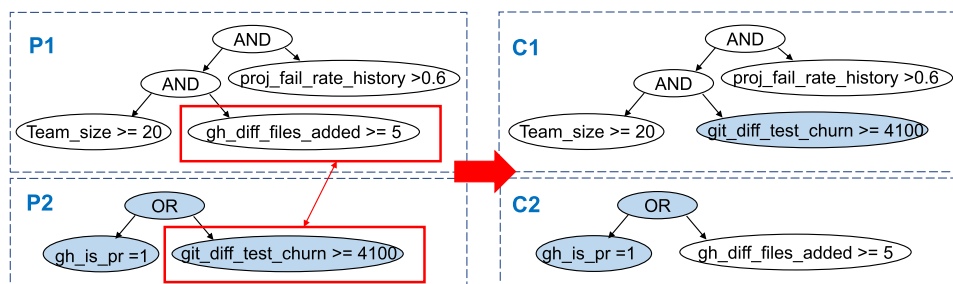

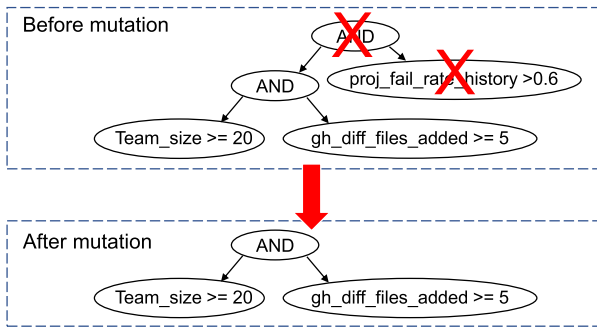**Fig. 3.** An example of crossover operator.

Fig. 4. An example of mutation operator.

leading to the automatic deletion of node AND (no left sub-tree). Thus, after applying the mutation operator the new rule will be:

IF team_size $\geq$ 20 OR gh_diff_files_added $\geq$ 5 **THEN** Failure.

iv. 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 [41], 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 [14]:

(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.

v. 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*) [42,43] that corresponds of the best compromise between TPR and FPR:

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

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

3.3. Dataset and CI-related metrics

To collect our data, we use TravisTorrent [34], 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 [9,27–29,32]. Table 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 [3].

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 [26].

4. 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 [26] for future replications and extensions.

Fig. 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
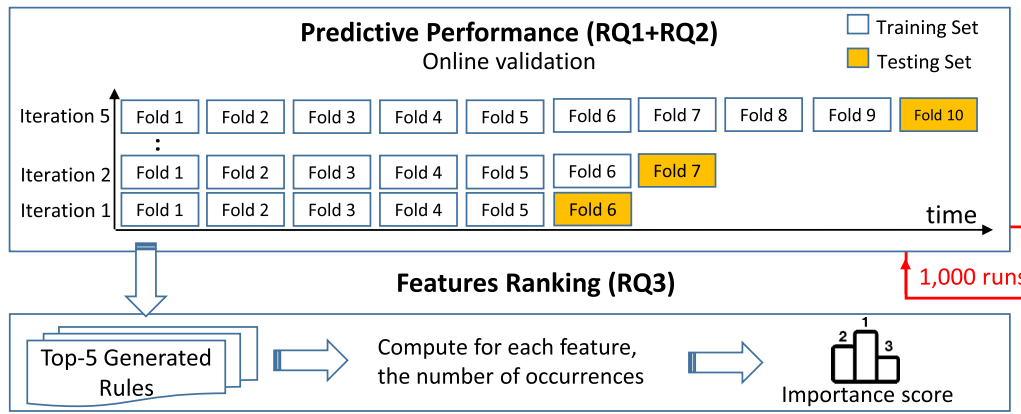
Fig. 5. Experimental design.

techniques used in this empirical study 1000 times to deal with the stochastic nature of these algorithms. To validate the predictive performance, we consider online validation [27]. 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.

4.1. 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?

4.2. Analysis method

4.2.1. 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 [45]. First, we compare our SBSE formulation against Random Search (RS) [37,46] 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 [37]. 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:

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

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 [47]:

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

Moreover, it is important to account for imbalance in a data set. Indeed, various researchers [25,48,49] 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) [49]. Higher balances are desirable for a model. The balance metric is defined as follows.

$$Balance = 1 - \sqrt{\frac{\left(0 - \frac{PF}{100}\right)^2 * \left(1 - \frac{PD}{100}\right)^2}{2}} \in [0, 1]. \tag{3}$$

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

4.2.2. 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 [36], Indicator-Based Evolutionary Algorithm (IBEA) [50] and Strength-Pareto Evolutionary Algorithm (SPEA2) [51], as they are among the most popular MOEAs and have been widely utilized in SBSE [13,15,15,38,52,53]. Additionally, all the search-based algorithms used in this paper are implemented using the MOEA framework [54], an open source framework for developing and experimenting with MOEAs [55].

Since the underlying goal of MOEAs is to determine a set of alternative solutions known as Pareto front approximations [55], we aim to compare the performance of each algorithm using Zitzler et al. [56]. 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 [57].
- **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 [57].

Table 2
Studies projects statistics.

Project Name	Language	# of Builds	Failure Rate	Age at CI (days)
CloudifySource/cloudify	java	4568	0.25	220
gradle/gradle	java	3822	0.08	1833
Graylog2/graylog2-server	java	3341	0.12	470
mitchellh/vagrant	ruby	3569	0.14	765
openMF/mifosx	java	2252	0.07	2
opf/openproject	ruby	5913	0.35	287
rails/rails	ruby	11,732	0.30	2354
rapid7/metasploit-framework	ruby	6391	0.07	2571
ruby/ruby	ruby	11,814	0.21	5099
SonarSource/sonarqube	java	2317	0.24	1013
Average	–	5,602	0.19	1,461

- **Spacing (SP):** is the most popular uniformity indicator [58]. 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 [8,27,29,32,44,59,59], 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 [60], 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 [27], 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 [26].

4.2.3. 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 [30–32] 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 3.3).

4.3. Subjects selection

Our experiments are based on TravisTorrent dataset[2], from which we selected top-10 Java and Ruby, the only supported languages in this dataset [34], projects according to the number of build records (after removing inadequate rows as described in Section 3.3). An overview about the studied projects is reported in Table 2. It is noteworthy that the data in all these projects is highly imbalanced. Our replication package is publicly available at [26].

Cloudify[3] is a cloud-enablement platform that on-boards applications to public and private clouds without architectural or code changes. Gradle[4] 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[5] 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[6] 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[7] is an open technology platform for financial inclusion that provides core functionalities to deliver financial services. OpenProject[8] is one of the leading open source web-based project management systems. Rails[9] 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[10] 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[11] is an interpreted object-oriented programming language often used for web development. Finally, SonarQube[12] 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.

---

[2] https://travistorrent.testroots.org/
[3] https://github.com/CloudifySource/cloudify
[4] https://github.com/gradle/gradle
[5] https://github.com/Graylog2/graylog2-server
[6] https://github.com/hashicorp/vagrant
[7] https://github.com/openMF/mifosx
[8] https://github.com/opf/openproject
[9] https://github.com/rails/rails
[10] https://github.com/rapid7/metasploit-framework
[11] https://github.com/ruby/ruby
[12] https://github.com/SonarSource/sonarqube

4.4. 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 1000 runs as suggested by Arcury and Briand guidelines [61] as well as recent works [38,39, 62–66]. 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 [67] in order to detect significant performance differences between the algorithms under comparison (α is set at 0.05). In this validation, each iteration is repeated 1000 times, for each algorithm and each project. It is worth mentioning that for RQ3, we choose the rule with the median value through 3000 runs of each iteration.

We also use Vargha-Delaney A (VDA) [68], a non-parametric effect size measure which is widely used in SBSE [69]. 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 [70]. Vargha-Delaney statistic also classifies the magnitude of the obtained effect size value into four different levels (*negligible, small, medium, and large*) [71].

4.5. 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 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.

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 [72] as the majority of the handled data is continuous.

5. Experimental results

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

5.1. 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. Fig. 6 plots the results while Table 4 highlights the statistical tests results of this comparison.

As shown in Fig. 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 × 1000 runs × 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 shows the results of MOEAs comparisons based on the hyper-volume (HV), Generational Distance (GD) and Spacing (SP) as described in Section 4.2.1. 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 Fig. 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).

5.2. RQ2. Results for the comparison with ML

Figs. 8 and 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. Table 7 reports the average (of 5 online validation iterations) balance and AUC scores while

Table 3
Algorithms parameters.

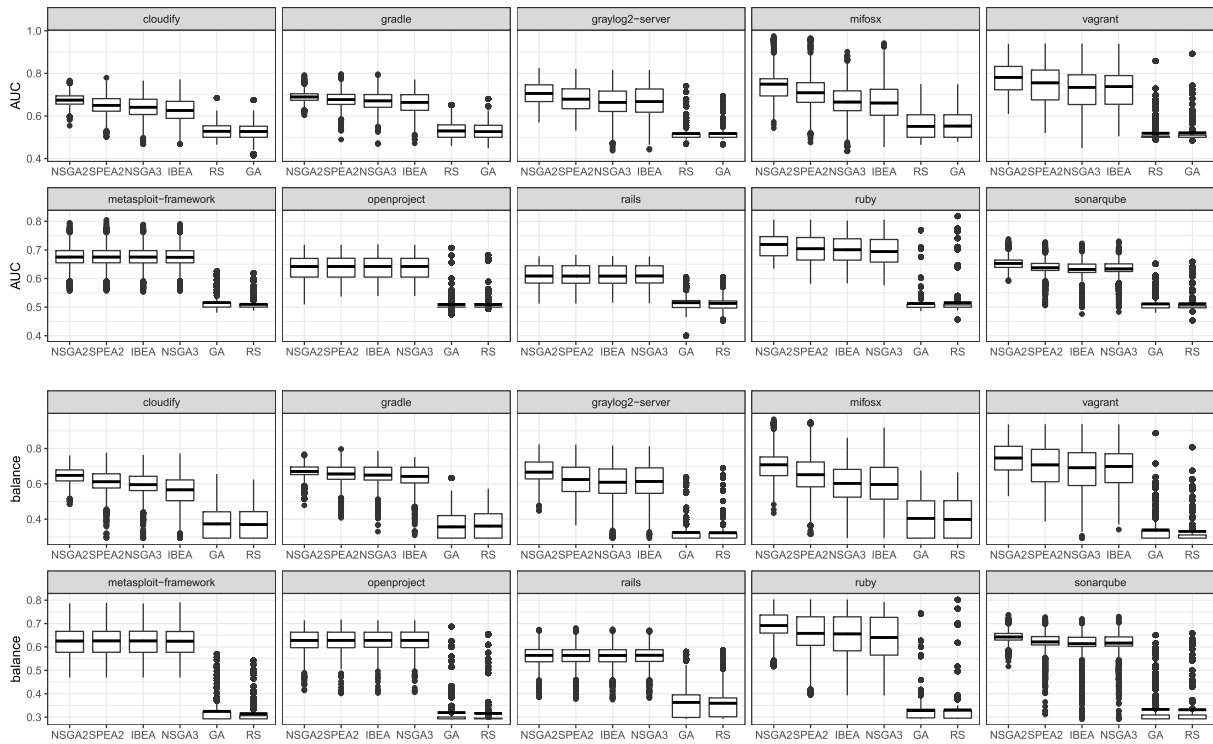| Algorithms	Parameters	Values
NSGA-II, NSGA-III, IBEA, SPEA2, GA, RS	Population size	100
	Maximum number of generations	500
	Maximum depth of the tree	10
	Crossover probability*	0.9
	Mutation probability*	0.1
RF	Maximum depth of the tree	10
	Number of estimators	200
DT	Maximum depth of the tree	10
NB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

* Not applied to RS

Fig. 6. Boxplots comparing scores of the search-based algorithms for the 5000 experiment instances (1,000 runs × 5 validation iterations) in each project.

Table 4
Statistical tests results of NSGA-II compared to other search-based techniques.

NSGA-II		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

Table 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 6 shows the statistical comparisons of these experiments. Note that NSGA-II, RF and DT were executed 1000 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 are failed) as can be noticed from Table 2. Moreover, we see from Table 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
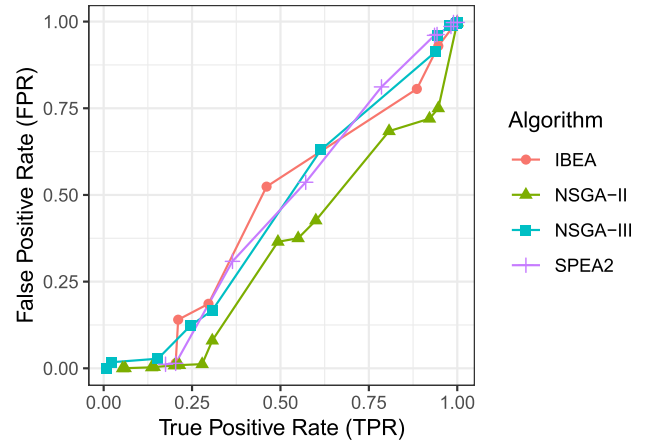

Fig. 7. An example of Pareto Front extracted from `mitchellh/vagrant` project.

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.

For instance, in the `Graylog2/graylog-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,
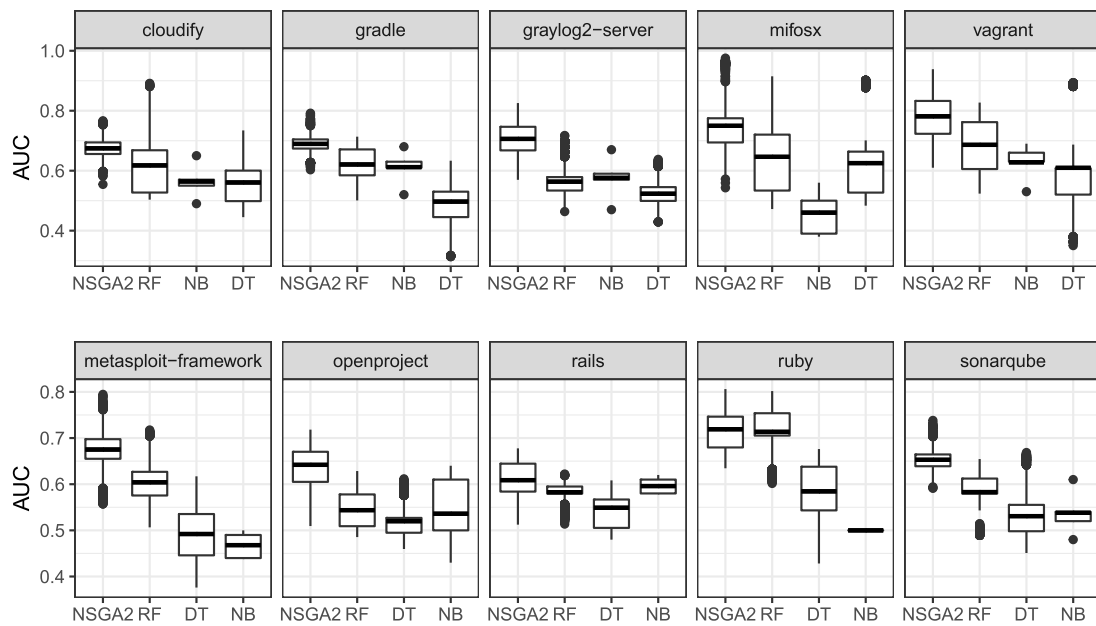
Fig. 8. Boxplots comparing the achieved AUC values for NSGA-II and each of the machine learning techniques, DT, NB and RF.
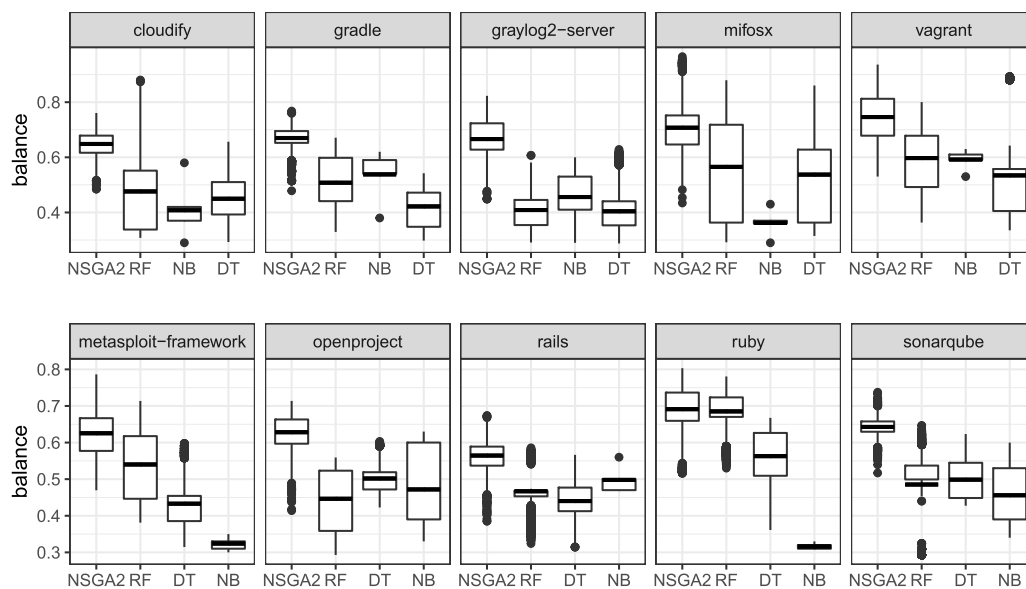

Fig. 9. Boxplots comparing the achieved balance values for NSGA-II and each of the machine learning techniques, DT, NB and RF.

Table 6
Statistical tests results of NSGA-II compared to ML techniques.

		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

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 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 [14].

5.3. RQ3. Results for feature analysis

In this RQ, we want to better understand what features contributed to achieving higher performances. Fig. 10 shows the results of feature ranking for each project while Table 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

Table 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
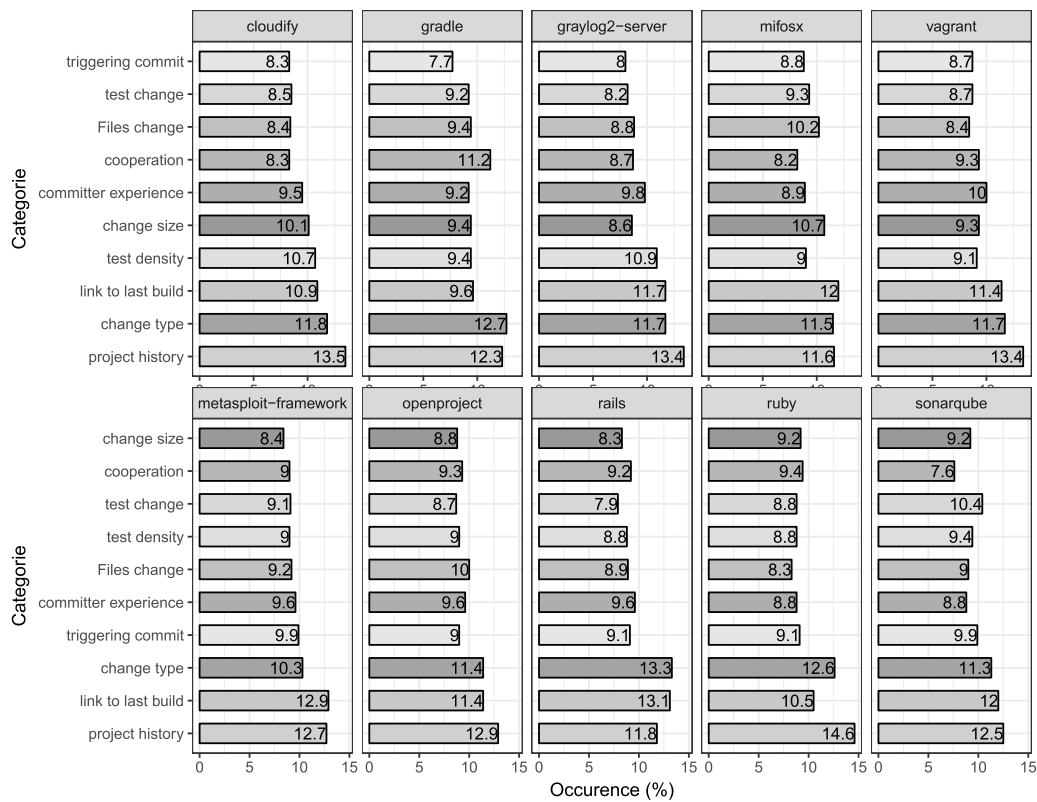
Fig. 10. Features ranking for each project.

Table 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

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 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 [9] 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 pr1o-vides additional support for our rules. As stated previously [8,9,30], 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 2567 builds where changes to only source code files
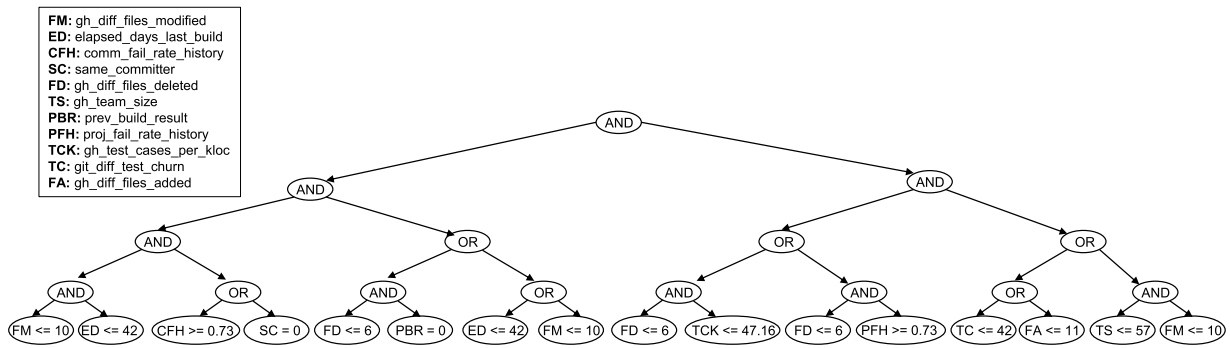
Fig. 11. An example of CI build failure prediction rule for the `mitchellh/vagrant` project.

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.

6. Discussion

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

6.1. 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, Fig. 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. Fig. 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 commitâs files and compute the CI metrics to determine 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.

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
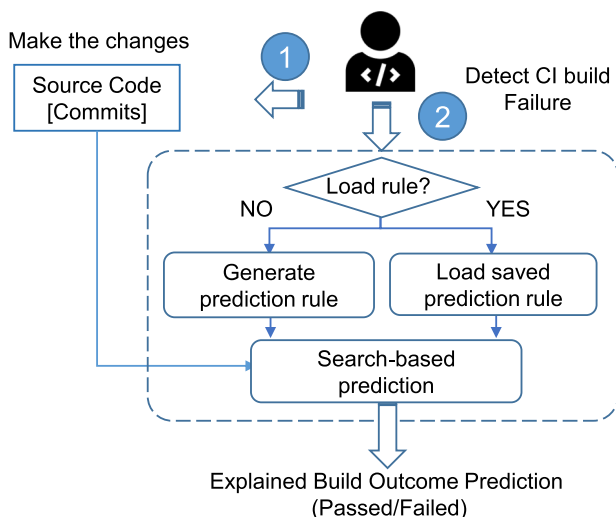

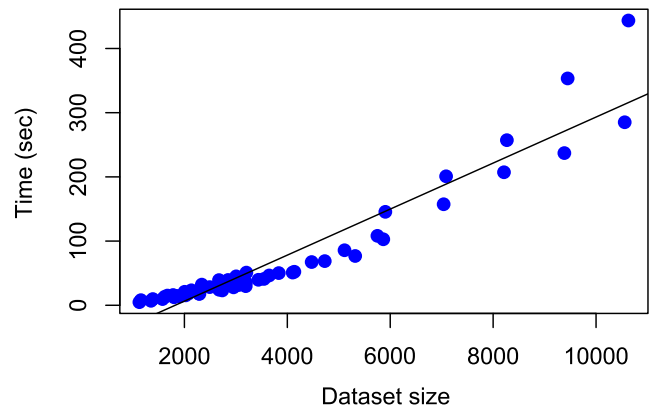**Fig. 12.** A usage workflow of our approach.

Fig. 13. The impact of the training dataset size on the NSGA-II execution time to build the prediction model.

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. Fig. 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., Fig. 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.

6.2. 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 6.1, 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 [30] 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.

6.3. 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 [73,74]. For instance, Fig. 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 ≥ 200 and the elapsed days ≤ 2; or if the number of modified files ≤ 2 and the source churn ≥ 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 [75]. 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.

Tool for recommending relevant files for build failures localisation. 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.

7. 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 [15,61]. To mitigate this threat, we performed 1000 runs of each experimentation instance and considered the
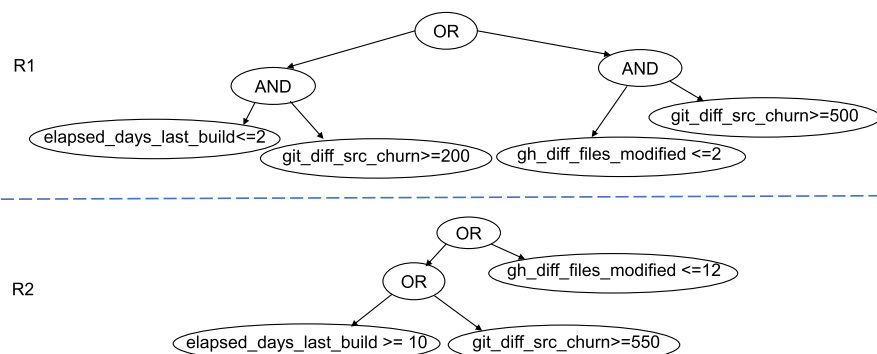

Fig. 14. An example showing how the prediction rules can be less effective when applied to other projects.

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 [14]. 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 [8,27,29,32,44,59,59]. 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 [8,29,32,44]. 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 [69]. 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.

8. 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.

CRediT authorship contribution statement

Islem Saidani: Conceptualization, Data curation, Formal analysis, Methodology, Software, Validation, Investigation, Writing - original draft. **Ali Ouni:** Conceptualization, Validation, Supervision, Resources, Writing - review & editing, Funding acquisition, Project administration. **Moataz Chouchen:** Software, Validation, Investigation. **Mohamed Wiem Mkaouer:** Methodology, Validation, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

References

[1] P.M. Duvall, S. Matyas, A. Glover, Continuous integration: Improving software quality and reducing risk, Pearson Education, 2007.
[2] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, V. Filkov, Quality and productivity outcomes relating to continuous integration in github. 10th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2015, 2015, pp. 805–816.
[3] M. Fowler, Continuous Integration, 2006, https://www.martinfowler.com/articles/continuousIntegration.html. Accessed: 2020-01-01.
[4] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig, Usage, costs, and benefits of continuous integration in open-source projects. 31st IEEE/ACM International Conference on Automated Software Engineering, in: ASE 2016, 2016, pp. 426–437.
[5] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, B. Vasilescu, The impact of continuous integration on other software development practices: A large-scale empirical study. 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 60–71.
[6] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, D. Dig, Trade-offs in continuous integration: assurance, security, and flexibility. 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 197–207.
[7] R. Abdalkareem, S. Mujahid, E. Shihab, J. Rilling, Which commits can be ci skipped? IEEE Trans. Software Eng. (2019).
[8] F. Hassan, X. Wang, Change-aware build prediction model for stall avoidance in continuous integration. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2017, pp. 157–162.
[9] A. Ni, M. Li, Cost-effective build outcome prediction using cascaded classifiers. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 455–458.
[10] U. Bhowan, M. Johnston, M. Zhang, Evolving ensembles in multi-objective genetic programming for classification with unbalanced data. Annual conference on Genetic and evolutionary computation (GECCO), 2011, pp. 1331–1338.
[11] U. Bhowan, M. Zhang, M. Johnston, Genetic programming for classification with unbalanced data. European Conference on Genetic Programming, Springer, 2010, pp. 1–13.
[12] U. Bhowan, M. Johnston, M. Zhang, X. Yao, Reusing genetic programming for ensemble selection in classification of unbalanced data, IEEE Trans. Evol. Comput. 18 (6) (2013) 893–908.
[13] I. Saidani, A. Ouni, M. Chouchen, M.W. Mkaouer, On the prediction of continuous integration build failures using search-based software engineering. Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, 2020, pp. 313–314.
[14] R. Malhotra, M. Khanna, An exploratory study for software change prediction in object-oriented systems using hybridized techniques, Automated Software Engineering 24 (3) (2017) 673–717.
[15] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: trends, techniques and applications, ACM Computing Surveys (CSUR) 45 (1) (2012) 11.
[16] J. Nam, W. Fu, S. Kim, T. Menzies, L. Tan, Heterogeneous defect prediction, IEEE Trans. Software Eng. 44 (9) (2017) 874–896.
[17] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Automated Software Engineering 20 (1) (2013) 47–79.
[18] J. Chen, V. Nair, R. Krishna, T. Menzies, æSamplingg as a baseline optimizer for search-based software engineering, IEEE Trans. Software Eng. 45 (6) (2018) 597–614.

[19] M. Kessentini, A. Ouni, Detecting android smells using multi-objective genetic programming. International Conference on Mobile Software Engineering and Systems, 2017, pp. 122–132.

[20] Z. Eckart, L. Marco, T. Lothar, Improving the strength pareto evolutionary algorithm for multiobjective optimi-zation, EUROGEN, Evol. Method Des. Optim. Control Ind. Problem (2001) 1–21.

[21] Y. Jin, B. Sendhoff, Pareto-based multiobjective machine learning: an overview and case studies, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 38 (3) (2008) 397–415.

[22] H. Zhao, A multi-objective genetic programming approach to developing pareto optimal decision trees, Decis Support Syst 43 (3) (2007) 809–826.

[23] U. Bhowan, M. Johnston, M. Zhang, X. Yao, Evolving diverse ensembles using genetic programming for classification with unbalanced data, IEEE Trans. Evol. Comput. 17 (3) (2012) 368–386.

[24] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii 6, 2002, pp. 182–197.

[25] R. Malhotra, A systematic review of machine learning techniques for software fault prediction, Appl Soft Comput 27 (2015) 504–518.

[26] Dataset for ci build prediction, 2020, (Available at : https://github.com/GP-CI-Build-Fail/replication-package).

[27] J. Xia, Y. Li, Could we predict the result of a continuous integration build? an empirical study. 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2017, pp. 311–315.

[28] Z. Xie, M. Li, Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization.. IJCAI, 2018, pp. 2875–2881.

[29] J. Xia, Y. Li, C. Wang, An empirical study on the cross-project predictability of continuous integration outcomes. 2017 14th Web Information Systems and Applications Conference (WISA), IEEE, 2017, pp. 234–239.

[30] T. Rausch, W. Hummer, P. Leitner, S. Schulte, An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 345–355.

[31] M. Beller, G. Gousios, A. Zaidman, Oops, my tests broke the build: An explorative analysis of travis CI with github. IEEE/ACM International Conference on Mining Software Repositories, 2017, pp. 356–367.

[32] Y. Luo, Y. Zhao, W. Ma, L. Chen, What are the factors impacting build breakage?. 2017 14th Web Information Systems and Applications Conference (WISA) IEEE, 2017, pp. 139–142.

[33] A. Atchison, C. Berardi, N. Best, E. Stevens, E. Linstead, A time series analysis of travistorrent builds: to everything there is a season. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 463–466.

[34] M. Beller, G. Gousios, A. Zaidman, Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017, pp. 447–450.

[35] T.A. Ghaleb, D.A. da Costa, Y. Zou, An empirical study of the long duration of continuous integration builds, Empirical Software Engineering (2019) 1–38.

[36] K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints, IEEE Trans. Evol. Comput. 18 (4) (2013) 577–601.

[37] M. Harman, P. McMinn, J.T. De Souza, S. Yoo, Search based software engineering: techniques, taxonomy, tutorial. Empirical software engineering and verification, Springer, 2010, pp. 1–59.

[38] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using NSGA-III, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (3) (2015) 17.

[39] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, 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.

[40] J.R. Koza, J.R. Koza, Genetic programming: On the programming of computers by means of natural selection 1, MIT press, 1992.

[41] M. Harman, J. Clark, Metrics are fitness functions too. 10th International Symposium on Software Metrics, 2004, pp. 58–69.

[42] A. Ouni, M. Kessentini, H. Sahraoui, M.S. Hamdi, Search-based refactoring: Towards semantics preservation. IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 347–356.

[43] A. Ouni, M. Kessentini, K. Inoue, M.O. Cinnéide, Search-based web service antipatterns detection, IEEE Trans. Serv. Comput. 10 (4) (2017) 603–617.

[44] M. Santolucito, J. Zhang, E. Zhai, R. Piskac, Statically verifying continuous integration configurations, Technical Report (2018).

[45] M. Harman, B.F. Jones, Search-based software engineering, Inf Softw Technol 43 (14) (2001) 833–839.

[46] D.C. Karnopp, Random search techniques for optimization problems, Automatica 1 (2–3) (1963) 111–121.

[47] J. Cervantes, X. Li, W. Yu, Using genetic algorithm to improve classification accuracy on imbalanced data. 2013 IEEE International Conference on Systems, Man, and Cybernetics, IEEE, 2013, pp. 2659–2664.

[48] M. Li, H. Zhang, R. Wu, Z.-H. Zhou, Sample-based software defect prediction with active and semi-supervised learning, Automated Software Engineering 19 (2) (2012) 201–230.

[49] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Software Eng. 33 (1) (2006) 2–13.

[50] F. di Pierro, S.-T. Khu, D.A. Savic, An investigation on preference order ranking scheme for multiobjective evolutionary optimization, IEEE Trans. Evol. Comput. 11 (1) (2007) 17–45.

[51] E. Zitzler, M. Laumanns, L. Thiele, Spea2: improving the strength pareto evolutionary algorithm, TIK-report 103 (2001).

[52] M. Harman, The current state and future of search based software engineering (2007) 342–357.

[53] A. Ouni, Search based software engineering: challenges, opportunities and recent applications. Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, 2020, pp. 1114–1146.

[54] D. Hadka, MOEA Framework, (http://moeaframework.org/). Accessed: 2020-01-01.

[55] D. Hadka, Moea Framework User Guide (2014).

[56] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, V.G. Da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, IEEE Trans. Evol. Comput. 7 (2) (2003) 117–132.

[57] N. Riquelme, C. Von Lücken, B. Baran, Performance metrics in multi-objective optimization. 2015 Latin American Computing Conference (CLEI), IEEE, 2015, pp. 1–11.

[58] M. Li, X. Yao, Quality evaluation of solution sets in multiobjective optimisation: a survey, ACM Computing Surveys (CSUR) 52 (2) (2019) 1–38.

[59] A. Ni, M. Li, Poster: Acona: Active online model adaptation for predicting continuous integration build failures. 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE, 2018, pp. 366–367.

[60] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: synthetic minority over-sampling technique, Journal of artificial intelligence research 16 (2002) 321–357.

[61] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering. International Conference on Software Engineering (ICSE), 2011, pp. 1–10.

[62] Y. Zhang, M. Harman, G. Ochoa, G. Ruhe, S. Brinkkemper, An empirical study of meta-and hyper-heuristic search for multi-objective release planning, ACM Transactions on Software Engineering and Methodology (TOSEM) 27 (1) (2018) 1–32.

[63] M. Paixao, M. Harman, Y. Zhang, Y. Yu, An empirical study of cohesion and coupling: balancing optimization and disruption, IEEE Trans. Evol. Comput. 22 (3) (2017) 394–414.

[64] F. Ferrucci, M. Harman, J. Ren, F. Sarro, Not going to take this anymore: multi-objective overtime planning for software engineering projects. 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 462–471.

[65] N. Almarimi, A. Ouni, S. Bouktif, M.W. Mkaouer, R.G. Kula, M.A. Saied, Web service api recommendation for automated mashup creation using multi-objective evolutionary search, Appl Soft Comput 85 (2019) 105830.

[66] S. Boukharata, A. Ouni, M. Kessentini, S. Bouktif, H. Wang, Improving web service interfaces modularity using multi-objective optimization, Automated Software Engineering 26 (2) (2019) 275–312.

[67] F. Wilcoxon, S. Katti, R.A. Wilcox, Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test, Selected tables in mathematical statistics 1 (1970) 171–259.

[68] A. Vargha, H.D. Delaney, A critique and improvement of the CL common language effect size statistics of mcgraw and wong, Journal of Educational and Behavioral Statistics 25 (2) (2000) 101–132.

[69] S. Nejati, G. Gay, Search-Based software engineering: 11th international symposium, SSBSE 2019, Tallinn, Estonia, August 31–September 1, 2019, proceedings 11664, Springer Nature, 2019.

[70] S.W. Thomas, H. Hemmati, A.E. Hassan, D. Blostein, Static test case prioritization using topic models, Empirical Software Engineering 19 (1) (2014) 182–212.

[71] S. Scalabrino, G. Grano, D. Di Nucci, R. Oliveto, A. De Lucia, Search-based testing of procedural programs: Iterative single-target or multi-target approach?. International Symposium on Search Based Software Engineering Springer, 2016, pp. 64–79.

[72] G.H. John, P. Langley, Estimating continuous distributions in bayesian classifiers, arXiv preprint arXiv:1302.4964 (2013).

[73] M. Choetkiertikul, H.K. Dam, T. Tran, T. Pham, A. Ghose, T. Menzies, A deep learning model for estimating story points, IEEE Trans. Software Eng. 45 (7) (2018) 637–656.

[74] F. Zhang, Q. Zheng, Y. Zou, A.E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 309–320.

[75] K. Janssen, K. Moons, C. Kalkman, D. Grobbee, Y. Vergouwe, Updating methods improved the performance of a clinical prediction model in new patients, J Clin Epidemiol 61 (1) (2008) 76–86.