A Machine Learning Approach to Improve the Detection of CI Skip Commits

Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab, *Senior Member, IEEE*

Abstract—Continuous integration (CI) frameworks, such as Travis CI, are growing in popularity, encouraged by market trends towards speeding up the release cycle and building higher-quality software. A key facilitator of CI is to automatically build and run tests whenever a new commit is submitted/pushed. Despite the many advantages of using CI, it is known that the CI process can take a very long time to complete. One of the core causes for such delays is the fact that some commits (e.g., cosmetic changes) unnecessarily kick off the CI process. Therefore, the main *goal* of this paper is to automate the process of determining which commits can be CI skipped through the use of machine learning techniques. We first extracted 23 features from historical data of ten software repositories. Second, we conduct a study on the detection of CI skip commits using machine learning where we built a decision tree classifier. We then examine the accuracy of using the decision tree in detecting CI skip commits. Our results show that the decision tree can identify CI skip commits with an average AUC equal to 0.89. Furthermore, the top node analysis shows that the number of developers who changed the modified files, the CI-Skip rules, and commit message are the most important features to detect CI skip commits. Finally, we investigate the generalizability of identifying CI skip commits through applying cross-project validation, and our results show that the general classifier achieves an average 0.74 of AUC values.

Index Terms—Continuous integration, travis CI, build status, machine learning

1 Introduction

Continuous integration (CI) is the process that provides developers with the ability to automatically build and test projects on every commit. In fact, prior work showed that CI increases developers' productivity and helps improve software quality [62]. Therefore, some of the biggest software companies such as Microsoft, Netflix, Facebook, and Google significantly use CI process [10], [44], [50], [69]. Moreover, open source projects have recently started using CI processes thanks to the popularity of online CI services such as Travis CI [5], [29].

However, like any solution, CI processes have several drawbacks. CI processes can take a very long time to complete [41], especially for large projects [10]. This can be particularly problematic for developers who kick off the CI process after each commit. The long completion time is mainly because the CI process needs to build and test in a clean container from scratch every time a new commit is pushed to the repository. This overhead can affect both, the speed of software development and the productivity of the developers (Duvall *et al*. [12], p. 87).

Due to the increased adoption of CI processes, most of the previous work focused on the study of its usage and benefits (e.g., [19], [36], [62]), reasons for failing builds [52], and even detect and fix CI configuration errors [19]. However, very few studies try to improve the efficiency of the CI process, especially for large projects that use CI processes. Recently, Abdalkareem *et al*. [1] investigated the reasons why developers skip the CI process and found that amongst other developers CI skip commits to saving development time and computation resources. In their effort to help developers automatically flag commits that can be CI skipped, Abdalkareem *et al*. [1] proposed a rule-based technique that achieves moderate performance and it is highly prone to false negative. A major problem with using the rule-based technique is that it is general and it does not consider the specific characteristics of a project. In the case of detecting CI skip commits, for example, different projects have different types of files used for documentation and also their team's development context is different.

Therefore, the main *goal* of our work is to automatically detect and label commits that can be CI skipped. To do so, we propose the use of machine learning (ML) techniques to detect CI skip commits. We started by analyzing ten open source Java projects that are using Travis CI as a CI service where their developers have explicitly CI skipped a number of their commits. Then, we extract 23 commit-level features and build a decision tree classifier to determine whether a commit is a CI skip commit or not. To evaluate the effectiveness of using the ML technique, we perform an empirical study to answer the following questions:

RQ1: Can we accurately detect CI skip commits using machine learning? We built a decision tree classifier using 23 features extracted from project repositories and compare its performance to the baseline, which is the ratio of CI skip commits in the studied projects. The results show that a decision tree

• *The authors are with the Data-driven Analysis of Software (DAS) Lab, Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H3G 1M8, Canada. E-mail: {rab_abdu, s_mujahi, eshihab}@encs.concordia.ca.*

Manuscript received 3 June 2019; revised 22 Nov. 2019; accepted 13 Jan. 2020. Date of publication 16 Jan. 2020; date of current version 10 Dec. 2021. (Corresponding author: Rabe Abdalkareem.) Recommended for acceptance by H. Rajan. Digital Object Identifier no. 10.1109/TSE.2020.2967380

classifier achieves higher F1-score of 0.79 (AUC = 0.89), on average. This improvement equates to an average improvement of 2.4X by the decision tree classifier, when it is compared to our baseline.

Then, we examine the most important features used by the decision tree to indicate CI skip commits, in order to provide insights to practitioners as to what factors best indicate commits that can be CI skipped. This leads us to our *RQ2: What features are the best indicators of CI skipped commits?* We find that the number of developers who changed the modified files, the devised rules, and the commit message written by the developers are the best indicators of CI skip commits. Lastly, to examine the generalizability of our proposed technique, we determine the effectiveness of our ML technique in predicting CI skip commits in across-projects. Particularly, we ask the *RQ3: How effective is the machine learning classifier when applied on cross-projects?* We build a general classifier and evaluated its performance using cross-projects validation. Our results show that our classifier achieves an average F1-score of 0.55 (AUC = 0.74), which is lower than the within-project classifiers, however, it is still a promising classifier that can be practically used. The result also show that cross-project classifier performance corresponds to an average F1-score improvement of 1.5X over our baseline.

To sum up, this work makes the following contributions:

- To the best of our knowledge, this is the first work to propose an ML technique to detect commits that can be CI skipped.
- We used 23 commit-level features in a machine learning classifier (decision tree) to detect CI skip commits.
- We conduct an empirical study to evaluate the performance of ML techniques where we examine 1) the performance of the CI skip classifier within-projects and across-projects. 2) We identify the most important indicators of CI skip commits used by the machine learning classifier.

The remainder of the paper is organized as follows. Section 2 presents our qualitative study about practitioners' perspective on the skipping of the CI process and discuss motivating examples. We describe our case study design in Section 3. We present our case study results in Section 4. We discussed our results in Section 5. The work related to our study is discussed in Section 6 and the threats to validity in Section 7. Section 8 concludes the our paper.

2 Motivation

At the time that the study was conducted there is little written in the scientific literature on CI skip commits except for the recent work by Abdalkareem *et al*. [1]. However, during our investigation of the topic, we found a rich discussion thread on the Github issue tracker of the Travis CI project. In that thread, many developers argue that the CI process should not be run on every commit, and Travis CI developers are asked to provide an advanced mechanism to automatically CI skip specific commits [35]. To gain more insight into the problem of CI skipping a commit from real-world developers, we manually examine the developers' opinions that are provided in a free-text. We first mined the thread from GitHub and collected the developers' contributions to the thread, which are in free-text. We also collected some background metadata of the developers that include the number of repositories they own, stars they received, and their followers on Github. Then, we perform a formal qualitative analysis, to elicit and understand how important the problem of CI skipping a commit is for developers [51]. The first two authors read the free-text contribution of the developers, and iterative coding them.

TABLE 1
Background of the Thread Participants

Number of	Minimum	Median	Average	Maximum
Repositories	3	69	97.21	427
Followers	1	69	158	1,900
Stars	0	75	648.9	9,000

We found that there are 47 unique developers that participated in the thread. Table 1 shows the minimum, median, average, and maximum number of repositories, stars, and followers for each developer. We observe that the developers have a high number of owned repositories with 69 on average. We also see a similarly high number for the stars and followers. These observations indicate that the developers are expert with knowledge of using different development techniques that include the use of the CI processes.

Our manual investigation reveals that 98 percent of the developers believe that it is essential to have the ability to CI skip some commits automatically. For example, one developer explained the need for such a feature as follows; *"Both software and documentation are in the same repository. I have a very large and slow testsuite which tests the software. I don't want to re-run that testsuite just because I changed something in the documentation."* [59]. Other developers reported some specific cases as one developers said: *"I also need this feature but my use case is a bit more complex. I run tests on both Linux and OSX (via travis) and I would like to skip the build on either or OSX or Linux depending on which files were changed in the GIT commit."* Additionally, another developer said *"Something like this would be really useful for our use-case case too where we have both client and server in the same repo. Having the ability to disable the server-side tests when only the client files have changed would save a lot of time running pointless tests. As you can see in our travis.yml we have a matrix for various node/db configs and then an include that adds the environment-independent client test suite at the end. Every time we make a client-only change we have 1hr of total build time on unrelated test suites before the client test suite even begins."* [2]. In addition, developers believe that building on every commit can be a waste of resources, which has been reported by Abdalkareem [1] as well, for example, another developer supports this by saying: *"This means that unwanted tasks still have to spin up a machine and do an initial git clone, which is rather wasteful of limited resources (the "skipped" tasks are shown as taking about 30s; idk how much hidden backend cost there is)."* [23]

To illustrate the problem that developers face when identifying and flagging commits as CI skip commits, we show real-life examples taken from the GeoServer project [45]. Table 2 shows the details of four Travis CI builds from the GeoServer project. For each build, the table shows the build

TABLE 2
Examples of Travis CI Builds Taken From the GeoServer Project

Build Id	#Jobs	Total Duration	Build Reason
11,097	3	1hr 37min 6sec	"Update PULL_REQUEST_TEMPLATE.md"
10,995	3	1hr 34min 56sec	"Adding some info to documentation, about the NoData indexer's propert"
10,904	3	1hr 38min 58sec	"fixed formatting"
10,508	3	1hr 24min 24sec	"Merge pull request #3325 from aaime/h2_migrate_docs"

The total duration of a build is presented in Hours (hr), Minutes (min), and Seconds (sec).

number, the number of jobs, the total duration, and the commit message written by the developer for the change that triggered the Travis CI build. As shown in the table, the four builds run for approximately an hour and a half and when we look at the commit message, we observe that all these builds have been triggered due to simple changes in the project, i.e., either a change in the readme file or formatting the source code. For example, Fig. 1 shows the details of the CI build #10904 (anonymized). It first shows the diff of the commit change (left side) that triggers the CI process from GitHub. It also shows the details of the build result from Travis CI. We see that the commit changes that trigger the CI process are just a simple formatting change (1) and this simple formatting requires more than an hour and a half of total CI run time (2).

Overall, the richness and thoughtfulness of this discussion highlighted the need for developing an advanced technique to detect CI skip commit automatically. In the next sections, we describe our ML technique to automatically detect CI skip commits that do not need to trigger the CI process.

3 Case Study Design

The main *goal* of our study is to determine the commits that can be CI skipped. To achieve this goal, we propose the use of machine learning techniques. We begin by collecting data of projects whose developers use the CI skip feature and explicitly skip a number of their commits, which we use as a labeled dataset. Then, we mine the selected software repositories to extract commit-level features and use them as dependent variables in our machine learning classifier. In the following subsections, we detail our labeled dataset and data extraction and processing steps.

3.1 Test Dataset

In this subsection, we introduce the dataset used in our study. To determine how effective the ML techniques are in detecting CI skip commits, we need to have a labeled test dataset that we can apply the ML techniques on. Our first criteria to build our testing dataset is to have projects with a sufficient number of CI skipped commits that are explicitly marked by developers. We resort to using the dataset provided in our previous work [1]. To build that dataset, we started by identifying projects that have a sufficient number of their commits skip the CI process. To do so, we searched GitHub for non-forked Java projects that use Travis CI and where their developers use the `'[ci skip]'` feature. To search for these projects on GitHub, we first use the BigQuery GitHub dataset, which provides a web-based console to allow the execution of a SQL query on the GitHub data [22]. We searched for all non-forked projects that 1) are written in the Java programming language; 2) contains the keywords `'[ci skip]'` or `'[skip ci]'` in more than 10 percent of their commit messages; and 3) do not exist in the TravisTorrent dataset [6]. We choose projects with $>$ 10 percent of skipped commits, since this is a good indicator that the developers of those projects are somehow familiar with the Travis CI skip feature. We also exclude the projects that exist in the TravisTorrent dataset, since in our prior work [1], we extracted the CI skip rules based on a manual analysis of the projects in the TravisTorrent dataset. It is important to note that using our previous build dataset [1] allows us to put our analysis in perspective and to be able to compare the performance of our ML technique to the state-of-the-art. The dataset contains ten open source projects written in Java that have some of their commits labeled as CI skip commit by developers. Also, it is worth mentioning that we only consider commits after the introduction of

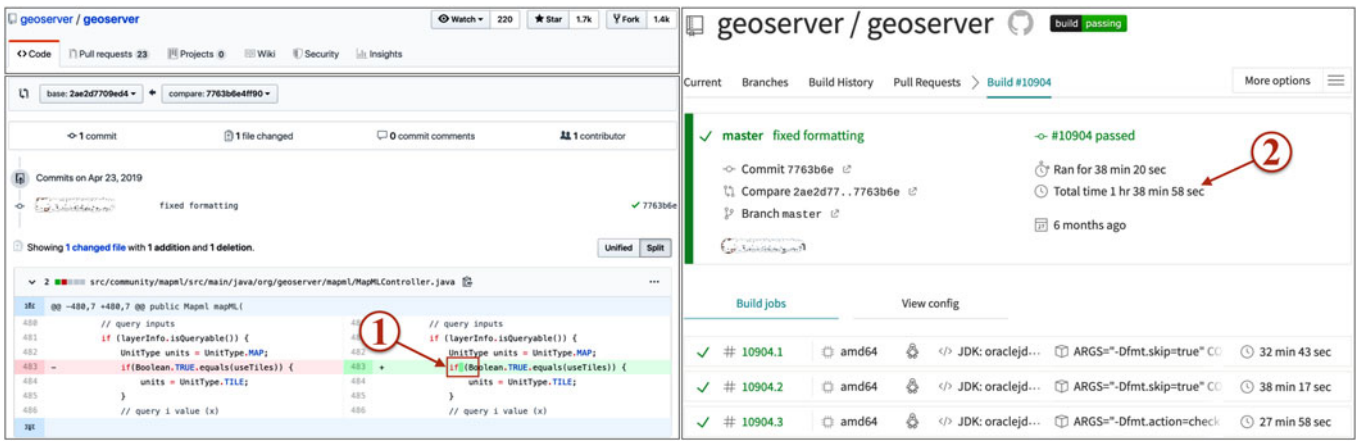

Fig. 1. Example of cosmetic code change and its build result mined from the GeoServer project. On the left, the figure shows the commit change difference and the type of code change (part (1)). On the right, the figure shows the build result of GeoServer project on this commit. It also shows that the time that token to finish the build (part (2)).

TABLE 3
Shows Projects in the Testing Dataset

Project	LOC	Classes	Methods	Time Period	# of Commits§	% of Skipped Commits
TracEE Context-Log	12,386	345	1,426	2014-12-12 - 2017-03-14	216	29.63
SAX	7,200	75	438	2015-06-20 - 2017-03-23	372	23.66
Trane.io Future	5,185	119	898	2017-02-05 - 2017-08-04	247	18.62
Solr-iso639-filter	2,335	82	299	2013-08-20 - 2015-06-16	408	41.42
jMotif-GI	4,689	54	336	2015-06-20 - 2017-03-29	345	12.17
GrammarViz	6,798	89	403	2014-08-22 - 2017-03-13	417	13.67
Parallec	12,452	242	1,112	2015-10-28 - 2017-07-09	129	56.59
CandyBar	13,702	206	1,025	2017-02-22 - 2017-09-26	242	69.01
SteVe	14,078	358	1,332	2015-10-07 - 2016-11-16	298	19.46
Mechanical Tsar	4,008	108	478	2015-05-06 - 2017-03-19	388	34.54
Average	**8,283.30**	**167.80**	**774.70**	-	**306.20**	**31.88**
Median	**6,999.00**	**113.50**	**688.00**	-	**321.50**	**26.65**

§*# of commits after the introduction of Travis CI service to the project.*

Travis CI to a project since it presents the period of the project life where its developers use the CI skip features.

Table 3 presents some statistics of our studied projects. It shows the number of commits after the introduction of Travis CI service to the project, and the percentage of CI skipped commits in the ten Java projects. In total, there are 3,062 (average = 306.20) commits in all the studied projects. The table also shows that the percentage of explicitly CI skipped commits range between 12.17 - 69.01 percent for the projects.

3.2 Features for CI Skip Commit Classification

Since our *goal* is to perform commit-level predictions to decide which commits should be CI skipped, we resorted to using the most commonly used commit-level features. These features were used in the past to predict, for example, whether a commit would introduce a bug or not [17], [30], [32], [40], [67]. However, we believe that some of these features can be used to determine the level of complexity of a commit, hence, providing useful information as to whether a commit should be CI skipped or not. We refer interested readers to the original paper by Kamei *et al.* [32] for the full details about these features. However, to make this paper self-sufficient, we describe the features used briefly. We extract these features using CommitGuru tool [49].

In addition to the previously mentioned features, we also extracted five specific features based on the rules that are related to CI skip commits. These features have been proposed by Abdalkareem *et al.* [1], and they are devised based on a manual examination of more than 1,800 CI skipped commits from the TravisTorrent dataset [6]. Finally, we employ the change's description, written by the developer as a feature, since prior work showed that textual descriptions of a commit provide valuable and discriminative information about changes [60]. For the sake of completeness, we present the features used in Table 4 and describe them next.

Diffusion Features. Diffusion features present the propagation of the change through the software system. We computed four measures to present diffusion features which are the number of modified subsystems, directories, files, and entropy. The entropy measures how the change is distributed across the different files.

Size Features. These features measure the size of code modified in a commit. For each commit, we measure the number of lines add and deleted, and number of lines of code in a file before the change. We also counted the number of different type of files changed in a commit using file extensions.

Purpose Features. For each change, we categorize the type of maintenance performed by the changes. The purpose group contains three features: the type of maintenance activities, if the change is a defect fixing changes, and if the commit is a merge commit. We extract the first two features by analyzing the commit message of the change and search for some keywords. For identifying defect fixing commits, we look for keywords such as *"fix"*, *"bug"*, or *"defect"* and their variants (i.e., capitalized first letter or, all capitalized). A similar approach to determine defect fixing changes was used in other work [32], [56]. Also, for identifying the maintenance activities, we use a different set of keywords such as *"refactoring"*, *"improve"*, or *"enhance"* and their variants [49]. For identifying merge commit, we consider the commit that have two parents as a merge commit.

History Features. Three features are computed in the history features which capture the history of a module modified in the change. We measure history features using; 1) the number of developers that changed the modified file in the past. 2) The average time interval between the current and the last time these files were modified. 3) the number of unique last changes of the modified files.

Experience Features. Experience features estimate the experience of the developer who made the change. We again use past changes to compute the developer experience features. We compute three features related to developer experience that are; 1) developer experience of change by computing the number of changes made by the developer before the current change; 2) recent developer experience, measured as $\frac{1}{1+n}$, where n is measured in years; 3) developer experience on subsystem modified by the change, which measures the number of previous changes performed by the developer to the changed subsystem(s).

CI-Skip Rule Features. We use the five features that are related to CI skip commits that are devised by Abdalkareem *et al.* [1]. These features are rules that are extracted based on analyzing more than 1,800 explicit CI skip commits marked by

TABLE 4
Features Used to Identify CI Skip Commits and the Rational of Using Them

Dim.	Name	Definition (Inspired from)	Rational
Diffusion	*NS*	Number of modified subsystems ([17], [32], [40]).	Changes that touch many subsystems are not, in general, trivial changes which are likely not to be CI skipped.
	ND	Number of modified directories ([17], [30], [32], [40], [67]).	Build commits that are based on changing several directories are more likely not to be CI skipped.
	NF	Number of modified files ([17], [30], [32], [40], [67]).	Build commits that changes multiple files are more likely to modify source code files which should be built.
	EN	Distribution of modified code across each file (i.e., Entropy) ([17], [30], [32], [67]).	Changes with high entropy are more likely to have a large number of file changes that make the changes more complicated and have high chances to modify the source code.
Size	*LA*	Lines of code added ([17], [30], [32], [40], [67]).	It is clear that the more lines of code added shows the need to build the system and run the test cases.
	LD	Lines of code deleted ([17], [30], [32], [40], [67]).	The more lines of code deleted the more the need to run the continuous integration process.
	LT	Lines of code in a file before the change ([17], [30], [32], [67]).	The size of the source code file that changed in a commit indicates the need for running the continuous integration process.
	TFC	The type of files change in the commit identified by their extensions([20], [26]).	The type of files changed in the commit indicate the need for the CI process to be run (i.e., changes related to source code files).
Purpose	*FIX*	Whether or not the change is a defect fix ([17], [30], [32], [67]).	Fixing a defect means that more code is modified or added that need to be tested after the change through running the continuous integration process.
	MR	If the commit is a merge commit.	The number of parents of a commit shows if the commit is a merge commit which required to running the continuous integration process to show that the merged code integrated safely to the project.
	CFT	The type of maintenance activities preformed.	Type of maintenance activities indicates the number of changes to the software project. As a result, it suggests the need to build and test the project.
History	*NDEV*	The number of developers that changed the modified files ([17], [30], [32], [40], [67]).	The larger the number of developers changed the modified files, the risker the changes are that require running the continuous integration process to make sure the changes do not break the project or fail tests.
	AGE	The average time interval between the last and the current change ([17], [30], [32], [40], [67]).	The lower the AGE, the more likely a defect will be introduced, and it shows the need to build and test the code.
	NUC	The number of unique changes to the modified files ([17], [30], [32], [40], [67]).	The larger the NUC, the more likely that commit introduces a defect to the projects which show the need to build and test the project.
Experience	*EXP*	Developer experience ([17], [30], [32], [40], [67]).	Experienced developers are more likely to be knowledgeable about the type of changes that can be CI skipped.
	REXP	Recent developer experience ([17], [30], [32], [40], [67]).	A developer who has often modified the files recently is more familiar with source code and recognize the type of changes that can be CI skip.
	SEXP	Developer experience on a subsystem ([17], [30], [32], [40], [67]).	A developer who is familiar with the subsystems modified by a commit is more likely to CI skip commits that do not need to be build or test.
Text	*CM*	Terms appear in the commit messages. We weight the terms using tf-tdf after removing English stop words.	Commit message are more likely to contain useful information about the type of changes in the commit (e.g., changes a readme files).
CI-skip Rules	*DOC*	If the commit changes non-source code files ([1]).	Based on the devised rules if a commit changes mainly non-source code, it is likely to be CI skipped.
	MET	If the commit modifies meta files such as git ignore ([1]).	Based on the devised rules if a commit changes mainly meta files, it is likely to be CI skipped.
	COM	If the commit modifies source code comments ([1]).	From the devised rules if the changes in a commit are mainly related to source code comments, it is likely to be a CI skip commit.
	FRM	If the commit changes the formating of the source code ([1]).	From the devised rules if the changes in a commit are mainly related to formatting source code, it is likely to be a CI skip commit.
	BLD	If the commit modifies the version in the project ([1]).	From the devised rules if the changes in a commit are mainly preparing for release or changing release version, it is likely to be a CI skip commit.

developers. We provide a detailed description and a rationale of each of features in Table 4.

Textual Features. For each commit message that is written by the developer, we preprocess the commit message by doing the following: (1) we remove any identification of skip commit by deleting the keyword '`[ci skip]`' or its variation '`[skip ci]`' from the commit message content, (2) we remove English stop words, (3) we remove numbers, or special characters, and (4) we lowercase the terms. We then convert the terms that appear in the commit messages into numerical values. To do so, we weigh the terms using Term Frequency - Inverse Document Frequency (TF/IDF) weighting scheme. TF/IDF is a statistical measure that is used to show how important a word is to a document in a collection. Table 4 also presents the rationale for using this feature.

3.3 Classification Models

To perform our prediction, we leveraged a decision tree classifier. We chose to use a decision tree to classify whether a commit is a CI skip commit or not since it provides an intuitive and easy to explain classification model. This enables developers to easily understand why a decision to skip a commit was made. In contrast, most other classifiers tend to produce "black box" models that do not explain which features affect the classification outcome [34]. We also discuss the use of other ML classifier in Section 5.1.

To build the decision tree classifier, we used the built-in libraries provided by Weka, a Java-based machine learning framework [24]. The well-known algorithm for building decision trees is C4.5 [46], which first creates a decision-tree based on the feature values of the training data such that internal nodes denote the different features, the branches correspond to the value of a certain features and the leaf nodes correspond to the classification of the dependent variables. The decision tree is made recursively by identifying the feature(s) that discriminate the various instances most clearly, i.e., having the highest information gain [46]. Once a decision tree is made, the classification for a new instance is done by checking the respective features and their values. In our work, we implement the decision tree classifier using the default parameter settings on top of Weka implementation [24].

Dealing With Imbalanced Data. One common issue with software engineering data is data imbalance [57]. Data imbalance occurs when one class occurs much more than the other in the dataset. In this case, the decision tree trains to learn from the features affecting the majority cases than the minority cases. As shown in Table 3, our testing dataset has on average only 31.88 percent (median = 26.65 percent) that labeled as CI skip commits, which means that the majority of the cases are not skipped commits. To deal with the imbalance problem in our dataset, we rely on the use of the re-sampling algorithm in Weka [24] to perform the under-sampling on the training data. We use the option "-B 1.0" (i.e., ensure the class distribution is uniform in the output data), and "- Z 100" (i.e., the final sample size is the same as the original dataset). It is important to note that we only applied the re-sampling step on the training dataset. We did not re-sample the testing dataset since we want to evaluate our classifier in a real-life scenario, where the data is imbalanced.

3.4 Performance Evaluation

To evaluate the performance of the ML technique, we compute precision, recall, and F1-score. In our study, recall is the percentage of correctly classified *Skip Commits* relative to all of the commits that are actually skipped (i.e., $Recall = \frac{TP}{TP+FN}$). Precision is the percentage of detected skipped commits that are actually skipped commits (i.e., $Precision = \frac{TP}{TP+FP}$), where TP is the number of skip commits that are correctly classified as CI skip commits; FP denotes the number of non CI skip commits classified as skip commits; and FN measure the number of classes of actual CI skip commits that identified as non skipped commits. We then combine both precision and recall of our classification technique in detecting skip commits using the well-known F1-score (i.e., $F1\text{-}score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$).

In addition, to mitigate the limitation of choosing a fixed threshold when calculating precision and recall, we also present the Area Under the ROC Curve (AUC) values. AUC is computed by measuring the area under the curve that plots the true positive rate against the false positive rate, while varying the threshold that is used to determine if a commit is classified as skipped or not. The main merit of the AUC is its robustness toward imbalanced data since its value is obtained by varying the classification threshold over all possible values. The AUC has a value that ranges between 0-1, and a larger AUC value indicates better classification performance.

4 Case Study Results

In this section, we present the results of our case study with respect to the three research questions. For each research question, we present the motivation for the question, the approach to answer the question, and results.

4.1 RQ1: Can we Accurately Detect CI Skip Commits Using Machine Learning?

Motivation. Since building the project after every commit can be wasteful, we want to be able to effectively determine commits that can be CI skipped. Even though prior work propose a rule-based techniques to automatically detect CI skip commit [1], their evaluation shows a moderate performance. Thus, the main goal of this research question is to investigate the use of supervised machine learning technique to assist developers in automatically identifying CI skip commits.

Approach. For each project in the testing dataset (Section 3.1), we use the selected 23 commit-level features shown in Table 4 to train a decision tree classifier to predict whether a commit is a CI skip commit or not. However, since a single source code commit can perform more than one change, some of the individual CI-Skip rules can be true and the rest is not true for the same commit. For example, a commit can change the source code comments and at the same time add source code. To make sure that our CI-Skip rules flag the CI skip commit, we combine the rules and then add its value as one feature to the decision tree. After that, for each project, we use 10-fold cross validation [14]. First, we divide the dataset for each project into 10 folds. We use 9 folds (i.e., 90 percent of the data) to train the decision tree, and use the remaining one fold to evaluate

TABLE 5
Performance of the Decision Tree Technique

Project	Precision	Recall	F1-Score (Relative F1-Score)	AUC
TracEE Context-Log	0.95	0.95	0.94 (2.6X)	0.96
SAX	0.76	0.92	0.83 (2.6X)	0.93
Trane.io Future	0.86	0.91	0.87 (3.2X)	0.94
Solr-iso639-filter	0.88	0.93	0.90 (2.0X)	0.95
jMotif-GI	0.75	0.86	0.78 (3.9X)	0.91
GrammarViz	0.57	0.81	0.66 (3.1X)	0.91
Parallec	0.91	0.90	0.89 (1.7X)	0.88
CandyBar	0.88	0.80	0.83 (1.4X)	0.82
SteVe	0.54	0.70	0.59 (2.1X)	0.83
Mechanical Tsar	0.59	0.67	0.62 (1.5X)	0.74
Average	**0.77**	**0.85**	**0.79 (2.4X)**	**0.89**
Median	**0.81**	**0.88**	**0.83 (2.3X)**	**0.91**

the performance of the classifier. We run this process 10 times for each fold (i.e., $1 \times$ 10-folds).

Finally, to evaluate the performance of the decision tree classifier in detecting CI skip commits, we compute the well-known evaluation metrics which are precision and recall and their combination F1-score as explained in Section 3.4 ten times for each fold. Also, we computed the Area Under the ROC Curve values. Then, to come up with one value for the ten runs, we compute the average of the evaluation metrics for 10-folds ten times (i.e., $1 \times$ 10-fold) for every project in our testing dataset.

Since one of main goals of using machine learning techniques is to improve the detection of CI skip commits, we measure how much better the performance of the decision tree is compared to the baseline for each project. In our case, the baseline classifier is a classifier where the precision is equal to the ratio of CI skip commits in the overall dataset (since this is the likelihood that a CI skipped commit will be detected through a baseline/random classifier) and the recall is 0.5 (since there are two classes to choose from, skipped or not) [53]. Then, we compare the values of F1-score of the decision tree against the baseline by calculating the relative F1-score (i. e., $Relative\ F1-score = \frac{decsion\ tree\ F1-score}{Baseline\ F1-score}$). Relative F1-score shows how much better our classifier does compared to the baseline. For instance, if a baseline achieves a F1-score of 10 percent, while the decision tree classifier achieves a F1-score of 20 percent, then the relative F1-score is $\frac{20}{10} = 2X$. In other words, the decision tree classifier performs twice as accurate as the baseline classifier. It is important to note that the higher the relative F1-score value the better the classifier is at detecting CI skip commits.

In addition, to put our analysis in perspective, we compare the performance of our ML technique to the state-of-the-art technique, which is the rule-based technique proposed by Abdalkareem *et al.* [1]. To do so, we run the rule-based technique on our testing dataset by using the publicly available tool developed by Abdalkareem *et al.* [1] and compare the results of the two techniques.

Results. Table 5 presents the precision, recall, F1-score (relative F1-score shown in parentheses), and AUC value of the decision tree classifier for the ten studied projects in our test dataset. First, the precision values obtained by the decision tree classifier range between 0.54 and 0.95, with an average of 0.77 (median = 0.81) while the recall values range between 0.67 and 0.95 with average value of 0.85 (median = 0.88). Also, the F1-score achieves values in the range between 0.59 and 0.94, with an average of 0.79 and median of 0.83. The values of the area under the ROC curve indicate that the decision tree is effective in detecting CI skip commits, achieving AUC values ranging between 0.74 - 0.96 with an average of 0.89.

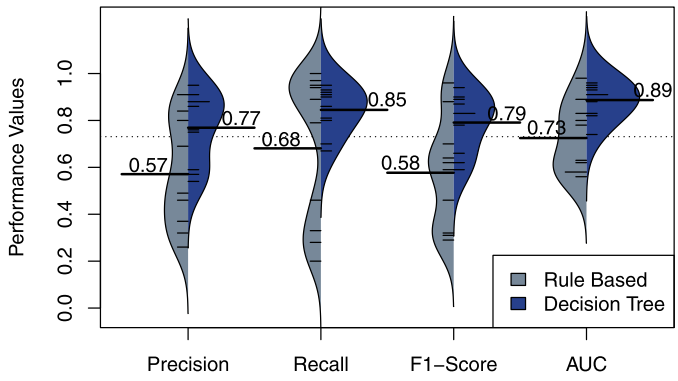

Fig. 2. Beanplots comparing the distributions of Precision, Recall, F1-score values for Rule-based and Decision Tree techniques. Dotted horizontal line shows the overall median.

Also, Table 5 shows the relative F1-score when comparing the performance of decision tree to the baseline. The computed relative F1-scores show an improvement of 2.4X on average over the baseline.

In particular, for all the ten projects, the decision tree outperforms the baseline with relative F1-score values ranging between 1.4X and 3.9X.

To compare the performance of the decision tree classifier against the rule-based technique [1], we present the distribution of precision, recall, F1-score, and AUC values for the ten studied projects. Fig. 2 summarizes the distribution of precision, recall, F1-score, and AUC values for the ten studied projects (each compared beanplot shows the distribution of evaluation metrics for the decision tree and the rule-based techniques). The figures show that in all cases the decision tree classifier outperform the rule-based approach. Fig. 2 shows that the decision tree classifier outperforms the rule-based techniques achieving an improvement of 56 percent on average (median = 41 percent).

Our decision tree classifier achieves an average F1-score of 0.79 (AUC of 0.89). Additionally, our results show that decision tree classifier can effectively improve the detection of CI skip commits with an average relative F1-score of 2.4X compare to the baseline while it achieves an improvement of 56 percent compare to state-of-the-art technique.

4.2 RQ2: What Features are the Best Indicators of CI Skipped Commits?

Motivation. So far, we saw that compared to our baseline, the decision tree classifier provides an improvement of 2.4X on average. Now, we want to better understand what features inputed to the decision tree assist in achieving such a high performance. For example, is it the CI-skip rules devised by Abdalkareem *et al.* [1] or other features are the best indicators of CI skip commits? Thus, we analyze the built decision trees to answer this research question.

TABLE 6
Top Node Analysis for Each Project

Project	Feature	Occurrence
TracEE Context-Log	*CI − Skip Rules*	8
	CM	2
SAX	*NDEV*	5
	CI − Skip Rules	2
	NUC	1
	CM	1
	TFC	1
Trane.io Future	*EXP*	9
	NUC	1
Solr-iso639-filter	*NUC*	9
	SEXP	1
jMotif-GI	*NDEV*	10
GrammarViz	*CI − Skip Rules*	10
Parallec	*CM*	10
CandyBar	*TFC*	9
	CM	1
SteVe	*SEXP*	10
Mechanical Tsar	*NDEV*	6
	LD	4

Approach. To identify the features that are the most important indicators of whether a commit is a CI skip commit or not, we perform Top Node Analysis [25], [54], [55]. Top Node Analysis is used to identify the most important features that are good indicators of whether or not a commit is CI Skip. In a decision tree classifier, the most important feature is the root node in the tree, and less important features will be placed in lower levels of the tree. Thus, the higher the level of the feature the more important it is. To determine the important features in detecting CI skip commits, we extract the features used in the top level of the created trees. Since we use 10-fold cross validation, the analysis produces 10 trees for each project. Thus, the same feature can occur multiple time in the top level in the ten created decision trees (i.e., 1x10-fold cross validation). That is, the number of occurrences of a feature in the top level also indicates how important that feature is in identifying CI skip commits and the higher the number of occurrence the more important it is. For example, if a feature appears in the top level of all 10 decision trees of a project, we say that this feature has an occurrence number of ten and it is the most important feature in detecting CI skip commits for this project. For each project, we extract the top level features and the number of their occurrence in the ten built decision trees.

In addition, to give a more general view of the most important indicators of CI skip commits in all studied projects, we aggregate the results of the top node analysis of each project and sum the number of occurrences of each feature. The higher the total number of occurrences of a feature in all the ten studied projects the more important it is in detecting CI skip commits. It should be mentioned that our process generated 100 decision tree (10x10) for all the projects, hence the sum up of features occurrences are out of 100.

Results. Table 6 shows the result of the top node analysis for each project. From Table 6, we note that there is no single feature that appeared to be important in all the studied projects. However, there are some features that are the strongest indicators of whether a commit is a CI skip commit or not for some projects. For example, the number of developers that changed the modified files is the most important feature for the `jMotif-GI` project. A closer examination of the `jMotif-GI` project, we found that the developers of this project CI skip commits when they change a specific file (readme file) in the project and there is only one developer who is responsible for performing all the changes for this file. The commit message is also the most important feature for four projects; `Parallec`, `TracEE Context-Log`, `SAX`, and `CandyBar` with top node analysis values equal to 10, 2, 1, and 1. For these projects a closer examination reveals that the content of the commit messages have a clear indication of the type of changes. For example, in a commit change in the `SAX` project, a developer changes the Java documentation in the source code and write *"updating javadocs"* as a commit message. Another feature that seems to be important as well is the CI-skip rule feature, which appears in the top node analysis of three projects namely `TracEE Context-Log`, `GrammarViz`, and `SAX` with top node analysis values equal to 8, 10, and 2 respectively. For example, a commit in the project `GrammarViz` added an image file to the project and the developer decided to CI skip the commit. On the other hand there are some features that appears in one project. For example, the feature number of lines deleted that appears in the `Mechanical Tsar` project. Again we examine some of the CI skip commit in this project. We found that the project's developers sometimes optimize the declaration of the used APIs and delete source code lines.

TABLE 7
Top Node Analysis for All the Projects

Feature	Number of Projects	Total Number of Occurrence§
NDEV	3	21
CI − Skip Rules	3	20
CM	4	14
NUC	3	11
SEXP	2	11
TFC	2	10
EXP	1	9
LD	1	4
Total	**-**	**100**

§*The number of occurrence is out of 100.*

Table 7 shows the result of the top node analysis for all the ten projects in our testing dataset. First, as the table shows in all the projects the number of developers who changed the modified files (*NDEV*), the devised rules (*CI − Skip Rules*), and the commit message (*CM*) are the most important features in identifying CI skip commits. They have occurrence values equal to 21, 20, and 14 for the number of developers who changed the modified files, devised CI-skip rules, and commit messages, respectively.

While it makes sense that the CI-skip rules are important in detecting CI skip commits as these rules are defined based on analyzing CI skip commits flagged by developers [1], it is more important to understand why the other two features (number of developers who change the modified file and the commit messages) are important as well.

To investigate why the number of developers who changed the modified files as feature is more important and effective than the other features in detecting CI skip commits, we also manually examine the built decision trees. We observed that when there is only one developer who changes the modified files, the classifier makes the decision that this commit is a CI skip commit. This is due to the practice that some files are special and modified usually by one developer. For example, in the `SAX` project, there is only one developer who is responsible for modifying the readme file.

Similarly, we investigate why the commit message is also important in detecting CI skip commits. We found that developers mention the type of activities performed by the change. For example, the following message "update docs" indicates that the developer changed documentation files in this commit.

Other features also important in indicating CI skip commits. The number of unique changes to the modified files (*NUC*) is a important indicator which appears in the top node analysis of three projects. The developer experience on a subsystem (*SEXP*) and the type of files change (*TFC*) appear in the top node analysis in two projects. Finally, developer experience (*EXP*) and number of lines of deleted code (LD) appear only on the top level of the top node analysis in one project each.

The results of the top node analysis indicate that number of developers who changed the modified files, the CI-skip rules, and commit message are the most important features in identifying CI skip commit using decision tree classifier.

4.3 RQ3: How Effective is the Machine Learning Classifier When Applied on Cross-Projects?

Motivation. Building an ML classifier to identify CI skip commits requires having labeled data to train on. However, many projects do not have sufficient historical labeled data to build a classifier (e.g., unlabeled, small or new project). Thus, we cannot train a machine learning classifier to detect CI skip commits on data from these projects. In this research question, we investigate to what extent and with which accuracy a commit can be automatically classified to be a CI skip commit using a cross-project machine learning classification.

Approach. To better understand the generalizability of the results achieved by the training classifier on data from one project and apply it on another project, we conduct a cross-project validation. In particular, we experiment 10 fold cross-projects validation. We conducted an experiment that trains a classifier on data from nine projects and uses the resulting classifier to determine whether a commit is CI skipped or not in the remaining one project, similar to prior work [3], [18]. We repeat this process ten times, once for each project in the testing dataset. To build the classifier, we train the decision tree machine learning classifier following the same approach described early in Section 4. Once again, we employ the well-known evaluation metrics where we compute the precision, recall, F1-score, and Area Under the ROC Curve values to measure the performance of the classifier. Finally, to examine the performance of the cross-project classifier with respect to the random baseline, we compute the relative F1-score.

Result. Table 8 presents the results of our experiment. It presents precision, recall, F1-score (relative F1-score shown in parentheses), and the AUC values for each project. Table 8 shows that the the general classifier achieve a performance of AUC value of 0.74 and 0.70 for the average and median, respectively. Five projects out of ten show good performance results. The projects, `TracEE Context-Log`, `SAX`, `jMotif-GI`, `Parallec`, and `GrammarViz` achieve high AUC values ranging between 0.70 - 0.92. Other projects show a moderate performance including `Solr-iso639-filter`, `SteVe`, and `CandyBar` with value of 0.69, 0.61 and 0.62 for the AUC.

TABLE 8
Performance of Cross-Projects Classification

Project	Precision	Recall	F1-Score (Relative F1)	AUC
TracEE Context-Log	0.43	0.57	0.48 (1.3X)	0.70
SAX	0.77	0.80	0.79 (2.1X)	0.92
Trane.io Future	0.19	0.41	0.25 (0.7X)	0.51
Solr-iso639-filter	0.49	0.89	0.63 (1.8X)	0.69
jMotif-GI	0.58	0.78	0.66 (1.7X)	0.90
GrammarViz	0.63	0.86	0.72 (1.9X)	0.93
Parallec	0.85	0.93	0.88 (2.5X)	0.92
CandyBar	0.78	0.52	0.62 (1.8X)	0.62
SteVe	0.41	0.20	0.26 (0.7X)	0.61
Mechanical Tsar	0.38	0.18	0.24 (0.7X)	0.56
Average	**0.55**	**0.61**	**0.55 (1.5X)**	**0.74**
Median	**0.54**	**0.68**	**0.63 (1.8X)**	**0.70**

However, there are two projects that have low F1-score and AUC values. In particular, project (`Trane.io Future`) has a poor performance with AUC value equal to 0.51. To investigate why this project achieves such a poor performance, we first look at the top node analysis from research question two, we found that at the top level of the top node analysis, developer experience (*EXP*) and the number of unique changes to the modified files (*NUC*) are the two important indicators to classify CI skip commits in this project. However, these two features are not found in the other projects except the number of modified files (*NUC*) appears in two projects, `Solr-iso639-filer` and `SAX` with occurrence values equal to 9 and 1, respectively. The project `Mechanical Tsar` also has a low F1-score and AUC values. For this project, top node analysis shows that two features are important, which are the number of developers who changed the modified files (*NDEV*) and number of lines of deleted code (*LD*). This shows that when building the decision tree on data from the other nine projects, the decision tree does not have sufficient evidence (information gain value) that these features are important for the examined project under test.

Finally, when we compare the performance of the cross-projects classifier to the baseline, our results show that cross-projects classifier shows an improvement of 50 percent on average over the baseline.

The results show that cross-projects machine learning classifier can provide comparable performances to within-project classifier of CI skip classification. For eight projects out of the ten studied projects, cross-projects classifier achieves AUC values range between 0.61 - 0.92 with an overall average equal to 0.74.

TABLE 9
Performance of Using Different Classifiers to Detect CI Skip Commits

Project	Decision Tree		Logistic Regres.		Naive Bayes		Random Forest		Random Tree		Decision Table		SVM	
	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC
TracEE Context-Log	0.94 (2.6X)	0.96	0.94 (2.5X)	0.98	0.96 (2.6X)	1.00	0.97 (2.6X)	**1.00**	0.94 (2.5X)	0.97	0.94 (2.6X)	0.96	0.95 (2.6X)	0.97
SAX	0.83 (2.6X)	0.93	0.78 (2.5X)	0.90	0.78 (2.5X)	0.95	0.90 (2.8X)	**0.98**	0.77 (2.4X)	0.92	0.86 (2.7X)	0.94	0.86 (2.7X)	0.92
Trane.io Future	0.87 (3.2X)	0.94	0.74 (2.8X)	0.94	0.83 (3.1X)	0.94	0.99 (3.7X)	**1.00**	0.64 (2.4X)	0.88	0.82 (3.0X)	0.93	0.91 (3.4X)	0.98
Solr-iso639-filter	0.90 (2.0X)	0.95	0.83 (1.8X)	0.85	0.76 (1.7X)	0.89	0.93 (2.1X)	**0.99**	0.65 (1.4X)	0.74	0.87 (1.9X)	0.93	0.87 (1.9X)	0.89
jMotif-GI	0.78 (3.9X)	0.91	0.57 (2.8X)	0.89	0.52 (2.6X)	0.95	0.86 (4.3X)	**0.97**	0.81 (4.0X)	0.90	0.71 (3.6X)	0.92	0.67 (3.3X)	0.86
GrammarViz	0.66 (3.1X)	0.91	0.68 (3.2X)	0.93	0.55 (2.6X)	0.93	0.78 (3.7X)	**0.96**	0.59 (2.8X)	0.88	0.63 (3.0X)	0.92	0.77 (3.7X)	0.91
Parallec	0.89 (1.7X)	0.88	0.81 (1.5X)	0.80	0.89 (1.7X)	0.91	0.95 (1.8X)	**0.97**	0.85 (1.6X)	0.78	0.91 (1.7X)	0.92	0.97 (1.8X)	0.96
CandyBar	0.83 (1.4X)	0.82	0.79 (1.4X)	0.76	0.88 (1.5X)	0.86	0.87 (1.5X)	**0.86**	0.75 (1.3X)	0.77	0.83 (1.4X)	0.81	0.83 (1.4X)	0.78
SteVe	0.59 (2.1X)	0.83	0.43 (1.6X)	0.72	0.48 (1.7X)	0.75	0.52 (1.9X)	**0.85**	0.32(1.1X)	0.64	0.61 (2.2X)	0.85	0.47 (1.7X)	0.69
Mechanical Tsar	0.62 (1.5X)	0.74	0.48 (1.2X)	0.58	0.59 (1.4X)	0.73	0.67 (1.6X)	**0.82**	0.58 (1.4X)	0.71	0.59 (1.4X)	0.70	0.57 (1.4X)	0.67
Average	**0.79 (2.4X)**	**0.89**	**0.71 (2.1X)**	**0.84**	**0.72 (2.1X)**	**0.89**	**0.84 (2.6X)**	**0.94**	**0.69 (2.1X)**	**0.82**	**0.78 (2.4X)**	**0.89**	**0.79 (2.4X)**	**0.86**
Median	**0.83 (2.3X)**	**0.91**	**0.76 (2.2X)**	**0.87**	**0.77 (2.1X)**	**0.92**	**0.89 (2.4X)**	**0.97**	**0.70 (2.0X)**	**0.83**	**0.83 (2.4X)**	**0.92**	**0.85 (2.3X)**	**0.90**

5 Discussion

In this section, we first examine the use other machine learning classifiers to detect CI skip commits. Second, we examine the amount of effort that can be saved by using our prediction of CI skip commits.

5.1 What Classifiers Provides the Best Accuracy for Identifying CI Skip Commits?

So far, we used decision tree classifier to determine CI skip commits, and it showed an effective improvement over the baseline. However, the decision tree is not the only supervised machine learning classifier. Thus, in this subsection, we investigate the use of other machine learning classifiers and compare their performance in identifying CI skip commits. We use our dataset prepared in Section 3.1 and the same approach described in Section 3.2 to train other six machine learning classifiers namely Logistic Regression, Naive Bayes, Random Forest, Random Tree, Decision Table, and Support Vector Machines. We choose to examine these ML classifiers, since they have different assumptions on the analyzed data, as well as having different characteristics in terms of execution speed and dealing with overfitting [9]. Also, they have been commonly used in the past in other software engineering studies (e.,g. [3], [4], [21], [27], [31], [32], [47], [58], [63]).

To provide a comprehensive comparison of the different classifiers, we compare them in two scenarios. First, we perform a within-project evaluation, where the classifiers are trained and tested using non-overlapping data from the same project. Second, we perform a cross-project evaluation, where the classifier is trained on data from several projects and tested on a completely different project. Finally, to examine the performance of within-project and cross-project classification of the different ML classifiers with respect to the random baseline, we compute the F1-score (relative F1-score) and AUC score as well.

Within-Project Classification. Table 9 shows the F1-score (relative F1-score shown in parentheses) and AUC values for the examined six classifiers and the decision tree as well. As Table 9 shows, on average, random forest produces the highest F1-score values with an average of 0.84 (median = 0.94), while the random tree classifier achieves the lowest performance with an average F1-score of 0.69 (median of 0.70), across all of the studied projects. The other classifiers achieve better performance than the random tree with F1-score values ranging between 0.71 - 0.79. This corresponds to a significant improvement ranging between 2.10X - 2.60X, on average, in F1-score over the baseline.

With average AUC scores ranging between 0.84 - 0.94, all the machine learning classifiers perform significantly high. *The highest AUC values achieved is produced again by the random forest classifier*. For example, seven projects in the testing dataset have AUC values greater than 0.95. The results suggest that the random forest classifier is the best machine learning classifier to detect CI skip commits.

*Cross-Project Classification.*Table 10 shows the results in terms of F1-score and AUC that are achieved by the seven classifiers for the cross-project validations. Once again, *the random forest classifier achieves the best results*, followed by support vector machines, and decision tree classifiers, with 0.59, 0.58, and 0.55 average F1-score values and 0.81, 0.72, and 0.74 average AUC values, respectively. The other classifiers also show moderate performance by achieving average AUC values range between 0.67 - 0.71.

Interesting are the cases of the `SAX`, `jMotif-GI`, `GrammarViz`, `CandyBar`, `Solr-iso639-filter`, `TracEE Context-Log`, and `Parallec` projects where the random forest classifier is able to achieve a high performance value ranging between 0.76 - 0.97 for the AUC values. Also, there are three projects that produce poor results which are `Trane.io Future`, `Mechanical Tsar`, and `SteVe`.

5.2 How Much Effort Can be Saved by Using ML Technique to CI Skip Commits?

Thus far, we have shown that our classification technique can accurately flag commits that should be CI skipped, however, one lingering question is—how much effort can we actually save using our classifiers? Of course, automatically detecting commits that can be CI skipped can reduce the amount of resources needed for the CI process, consequently speeding up the overall development cycle, making code reach the customers faster. In this subsection, we investigate the amount of effort that can be saved by applying our classifier to nineteen projects from the TravisTorrent dataset [6].

TABLE 10
Performance of Using Different Classifiers to Detect CI Skip Commits Cross-Project Validation

Project	Decision Tree		Logistic Regres.		Naive Bayes		Random Forest		Random Tree		Decision Table		SVM	
	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC	F1 (R.-F1)	AUC
TracEE Context-Log	0.48 (1.3X)	0.70	0.20 (0.5X)	0.78	0.83 (2.3X)	0.92	0.79 (2.1X)	**0.97**	0.61 (1.7X)	0.75	0.63 (1.7X)	0.79	0.55 (1.5X)	0.69
SAX	0.79 (2.1X)	0.92	0.68 (1.8X)	0.81	0.46 (1.2X)	0.93	0.87 (2.3X)	**0.97**	0.76 (2.0X)	0.93	0.53 (1.4X)	0.81	0.81 (2.1X)	0.91
Trane.io Future	0.25 (0.7X)	0.51	0.31 (0.8X)	0.55	0.14 (0.5X)	0.36	0.25 (0.7X)	**0.52**	0.20 (0.5X)	0.49	0.19 (0.5X)	0.46	0.22 (0.6X)	0.52
Solr-iso639-filter	0.63 (1.8X)	0.69	0.41 (1.2X)	0.57	0.56 (1.6X)	0.66	0.50 (1.4X)	**0.77**	0.33 (1.0X)	0.56	0.44 (1.3X)	0.56	0.67 (1.9X)	0.74
jMotif-GI	0.66 (1.7X)	0.90	0.45 (1.2X)	0.74	0.35 (0.9X)	0.91	0.78 (2.0X)	**0.93**	0.64 (1.6X)	0.88	0.68 (1.7X)	0.88	0.66 (1.7X)	0.85
GrammarViz	0.72 (1.9X)	0.93	0.51 (1.3X)	0.79	0.31 (0.8X)	0.91	0.84 (2.1X)	**0.97**	0.67 (1.7X)	0.93	0.66 (1.7X)	0.90	0.75 (1.9X)	0.91
Parallec	0.88 (2.5X)	0.92	0.86 (2.4X)	0.85	0.84 (2.3X)	0.95	0.90 (2.5X)	**0.93**	0.81 (2.3X)	0.83	0.80 (2.2X)	0.82	0.90 (2.5X)	0.86
CandyBar	0.62 (1.8X)	0.62	0.50 (1.5X)	0.64	0.79 (2.3X)	0.77	0.45 (1.3X)	**0.76**	0.52 (1.5X)	0.57	0.58 (1.7X)	0.69	0.63 (1.9X)	0.63
SteVe	0.26 (0.7X)	0.61	0.29 (0.8X)	0.48	0.33 (0.9X)	0.55	0.26 (0.7X)	**0.59**	0.32 (0.8X)	0.59	0.04 (0.1X)	0.58	0.32 (0.9X)	0.58
Mechanical Tsar	0.24 (0.7X)	0.56	0.39 (1.1X)	0.51	0.54 (1.5X)	0.56	0.26 (0.7X)	**0.65**	0.41 (1.1X)	0.58	0.27 (0.7X)	0.56	0.33 (0.9X)	0.51
Average	**0.55 (1.5X)**	**0.74**	**0.46 (1.3X)**	**0.67**	**0.52 (1.4X)**	**0.75**	**0.59 (1.6X)**	**0.81**	**0.53 (1.4X)**	**0.71**	**0.48 (1.3X)**	**0.71**	**0.58 (1.6X)**	**0.72**
Median	**0.63 (1.8X)**	**0.70**	**0.43 (1.2X)**	**0.69**	**0.50 (1.3X)**	**0.84**	**0.64 (1.7X)**	**0.85**	**0.57 (1.6X)**	**0.67**	**0.56 (1.5X)**	**0.74**	**0.65 (1.8X)**	**0.72**

We evaluate the effort-saving by applying the trained classifier on projects from the TravisTorrent dataset [6]. We select the twenty projects with the highest number of build commits from the TravisTorrent dataset. We choose these projects because we want to examine the effort-saving on real projects. For every project, we first clone the project and then we extract the features described in Section 3.2. We were not able to extract features from one project that has many branches that we could not extract the commit-level features for. Thus, we ended up with nineteen projects to analyze. Table 11 lists the statistics of the studied Java projects that include number of LOCs, classes, methods and time period. It also shows the list of the analyzed projects and number of commits in each project that are computed after the introduction of Travis CI. We then applied the cross-project classification. We start by building a random forest classifier by training the classifier on all the labeled data from the ten projects in our testing datasets from Section 3.1. We choose to use random forest classifier since it is the most accurate classifier when it uses in cross-project classification. We applied the trained random forest classifier on all the commits in the nineteen projects and flag commits that should be CI skipped. Finally, we report the percentage of commits that are flagged to be CI skipped.

The last two columns of Table 11 show the number and percentage of the commits that are flagged to be CI skipped in all the selected nineteen projects. It shows that the percentage of detected CI skip commits range between 1.46 – 11.06, and with an average of 4.98 percent of the commits in the studied projects can be CI skipped. As the table shows, for eight projects, the percentage of CI skipped commits is more than 5 percent ranging between 5.42 – 11.06 percent. This finding shows the importance of skipping commits that unnecessary kick off the CI process, which in return helps developers speed up the development and release process.

TABLE 11
Performance of Using Random Forest Classifier on Unseen DataSet

Project	LOC	Classes	Methods	Time Period	# of Commits[§]	Predicted	Percentage
Gradle Build Tool	217,009	7,401	30,375	2012-10-25 - 2017-12-13	38,613	3,018	7.82
Apache Jackrabbit Oak	432,413	6,864	39,406	2012-06-18 - 2017-12-08	13,399	348	2.60
LanguageTool	85,734	1,380	6,626	2014-05-14 - 2017-12-14	12,823	1,418	11.06
grayLog	105,380	2,114	10,734	2012-04-16 - 2017-12-13	10,858	589	5.42
sonarQube	525,849	8,315	56,320	2015-03-17 - 2017-12-14	10,857	170	1.57
Singularity Mesos	58,902	815	6,454	2014-08-22 - 2017-12-14	7,610	358	4.70
Cloudify	78,162	1,103	6,920	2012-06-01 - 2014-11-27	6,864	100	1.46
Structr	154,405	2,118	13,428	2012-12-20 - 2017-12-14	6,618	270	4.08
Orbeon Forms	66,976	1,266	7,194	2012-03-13 - 2017-12-13	6,465	249	3.85
ownCloud App	40,030	509	2,909	2014-01-21 - 2017-12-14	4,765	303	6.36
BuildCraft	155,649	2,555	15,254	2014-05-04 - 2017-12-12	4,308	90	2.09
Grails Web Application	52,691	755	5,944	2014-05-12 - 2017-12-12	4,305	452	10.50
FenixEdu Academic	303,621	4,808	37,598	2013-05-23 - 2017-11-20	4,067	253	6.22
Unidata's THREDDS	474,494	5,686	43,281	2013-10-01 - 2017-12-14	4,008	142	3.54
DSpace Platform	172,969	2,102	12,813	2013-07-25 - 2017-12-14	3,956	141	3.56
Traccar	48,559	1,058	2,869	2015-03-03 - 2017-12-15	3,852	82	2.13
Brightspot	45,117	815	3,645	2015-06-12 - 2017-12-08	3,547	82	2.31
Apache Fineract	193,841	2,774	15,008	2013-02-15 - 2016-02-01	3,116	261	8.38
Linux.org.ru	23,201	354	2,067	2012-10-09 - 2017-12-14	2,763	192	6.95
Average	**170,263**	**2,779**	**16,781**	**-**	**8,041.79**	**448.32**	**4.98**
Median	**105,380**	**2,102**	**10,734**	**-**	**4,765**	**253**	**4.08**

§*# of commits after the introduction of Travis CI to the project.*

To make sure that the ML technique detects the correct commits to be CI skip commits, we manually examined all the identified CI skip commits using the ML technique. The ML technique (random forest classifier) detects 8,518 commits as CI skip commits. To examine these commits, the first author manually examined all the identified CI skip commits to make sure that ML classifier detects commits that are unnecessary to kick off CI process CI for. Since this process is subject to human bias and to examine the validity of our manual examination, we had the second author (PhD student) independently examine a statistically significant sample of the detect CI skip commits by our ML classifier to reach a 95 percent confidence level using a 5 percent confidence interval. The second author manually examined the statistically significant sample of 368 detected CI skip commits. We then use the Cohen's Kappa coefficient to evaluate the level of agreement between the two authors [15]. In our analysis, we found the level of agreement between the annotators to be 0.89, which is considered to be excellent agreement [16]. Finally, for the cases that the two authors did not agree on, they hold a final round to clarify their classification and come up with an agreement.

Out of the 8,518 identified CI skip commit by the ML technique, we found that 7,267 (85.31 percent) are commits that can be CI skip. Also, for 1,251 commits that the ML technique identified to be CI skip, the manual investigation reveals that these may not be CI skip or false positive. From our manual analysis, We found that false positive cases are mainly related to four categories. The most frequent false positive cases are related to code changes to other programming languages (in 76.7 percent of the commits) that include JavaScript code (38.7 percent), Groovy code (26.9 percent), Scala (9.4 percent), Python (1.3 percent), and Ruby (1.2 percent). From our analysis of the commits related to the category mentioned above, we observed that these commits are related to projects that contain other programming language code than Java. For example, the project `sonarQube` has some code related to front-end web development and contains JavaScript code. Also, we found that 13 percent of the false positive cases are related to changes in the configuration and infrastructure code, such as Docker files. Interestingly, we found that 6.9 percent of the false positive commits change simple Java code, where a developer renames one variable in a Java file. Lastly, we found a small number of the flagged CI skipped commits (2.8 percent) are related changes that modify SQL statements in SQL code files.

6 Related Work

The *goal* of this paper is to use ML to classify commits that can be CI skip using features extracted from commit-level granularity. Thus, we divide the prior work into two parts; work related to the continuous integration and work related to commit-level prediction.

6.1 Continuous Integration

Most recently, Abdalkareem *et al.* [1] proposed a rule-based techniques to detect CI skip commits. Their technique is based on a set of rules devised through a manual examination of approximately 1,800 explicit CI skip commits by developers. Their evaluation reveal that the rule-based technique achieves an average Areas Under the Curve equal to 0.73. Despite the fact that we share the same goal with the work of Abdalkareem *et al.* [1], which detecting CI skip commits, our work is different in that we focus exclusively on the use of ML techniques to detect CI skip. In addition, we compare the performance of using ML techniques to Abdalkareem *et al.* [1] and found that ML achieve higher performance.

Previous studies also investigated the possibility of predicting whether a CI process will result in a pass or a fail build. Ni and Li [43] proposed the use of cascade classifiers to predict build failures. Their classifiers achieve an AUC of 0.75, which is higher than other basic classifiers they compare to, such as decision trees and Naive Bayes. Their analysis also shows that historical committers and project statistics are the best indicators of build statuses. Hassan *et al.* [26] propose to predict whether the build outcome will be successful of not using a combined features of build related metadata and code change information of commits. Their experiment is based on a dataset of more than 250,000 build instances over a period of 5 years recorded by Travis CI and their results show that the proposed prediction model can predict build outcome with an F1-score of more than 0.89. Xie and Li [65] proposed an advanced semi-supervised AUC optimization method that deals with the imbalance and unlabeled build data. They then examined the performance of the proposed method on build data from eight projects and found that their method achieves an average AUC of 0.74. Also, Ghaleb *et al.* [20] investigated the characteristics of the builds that are associated with the long build durations. To do so, they built logistic models to model long build durations across projects and one of their main findings suggests that caching content that rarely changes can speed up builds. Our work differs from these studies in that we focus exclusively on the detecting commits that can be CI skipped. In many ways, our study complements prior studies since we share the same goal of reducing CI time, however, our focus is different given our exclusive focus on CI skip commits.

Related to improving the CI tools, Brandtner *et al.* [7] introduced a tool called SQA-Mashup that integrates data from different CI tools to provide a comprehensive view of the status of a project. Campos *et al.* [8] propose an approach to generate unit tests as part of the CI process automatically. Other researchers investigate to enhance communication channel between developers who use CI in their projects. They find that CI provides a mechanism to send notifications of build failures [13], [38]. Downs *et al.* [11] conducted an empirical study by interviewing developers and found that the use of CI substantially affects the team's work-flow. Based on their findings, a number of guidelines were suggested to improve CI monitoring and communication when using CI.

Other work has focused on detecting the status of builds and investigated the reasons for build failures. Rausch *et al.* [48] collected a number of build metrics (e.g., file type and the number of commits) for 14 open-source Java project that uses Travis CI to better understand build failures. Among other findings, their study showed that cosmetic changes sometimes break builds, but this often indicates unwanted flakiness of tests or the built environment. Seo *et al.* [52] studied the characteristics of more than 26 million builds

done in C++ and Java at Google. They found that the most common reason for builds failures is the dependencies between components. Ziftic and Reardon [69] propose a technique to detect fail regression tests of CI build automatically. The technique uses heuristics that filter and rank changes that might introduce the regression. Miller [42] reported the use of continuous integration in an industrial setting and showed that compilation errors, failing tests, static analysis tool issues, and server issues are the most common reasons for build failures. Other studies investigate the cost, benefits, and usage of CI in open source and property projects [28], [29], [36], the quality outcomes for open-source projects that use CI services [62], [68], and configuration problem of CI services [19].

As shown in the work mentioned above, CI can improve the quality and productivities of software development. However, getting results from CI can take considerable time for some projects. Hence, our work addresses this issue by using ML technique to detect commits that can be CI skipped.

6.2 Commit-Level Prediction

A plethora of studies exists that uses machine learning to understand various characteristics of software engineering. In particular, work that predicts the characteristic of software based on commit-level. Hassan and Zhang [25] used classifiers to predict whether a build would pass a certification process. McIntosh *et al.* [39] build a classifier that determines whether or not a software change will be related to built changing. Xia *et al.* [64] propose cross-project build co-change identification classifier to improve the performance of build co-change identification in projects in the initial development phases. Recently, Macho *et al.* [37] improve the existing classifier performance by taking into consideration the use of commit-level details.

Other work aims to predict defect changes. For example, Kim *et al.* [33] classify each software change as buggy or non-buggy by using commit-level metrics including added and deleted source code and textual features in change message. Kamei *et al.* [32] conducted a large-scale empirical study of change-level quality assurance on eleven open source and commercial projects. They aimed to provide an effort-reducing way to focus on predicted defective change. Along the same lines, Yang *et al.* [67] use simple unsupervised classifies approaches for defective change prediction. They observed that simple unsupervised classifier can perform better than supervised classifiers on defective change prediction. Recently, Yan *et al.* [66] examine the use of a machine learning approach to predict whether a source code change contains self-admitted technical debt in the software system.

Along the same line with those mentioned above studied, our work focuses on the prediction of CI skip commits using ML classifiers and commit-level features such as the number of added lines and developer experience. However, our work aims to identify whether or not a commit can be CI skip.

7 Threats to Validity

This section describes the threats to the validities of our study.

7.1 Internal Validity

Concerns with factors that could have influenced our study set up. We rely on the git command line to analyze the history of a project. Developers could modify the history of their git repositories using, `reset` for example. From our analysis, we only found one project that had a modified history. To identify the type of file changes in a commit, we use a list of extensions of the most common file types (e.g., readme files, etc.) from [61]. In some cases, the list of file types we use may not be comprehensive.

7.2 Construct Validity

Considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. The different configurations of Travis CI could affect our results. Our labeled dataset is based on projects having CI skip commits that are explicitly marked as so by developers. In some cases, developers may forget to label commits that should be skipped with `[skip ci]` or `[ci skip]`. To evaluate the machine learning techniques, we selected ten open source Java projects where developers explicitly mark at least 10 percent of the commits as CI skip commits. Additionally, we compare the performance of the ML technique to the defined baseline which is simply the ratio of skipped commits over all the commits in our test dataset. It simply serves as an indicator of what a random guess would do. It is a lower bound, hence we use it to show that at the least, the ML technique is better than this lower bound. In addition, to classify commits that are CI skip commits, we use seven ML classifiers with their default configuration parameters provided by the Weka framework. Changing these parameters may yield different results.

7.3 External Validity

Threats to external validity concern the generalization of our findings. Our study is based solely on Java projects; hence our results may not hold for projects written in other programming languages. However, commit-level features and the ML technique can be easily generalized to other programming languages by analyzing the skip commits of the other projects written in different programming languages. Second, the datasets used in our study present only open source project hosted on GitHub that do not reflect proprietary projects. Furthermore, we examine projects that use Travis CI for their continuous integration services, and different CI platforms could have more advanced features for controlling skip commits. That said, Travis CI is one of the most popular CI services on GitHub that have a basic feature of skipping unrequited build commits.

8 Conclusion

In this paper, we study CI skip commits that developers tend not to build a project on. Specifically, our goal is to use ML techniques to detect commits that can be CI skip. To do so, we propose the use of 23 commit-level features extracted from ten Java projects that use Travis CI. Then, we build a decision tree classifier. We found that the decision tree can effectively improve the detection of CI skip commits with an average F1-measure of 0.79 (median = 0.83). It also achieves an average AUC of 0.89 (median = 0.91), which represent an

improvement of 56 percent on average over the state-of-the-art (the rule-based technique [1]). Regarding the most important features used by the decision tree classifiers to indicate CI skip commits, we found that the number of developers who modified the changed files and CI-Skip rules to be the most important indicators of CI skip commits. Additionally, we investigate the generalizability of detecting CI skip commits when we use cross-projects validation. Our results show that the cross-projects validation achieves on average 0.55 and 0.74 for F1-measure and AUC, respectively.

REFERENCES

[1] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be CI skipped?" *IEEE Trans. Softw. Eng.*, pp. 1–17, 2019.

[2] K. Ansfield, "[ci skip] for the build matrix - issue #4713 - travis-ci/travis-ci," 2015, Accessed: Feb. 10, 2019. [Online]. Available: https://github.com/travis-ci/travis-ci/issues/4713

[3] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 375–385.

[4] L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li, "Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 170–181.

[5] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis CI with GitHub," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 356–367.

[6] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing travis CI and GitHub for full-stack research on continuous integration," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 447–450.

[7] M. Brandtner, E. Giger, and H. Gall, "Supporting continuous integration by mashing-up software quality information," in *Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance Reeng. Reverse Eng.*, 2014, pp. 184–193.

[8] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: Enhancing continuous integration with automated test generation," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 55–66.

[9] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proc. 23rd Int. Conf. Mach. Learn.*, 2006, pp. 161–168.

[10] M. A. Cusumano and R. W. Selby, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York, NY, USA: The Free Press, 1995.

[11] J. Downs, J. Hosking, and B. Plimmer, "Status communication in agile software teams: A case study," in *Proc. 5th Int. Conf. Softw. Eng. Adv.*, 2010, pp. 82–87.

[12] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Boston, MA, USA: Addison-Wesley Professional, 2007.

[13] S. Dsinger, R. Mordinyi, and S. Biffl, "Communicating continuous integration servers for increasing effectiveness of automated testing," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 374–377.

[14] B. Efron, "Estimating the error rate of a prediction rule: Improvement on cross-validation," *J. Amer. Statist. Assoc.*, vol. 78, no. 382, pp. 316–331, 1983.

[15] J. Fleiss, "The measurement of interrater agreement," in *Statistical Methods for Rates and Proportions*. Hoboken, NJ, USA: Wiley, 1981, pp. 212–236.

[16] J. L. Fleiss and J. Cohen, "The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability," *Educ. Psychol. Meas.*, vol. 33, no. 3, pp. 613–619, 1973.

[17] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 72–83.

[18] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 172–181.

[19] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis CI," *IEEE Trans. Softw. Eng.*, vol. 46, no. 1, pp. 33–50, Jan. 2020.

[20] T. A. Ghaleb, D. A. da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2102–2139, Aug. 2019.

[21] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 789–800.

[22] GitHub, "GitHub activity data | marketplace - Google cloud platform," Accessed: Jul. 03, 2019, 2019. [Online]. Available: https://console.cloud.google.com/marketplace/details/github/github-repos?filter=solution-type:dataset&id=46ee22ab-2ca4–4750-81a7-3ee0f0150dcb

[23] J. Graham, "Conditionally ignore some of the builds in a matrix - issue #6685 - travis-ci/travis-ci," 2016, Accessed: Feb. 10, 2019. [Online]. Available: https://github.com/travis-ci/travis-ci/issues/6685

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *ACM SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[25] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 189–198.

[26] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *Proc. 11th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2017, pp. 157–162.

[27] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Autom. Softw. Eng.*, vol. 19, no. 2, pp. 167–199, Jun. 2012.

[28] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 197–207.

[29] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 426–437.

[30] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empir. Softw. Eng.*, vol. 24, no. 5, pp. 2823–2862, Oct. 2019.

[31] H. Iba, "Random tree generation for genetic programming," in *Proc. 4th Int. Conf. Parallel Problem Solving Nature*, 1996, pp. 144–153.

[32] Y. Kamei *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[33] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

[34] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas, "Machine learning: A review of classification and combining techniques," *Artif. Intell. Rev.*, vol. 26, no. 3, pp. 159–190, Nov. 2006.

[35] P. Ladenburger, "Exclude files from triggering a build - issue #6301 - travis-ci/travis-ci," Accessed: Nov. 29, 2017, Jul. 2016. [Online]. Available: https://github.com/travis-ci/travis-ci/issues/6301

[36] M. Leppanen *et al.*, "The highways and country roads to continuous deployment," *IEEE Softw.*, vol. 32, no. 2, pp. 64–72, Mar. 2015.

[37] C. Macho, S. McIntosh, and M. Pinzger, "Predicting build co-changes with source code change and commit categories," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 541–551.

[38] K. Matsumoto, S. Kibe, M. Uehara, and H. Mori, "Design of development as a service in the cloud," in *Proc. 15th Int. Conf. Netw.-Based Inf. Syst.*, 2012, pp. 815–819.

[39] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 241–250.

[40] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, May 2018.

[41] J. Micco, "Tools for continuous integration at Google scale - YouTube," Jun. 2012. [Online]. Available: https://ww.youtube.com/watch?v=KH2 sB1A6lA

[42] A. Miller, "A hundred days of continuous integration," in *Proc. Agile Conf.*, 2008, pp. 289–293.

[43] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 455–458.

[44] C. Parnin *et al.*, "The top 10 adages in continuous deployment," *IEEE Softw.*, vol. 34, no. 3, pp. 86–95, May 2017.
[45] G. project, "geoserver/geoserver: Official geoserver repository," 2014, Accessed: Oct. 07, 2019. [Online]. Available: https://github.com/geoserver/geoserver
[46] R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA, USA: Morgan Kaufmann Publishers, 1993.
[47] M. M. Rahman, C. K. Roy, and R. G. Kula, "Predicting usefulness of code review comments using textual features and developer experience," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 215–226.
[48] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 345–355.
[49] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 966–969.
[50] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm, "Continuous deployment of mobile software at Facebook (showcase)," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 12–23.
[51] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 557–572, Jul. 1999.
[52] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 724–734.
[53] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 62:1–62:11.
[54] E. Shihab *et al.*, "Studying re-opened bugs in open source software," *Empir. Softw. Eng.*, vol. 18, no. 5, pp. 1005–1042, Oct. 2013.
[55] J. S. Shirabad, T. C. Lethbridge, and S. Matwin, "Supporting software maintenance by mining software update records," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2001, pp. 22–31.
[56] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5.
[57] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 12, pp. 1253–1269, Dec. 2019.
[58] F. Thung, D. Lo, L. Jiang, Lucia, F. Rahman, and P. T. Devanbu, "When would this bug get reported?" in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, Sep. 2012, pp. 420–429.
[59] M. Unterwaditzer, "Feature request: Skip build if given set of files didn't change issue #2086 travis-ci/travis-ci," Accessed: Jul. 18, 2018, Mar. 2014. [Online]. Available: https://github.com/travis-ci/travis-ci/issues/2086
[60] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 72–81.
[61] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload–A case study of the gnome ecosystem community," *J. Empir. Softw. Eng.*, vol. 19, no. 4, pp. 955–1008, 2014.
[62] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 805–816.
[63] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proc. 10th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2016, pp. 29:1–29:10.
[64] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering*, 2015, pp. 311–320.
[65] Z. Xie and M. Li, "Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 2875–2881.
[66] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang, "Automating change-level self-admitted technical debt determination," *IEEE Trans. Softw. Eng.*, vol. 45, no. 12, pp. 1211–1229, Dec. 2019.
[67] Y. Yang *et al.*, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 157–168.
[68] Y. Yu, B. Vasilescu, H. Wang, V. Filkov, and P. T. Devanbu, "Initial and eventual software quality relating to continuous integration in GitHub," *CoRR*, vol. abs/1606.00521, 2016.
[69] C. Ziftci and J. Reardon, "Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale," in *Proc. 39th Int. Conf. Softw. Eng. Softw. Eng. Pract. Track*, 2017, pp. 113–122.

Rabe Abdalkareem received the master's degree in applied computer science from Concordia University, Montreal, Canada, and the PhD degree in computer science and software engineering from Concordia University, Montreal, Canada. He is a postdoctoral fellow with the Software Analysis and Intelligence Lab (SAIL), Queen's University, Canada. His research investigates how the adoption of crowdsourced knowledge affects software development and maintenance. His work has been published at premier venues such as FSE, ICSME and MobileSoft, as well as in major journals such as the *IEEE Transactions on Software Engineering*, *IEEE Software*, *Empirical Software Engineering*, and the *Information and Software Technology*. For more information: http://users.encs.concordia.ca/rababdu.

Suhaib Mujahid received the bachelor's degree in information systems from Palestine Polytechnic University, Hebron, and the master's degree in software engineering from Concordia University, Canada, in 2017, where his work focused on detection and mitigation of permission-related issues facing wearable app developers. He is working toward the PhD degree in the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. His research interests include wearable applications, software quality assurance, mining software repositories and empirical software engineering. For more information: http://users.encs.concordia.ca/s mujahi.

Emad Shihab is an associate professor and Concordia University research chair in the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. His research interests are in software engineering, mining software repositories, and software analytics. His work has been published in some of the most prestigious SE venues, including ICSE, ESEC/FSE, MSR, ICSME, EMSE, and TSE. He serves on the steering committees of PROMISE, SANER and MSR, three of the leading conferences in the software analytics areas. His work has been done in collaboration with and adopted by some of the biggest software companies, such as Microsoft, Avaya, BlackBerry, Ericsson, and National Bank. He is a senior member of the IEEE. For more information: http://das.encs.concordia.ca.