Automated Software Engineering (2022) 29:21
https://doi.org/10.1007/s10515-021-00319-5

Improving the prediction of continuous integration build failures using deep learning

Islem Saidani[1] · Ali Ouni[1] · Mohamed Wiem Mkaouer[2]

Received: 16 June 2020 / Accepted: 17 December 2021 / Published online: 20 January 2022

Abstract

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

Keywords Continuous integration · Build prediction · Travis CI · Genetic algorithm · Long short term memory · Machine learning · Hyper-parameters optimization · Concept drift

Extended author information available on the last page of the article

1 Introduction

Continuous integration (CI) (Duvall et al. 2007) is a set of software development practices that are widely adopted in commercial and open source environments (Vasilescu et al. 2015). A typical CI system, such as Travis CI https://travis-ci.org/, a widely-used cloud-based platform for providing CI services to software projects, advocates to continuously integrate code changes, introduced by different developers, into a shared repository branch. The key to making this possible, according to Fowler (Fowler 2006), is automating the process of building and testing, which reduces the cost and risk of delivering defective changes. From the academic side, the study of CI adoption has become an active research topic and it has already been shown that CI improves developers' productivity (Hilton et al. 2016; Saidani et al. 2020, 2022), helps to maintain code quality (Vasilescu et al. 2015; Saidani et al. 2021a, b) and allows for a higher release frequency (Zhao et al. 2017).

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

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

- *Feature engineering:* Traditional ML techniques rely on a set of features manually designed for characterizing a given problem, *e.g.*, CI builds. Generally, the feature engineering task is tedious, time-consuming, error-prone and requires substantial expertise in the field (Li et al. 2018; Shan et al. 2016; Bouktif et al. 2018; Sundsøy et al. 2016). Additionally, the accuracy of prediction models depends highly on the relevance of the selected features. Broadly speaking, the build failure prediction problem is not yet resolved as the reasons behind the build failure is still ambiguous.

- *Temporal information:* Previous work on CI build prediction is focused on TravisTorrent-based measures (*e.g.*, number of all built commits, number of distinct authors, etc.) to predict new CI build results in the future, without taking temporal information into account, *i.e.*, chronological order of CI build outcomes. As a result, these works achieved a limited prediction accuracy. The CI builds outcome data is by nature a time series data (Atchison et al. 2017) where the temporal dimension is of crucial importance. However, such time series data can be highly erratic and complex with much noise and high dimensionality especially with unexpected or repetitive build failures over time (Längkvist et al. 2014).
- *Data imbalance:* Another innate issue to classic ML-based approaches is related to the imbalanced distribution of class examples as failed builds are typically likely to occur less than passed ones (Xie and Li 2018). This challenges their applicability due to the performance bias that can occur when an imbalanced distribution of class examples is used (Bhowan et al. 2011, 2010, 2013). Furthermore, this imbalanced nature of the training data was rarely discussed in existing works. However, in CI context, a good accuracy on the failed builds prediction is more important than the passed builds accuracy.

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

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

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

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

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

The remainder of this paper is organized as follows. Section 2 provides the motivation for time-series prediction then we present an overview of the CI build process as well as LSTM-based modeling. We present our approach in Section 3. Section 4 shows the experimental setup of our empirical study while Section 5 presents the obtained results. Section 6 discusses the results implications on CI developers, researchers and tool builders. Section 7 lists the related work. Section 8 reviews the threats to the validity of our results. Finally, Section 9 concludes the paper and outlines avenues for future work.

2 Motivation and background

In this section, we provide a motivating example. Then, we present a brief backgrounds on CI build process, LSTM-RNN as well as genetic algorithms.

2.1 Motivating example

Figure 1 depicts an example of CI build outcome fluctuations over time for Jruby GitHub project https://github.com/jruby/jruby that uses the Travis CI system. As shown in the figure, we can observe that there exist data patterns and an explicit dependency on the time variable that may have a strong association with build outcome. In practice, it is common that after a CI build failure, developers proceed to take the right actions to fix the cause of the build failure which may lead to a sequence of build failures followed by a succeeded build, as can be seen in Figure 1. Conversely, after a sequence of successful builds, build failures often happen in an unexpected manner over time. Moreover, previously experienced bugs and failures can be the root cause of new failures during the CI build, while other unforeseen failures may happen in an independent manner resulting into temporal dynamic behavior and complex non-linear dependencies between failures. Thus, individual observations, *i.e.*, build results, cannot be predicted independently on each other. This makes the CI build failures a time-series data involving a sequence of observations over regularly spaced intervals. Indeed, time series data consists typically of sampled data points taken from a continuous, real-valued process over time, such as the CI build process.

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

2.2 CI build process

CI aims to build healthier software systems by developing and testing in smaller increments without compromising software quality. The basic notion of CI, as described by Folwer (Fowler 2006) is to support developers' work by automating the code compilation, dependencies collection and tests running. This process is an

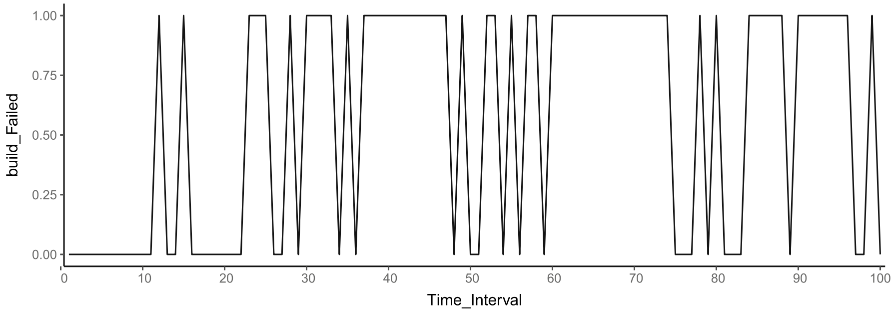


Fig. 1 A snapshot of build outcome fluctuations in JRuby between 2016-08-01 and 2016-08-31."1" means a failed build, and "0" for a passed build

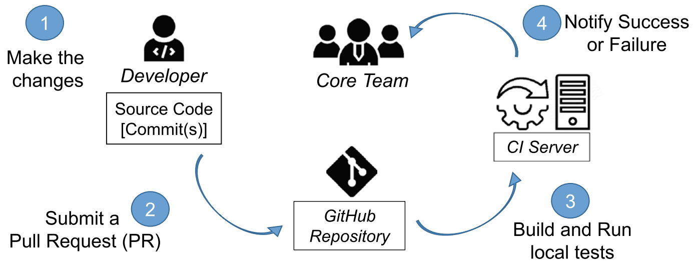


Fig. 2 CI build process

enduring check on the quality of contributed code that mitigates the risk of "breaking the build" as regressions can be detected and fixed immediately.

CI has a well-defined life-cycle when generating builds. The main phases of the CI build life-cycle are depicted in Figure 2. First of all, a contributor forks, *i.e.*, clones, the project repository, makes some changes, as creating a new feature or by fixing some bugs, on the code base (1). When the work is done, the contributor submits the changes to the original repository (2). At this point, the CI service carries out a series of tasks to build and test these changes (3). Then, it provides immediate feedback on the outcome of the test to the core team (4), *i.e.*, developers who dispose of write access to a project's code repository (Vasilescu et al. 2015). When one or more of those tasks fail, the build is considered *failed*, otherwise it will be *passed* and core team members proceed to do a code review and, if necessary, the submitter would be requested for modifications. After a cycle of code reviews, automatic building and testing, if everyone is satisfied, the submitted changes will be merged to the mainline branch.

2.3 Long short-term memory network

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

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

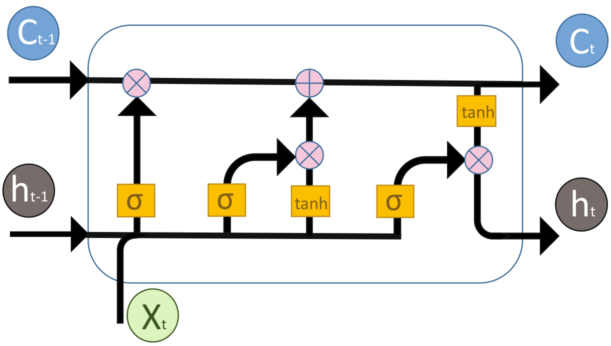


Fig. 3 An overview of information flow in LSTM

which decides what to output from memory. These gates allow efficient management of LSTM internal cell memory.

Figure 3 shows the information flow and the set of gates within LSTM cells http://colah.github.io/posts/2015-08-Understanding-LSTMs/. In this diagram, the pink circles represent point-wise operations (e.g. "+" operation for addition), while the yellow boxes stand for neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied to different locations.

The first step in LSTM is to decide what information to be erased from the cell state. The key to this is the top line of the diagram which represents the memory pipe. The input to this pipe is the old memory (a vector noted C_{t-1}) that passes through the forget gate layer. The latter is controlled by (*i*) a *sigmoid* layer neural network describing how much of each h_{t-1} (the output of the previous LSTM block) and X_t (the input for the current LSTM block) should be passed and (*ii*) a point-wise multiplication operation that is activated when we want to ignore the old memory. The next step is to decide what new information to be stored in the cell state using the input gate layer composed of (*i*) a *sigmoid* layer that decides which values of h_{t-1} and X_t to be updated and (*ii*) a *tanh* neural layer that creates a vector of new candidate values to be added to the cell state. These two elements will be then combined (with + operation) to change the old cell state C_{t-1} to the new cell state C_t. Finally, we need to set the output h_t. We filter C_t through *tanh* operation and multiply it by the output of the *sigmoid* gate, so that we only output the parts we decided to.

There exist several variants of the LSTM architecture for RNNs like GRU (Dey and Salemt 2017) where the structure is similar to a LSTM cell but with only two gates namely update (combination of forget and input gates) and reset gates.

2.4 Genetic algorithm

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

Algorithm 1 High level pseudo code of the Genetic Algorithm (GA)

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

3 Our proposed approach

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

3.1 Methodology overview

The main *goal* of our approach is to help developers cutting off such expenses by effectively predicting the CI build outcome before they happen. Indeed, CI build failures are generally time and resource-consuming and can cause disruptions in the development process and delays in the software product release dates (Ghaleb et al. 2019a). In particular, we handle the problem of CI build failures as a time series prediction problem by estimating the outcome of a given build based on the history of observed build processes. We use LSTM-based Recurrent Neural Network (RNN) to model the CI build process sequential data. Figure 4 provides an overview of our proposed approach.

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

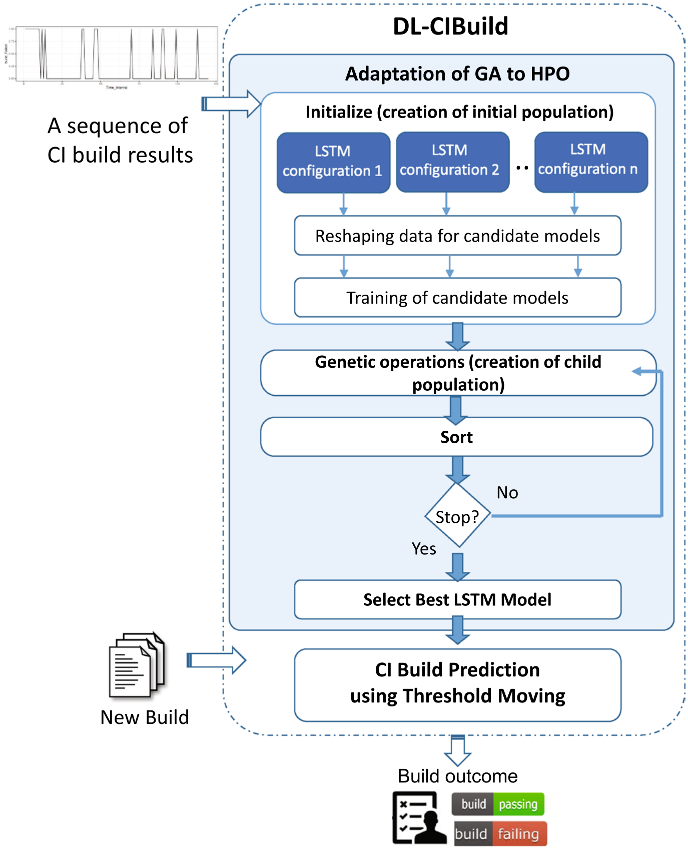


Fig. 4 *DL-CIBuild* overview

would fail or succeed. The hyper-parameters to be tuned, the data preprocessing and the adaptation of GA are described in the following sub-sections.

3.2 LSTM model construction and hyper-parameters tuning

We first need to design and configure our LSTM-RNN model by choosing the architecture, setting up the initial hyper-parameters, and selecting the mathematical components such as activation functions, loss functions, and gradient-based optimizers (Jebnoun et al. 2020; Bouktif et al. 2020). Obtaining good results using LSTM networks is not trivial, as it requires consideration of the tuning of many parameters. Unfortunately, applying LSTM models may not produce acceptable or optimal results, not only because of the nature of the analyzed data but also due to the naive selection of its hyper-parameter values. Table 1 lists the parameters to be optimized for the LSTM model.

To construct the model, the first LSTM parameters to be tuned are the numbers of hidden layers and neurons per layer. As a neural network, LSTM depends highly on

Table 1 List of parameters for the LSTM model

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

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

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

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

3.3 GA adaptation for HPO of LSTM

In this section, we describe how we adapted Genetic Algorithm (GA) for LSTM model configuration problem, then we provide the hyper-parameters to be optimized for our LSTM model. In particular, as described in Section 2.4, for any attempt to use GA in a real-world problem, a number of key elements need to be defined such as the solution representation, genetic operators and the fitness function.

3.3.1 Individual representation

A candidate solution, *i.e.*,a set of parameters configurations, is represented as an array where each cell corresponds to a randomly generated value for a specific parameter as depicted in Figure 5. The initial population is composed of *N* solutions created randomly.

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

Fig. 5 An example of solution encoding for the GA

3.3.2 Genetic operators

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

- *Crossover:* is used to combine the genetic information of two parents. In our adaptation, we use the standard single-point crossover operator. A sub-list is extracted from each parent. Then, the crossover operator exchanges the two sub-lists between parents. Figure 6 shown an example of the crossover operator applied to Parents 1 and 2 to produce two offspring solutions Child 1 and Child 2.
- *Mutation:* The mutation operator aims at adding slight modifications to a candidate solution. In our adaptation, the mutation operator first randomly selects one or more cells from a given candidate solution. Then, the selected cell(s) will be replaced by new randomly generated values. Figure 7 shows an example of mutation operators where three random cells from a parent solution are selection, *i.e.*, the *number of layers*, the *number of epochs* and the *optimizer*, and randomly replaced by other values.

3.3.3 Solution evaluation (fitness function):

Each candidate solution should be evaluated to assess how good it is in solving the problem at hand. An appropriate fitness function should be defined to evaluate the fitness of a candidate solution, *i.e.*, the selected hyper-parameters to build out model. In this paper, we aim to optimize the architecture of our LSTM-RNN model by minimizing the validation loss (Li et al. 2020).

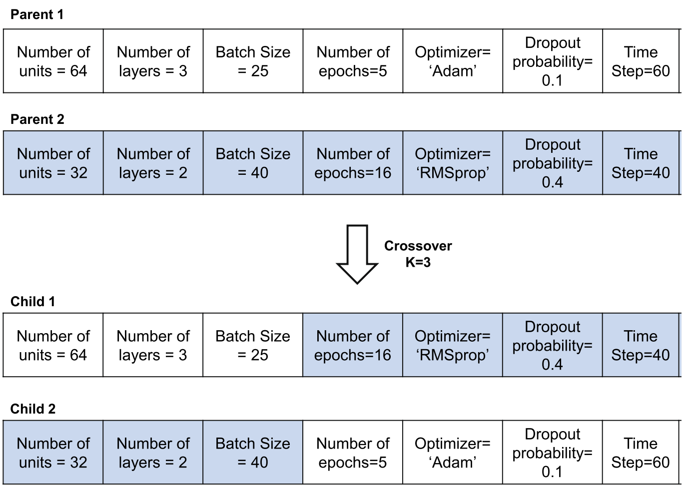


Fig. 6 An example of crossover operator for the GA

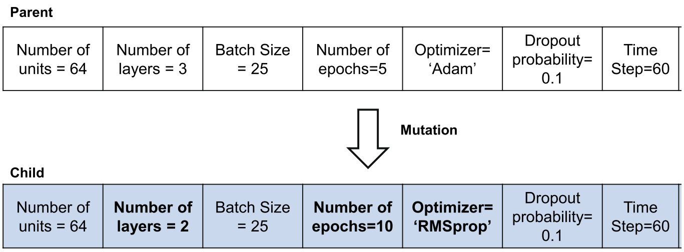


Fig. 7 An example of mutation operator for the GA

3.4 Data preprocessing

To train the model, we must first transform the data to a specific encoding that could be modeled with LSTM. The input data for LSTM, which consists of set of CI builds results, needs to be reshaped into a 3D array with the following dimensions *[samples, time steps, features]*, where the *samples* are the input data, *time steps* are the number of previous observations (which is tuned by GA) used to predict the next build result and *features* is the number of features considered to feed the network which corresponds to 1 as we use a single LSTM model. In Figure 8, we provide an example of input data (*i.e.* a sequence of builds outcomes) and how it reshaped with a time step = 5.

3.5 CI build prediction based on threshold moving strategy

In our case, LSTM works as a binary classifier where the output is the probability of class membership (*i.e.*, the probability of new build to fail) and this must be interpreted before it can be mapped to a class label (*e.g.*, failed or passed). Hence, it is crucial to set the decision threshold above which all values are mapped to one class and all other values are mapped to another class. Threshold moving has brought the attention from the DL research community (Collell et al. 2018; Buda et al. 2018; Krawczyk and Woźniak 2015; Zhou and Liu 2005; Zheng 2010) as a solution to handle the imbalanced distribution of class examples in time series data, which is

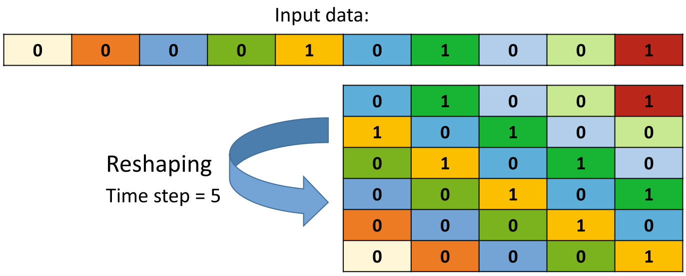


Fig. 8 An example of data preprocessing

indeed the case of build failure prediction (Xie and Li 2018). This solution refers to tuning the threshold used to map probabilities to class labels as the default value (=0.5) can lead to a poor predictive performance when the data is imbalanced Foster Provost (2000).

4 Empirical study setup

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

Figure 9 provides an overview of our experimental design. First, we start by selecting the suitable optimization technique for our DL approach (RQ1). Then to validate the predictive performance, we compare our results with five widely-used Machine Learning (ML) techniques including Decision Tree (DT) (Quinlan 2014), Random Forest (RF) (Breiman 2001), AdaBoost (ADA) (Schapire 2013; Ni and Li 2017), Support Vector Classification (SVC) (Hsu et al. 2003) and Logistic Regression (LR) CM Bishop (2006). We first consider online validation (Xia and Li 2017) (RQ2). Then, we investigate the generalizability of identifying CI build failures by applying cross-project validation using the *Bellwether strategy* (Xia et al. 2017a) (RQ3). Lastly, we evaluate the sensitivity of our DL approach to the training size while comparing its performance against the other ML techniques (RQ4). In the following, we describe our validation in detail.

4.1 Replication package

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

4.2 Data

Our experiments are based on TravisTorrent dataset, from which we selected top-10 projects according to the number of build records. An overview about the studied projects is reported in in Table 2. It is worth noting that the dataset is highly imbalanced as reported in Table 2 with an average failure rate of 0.3.

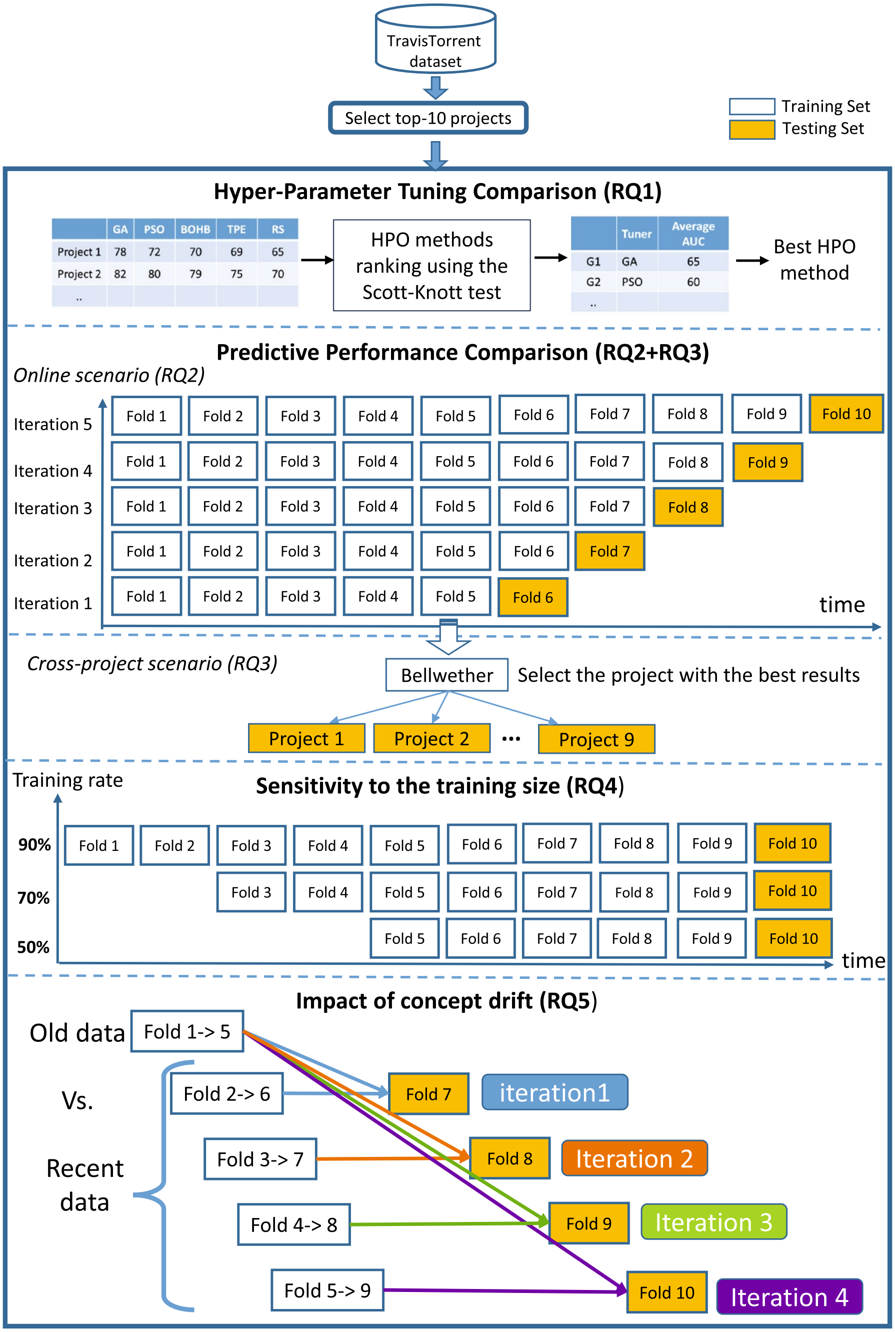


Fig. 9 Experimental design

Table 2 Studies projects statistics

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

`Rails`[1] is a web application framework that provides several features needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. `Ruby`[2] is an interpreted object-oriented programming language often used for web development. `JRuby` https://github.com/jruby/jruby is an implementation of Ruby on the JVM. `Metasploit`[3] is a penetration testing platform that enables to write, test, and execute code with a suite of tools. `Jackrabbit Oak`[4] is a scalable, high-performance hierarchical content repository designed for use as the foundation of modern web sites and content applications. `OpenProject`[5] is one of the leading open source web-based project management systems. `Cloudify`[6] is a cloud-enablement platform that on-boards applications to public and private clouds without architectural or code changes. Graylog2-server[7] is a log management system that centrally captures, stores, and enables real-time search and log analysis. `Vagrant`[8] is a tool for building and distributing development environments with a declarative configuration file. SonarQube[9] is a popular platform for continuous inspection of code quality. Finally, the `Open Build Service`[10] is a generic system to build and distribute binary packages from sources in an automatic, consistent and reproducible way.

1 https://github.com/rails/rails.
2 https://github.com/ruby/ruby.
3 https://github.com/rapid7/metasploit-framework.
4 https://github.com/apache/jackrabbit-oak.
5 https://github.com/opf/openproject.
6 https://github.com/CloudifySource/cloudify.
7 https://github.com/Graylog2/graylog2-server.
8 https://github.com/hashicorp/vagrant.
9 https://github.com/SonarSource/sonarqube.
10 https://github.com/openSUSE/open-build-service.

Table 3 Configuration space for the hyper-parameters of LSTM

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

4.3 Research questions

We designed our experiments to answer five research questions:

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

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

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

1. *Random Search (RS):* A HPO technique that belongs to the family of model-free algorithms. This method was proposed to overcome certain limitations of Grid Search (GS) related mainly to the computational costs random search (RS). RS is similar to GS; but, instead of testing all values in the search space, RS randomly selects a pre-defined number of samples between the upper and lower bounds. To implement RS, we use `Hyperopt` http://hyperopt.github.io/hyperopt/. (the HPO Python framework) while selecting `rand.suggest` algorithm. With regards to its performance, Bergstra and Bengio (2012) argued that RS is more effective than GS.
2. *Tree-structured Parzen Estimators (TPE):* is a Bayesian optimization (BO) based method that, unlike GS and RS, determines the future evaluation points based on the previously-obtained results. This technique has been widely applied in practice (Xia et al. 2017b; Guo et al. 2019). We use the implementation of this technique as provided by `Hyperopt` using the algorithm `tpe.suggest`.
3. *Bayesian Optimization HyperBand (BOHB):* is a multi-fidelity optimization technique that uses a subset of the original to solve the constraint of limited time and resources. It has been shown that BOHB outperforms many other optimization techniques when tuning SVM and DL models (Falkner et al. 2018). To implement this technique, we use `HpBandSter` Python library https://automl.github.io/HpBandSter/build/html/index.html..

4. *Particle Swarm Optimization (PSO):* is another meta-heuristic conceived by Shi and Eberhart (1998) that has been widely adopted for complex HPO problems (Tharwat and Hassanien 2019; Lorenzo et al. 2017). Note that PSO is supported in `Optunity` HPO framework https://optunity.readthedocs.io/en/latest/. as the default option.

In order to ensure a fair comparison, some constraints should be satisfied. First, we evaluate the different HPO methods using the same hyper-parameter configuration space. Table 3 summarizes the configuration space for LSTM model. Additionally, since the studied HPO methods evaluate the candidate configurations based on an objective function to be optimized, we use the same function to be optimized namely the *validation loss*. The training loss functions are threshold-independent metrics *i.e.* not sensitive to imbalanced data (Tantithamthavorn et al. 2018a).

After that, to deal with the stochastic nature of HPO methods, we repeat each experiment 31 times and the median performance is reported as the performance estimate, as recommended by Arcuri and Briand (2011). On the other hand, we set the the maximum number of iterations to 50 for RS, TPE, PSO and BOHB; while we set the number of generations and population size to 5 and 10 respectively for GA ($5 * 10 = 50$).

In the next step, the performance metrics are selected. For each experiment on the selected ten datasets, online validation (cf. Section 4.3) is considered to evaluate the studied HPO methods. First, the Area Under the ROC Curve (AUC) (cf. Section 4.4) is used as the classification performance metric. Additionally, the computational time (CT), the total time needed to complete an experimentation, is also used as the efficiency metric (Yang and Shami 2020; Wicaksono and Supianto 2018; Xia et al. 2017b; Tantithamthavorn et al. 2018a). Note that AUC is computed on the testing set while CT is calculated on the training set as the HPO methods are applied on this set.

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

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

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

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

$j - 1$ folds are selected as training set to train the model. It is worthy to mention, that we verified for each project and validation iteration, the existence of failed builds.

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

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

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

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

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

Approach. Using the same dataset described in Section 4.2, we train and evaluate our approach against baseline techniques based on different training sizes. Similarly to RQ2, we split the data into 10 folds sorted by the time of the build; then, we vary the size of the training set while using the same testing fold in each experiment. In the first experiment, 50% of the datasets are used to construct the predictive models. In the second experiment, the datasets used for the model construction are increased to 70%. In the third experiment, the datasets used for model construction are increased to 90%. In the testing phase, we compare the predictive performance as described in the next section.

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

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

Approach. To study the possible concept drift, we train and test the predictive performance of our approach over time against baseline techniques. As shown in Figure 9, we first split the data into 10 folds sorted by the time of the build. In the first iteration, we train the models using folds 1 to 5 (old data) and folds 2 to 6 (recent data) and compare the predictive performance on the fold 7. In the second iteration, we compare the old data (*i.e.* folds from 1 to 5) to the folds 3 to 7 and test both data on fold 8 etc. In this way, we assess the effectiveness of the approaches based on data from two different time periods in order to assess whether the predictive performance drifts over time.

4.4 Evaluation metrics

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

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

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

$$Recall = \frac{TP}{TP + FN} \in [0, 1] \tag{2}$$

$$Precision = \frac{TP}{TP + FP} \in [0, 1] \tag{3}$$

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

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

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \in [0, 1] \tag{4}$$

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

$$AUC = \frac{1 + \frac{TP}{TP+FN} - \frac{FP}{FP+TN}}{2} \in [0, 1] \tag{5}$$

4.5 Machine learning benchmark

We compare the prediction performance of our *DL-CIBuild* approach with five widely-used ML techniques in previous CI and software engineering research (Xia et al. 2017a; Ni and Li 2018; Xia and Li 2017; Luo et al. 2017; Hassan and Wang 2017; Ni and Li 2018; Santolucito et al. 2018), namely Decision Tree (DT), Random Forest (RF), AdaBoost (ADA), Support Vector Classification (SVC) and Logistic Regression (LR). The initial input to these models is a set of features comprising 21 state-of-the-art CI features from TravisTorrent dataset (Xia et al. 2017a; Ni and Li 2018; Xia and Li 2017; Luo et al. 2017; Hassan and Wang 2017; Ni and Li 2018; Santolucito et al. 2018). These features are summarised in Table 4.

4.5.1 Data pre-processing

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

4.5.2 Parameter tuning for machine learning techniques

We use the best HPO method as the one to be revealed in RQ1. In order to facilitate the replication of our results, we provide the selected main parameters and their respective search spaces for ML techniques as shown in Table 5.

4.6 Inferential statistical test methods used

When applied to the same problem instance, ML and LSTM models may provide different results on each run. To deal with this stochastic nature, it is important to assess their effectiveness by performing a large number of runs, at least 30 runs as suggested in (Arcuri and Briand 2011). Additionally, it is essential to use the statistical tests that provide support for/rejection of the conclusions derived by analyzing the obtained results.

Table 4 CI-related features extracted from TravisTorrent

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

Table 4 (continued)

Metric	Description	Reference
gh_sloc	Number of source lines of code, in the entire repository at the time of this build.	(Xia et al. 2017a; Xia and Li 2017; Luo et al. 2017; Xie and Li 2018)
gh_test_lines_per_kloc	Number of lines in test cases per 1000 gh_sloc.	(Xia and Li 2017; Luo et al. 2017)
gh_test_cases_per_kloc	Number of test cases per 1000 gh_sloc.	(Xia and Li 2017; Luo et al. 2017)
gh_asserts_cases_per_kloc	Number of assertions per 1000 gh_sloc.	(Xia et al. 2017a; Xia and Li 2017; Luo et al. 2017)

Table 5 Configuration space for the hyper-parameters of ML models

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

4.6.1 Statistical tests for RQ1, RQ4 and RQ5

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

4.6.2 Statistical tests for RQ2 and RQ3

W employ Wilcoxon signed rank test (Wilcoxon et al. 1970) in order to detect significant performance differences between the algorithms under comparison (α is set at 0.05). We also use the Cliff's delta, δ, a non-parametric effect size measure for ordinal data (Cliff 1993) to assess the difference magnitude. The effect size is

Table 6 The ranking of the HPO methods for LSTM, divided into distinct groups that have a statistically significant difference in the mean

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

considered negligible when $|\delta| < 0.147$, small when $0.147 \leq |\delta| < 0.33$, medium when $0.33 \leq |\delta| < 0.474$ and large otherwise (Romano et al. 2006).

5 Experimental results

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

5.1 RQ1. Results of HPO comparison

The experiments of applying GA and other four different HPO methods when applied to LSTM models are summarized in Table 6. This table shows the average performance of each HPO methods evaluated based on AUC and the Computational Time (CT).

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

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

Table 7 Performance of *DL-CIBuild* vs ML techniques under online validation

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

The highest values of the performance metric are given in bold

Table 8 Statistical tests results of *DL-CIBuild* compared to ML techniques under online validation

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

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

5.2 RQ2. Results of online validation

Table 7 reports the average (of 5 online validation iterations) AUC, F1 and accuracy scores for each studied project. Note that *DL-CIBuild*, all the involved techniques are executed 31 times to deal with their stochastic nature. Then, we computed the median values of each experiment. Moreover, Table 8 shows the statistical comparisons of these experiments.

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

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

Looking at F1-scores, we also see that *DL-CIBuild* achieved the best results for 6 out of 10 projects with a median score of 49% with an improvement of 10% as compared to the results achieved by SVC (the best ML performing technique). The statistical tests reveal that *DL-CIBuild* outperforms ML with medium (compared to

ADA, DT, LR and SVC) to large effect sizes (with RF). Exceptionally, in `sonar-qube` project, we found evidence for LR algorithm to be better than *DL-CIBuild*. But overall, we clearly see that LR achieved poor performances in terms of F1-scores of 33% in median. This is especially the case for `graylog2-server` and `metasploit-framework` projects as LR turns out to be inefficient to correctly detect failed builds.

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

As for the accuracy scores, the obtained results also show that *DL-CIBuild* is a better performer than the five considered ML techniques, with a significant improvement of 11% in median, and large effect sizes as shown in Table 8. Additionally, the accuracy scores of our approach range from 63% to 85% while achieving in median a high score of 72% and for 9 out of 10 projects, the accuracy values of *DL-CIBuild* exceed those of ML techniques.

To sum up, it is worth noting that, due to the highly imbalanced nature of the analyzed data (*i.e.*, only a small portion of the builds are failed) as can be seen from Table 2, the achieved AUC, F1 and accuracy results by *DL-CIBuild* are considered significant. Furthermore, we can see from the statistical results, that ML modest performance may not be only related to the nature of the dataset as we applied resampling to the training data using SMOTE, but this could be related to the complex and erratic temporal dependencies between the builds that are hard to capture with traditional ML techniques. Thus, DL-based time series models seem more appropriate to such a problem.

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

5.3 RQ3. Results of cross-projects validation

As mentioned earlier, `jruby` project exhibited the highest prediction capability among the studied projects by achieving the best scores on average (and in median) and it is considered as the *Bellwether* for cross-project strategy. Hence, we train *DL-CIBuild*, based on *jruby* project, using our evaluation metrics, the Area Under the ROC Curve (AUC), F1-score, and accuracy values, to measure the performance of our classifier. Table 9 presents the effectiveness of cross-project modeling compared to ML techniques while Table 10 reports the statistical tests results.

First, the results show that *DL-CIBuild* achieves a performance of AUC value of 72% in median which ranges from 63-82%. Six projects out of nine show good performance results ($\geq$ 70%) and `cloudify` achieves high AUC value of 82%. Compared to within-project validation, these results show that our approach can achieve with cross-projects a significant improvement of 7% in median over

Table 9 Performance of *DL-CIBuild* vs ML techniques under cross-project

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

The highest values of the performance metric are given in bold

Table 10 Statistical tests results of *DL-CIBuild* under cross-projects compared to its achieved within-project results as well as ML techniques

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

online validation with a large effect size. Except for `ruby` whose AUC score slightly decreased from 74% to 73%, may be because the data in this project is larger than the bellwether data, all the studied projects show a better performance which indicates that *DL-CIBuild* a very promising solution to mitigate the lack of data, especially for new software projects. Additionally, we observe an improvement of 17% in median over ML techniques whose results are worse than their within-project scores. The statistical tests results show that the difference is significant with large effect sizes.

The same observations for AUC can be applied to F1-score for which we recorded for *DL-CIBuild* a significant improvement of 8% compared to within-project results with a medium effect size. Also, *DL-CIBuild* is the best technique across all the studied projects by achieving in median 57% compared to ML techniques that showed modest to low F1-scores of 37% for SVC, 35% for LR and ADA, 34% for RF and 31% for DT. The statistical tests results show that *DL-CIBuild* is significantly better with large effect sizes, as reported in Table 10. Another observation to report from these results is that all ML techniques have shown a drop in F1-scores; which confirms previous findings in the literature who pointed out that ML techniques are less effective for cross-project prediction (Choetkiertikul et al. 2018; Abdalkareem et al. 2020; Zhang et al. 2016). This result shows that when building ML techniques under cross-project prediction, the target project has a low collinearity with the source project features.

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

> Results of RQ3 show a substantial improvement for *DL-CIBuild* compared to online validation results by achieving 72%, 57% and 78% in terms of AUC, F1-score and accuracy, respectively. These results indicate that our approach is effective when learning from a cross-project training corpus. We explain these results by the fact that in a cross-project setting, our approach is fed with more

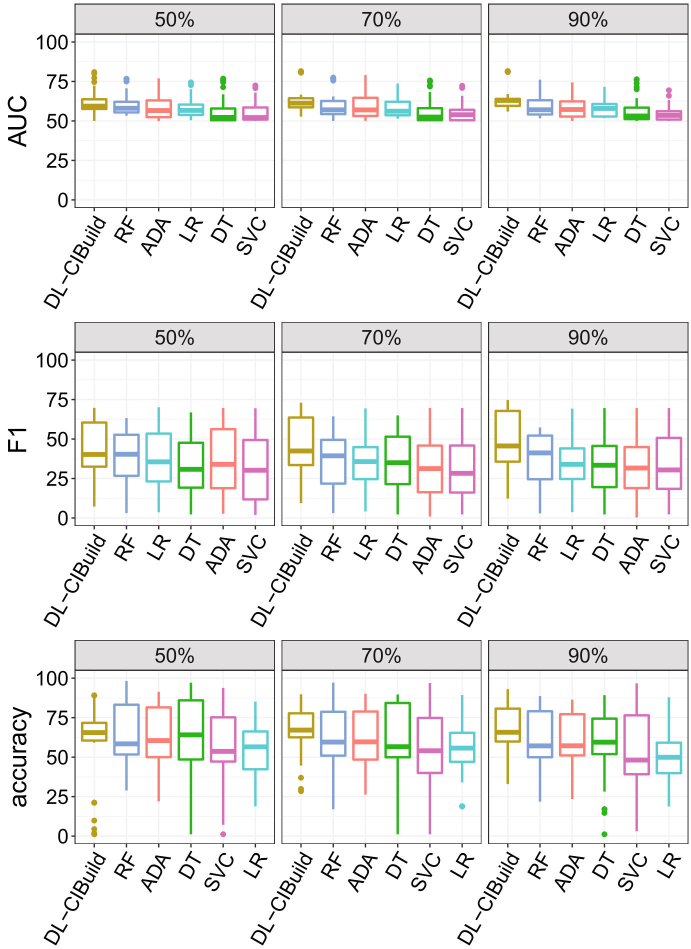


Fig. 10 Comparison of the prediction performance with different training sets sizes

data. Moreover, our proposed approach still outperforms state-of-the-art ML techniques.

5.4 RQ4. Results of the sensitivity to training size

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

Figure 10 presents the performance (in terms of AUC, F1 and accuracy) on the test dataset, of the studied approaches, after training for 31 times with 50%, 70% and 90% of the dataset. Additionally, Table 11 shows the rank differences for all of the studied approaches when varying the training sizes.

Table 11 The ranking of the approaches when varying the training size, divided into distinct groups that have a statistically significant difference in the average (Avg)

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

Looking at the plotted boxplots of *DL-CIBuild*, we observe that, for all the computed measures, the performance of our approach increases up to 90% of the datasets for which the best scores were recorded. For instance, increasing the training from 50% to 90% of the datasets, results in an improvement of 3%, 5% and 9% in terms of AUC, F1 and accuracy in median respectively. Moreover, Table 11 shows that there is a clear separation of AUC, F1 and accuracy scores of *DL-CIBuild* into distinct Scott-Knott ranks for 50% and 90% of training data. However, the scores seem comparable when training on 70% and 90% of the datasets; which means that our approach plateaus out from 70%. Nevertheless, we can conjecture that an advantage of using our approach in practice is that, as the project ages and more CI build records are available, *DL-CIBuild* will reach higher scores.

As compared to ML techniques, we clearly see that *DL-CIBuild* is better across different training set sizes. Moreover, Table 11 shows that for AUC scores, *DL-CIBuild* is statistically better than other techniques even when trained only on 50% of the datasets. As for F1 and accuracy scores, we see that *DL-CIBuild* share the same ranking with other ML but achieves better scores. Overall, we conjecture that, for different training sizes, our approach is more suitable than the ML techniques.

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

5.5 Results of the concept drift evaluation

In the last evaluation, we study the extent to which the approaches under evaluation suffer from concept drift (*i.e.* the degradation in the predictive performance over time (Widmer and Kubat 1996)). Figure 11 presents the performance (in terms of AUC, F1 and accuracy) on the test dataset, of the studied approaches, after training for 31 times using old (in red) and recent (in blue) training data. Additionally, Table 11 shows the ranks for all of the studied approaches when training in different time intervals.

As shown in Figure 11, we observe that, for all the computed measures, the performance of *DL-CIBuild* slightly increases when training the models on recent training data. For instance, we recorded a median improvement of 2% in terms of AUC when training on more recent data. However, when looking at Table 12, we see that the obtained results on both recent and old data seem to be comparable (same ranking group). These results indicate that while predicting the next builds is better when training on more recent build records, the data stream does not seem to be drifting

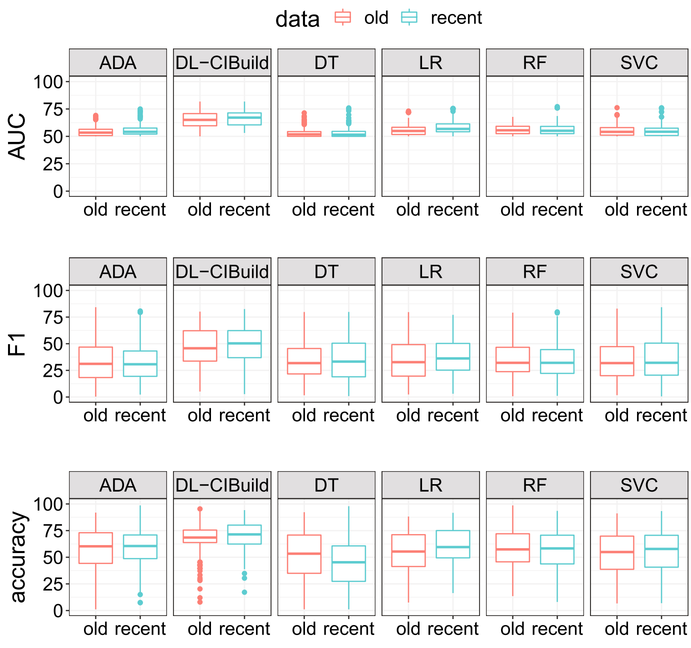


Fig. 11 Comparison of the prediction performance of *DL-CIBuild* against ML techniques trained on old and recent data

Table 12 The ranking of the approaches when varying the training time, divided into distinct groups that have a statistically significant difference in the average (Avg)

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

for our approach. Thus, the models of *DL-CIBuild* do not need to be frequently retrained.

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

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

6 Discussions and implications

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

6.1 For CI developers

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

***DL-CIBuild* can run faster.** One of the acknowledged drawbacks of using our approach is that it is computationally expensive due to the massive training time of LSTM models as well as using Genetic Algorithm (GA) for Hyper-parameters Optimization (HPO). In order to mitigate this issue, we improve the efficiency of GA by enabling the parallel evaluation of the configurations in each generation (which includes the training of LSTM models using the candidate configurations). By integrating this parallelization mechanism, we significantly reduced the execution time of GA as can be shown in Figure 12. As we clearly see, the optimized version of GA can even run faster than BOHB technique that also supports the parallelization (Yang and Shami 2020).

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

6.2 For researchers

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

Deep learning LSTM is a suitable modelling choice for software engineering problems for which the temporal dimension is important. To the best of our knowledge, our work is the first attempt to use deep learning LSTM for the problem of learning CI build failures. The use of LSTM models has allowed to automatically learn the periodicity of build results and use this for predicting build failure. The evaluation results demonstrate the significant improvement that our DL approach has brought in terms of predictive performance especially with comparison to ML techniques. These results represent a significant improvement that can help researchers to mitigate the issues related to feature engineering which is a tedious and error-prone process that needs specific expertise with the domain knowledge to generate features for ML models. Moreover, *DL-CIBuild* has shown effectiveness in handling

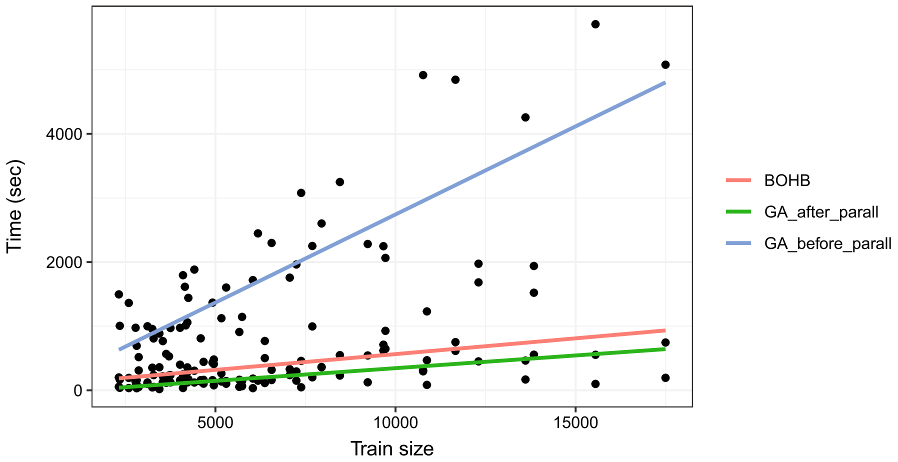
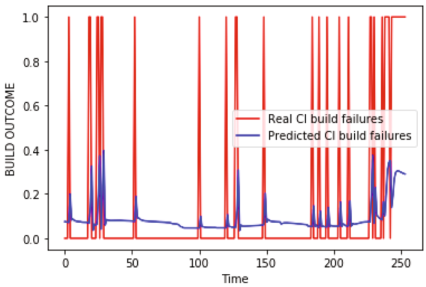


Fig. 12 Theimpact of the training set size on the execution time to run GA before (blue) and after (green) parallelization compared to BOHB (in red)

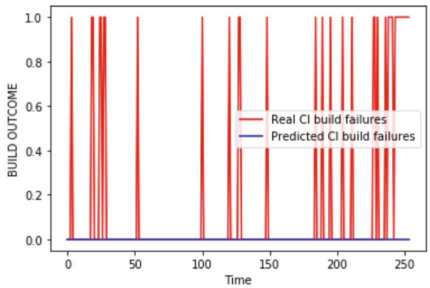
the lack of data considering cross-project validation while no existing solution has been demonstrated to work at this performance scale. Knowing that software engineering tasks are process-based where the temporal dimension is of crucial importance, our proposed approach can serve as the baseline for further research in the application of DL and LSTM models to time series problems in software engineering.

Dynamic selection of the classification threshold. Another possible direction to enhance the prediction accuracy of deep learning LSTM models is to accurately set the classification threshold (above which a build is considered failed) which can highly impact the prediction results. As illustrated in Figure 13, we can see an example highlighting the importance of threshold moving from the `Ruby` project. In this figure, the chart (a) plots the output of our LSTM model (which is following the real trend), while the chart (b) shows the prediction results when the classification threshold is set by default (=0.5) which results in classifying all the builds as succeeding (none of the failed builds can be detected). Based on these observations, an important research direction for CI researchers is to consider adaptive threshold selection over time when conceiving DL-based models. This selection can be performed dynamically over time, *i.e.*, adapted depending on the project's activity period such as major/minor releases, new features, library dependencies upgrade/migration, code reengineering, code optimization, etc. We conjecture that a dynamic selection can be an effective solution for deep learning LSTM based prediction.

Can the predictive performance be improved with re-sampling? So far, we showed that *DL-CIBuild* provides an effective improvement over the ML techniques without re-sampling but instead using Threshold Moving (TM). Unlike sampling, TM does not rely on the manipulation of the training set but instead on manipulating the classifier output. However, one can argue that the use of resampling can further improve the identification of CI build failures for *DL-CIBuild*, even though the latter has shown less sensitivity to the class imbalance problem as pointed out in



(a) Failure prediction probability.



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

Fig. 13 An example showing the impact of the threshold moving on the prediction accuracy extracted from the project `Ruby`

our previous research questions (RQ2+RQ3). Thus, we conduct a set of additional experiments to re-balance the input data prepared in Section 2(online and cross-project validations) using SMOTE (Chawla et al. 2002) the standard oversampling approach; and re-run the LSTM-RNN learning process on the new balanced data. Similar to (Buda et al. 2018), the combination of TM and SMOTE is also tested. To provide a comprehensive comparison, we compute the F1-score, AUC score and the overall accuracy. Figure 14 shows the obtained results using SMOTE (in red), TM (in blue) and by combining the two approaches (in green). Considering online validation, we observe that framework shows a better performance using TM than when applying SMOTE with a statistically significant (but small) improvement of 4% in terms of AUC and F1 respectively.

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

6.3 For tool builders

Feedback mechanisms to predict build failures. CI services such as Travis CI could provide mechanisms for developers to estimate the likelihood that their current build would fail. Information about the predicted build failures can help the software development team to avoid time overhead. Such information would provide decision support to avoid useless build runs or suggest running builds during project

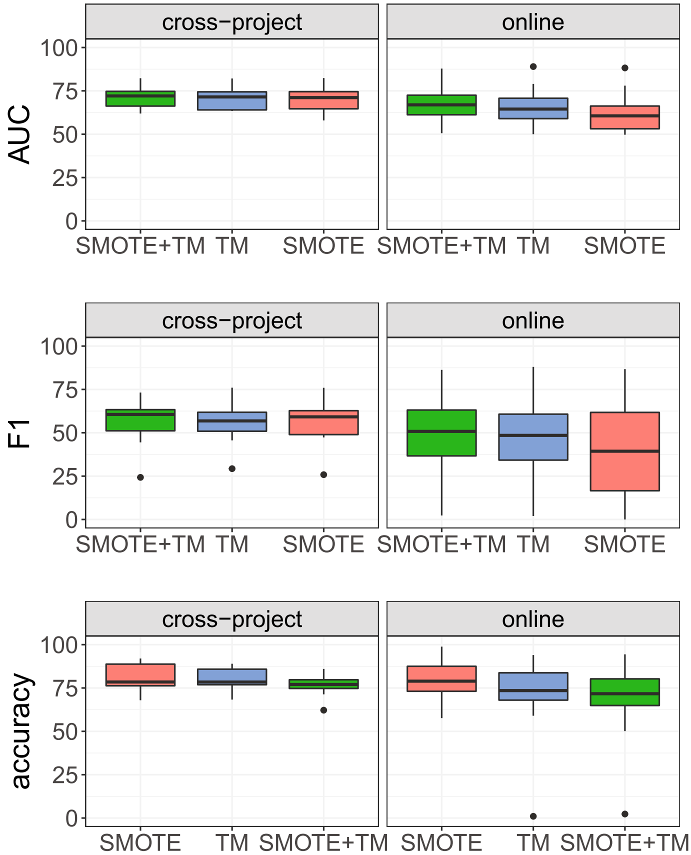


Fig. 14 Comparison of *DL-CIBuild* results with SMOTE (red), Threshold Moving (blue) and by combining them (green)

inactivity periods (*e.g.*, out of the working hours) in order to avoid the risk of reducing the team's productivity and release delays.

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

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

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

7 Related work

This section presents the research around this topic. First, we review research works about CI builds prediction. Then, we present the studies performed to analyse CI builds. Finally, we provide insights into the application of HPO of ML/DL approaches.

7.1 Prediction of CI builds

Many research works have introduced prediction models to predict the build status. However, we only focus on works dedicated to CI environment that has different workflow and can suffer from different latency as stated by Hassan and Wang (2017) and Hilton et al. (2016). For the sake of clarity and completeness of the reporting, we summarise them in Table 13, presenting their key information, *e.g.*, used models, along with a brief description of the methodology employed to address their objectives and the achieved results.

Xia and Li (2017) compared nine ML models to construct CI prediction models of 126 open source projects hosted on GitHub. Their experiments were based on both cross-validation and online scenarios. In cross-validation, their models achieved an Area Under the ROC Curve (AUC) score of over 70%. However, under the online scenario, they observed a tendency for their prediction scores to decrease up to 60% of AUC. In both scenarios, they found that Decision Tree (DT) and Random Forest (RF) achieved the best performance scores.

Ni and Li (2017) employed AdaBoost (ADA) to predict CI build failures of 532 CI projects. This adaption achieved an AUC of 75%, using 50% of the dataset as training set and last 50% instances as test set.

Hassan and Wang (2017) proposed the prediction model of CI build failure on three build systems, namely Ant, Maven and Gradle, under the cross-project prediction and cross-validation scenarios. Using RF, they achieved over 90% of AUC scores for the considered build systems. Additionally, the cross-validation provided better results. However, when we looked at the provided dataset, we found that there is a large number of redundant lines that may influence the validity of the reported results. We also found that the dataset is perfectly balanced (45% of failed builds) which is not the case in practice as it is generally known that failed builds are much

Table 13 An overview of the literature on the prediction of CI build outcome

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

less to occur than passed ones (Xie and Li 2018). In this paper, we found that when applying RF to our generated dataset, our approach can achieve better results.

Xia et al. (2017a) conducted an empirical study to evaluate the predictive performance of six common ML models including RF and DT considering cross-project validation. For dataset selection, they compared three methods namely Random Selection, Burak Filter based on build-level and Bellwether Strategy based on project-level. According to the results of their experiments, they found that Bellwether strategy performs better than the two other methods. And among the used models, they found that DT classifier performs the best achieving a score of 17% for F1-score on average.

Luo et al. (2017) have used the features of TravisTorrent dataset to predict the result of a build. Additionally, they compared Support Vector Machine (SVM), DT, RF and Logistic Regression (LR). Based on 10-fold cross-validation, the results reveal that LR and SVM were the best performers.

Although these research efforts have advocated that predicting CI build failures is possible, these works achieved a limited prediction accuracy that is sometimes comparable to the performance of random guessing (Xia et al. 2017a). Another main issue to classic ML-based approaches is related to the imbalanced distribution of build results. This challenges their applicability due to the performance bias that can occur when an imbalanced distribution of class examples is used (Saidani et al. 2020). Furthermore, this imbalanced nature of the training data was rarely discussed in existing works. However, in CI context, a good accuracy on the failed builds prediction is more important than the passed builds accuracy. The existence of these issues suggest that the build failure prediction problem is not yet resolved. In our paper, we showed that the usage of *DL-CIBuild*, the DL-based approach, can effectively predict the CI build failure while considering the imbalance nature of the data.

7.2 Analysis of CI builds

The analysis of build failure is a growing topic as many research works (Beller et al. 2017; Luo et al. 2017) attempted to discover the reasons behind build breakage that are related mainly to the development activities. Other studies investigated the temporal trends of CI builds and investigated the link between the current build and previous ones. In particular, Rausch et al. (2017) observed by analyzing the build logs that for 10 selected projects using Travis-CI, that more than 50% (up to 80%) of all failed builds follow a previous build failure. That is, if the build failed, the next build is more likely to fail as well. Ni and Li (2017) also found that features linked to the last build such as previous build result can be effective in predicting the current build outcome. These results lend support to our findings as our study shows that the sequence of build results is a strong predictor for future failures. Atchison et al. (2017) also observed a clear seasonality in build activity, as their approach was able to estimate the number of builds to be generated in the future, with an average accuracy of 86%. However, that study did not investigate how the build outcome evolves over time, nether it estimated the build results. Ghaleb et al. (2019b) revealed that some builds may break while generating the build. Such kinds of build breakages introduce noises to build breakage data. Particularly, they found that 33% of the Travis CI build failures are due to environmental factors, 29% are due to errors in previous builds, and 9% are due to build jobs that were later deemed by developers as noisy. Gallaba et al. (2018) also found that CI builds are noisy.

Despite this much research, the reasons behind the build failure is still ambiguous. The findings of our paper encourage researchers to analyse further the periodicity of CI builds that may be linked to other software engineering activities *e.g.* release preparation.

7.3 Application of optimization techniques to ML/DL algorithms

Very often a learning algorithm performance can be significantly improved through the optimization of its parameters selection (Tantithamthavorn et al. 2018a). Hyper-parameter tuning can be interpreted as an optimization problem where the objective is to find a configuration that optimizes an objective (*e.g.* minimize the loss function).

Grid Search(GS) is widely used method for HPO (Bergstra et al. 2011). This approach consists of trying all possible combinations of an existing set of parameters. However, with the growing complexity of the search space due to the increase of parameters and their possible values, GS becomes a non practical choice for configuring complex learning algorithms (Yang and Shami 2020).

Random Search (RS) was proposed by Bergstra and Bengio (2012) to overcome certain limitations of GS. Instead of testing all possible combinations in the search space, RS randomly selects the candidate hyper-parameter values. . In (Bergstra and Bengio 2012), the authors showed that RS is more efficient than RS for tuning neural networks. However, RS may involve many unnecessary evaluations, which decrease its efficiency (Bergstra and Bengio 2012).

Bayesian Optimization (BO) (Snoek et al. 2012) is an iterative algorithm that, unlike GS and RS, determines the future evaluation points based on the previously-obtained results. One of the most commonly used BO-based methods is **Tree-structured Parzen estimator (TPE)** (Bergstra et al. 2011) which has proved its effectiveness. For instance, by tuning XGBoost, Xia et al. (2017b) have found that TPE performs significantly better than RS, GS and manual search in terms of accuracy but requires more time due to the additional computational costs related to the creation of Parzen estimators. Guo et al. (2019) have also found that TPE is better than RS in terms of accuracy and precision.

Bayesian Optimization HyperBand (BOHB) (Falkner et al. 2018) is a recent approach (in 2018) that combines Bayesian optimization and HyperBand (Li et al. 2017) by replacing HyperBand's random search by TPE. Recently, Haris et al. (2021) have shown that this approach is better than TPE and HyperBand for tuning deep learning.

To solve complex and large search space problems, meta-heuristics, including **Genetic algorithms (GAs)** and **Particle Swarm Optimization (PSO)**, are the most prevalent (Yang and Shami 2020). For instance, Lorenzo et al. (2017) found that PSO improves the performance of RS and GS. The same conclusion was dropped by Tharwat and Hassanien (2019). Wicaksono and Supianto (2018) have shown that GA is better than GS to tune Support Vector Machine (SVM), Random Forest (RF), Adaptive Boosting (AdaBoost) and K-Nearest Neighbour (KNN). Di Martino et al. (2011) also used GA to configure SVM models.

There exist other HPO methods that were described in detail in Yang and Shami (2020) review. In our paper, we showed that GA is the most suitable, in terms of AUC and time, HPO technique when applied to LSTM in the context of CI build failure.

8 Threats to validity

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

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

It would be also interesting to compare the performance of the Threshold Moving against other sampling techniques like *MAHAKIL* (Bennin et al. 2017) or *SMOTUNED* (Agrawal and Menzies 2018), which would be an interesting future work

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

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

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

Reliability validity concerns the possibility of replicating this study. All the studies projects are publicly available. Moreover, the Python implementation of our approach is provided in our replication package https://github.com/stilab-ets/DL-CIBuild.

9 Conclusion

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

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

A optimal parameters for online validation (RQ1 and RQ2)

See Table 14.

Table 14 Optimal Parameters for RQ1 and RQ2

Project	Experiment	nb_epochs	nb_batch	time_step	drop_proba	nb_layers	nb_units	Optimizer
cloudify	1	7	38	38	0.11	3	72	adam
	2	6	26	36	0.07	3	50	rmsprop
	3	7	14	36	0.02	2	94	adam
	4	7	22	44	0.12	3	76	adam
	5	7	23	60	0.03	3	64	adam
graylog2-server	1	7	27	31	0.06	2	44	adam
	2	6	16	54	0.19	4	76	adam
	3	7	64	39	0.05	2	52	rmsprop
	4	4	22	33	0.16	2	54	adam
	5	4	24	39	0.05	4	63	rmsprop
jackrabbit-oak	1	7	14	34	0.12	4	60	adam
	2	7	45	36	0.14	2	92	adam
	3	7	11	30	0.16	4	72	adam
	4	7	14	45	0.18	2	57	rmsprop
	5	7	15	49	0.15	3	94	adam
jruby	1	7	21	38	0.12	2	47	adam
	2	6	22	48	0.08	2	52	adam
	3	6	5	30	0.05	4	49	adam
	4	5	12	31	0.05	2	82	adam
	5	6	16	40	0.06	4	39	rmsprop
metasploit-framework	1	5	52	56	0.19	2	96	rmsprop
	2	6	24	50	0.15	2	45	adam
	3	4	16	45	0.19	2	61	rmsprop
	4	5	16	49	0.04	2	79	rmsprop
	5	6	56	34	0.07	3	53	adam

Table 14 (continued)

Project	Experiment	nb_epochs	nb_batch	time_step	drop_proba	nb_layers	nb_units	Optimizer
open-build-service	1	6	28	32	0.07	2	68	adam
	2	5	52	33	0.03	3	78	rmsprop
	3	5	63	42	0.08	2	92	adam
	4	6	13	39	0.07	3	95	adam
	5	6	26	31	0.10	2	40	adam
openproject	1	7	12	37	0.14	2	94	adam
	2	7	30	40	0.16	2	65	rmsprop
	3	5	20	60	0.13	4	46	rmsprop
	4	5	18	35	0.13	2	45	adam
	5	7	28	38	0.11	3	46	adam
rails	1	5	58	36	0.05	2	45	adam
	2	7	49	33	0.20	3	67	rmsprop
	3	5	43	47	0.11	3	68	rmsprop
	4	6	63	50	0.14	3	68	adam
	5	4	62	55	0.11	3	55	adam
ruby	1	6	22	38	0.08	2	82	adam
	2	7	10	48	0.05	4	83	adam
	3	7	26	58	0.18	3	67	adam
	4	6	55	50	0.06	3	73	adam
	5	4	4	30	0.06	4	48	adam

Table 14 (continued)

Project	Experiment	nb_epochs	nb_batch	time_step	drop_proba	nb_layers	nb_units	Optimizer
sonarqube	1	6	7	55	0.06	4	79	rmsprop
	2	6	8	51	0.18	4	87	rmsprop
	3	5	4	31	0.01	2	58	rmsprop
	4	6	16	51	0.02	2	86	rmsprop
	5	5	43	43	0.12	2	90	adam

B optimal parameters for cross-project validation (RQ3)

See Table 15.

Table 15 Optimal Parameters for cross-project validation (Jruby is the training project)

Parameter	Optimal value
nb_epochs	5
nb_batch	16
time_step	60
drop_proba	0.2
nb_layers	4
nb_units	32
optimizer	"adam"

C optimal parameters for RQ4

See Table 16.

Table 16 Optimal Parameters for RQ4

Project	Experiment	nb_epochs	nb_batch	time_step	drop_proba	nb_layers	nb_units	Optimizer
cloudify	1	4	42	58	0.11	2	81	adam
	2	6	10	50	0.11	3	76	adam
	3	3	5	46	0.02	3	72	adam
graylog2-server	1	5	45	46	0.16	2	60	adam
	2	5	45	49	0.06	2	91	adam
	3	4	17	35	0.08	2	86	adam
jackrabbit-oak	1	6	25	59	0.20	2	76	adam
	2	2	29	52	0.02	3	90	adam
	3	5	62	55	0.09	2	73	adam
jruby	1	3	44	45	0.07	2	74	adam
	2	5	20	53	0.14	2	69	adam
	3	5	23	42	0.04	2	81	adam
metasploit-framework	1	4	5	55	0.06	3	33	adam
	2	5	24	46	0.09	3	34	adam
	3	5	34	31	0.01	2	40	adam
open-build-service	1	4	34	30	0.16	2	37	adam
	2	6	54	56	0.14	2	73	adam
	3	6	54	54	0.19	3	78	adam
openproject	1	2	17	50	0.07	2	70	adam
	2	6	36	37	0.06	2	84	adam
	3	4	57	53	0.02	2	42	adam
rails	1	4	5	59	0.03	2	38	adam
	2	3	11	53	0.14	2	51	adam
	3	5	49	38	0.11	3	61	adam

Table 16 (continued)

Project	Experiment	nb_epochs	nb_batch	time_step	drop_proba	nb_layers	nb_units	Optimizer
ruby	1	3	55	50	0.09	3	88	adam
	2	3	35	37	0.15	2	66	adam
	3	5	32	41	0.03	3	86	adam
sonarqube	1	5	18	48	0.07	3	62	adam
	2	4	15	52	0.19	3	48	adam
	3	3	45	59	0.18	3	74	adam

Optimal parameters for RQ5

See Tables 17 and 18.

Table 17 Optimal Parameters for RQ5 (old data)

Project	Experiment	nb_epochs	nb_batch	time_step	nb_units	Optimizer	drop_proba	nb_layers
cloudify	1	6	4	41	32	rmsprop	0.15	4
	2	5	4	37	32	rmsprop	0.17	3
	3	5	16	44	32	adam	0.07	1
	4	5	16	33	32	adam	0.03	2
graylog2-server	1	4	32	31	32	rmsprop	0.03	1
	2	5	32	48	32	adam	0.19	1
	3	6	8	59	32	rmsprop	0.09	1
	4	6	16	50	32	rmsprop	0.03	3
jackrabbit-oak	1	4	4	32	32	rmsprop	0.03	4
	2	5	32	55	32	rmsprop	0.1	4
	3	6	8	44	32	adam	0.12	4
	4	4	8	57	32	adam	0.19	3
jruby	1	6	4	44	32	adam	0.05	2
	2	5	32	42	32	adam	0.06	4
	3	5	4	49	32	adam	0.07	4
	4	6	32	45	32	adam	0.2	3
metasploit-framework	1	5	8	59	32	adam	0.1	1
	2	4	4	30	32	adam	0.04	1
	3	6	16	60	32	rmsprop	0.07	1
	4	6	32	46	32	rmsprop	0.17	1

Table 17 (continued)

Project	Experiment	nb_epochs	nb_batch	time_step	nb_units	Optimizer	drop_proba	nb_layers
open-build-service	1	4	8	54	32	adam	0.02	1
	2	6	4	33	32	rmsprop	0.17	3
	3	4	8	37	32	rmsprop	0.07	1
	4	6	8	42	32	rmsprop	0.06	3
openproject	1	6	8	54	32	rmsprop	0.08	1
	2	4	16	35	32	rmsprop	0.1	2
	3	6	16	36	32	rmsprop	0.07	1
	4	6	8	57	32	rmsprop	0.11	4
rails	1	6	16	59	32	adam	0.14	4
	2	6	4	51	32	adam	0.16	4
	3	5	8	35	32	adam	0.01	4
	4	6	8	38	32	adam	0.01	4
ruby	1	5	4	51	32	adam	0.19	1
	2	5	4	39	32	rmsprop	0.09	4
	3	5	4	42	32	rmsprop	0.2	3
	4	5	8	53	32	adam	0.14	2
sonarqube	1	6	32	39	32	rmsprop	0.1	1
	2	6	16	32	32	adam	0.14	1
	3	6	32	48	32	rmsprop	0.04	3
	4	6	4	56	32	rmsprop	0.1	1

Table 18 Optimal Parameters for RQ5 (recent data)

Project	Experiment	nb_epochs	nb_batch	time_step	nb_units	Optimizer	drop_proba	nb_layers
cloudify	1	5	8	52	32	rmsprop	0.1	1
	2	4	8	38	32	adam	0.19	4
	3	5	32	45	32	rmsprop	0.04	1
	4	4	4	41	32	rmsprop	0.17	4
graylog2-server	1	5	8	60	32	adam	0.05	2
	2	6	4	39	32	adam	0.09	2
	3	5	16	32	32	adam	0.11	1
	4	5	8	60	32	adam	0.05	2
jackrabbit-oak	1	4	32	43	32	rmsprop	0.2	1
	2	4	16	51	32	adam	0.02	4
	3	4	32	43	32	rmsprop	0.2	1
	4	4	16	51	32	adam	0.02	4
jruby	1	4	8	43	32	adam	0.1	2
	2	6	8	59	32	adam	0.17	4
	3	5	4	30	32	adam	0.04	2
	4	4	8	43	32	adam	0.1	2
metasploit-framework	1	6	8	52	32	rmsprop	0.1	2
	2	4	4	37	32	rmsprop	0.02	1
	3	6	4	41	32	rmsprop	0.14	3
	4	4	32	54	32	rmsprop	0.19	2
open-build-service	1	4	4	32	32	rmsprop	0.13	3
	2	6	32	35	32	adam	0.15	4
	3	4	4	32	32	rmsprop	0.13	3
	4	6	64	38	32	rmsprop	0.19	2

Table 18 (continued)

Project	Experiment	nb_epochs	nb_batch	time_step	nb_units	Optimizer	drop_proba	nb_layers
openproject	1	6	8	45	32	adam	0.1	1
	2	6	16	47	32	rmsprop	0.03	3
	3	6	32	37	32	adam	0.04	2
	4	6	32	37	32	adam	0.04	2
rails	1	4	16	32	32	adam	0.03	2
	2	6	32	31	32	rmsprop	0.02	4
	3	6	32	31	32	rmsprop	0.02	4
	4	4	16	32	32	adam	0.03	2
ruby	1	4	8	51	32	rmsprop	0.2	3
	2	6	32	33	32	rmsprop	0.14	3
	3	5	4	46	32	rmsprop	0.02	1
	4	4	8	51	32	rmsprop	0.2	3
sonarqube	1	5	16	42	32	rmsprop	0.03	1
	2	6	8	30	32	rmsprop	0.18	4
	3	5	8	42	32	adam	0.18	4
	4	5	16	42	32	rmsprop	0.03	1

Acknowledgements This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Discovery Grant RGPIN-2018-05960.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Abdalkareem, R., Mujahid, S., Shihab, E.: A machine learning approach to improve the detection of ci skip commits. *IEEE Trans. Softw. Eng.*, (2020)

Agrawal, A., Menzies, T.: Is" better data" better than" better data miners"? In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1050–1061 (2018)

Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1–10 (2011)

Atchison, Abigail, Berardi, Christina, Best, Natalie, Stevens, Elizabeth, Linstead, Erik: A time series analysis of travistorrent builds: to everything there is a season. In *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 463–466. IEEE (2017)

Athiwaratkun, Ben, Stokes, Jack W.: Malware classification with lstm and gru language models and a character-level cnn. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2482–2486 (2017)

Beller, M., Gousios, G., Zaidman, A.: Oops, my tests broke the build: An explorative analysis of travis ci with github. In: *IEEE/ACM International Conference on Mining Software Repositories* 356–367, (2017)

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

Bergstra, James, Bengio, Yoshua: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**(Feb), 281–305 (2012)

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: *25th annual conference on neural information processing systems (NIPS 2011)*, volume 24. Neural Information Processing Systems Foundation, (2011)

Bhowan, Urvesh, Johnston, Mark, Zhang, Mengjie, Yao, Xin: Reusing genetic programming for ensemble selection in classification of unbalanced data. IEEE Trans. Evolut. Comput. **18**(6), 893–908 (2013)

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

Bhowan, Urvesh, Zhang, Mengjie, Johnston, Mark: Genetic programming for classification with unbalanced data. In *European Conference on Genetic Programming*. Springer, 1–13 (2010)

Bhowan, U., Johnston, M., Zhang, M.: Differentiating between individual class performance in genetic programming fitness for classification with unbalanced data. In: *2009 IEEE Congress on Evolutionary Computation*. IEEE, 2802–2809 (2009)

Bishop, C.M.: Pattern recognition and machine learning (information science and statistics). Springer-Verlag New York, inc., Secaucus, NJ, USA, (2006))

Bouktif, Salah, Fiaz, Ali, Ouni, Ali, Adel Serhani, Mohamed: Multi-sequence lstm-rnn deep learning and metaheuristics for electric load forecasting. Energies **13**(2), 391 (2020)

Bouktif, Salah, Fiaz, Ali, Ouni, Ali, Serhani, Mohamed Adel: Optimal deep learning lstm model for electric load forecasting using feature selection and genetic algorithm: Comparison with machine learning approaches. Energies **11**(7), 1636 (2018)

Breiman, Leo: Random forests. Mach. Learn. **45**(1), 5–32 (2001)

Buda, Mateusz, Maki, Atsuto, Mazurowski, Maciej A.: A systematic study of the class imbalance problem in convolutional neural networks. Neural Netw. **106**, 249–259 (2018)

Cervantes, J., Li, X., Yu, W.: Using genetic algorithm to improve classification accuracy on imbalanced data. In: *2013 IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 2659–2664 (2013)

Chawla, Nitesh V., Bowyer, Kevin W., Hall, Lawrence O., Kegelmeyer, W Philip: Smote: synthetic minority over-sampling technique. J. Artif. Intell. Res. **16**, 321–357 (2002)

Choetkiertikul, Morakot, Dam, Hoa Khanh, Tran, Truyen, Pham, Trang, Ghose, Aditya, Menzies, Tim: A deep learning model for estimating story points. IEEE Trans. Softw. Eng. **45**(7), 637–656 (2018)

Cliff, Norman: Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol. Bull. **114**(3), 494 (1993)

colah'sblog. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/,. Accessed: 2020-03-01

Collell, Guillem, Prelec, Drazen, Patil, Kaustubh R.: A simple plug-in bagging ensemble based on threshold-moving for classifying binary and multiclass imbalanced data. Neurocomputing **275**, 330–340 (2018)

Cui, Yiming,Wang, Shijin, Li, Jianfeng: Lstm neural reordering feature for statistical machine translation. arXiv preprint arXiv:1512.00177, (2015)

Dey, Rahul., Salemt, F.M.: Gate-variants of gated recurrent unit (gru) neural networks. In: *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*. IEEE 1597–1600 (2017)

Di Martino, S., Ferrucci, F., Gravino, C., Sarro, F.: A genetic algorithm to configure support vector machines for predicting fault-prone components. In: *International Conference on Product Focused Software Process Improvement*. Springer, Berlin, Heidelberg 247–261 (2011)

Duvall, P.M., Matyas, S., Glover, A.: *Continuous integration: improving software quality and reducing risk*. Pearson Education, (2007)

Ebo Bennin, Kwabena, Keung, Jacky, Phannachitta, Passakorn, Monden, Akito, Mensah, Solomon: Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Trans. Softw. Eng. **44**(6), 534–550 (2017)

Ekanayake, J., Tappolet, J., Gall, H.C., Bernstein, A.: Tracking concept drift of software projects using defect prediction quality. In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 51–60 (2009)

Falkner, S., Klein, A., Hutter, F.: BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*. PMLR, 1437–1446 (2018)

Fowler, Martin: Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html, (2006). Accessed: 2020-01-01

Gallaba, K., Macho, C., Pinzger, M., McIntosh, S.: Noise and heterogeneity in historical build data: an empirical study of travis ci. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 87–97, (2018)

Ghaleb, T.A., Da Costa, D.A., Zou, Y.: An empirical study of the long duration of continuous integration builds. Empir. Softw. Eng. **24**(4), 1–38 (2019)

Ghaleb, T.A., da Costa, D.A., Zou, Y., Hassan, A.E.: Studying the impact of noises in build breakage data. *IEEE Trans. Softw. Eng.*, (2019b)

Ghotra, B., McIntosh, S., Hassan, A.E.: Revisiting the impact of classification techniques on the performance of defect prediction models. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering **1**, 789–800 (2015)

Goldberg, David E.: Genetic algorithms in search. *Optimization, and Machine Learning*, (1989)

Graves, Alex, Schmidhuber, Jürgen.: Framewise phoneme classification with bidirectional lstm and other neural network architectures. Neural Netw. **18**(5–6), 602–610 (2005)

Graves, Alex, Jaitly, Navdeep, Mohamed, Abdel-rahman: Hybrid speech recognition with deep bidirectional lstm. In: *2013 IEEE workshop on automatic speech recognition and understanding*. IEEE 273–278 (2013)

Guo, Baosu, Jingwen, Hu., Wenwen, Wu., Peng, Qingjin, Fenghe, Wu.: The tabu_genetic algorithm: a novel method for hyper-parameter optimization of learning algorithms. Electronics **8**(5), 579 (2019)

Haris, Muhammad, Hasan, Muhammad Noman, Qin, Shiyin: Early and robust remaining useful life prediction of supercapacitors using bohb optimized deep belief network. Appl. Energy **286**, 116541 (2021)

Harman, Mark, Mansouri, S Afshin, Zhang, Yuanyuan: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. (CSUR) **45**(1), 11 (2012)

Harman, Mark, McMinn, Phil, De Souza, Jerffeson Teixeira, Yoo, Shin: Search based software engineering: Techniques, taxonomy, tutorial. In: *Empirical software engineering and verification*. Springer, Heidelberg 1–59 (2010)

Hassan, F., Wang, X.: Change-aware build prediction model for stall avoidance in continuous integration. In: *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 157–162 (2017)

Hastie, Trevor, Tibshirani, Robert, Friedman, Jerome: The elements of statistical learning: data mining, inference, and prediction. Springer Science & Business Media, Berlin, Heidelberg (2009)

Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D.:Trade-offs in continuous integration: assurance, security, and flexibility. In: *11th Joint Meeting on Foundations of Software Engineering*, 197–207. ACM. (2017)

Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: *31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 426–437, (2016)

Hochreiter, Sepp, Schmidhuber, Jürgen.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)

Hochreiter, Sepp, Schmidhuber, Jürgen: Lstm can solve hard long time lag problems. In: *Advances in neural information processing systems*, 473–479 (1997b)

Hpbandster. https://automl.github.io/HpBandSter/build/html/index.html,. Accessed: 2021-12-16

Hsu, Chih-Wei, Chang, Chih-Chung, Lin, Chih-Jen, et al.: A practical guide to support vector classification, (2003)

Hyperopt framework. http://hyperopt.github.io/hyperopt/,. Accessed: 2021-12-16

jruby project. https://github.com/jruby/jruby,. Accessed: 2021-12-16

Javier Ordóñez, Francisco, Roggen, Daniel: Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. Sensors **16**(1), 115 (2016)

Jebnoun, Hadhemi, Ben Braiek, Houssem, Masudur Rahman, Mohammad, Khomh, Foutse: The scent of deep learning code: An empirical study. (2020)

Jelihovschi, Enio G., Faria, José Cláudio., Allaman, Ivan Bezerra: Scottknott: a package for performing the scott-knott clustering algorithm in R. TEMA (São Carlos) **15**(1), 3–17 (2014)

Karpathy, Andrej, Johnson, Justin, Fei-Fei, Li: Visualizing and understanding recurrent networks. arXiv preprint arXiv:1506.02078, (2015)

Krawczyk, Bartosz, Woźniak, Michał: Cost-sensitive neural network with roc-based moving threshold for imbalanced classification. In: *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, Heidelberg 45–52 (2015)

Krishna, R., Menzies, T., Fu, W.: Too much automation? the bellwether effect and its implications for transfer learning. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM 122–131 (2016)

Li, Hang, Gong, Xiu-Jun., Hua, Yu., Zhou, Chang: Deep neural network based predictions of protein interactions using primary sequences. Molecules **23**(8), 1923 (2018)

Li, Lisha, Jamieson, Kevin, DeSalvo, Giulia, Rostamizadeh, Afshin, Talwalkar, Ameet: Hyperband: A novel bandit-based approach to hyperparameter optimization. J. Mach. Learn. Res. **18**(1), 6765–6816 (2017)

Li, Yanxi, Dong, Minjing, Wang, Yunhe, Xu, Chang: Neural architecture search in a proxy validation loss landscape. In: *International Conference on Machine Learning*. PMLR, 5853–5862 (2020)

Lorenzo, P.R., Nalepa, J., Kawulok, M., Ramos, L.S., Pastor, J.R.: Particle swarm optimization for hyperparameter selection in deep neural networks. In: *Proceedings of the genetic and evolutionary computation conference*, 481–488, (2017)

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

Längkvist, Martin, Karlsson, Lars, Loutfi, Amy: A review of unsupervised feature learning and deep learning for time-series modeling. Patt. Recog. Lett. **42**, 11–24 (2014)

Malhotra, Ruchika, Khanna, Megha: An exploratory study for software change prediction in object-oriented systems using hybridized techniques. Auto. Softw. Eng. **24**(3), 673–717 (2017)

Mkaouer, Wiem, Kessentini, Marouane, Shaout, Adnan, Koligheu, Patrice, Bechikh, Slim, Deb, Kalyanmoy, Ouni, Ali: Many-objective software remodularization using nsga-iii. ACM Trans. Softw. Eng. Methodol. (TOSEM) **24**(3), 17 (2015)

Ni, A., Li, M.: Cost-effective build outcome prediction using cascaded classifiers. In: *International Conference on Mining Software Repositories (MSR)*, 455–458 (2017)

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

Optunity framework. https://optunity.readthedocs.io/en/latest/,. Accessed: 2021-12-16

Ouni, Ali, Kessentini, Marouane, Sahraoui, Houari, Inoue, Katsuro, Deb, Kalyanmoy: Multi-criteria code refactoring using search-based software engineering: An industrial case study. ACM Trans. Softw. Eng. Methodol. (TOSEM) **25**(3), 23 (2016)

Pascanu, Razvan, Mikolov, Tomas, Bengio, Yoshua: On the difficulty of training recurrent neural networks. In: *International conference on machine learning*, 1310–1318, (2013)

Provost, F.: Machine learning from imbalanced data sets 101. In: Proceedings of the AAAI'2000 workshop on imbalanced data sets, Vol. 68, No. 2000, pp. 1-3. AAAI Press (2000)

Rausch, T., Hummer, W., Leitner, P., Schulte, S.: An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: *Proceedings of the 14th international conference on mining software repositories*. IEEE Press, 345–355 (2017)

Replication package for ci build prediction. Available at : https://github.com/stilab-ets/DL-CIBuild, (2020)

Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J.: Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In: *annual meeting of the Florida Association of Institutional Research*, 1–33, (2006)

Ross Quinlan, J.: C4. 5: A programs for machine learning. Elsevier, Amsterdam (2014)

Saidani, Islem, Ouni, Ali, Mkaouer, Mohamed Wiem, Palomba, Fabio: On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. Inf. Softw. Technol. **138**, 106618 (2021)

Saidani, I., Ouni, A., Mkaouer, W.: Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Trans. Softw. Eng.* (2022)

Saidani, I., Ouni, A., Chouchen, M., Mkaouer, M.W.: Predicting continuous integration build failures using evolutionary search. *J. Inf. Soft. Technol.* 128, (2020)

Saidani, I., Ouni, A., Chouchen, M., Mkaouer, M.W.: Bf-detector: an automated tool for ci build failure detection. In: *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1530–1534 (2021b)

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

Schapire, Robert E.: Explaining adaboost. In: *Empirical inference*. Springer, Heidelberg 37–52 (2013)

Shan, Y., Hoens, T.R., Jiao, J., Wang, H., Yu, D., Mao, J.C.: Deep crossing: Web-scale modeling without manually crafted combinatorial features. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 255–262 (2016)

Shi, Y., Eberhart, R.C.: Parameter selection in particle swarm optimization. In: *International conference on evolutionary programming*. Springer, Berlin 591–600 (1998)

Shrikanth, NC., Majumder, S., Menzies, T.: Early life cycle software defect prediction. why? how? arXiv preprint arXiv:2011.13071, (2020)

Singh, A., Walenstein, A., Lakhotia, A.: Tracking concept drift in malware families. In: *Proceedings of the 5th ACM workshop on Security and artificial intelligence*. 81–92, (2012)

Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. arXiv preprint arXiv:1206.2944, (2012)

Sundermeyer, Martin, Schlüter, Ralf, Ney, Hermann: Lstm neural networks for language modeling. In: *Thirteenth annual conference of the international speech communication association*, (2012)

Sundsøy, Pål , Bjelland, Johannes, Reme, Bjørn-Atle, Iqbal, Asif M., Jahani, Eaman: Deep learning applied to mobile phone data for individual income classification. In *Proceedings of the 2016 International Conference on Artificial Intelligence: Technologies and Applications, Bangkok, Thailand*, 24–25 (2016)

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. **43**(1), 1–18 (2017)

Tantithamthavorn, Chakkrit, McIntosh, Shane, Hassan, Ahmed E., Matsumoto, Kenichi: The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng. **45**(7), 683–711 (2018)

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: The impact of automated parameter optimization for defect prediction models. IEEE Trans. Softw. Eng. **45**(7), 683–711 (2018)

Tharwat, Alaa, Hassanien, Aboul Ella: Quantum-behaved particle swarm optimization for parameter optimization of support vector machine. J. Classif. **36**(3), 576–598 (2019)
Travis ci. https://travis-ci.org/. Accessed: 2021-12-16
Tsymbal, Alexey: The problem of concept drift: definitions and related work. Comput. Sci. Depart., Trinity Coll. Dublin **106**(2), 58 (2004)
Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V.: Quality and productivity outcomes relating to continuous integration in github. In: *10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 805–816, (2015)
Wang, Yequan, Huang, Minlie, Zhu, Xiaoyan, Zhao, Li: Attention-based lstm for aspect-level sentiment classification. In: *Proceedings of the 2016 conference on empirical methods in natural language processing*, 606–615, (2016)
Wicaksono, Ananto Setyo, Supianto, Ahmad Afif: Hyper parameter optimization using genetic algorithm on machine learning methods for online news popularity prediction. Int. J. Adv. Comput. Sci. Appl **9**(12), 263–267 (2018)
Widmer, Gerhard, Kubat, Miroslav: Learning in the presence of concept drift and hidden contexts. Mach. Learn. **23**(1), 69–101 (1996)
Wilcoxon, Frank, Katti, S.K., Wilcox, Roberta A.: Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank T. Select. Tab. Math. Stat. **1**, 171–259 (1970)
Xia, Yufei, Liu, Chuanzhe, Li, YuYing, Liu, Nana: A boosted decision tree approach using bayesian hyper-parameter optimization for credit scoring. Expert Syst. Appl. **78**, 225–241 (2017)
Xia, Jing, Li, Yanhui: Could we predict the result of a continuous integration build? an empirical study. In: *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 311–315, (2017)
Xia, Jing, Li, Yanhui, Wang, Chuanqi: An empirical study on the cross-project predictability of continuous integration outcomes. In: *14th Web Information Systems and Applications Conference (WISA)*, 234–239. IEEE, (2017a)
Xie, Zheng, Li, Ming: Cutting the software building efforts in continuous integration by semi-supervised online auc optimization. In *IJCAI*, 2875–2881 (2018)
Yang, Li., Shami, Abdallah: On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing **415**, 295–316 (2020)
Zenisek, Jan, Holzinger, Florian, Affenzeller, Michael: Machine learning based concept drift detection for predictive maintenance. Comput. Indus. Eng. **137**, 106031 (2019)
Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E.: Cross-project defect prediction using a connectivity-based unsupervised classifier. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 309–320 (2016)
Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B.: The impact of continuous integration on other software development practices: A large-scale empirical study. In: *32nd IEEE/ACM International Conference on Automated Software Engineering*, 60–71 (2017)
Zheng, Jun: Cost-sensitive boosting neural networks for software defect prediction. Expert Syst. Appl. **37**(6), 4537–4543 (2010)
Zhou, Zhi-Hua., Liu, Xu-Ying.: Training cost-sensitive neural networks with methods addressing the class imbalance problem. IEEE Trans. Knowl. Data Eng. **18**(1), 63–77 (2005)

Authors and Affiliations

Islem Saidani[1] · **Ali Ouni**[1] · **Mohamed Wiem Mkaouer**[2]

✉ Ali Ouni
ali.ouni@etsmtl.ca

Islem Saidani
islem.saidani.1@ens.etsmtl.ca

Mohamed Wiem Mkaouer
mwmvse@rit.edu

1 ETS Montreal, University of Quebec, Montreal, Canada

2 Rochester Institute of Technology, Rochester, USA