Noname manuscript No.
(will be inserted by the editor)

An Investigation of Online and Offline Learning Models for Online Just-in-Time Software Defect Prediction

George G. Cabral · Leandro L. Minku · Adriano L. I. Oliveira · Dinaldo A. Pessoa · Sadia Tabassum*

Received: date / Accepted: date

Abstract Just-in-Time Software Defect Prediction (JIT-SDP) operates in an online scenario where additional training data is received over time. Existing online JIT-SDP studies used online Oza ensemble learning methods with Hoeffding Trees as base learners to learn and update JIT-SDP models over time in this scenario. However, it is unknown how these approaches compare against offline learning approaches adapted to operate in online scenarios, and how the use of any other online or offline base learners would affect online JIT-SDP in terms of predictive performance and computational cost. We therefore propose a new approach called Batch Oversampling Rate Boosting (BORB) that is able to use offline base learners in an online JIT-SDP scenario. Based on 10 open source projects, we provide a comprehensive evaluation of BORB with 5 different base learners and the existing online approach Oversampling Rate Boosting with 4 different base learners, both in within-project and cross-project online JIT-SDP scenarios. The results show that offline learning can lead to better predictive performance than the top performing online learning approaches considered in our study, at a higher computational cost. Cross-

* Authors in alphabetical order.

George G. Cabral
Department of Computing, Federal Rural University of Pernambuco, 52.171-900, Recife, BR
E-mail: george.gcabral@ufrpe.br

L.L. Minku (corresponding author) and S. Tabassum
School of Computer Science, University of Birmingham, Edgbaston, Birmingham, B15 2TT, UK
E-mail: sxt901@student.bham.ac.uk and E-mail: l.l.minku@bham.ac.uk

Adriano L. I. Oliveira
Center for Informatics, Federal University of Pernambuco, Recife, 50.740-560, BR
E-mail: alio@cin.ufpe.br

Dinaldo A. Pessoa
Banco Central do Brasil, R. da Aurora, 1259, Santo Amaro, Recife, 50040-090, BR
E-mail: dinaldoap@gmail.com

project data was helpful to improve predictive performance both for offline and online learning, but especially for online learning.

Keywords Within-project software defect prediction · cross-project software defect prediction · online learning · offline learning

1 Introduction

Software systems are nowadays essential in our everyday lives. The structures of these systems have been growing larger and ever more complex to fulfill the increasing demands from various different sectors. Hence, ensuring the quality of software systems has become a critical task. Improving software quality highly depends on reducing the amount of software defects. Currently, one of the most active research areas in the Software Engineering domain is Software Defect Prediction (SDP) [8, 18, 27]. The main objective of SDP is to predict which parts of the software are likely to contain defects so that resources such as time and budget can be more effectively allocated to support software quality assurance activities.

Early work on SDP typically focused on predicting defects in files or modules. In recent years, another branch of SDP has emerged that focuses on predicting defect-inducing software changes. This is known as Just-in-Time Software Defect Prediction (JIT-SDP). JIT-SDP predicts whether a change in the code will induce defects or not as soon as it is committed to a software repository. The main advantages of JIT-SDP over file level prediction are [22]: (i) predictions are performed at fine granularity, helping to reduce the effort required to fix the defects; (ii) the defect fixing task can be assigned to the right developer as changes can be easily mapped to the person who committed them; and (iii) defect prediction is made immediately after committing the change so that the code is still fresh in the developer's mind, facilitating code inspection.

Most of the existing JIT-SDP studies implicitly assume an *offline scenario*, where a pre-existing training set is available beforehand and additional training examples are never received anymore [21, 22]. However, in practice, JIT-SDP operates in an *online scenario*, where software changes become labelled as either clean or defect-inducing and become available as training data over time. McIntosh and Kamei [30] showed that there can be fluctuations in the importance of the characteristics of defect-inducing software changes over time, which may be a result of concept drift during the software development process. Concept drift can be described as a change in the data generating process, affecting the underlying probability distribution of the data. It may negatively impact predictive performance of the models, if they are predominantly built on old data. To deal with this issue, it is important for models to be able to learn and adapt to new data over time.

Both online and offline learning models can be used to learn additional data received over time in online scenarios. Online learning models are models that consider training examples one at a time, with the model parameters

being updated after the presentation of each training example [3]. Therefore, they naturally fit online scenarios, as they can be updated with each new training example generated by the online scenario separately, without requiring access to past data. Offline learning models are models that process the entire training set in one go [3]. Even though they require the whole training set to be available before learning commences and cannot process each training example separately upon arrival, they can also be adapted for use in online scenarios. This can be done by retraining the model on new data together with (a sufficient amount of) past data. Both offline and online learning models need to be combined with special strategies to deal with concept drift to be able to address changes in the underlying data distribution. However, as online learning models do not require retraining on past data, their training process is usually faster. Such advantage can also become a weakness depending on the problem being learned. In particular, as online learning models do not conduct multiple learning passes through the data, they may present poorer predictive power, for instance as a result of catastrophic forgetting [29].

Therefore, this paper aims at analyzing whether offline learning models can improve predictive performance in online JIT-SDP scenarios compared to online learning models, and whether this would be at the cost of higher computational requirements. This investigation will be carried out both on Within-Project (WP) and Cross-Project (CP) online JIT-SDP scenarios. In particular, these two scenarios may lead to different conclusions in terms of which type of learning models perform better, due to the different amounts of training data used. The following research questions (RQs) are addressed:

RQ1 Can offline learning help to improve predictive performance compared to online learning in online WP JIT-SDP scenarios? Which base learners usually perform best?
RQ2 How beneficial is CP data to improve predictive performance of offline models compared to online models in online CP JIT-SDP scenarios?
RQ3 How high is the computational cost of offline learning in online JIT-SDP scenarios compared to that of online learning models?

To answer the above research questions, we propose a new approach called Batch Oversampling Rate Boosting (BORB) that is able to use different offline base learners and can operate in the online JIT-SDP scenario by learning from new training data. BORB is an offline version of the online JIT-SDP approach Oversampling Rate Boosting (ORB) [7]. It translates the online resampling concepts of ORB which have been previously shown to be useful for online JIT-SDP scenarios [7] into an offline resampling approach. Therefore, in this paper, offline learning implies using BORB, whereas online learning implies using ORB, unless stated otherwise.

BORB and ORB approaches are compared using 5 and 4 different base learners, respectively. The use of different base learners enables a more complete investigation of offline and online learning, as different base learners may lead to different conclusions. Our experiments based on ten open source

projects show that offline learning (BORB) helped to improve predictive performance compared to online learning (ORB) when using most base learners with WP data. Even though CP data helped to improve BORB's predictive performance further, it was more helpful to improve ORB's predictive performance. The training process of online learning through ORB was less computationally expensive than that of offline learning through BORB. However, the magnitude of the differences in predictive performance and computational cost between the top ORB and BORB approaches were not very large.

The contributions of this work are following:

- We provide the first comparison between offline base learners and online base learners in a realistic online JIT-SDP scenario, revealing that offline learning can slightly improve predictive performance compared to online learning. Therefore, if researchers or practitioners have predictive performance as a priority when choosing a JIT-SDP model, we recommend them to consider offline JIT-SDP as a possible choice.
- We show how to adapt offline base learners so that they can use adaptive resampling rates to deal with class imbalance in online scenarios for JIT-SDP.
- We show that CP data can improve predictive performance. This happens both when using offline BORB and online ORB, even though CP data was particularly beneficial for online ORB. Therefore, we recommend researchers and practitioners to consider adopting CP learning especially if they are using online ORB.
- We show that online learning required less computational cost than offline learning. Therefore, we recommend researchers and practitioners to consider online learning if computational cost is a concern for them. This may be a concern when there is a need for comparing several JIT-SDP models to decide which one to adopt. However, it may not be a concern when adopting a single online or offline JIT-SDP model over time as the cost of these approaches becomes negligible in this context.
- While one may intuitively assume that it is better to use *online* learning models for *online* JIT-SDP scenarios, we show that both *online* and *offline* learning models can bring benefits in such realistic scenarios and are worth further exploring.

This paper is further organized as follows. Section 2 presents related work. Section 4 introduces the proposed approach. Section 5 presents the details of the investigated datasets. Section 6 explains the experimental setup for answering the RQs. Section 7 explains the results of the experiments. Section 8 presents threats to validity. Section 9 presents the conclusions and future work.

2 Related Work

This section discusses online and offline models for JIT-SDP using both WP and CP data. As our previous work [37, 38] also involves online CP JIT-SDP,

there are some commonalities between the related work listed here and that of those studies.

2.1 Offline WP JIT-SDP

Kim et al. [23] conducted one of the first studies on JIT-SDP. They proposed an approach to classify software changes, as defect-inducing or not, based on features extracted from the change metadata such as author name, commit hour, code entropy, lines of comments, cyclomatic complexity, etc. Their approach achieved an accuracy rate of 78% on average in a study involving 12 open source software projects. Śliwerski et al. [35] investigated the connection among defects in a defect-tracking system and a control version system in order to identify 'fix-inducing changes' (changes able to identify previous defect-inducing changes). They investigated which properties of these changes are correlated with inducing fixes. They showed, for example, that if the change is large it is more likely to induce a fix. Eyolfson et al. [14] showed that commits submitted during certain time of the day, day of the week and the daily commit frequency of the developer may influence the "bugginess" of a commit. Kamei et al. [22] performed a large scale investigation of JIT-SDP by building logistic regression models using 6 open source and 5 commercial projects. They used 14 different features extracted from code changes to predict defect inducing changes and achieved an average accuracy of 68% and an average recall of 64%.

Other studies focused on the machine learning approach being used to create JIT-SDP models. Chen et al. [10] considered JIT-SDP as a multi objective problem by maximizing the number of identified defect-inducing changes and minimizing efforts to fix them. They used logistic regression models to conduct the prediction using 6 open source datasets considering the 14 metrics described in [22]. Their approach managed to identify 63.8% of the defect-inducing changes on average when using only 20% of the software quality team effort. Yang et al. [45] proposed a two-layer ensemble learning approach TLEL. In the inner layer, they used Decision Trees and Bagging models to create a Random Forest model. In the outer layer, they grouped many different Random Forest models using stacking [1]. They have also investigated other base learners than Decision Trees, including Naive Bayes, Support Vector Machines, Linear Discriminant Analysis and Nearest Neighbor Classifiers. They showed that ensembles of Decision Trees to create Random Forests performed better than using other base learners in 5 out of 6 open source projects. Their approach detected 70% of defect-inducing changes by reviewing 20% of the code. The TLEL also achieved higher F1-score compared to three baseline approaches – Deeper, DNC and MKEL. Yang et al. [46] proposed a deep learning method called 'Deeper' for JIT-SDP. They compared their approach with the approach proposed by Kamei et al. [22] and showed that their approach was able to discover 32.22% more defects, based on a study with 6 open source datasets. Li et al. [26] investigated the impact of different combinations of base

learners such as Support Vector Machines (SVMs), Logistic Regression (LR), Random Forest (RF), Multi-layer Perceptron (MLP), Naive Bayes (NB) and Decision Trees (DTs). They showed that the diversity of base learners plays an important role for achieving promising performance.

Some studies have also suggested effort-aware prediction of defect-inducing software changes, leading to approaches such as EALR [22], CBS [19] and CBS+ [20]. However, the effort-aware components of these approaches require a whole set of software changes to be available for sorting in order of inspection priority. As being able to make predictions "just-in-time", at commit time, is one of the key advantages of JIT-SDP [22], it is unsuitable to wait for such whole set of changes to be produced for sorting in JIT-SDP.

All of the above discussed studies considered JIT-SDP in an offline scenario (i.e., all the learning algorithms used in these studies are offline and were not retrained with new data over time). These studies did not take into account the fact that the label of a training data may not be available immediately after the software change submission, i.e., they overlook a problem known as verification latency which consists in the delay for obtaining the class (or label) of a software change. The chronology of the data was also disregarded. As a result, in these works, future examples may have been used to train models for predicting past data. Hence, these offline WP JIT-SDP studies are not applicable in a realistic scenario.

2.2 Online WP JIT-SDP

Tan et al. [39] investigated JIT-SDP in a scenario where new batches of training examples arrive over time and can be used for updating the classifiers. To the best of our knowledge, even though previous work considered verification latency in defect models that are updated online [24], Tan et al. were the first work to consider this issue in JIT-SDP. Regarding the classifiers, they used 7 updatable algorithms based on Naive Bayes (Bayes, LWL), instance-based learning (IBK, KStar), boosting (LogitBoost), nearest-neighbors (NNge), and Support Vector Machines (SPegasos) to learn over time. In addition, they used resampling techniques to tackle the inherent class imbalance problem. Based on a study with one proprietary and six open source projects, the authors claim that both resampling techniques and the updatable classification improve the precision by 12.2-89.5%. In this work they mention that overlooking the data chronology and the verification latency problem lead to a false impression of higher predictive performance. Therefore, it is important to take the data chronology and verification latency problem into account in order to reproduce more realistic scenarios. However, their approach assumes that there is no concept drift, i.e., that the defect generating process does not suffer variations over time. Their approach also assumes a fixed gap of time between the training and test examples, where no training examples can be produced. In practice, some software changes may be found to be defect-inducing during that gap,

but their use for training will be delayed by their approach. Moreover, they have not compared online versus offline learning models in their work.

McIntosh et al. [31] performed a longitudinal case study of 37,524 changes from the rapidly evolving QT and OPENSTACK systems and found that fluctuations in the importance of the features of fix-inducing changes can impact the performance of JIT-SDP models. They showed that JIT-SDP models typically lose predictive power after one year, possibly as a result of concept drift. Hence, they suggest to continuously update the JIT-SDP model with recent data.

Cabral et al. [7] proposed a method called Oversampling Rate Boosting (ORB) to tackle a type of concept drift called class imbalance evolution, where the proportion of examples of the defect-inducing and clean classes change over time. Their work investigates an online JIT-SDP scenario taking verification latency into account. They considered a waiting time (w days) after the commit time to safely label the change as clean.If a defect is found within w days, the change is labeled as defect-inducing and used for training. If no defect associated to a change has been found in w days from its commit time, this gives confidence that this change is clean and therefore be labeled and used for training. If a change that has already been labeled as clean is found to be defect-inducing after w days, the training example corresponding to that change will be updated with the correct label and be presented again for learning.

ORB has a resampling rate to tackle class imbalance evolution based on the moving average over the predictions provided by the JIT-SDP model. In [7] this mechanism has shown to be able to improve predictive performance over JIT-SDP approaches that assume a fixed level of class imbalance. ORB achieved better $|R_0 - R_1|$ up to 45.38% and 63.59% compared to the state-of-the art class imbalance evolution algorithms Undersampling Online Bagging (UOB) and Improved Oversampling Online Bagging (OOB) [43], respectively.

2.3 Offline CP JIT-SDP

JIT-SDP classifiers require sufficient amount of training data to provide useful predictions. Such data is not available at the beginning of a software project as data arrives sequentially over time. Cross-Project (CP) JIT-SDP can overcome this issue by using data from past projects to build the classifier. Several studies investigated CP JIT-SDP. Kamei et al. [21] conducted one of the first studies. They carried out an empirical evaluation of CP JIT-SDP performance by using data from 11 open source projects. They investigated five CP JIT-SDP approaches based on project similarity, three variations of data merging approaches, and ensemble approaches where each model was trained on data from a different project. All approaches employed random forests as base learners. They found that simple merging of all CP data into a single training set and ensemble approaches obtained similar predictive performance to that of WP models. Different from SDP at the component level, other more complex

approaches, including similarity-based approaches, did not offer any additional advantage compared to these. Another study [9] investigated CP JIT-SDP for mobile platforms using 14 apps extracted from the CommitGuru platform [34]. They compared the CP performance of four different well-known classifiers and four ensemble techniques. Naive Bayes performed best compared to other classifiers and some ensemble techniques. They did not check how CP compared against WP results.

Chen et al. [10] considered JIT-SDP as a multi-objective problem to maximize the number of identified defect-inducing changes while minimizing the effort required to fix the defects. They proposed a multi-objective optimization-based supervised method called MULTI to build logistic regression JIT-SDP models. They used six open source projects. MULTI was evaluated on three different performance evaluation scenarios (cross-validation, cross-project-validation, and timewise-cross-validation) against 43 state-of-the-art supervised and unsupervised methods. They found that it can perform significantly better than WP methods in terms of Accuracy and P_{opt} metrics. Zhu et al. [47] proposed a JIT-SDP approach called DAECNN-JDP based on denoising autoencoder and convolutional neural networks. WP and CP defect prediction experiments were performed on six large open source projects and DAECNN-JDP was compared with 11 baseline models, including eight machine learning models, EALR, Deeper and CNN-JDP. The results show that DAECNN-JDP achieved better predictive performance than the baseline models for both CP and WP JIT-SDP. However, the predictive performances of CP and WP approaches were not compared against each other.

The studies above considered offline scenarios where the model is never updated with new data and, hence, cannot deal with concept drift. They did not take into account the chronology and verification latency of the data as well. It is unknown whether their conclusions would hold in realistic online JIT-SDP scenarios.

2.4 Online CP JIT-SDP

Tabassum et al. [37, 38] first investigated CP learning for online JIT-SDP based on OOB [42] and ORB [7]. They proposed three online CP approaches called AIO (that builds a single model by training with all WP and CP data together), Filtering (that filters out CP instances dissimilar to target project) and Ensemble (that builds an ensemble of models, where each model is trained by data from a different project) based on Hoeffding tree as base learners. Their study based on 10 open source and 9 proprietary datasets showed that their online CP approaches (AIO and Filter) achieved up to 53.89%, 37.35% and 29.03% improvements in terms of G-Mean compared to a WP online approach. They have also shown that enabling the CP approaches to be updated with additional training data received over time in an online CP scenario leads to better predictive performance than adopting an offline CP scenario, where only CP data available before the target project commences is used for training.

3 Background

This section explains the online JIT-SDP scenario adopted in this work and some background required to understand it. It also explains the ORB approach upon which BORB is based.

3.1 Definitions

Definition (Data Stream): A data stream is a potentially infinite sequence of training examples $\mathcal{S} = \{(\vec{x}_i, y_i)\}_{i=1}^{\infty}$, where i is a natural sequential number (time step) indicating the order with which the training examples were labeled, $\vec{x}_i$ are the input features describing example i, and y_i is the label of example i. In JIT-SDP, the input features are features describing the software change, as will be explained in Section 5. The label is defect-inducing or clean.

Definition (Online Scenario With Verification Latency): An online scenario is a scenario where training examples are produced over time, forming a data stream. JIT-SDP operates in an online scenario where the labels of the software changes arrive with a delay, which is referred to as verification latency [12]. Specifically for JIT-SDP, labelled examples can be produced following the procedure defined by Cabral et al. [7]. When a change is committed to the repository the developers hope it to be clean, but it may, instead, induce a defect. To label this change as clean, we need to wait for a period of time (waiting period w) to be confident that the change is really clean. If no defect is reported to be associated to this change within w days, the change is labeled as clean at the end of the waiting period, producing a training example. If, on the other hand, a defect is found to be linked to this change during these w days, the change is immediately labeled as defect-inducing, without having to wait until the end of the waiting period. It may also happen that a change that was initially labeled as clean is found to be defect-inducing after the w days. When this happens, this change is relabeled as defect-inducing, producing a new labeled training example of the defect-inducing class. This procedure respects chronology, being able to capture a realistic scenario that reflects the labelling procedure that would be observed in practice. The waiting period w can be considered as a pre-defined parameter.

Definition (Online Learning): Given a data stream formed by training examples ordered by the time they were produced $\mathcal{S} = \{(\vec{x}_i, y_i)\}_{i=1}^{\infty}$, an online learning model is a model that is immediately updated whenever a new training example $(\vec{x}_i, y_i) \in \mathcal{S}$ becomes available. Strict online learning models must be able to process (learn) each training example once and only once. So, the classifier is always updated with new examples, without requiring any re-training on past examples. This is useful to speed up learning for cases where storing and reprocessing past training examples may be computationally infeasible, e.g., for very large data streams, or data streams where the frequency of incoming data is very large. However, some (non-strict) online learning algorithms may access a memory containing past training examples to support

the learning process. The classifier may also have strategies to speed up adaptation to changes (a.k.a., concept drifts) that may affect the underlying data generating process.

Definition (Offline Learning): Consider a finite set $\tau = \{(\vec{x}_i, y_i)\}_{i=1}^{n}$ containing n examples that are available for training at a given time. This set can be referred to as a batch. Being a set and not a stream, the time order of these examples is ignored. An offline learning model is trained on τ such that the training and testing phases cannot intersect in time, i.e., the classifier is only available to use when the training procedure has ended.

3.2 Discussion on Adopting Offline Learning in Online Scenarios

Even though the time order of the examples within τ is ignored by the offline learning procedure, it is still possible to create a sequence of training sets τ_j, $j \geq 0$, where each training set τ_j is updated with one or more new training examples that may become available until the current time step. We refer to the number of time steps that we wait before creating a new training set as retraining period (rp). At every rp time steps, τ_j is created with all τ_{j-1} training examples plus the new rp training examples. The batch τ_j is then used to retrain the offline learning model from scratch. The larger the rp, the longer we will have to wait before the predictive model can be retrained. If a concept drift happens during this period, the outdated model is unable to react to this drift until the new τ_j is created, potentially hindering the predictive performance. On the other hand, the larger the rp, the higher the computational cost of the approach, as the model is retrained from scratch more often.

Despite the training process of the offline learning model ignoring the time order of examples *within* a given training set, this process would still ensure that only training data that is really available at a given point in time would be used for training, i.e., the JIT-SDP online scenario described in this section would still be respected.

As the data stream generated by the software changes submitted to a software repository is not a high frequency stream, it may be computationally acceptable to store past changes and rebuild classifiers from scratch when new training sets become available. Moreover, managing the whole historical data stream enables us to access all the benefits of offline learning over online learning. In particular, by ignoring the time order of examples, offline learning models frequently process the training set several times, which can help to produce stronger (more accurate) classifiers. In face of verification latency, revisiting all software changes labeled so far allows us to delete any training examples whose label was incorrectly assigned as clean for software changes that have now been found to be defect inducing. This prevents the classifier to learn noisy information, different from online learning models such as ORB [7], where the mislabeled training example is definitely incorporated into the classifier. Nevertheless, a potential disadvantage of using offline learning for

online scenarios is that this could make it more difficult to deal with certain types of change in the defect generating process, as each given training set may contain a mix of examples produced by different defect generating processes.

3.3 The ORB Approach

Cabral et al. [7] tackled the problem of the class imbalance evolution over time when dealing with online JIT-SDP. They showed that this evolution negatively impacts the predictive performance by making the classifier to become highly skewed towards one of the classes during different periods of the project. They also considered the verification latency problem for receiving the class labels. Their proposed Oversampling Rate Boosting (ORB) approach was able to improve the predictive performance in comparison to algorithms that assume a fixed imbalance ratio over time and to the existing class imbalance evolution algorithms Undersampling Online Bagging (UOB) and Improved Oversampling Online Bagging (OOB) [43].

Algorithm 1 Oversampling Rate Boosting (ORB) [7]

Input: Ensemble size n, incoming training example d, parameters of the adjustment function (th, l_0, l_1, m), noise mechanism parameter o, decay factor θ', window size w_s

1: **for** each training example $d^{(t)} = (x^{(t)}, y^{(t)})$, $t \leftarrow 0$ to ∞ **do**
2: Obtain the ensemble prediction $y^{(t)}$ for $x^{(t)}$
3: Compute the average $ma^{(t)}$ over the predictions on the most recent w_s examples, including $d^{(t)}$
4: Update the proportions $\rho_0^{(t)}$ and $\rho_1^{(t)}$ of each class using Eq. 1
5: **for** $i \leftarrow 0$ to n **do**
6: $\lambda = 1$
7: **if** $y^{(t)} == 1$ and $\rho_1^{(t)}$ ¡ $\rho_0^{(t)}$ **then**
8: $\lambda = \rho_0^{(t)} / \rho_1^{(t)}$
9: **end if**
10: **if** $y^{(t)} == 1$ and $\rho_0^{(t)}$ ¡ $\rho_1^{(t)}$ **then**
11: $\lambda = \rho_1^{(t)} / \rho_0^{(t)}$
12: **end if**
13: Set $k \sim Poisson(\lambda)$
14: Calculate $OBF^{(t)}$ $(ma^{(t)}, th, l0, l1, m)$ using Eq. 2 or Eq. 3
15: $k = k \; . \; OBF^{(t)}$
16: Run noise safety mechanism with parameter o
17: /* Depending on the noise safety mechanism outcome, update the ith Hoeffding tree with k copies of d_t */
18: Update(HT_i, k, d_t)
19: **end for**
20: **end for**

$$\rho_c^{(t)} = \theta' \rho_c^{(t-1)} + (1 - \theta')(y^{(t)} == c) \; , \tag{1}$$

Algorithm 1 shows the pseudocode of the ORB approach. It is important to note that the ORB is built upon the OOB [43] approach. Thus, in Algorithm 1, the numbered black lines correspond to the original OOB while the

blue lines correspond to the ORB. ORB calculates the moving average of the predictions $ma^{(t)}$ using a time window of size w_s. JIT-SDP is a binary problem where 0 represents the clean class and 1 represents the defect-inducing class. Calculating the moving average allows us to detect a bias in the predictions towards any particular class. Depending on this bias, the resampling rate of one of the classes is boosted (increased).

As JIT-SDP is a class imbalanced problem, an effective classifier would provide class imbalanced predictions. So, ORB is set to make the predictions rate as close as possible to a parameter th that represents the desired imbalance ratio of the predictions. For example, if $ma^{(t)}$ is close to 1, it means that the classifier is producing many false alarms, then the resampling rate for the class 0 (clean class) will be increased in order to reduce the classifier's skew, making it closer to the desired skew th. The adjustment in the resampling rate is made through boosting factors computed according to Equations 2 and 3. The final oversampling rate is then the product between obf_0 or obf_1 and the resampling rate k necessary to balance the classes computed by OOB. These boosting factors are responsible for adding an extra emphasis to one of the classes in order to yield balanced predictions.

$$OBF_0^{(t)}(P_0) = \begin{cases} \left(\frac{m^{ma^{(t)}} - m^{th}}{(m - m^{th})} * l_0 \right) + 1, \text{if } ma^{(t)} > th \\ 1, \text{ otherwise} \end{cases} \tag{2}$$

$$OBF_1^{(t)}(P_1) = \begin{cases} \left(\frac{m^{(th - ma^{(t)})} - 1}{(m^{th} - 1)} * l_1 \right) + 1, \text{if } ma^{(t)} \leq th \\ 1, \text{ otherwise} \end{cases} \tag{3}$$

In equations 2 and 3, P_0 and P_1 are sets of hyperparameters containing the parameters: m - determines the growth of the exponential function, th - stands for the threshold that indicates the desired class imbalance in the predictions; ma^t - the predictions moving average at time t; l_0 and l_1 - control the maximum boosting factor values.

In short, if $ma^{(t)} \leq th$, this suggests that less than th% of the commits are being classified as defect-inducing. Hence, the resampling rate of the defect-inducing class should be boosted. If $ma^{(t)} > th$, then more than th% of the commits are classified as defect-inducing. Hence, the resampling rate of the clean class should be boosted. For further details regarding the ORB, please refer to [7].

4 Proposed Approach

To investigate the influence of offline learning in online JIT-SDP scenarios we propose a novel approach called Batch Oversampling Rate Boosting (BORB), which consists of an adaptation of Oversampling Rate Boosting (ORB) [7]. Our RQs require us to isolate the effects of offline vs online learning as much as possible, so that we can analyze the potential benefit of offline learning without

Algorithm 2 BORB's testing and training procedure

Input: data stream $\mathcal{S}$, ORB parameters $(th, l0, l1, m)$, window size w_s, training sample size n, retraining time period rp, number of training iterations it, waiting time period w

$\theta \leftarrow 0.5$ ▷ decision threshold adopted by the classifier
$\mathcal{X} \leftarrow \emptyset$ ▷ set of software changes received so far
for $x^{(t)} \in \mathcal{S}$ **do** ▷ $x^{(t)}$ change (example) committed at timestep t
  if clf is already trained **then**
    $\vec{c} \leftarrow \emptyset$
    for $i \leftarrow (w_s - t)$ to t **do**
      $\vec{c} \leftarrow \vec{c} \cup score(clf, \theta, x^{(i)})$
    end for
    $\theta \leftarrow quantile(\vec{c}, th)$
    $\hat{y}^{(t)} \leftarrow pred(clf, x^{(t)}, \theta)$
  else
    $\hat{y}^{(t)} \leftarrow 0$ ▷ for this problem, class 0 refers to the clean class
  end if
  $\mathcal{X}.add(x^{(t)})$
  $\tau \leftarrow updateTrainingSet(\tau, \mathcal{X}, w)$ ▷ update the training set with any new label
  if modulo$(t, rp) == 0$ **then** ▷ an entire training procedure is performed
    $clf \leftarrow$ restart(clf) ▷ restart or instantiate a new classifier
    $obf_0 \leftarrow 1$
    $obf_1 \leftarrow 1$
    for $i \leftarrow 0$ to it **do**
      $S \leftarrow skewedSample(\tau, obf_0, obf_1, n)$
      $clf \leftarrow train(clf, S)$
      $ma \leftarrow \frac{1}{w_s} \sum_{j=t-w_s}^{t} pred(clf, x^{(j)}, 0.5)$
      $obf_0 \leftarrow OBF_0(ma, th, l_0, m)$
      $obf_1 \leftarrow OBF_1(ma, th, l_1, m)$
    end for
  end if
end for

being affected by other mechanisms that one may design to further improve predictive performance of the state-of-the-art. Therefore, such adaptation was designed to be as similar as possible to the ORB approach, but using core offline learning mechanisms instead of online ones.

BORB is an offline learning algorithm which periodically rebuilds its JIT-SDP models incorporating newly labeled training examples. This is achieved by updating the training set with the most recently labeled examples. The updated training set, containing all training examples received so far, is then used as a batch for retraining the JIT-SDP model from scratch based on an offline learning algorithm. Different from the offline learning approaches presented in Section 2.1, such training process ensures that only training examples whose labels are already available with respect to the dataset chronology can be used for training (i.e., it takes the verification latency problem into account and follows the online JIT-SDP scenario explained in Section 3).

Overall, BORB and ORB share the following similarities:

– Both methods use resampling to deal with class imbalanced based on the same oversampling rate boosting function (Equations 2 and 3).

- They are both capable of detecting when the classifier is performing badly based on the rule involving ma and th explained in Section 3.3 and to react to it by adjusting the resampling rate based on the above mentioned oversampling rate boosting function.
- They are both able to take into account verification latency through the waiting time strategy from [7].
- They both ensure that the online scenario explained in Section 3.1 is respected. In particular, both of them ensure that only training examples that are already available at a given point in time can be used for training at this point in time.

Their key differences are related to replacing the core online mechanisms of ORB by core offline ones:

- Being an offline learning approach, BORB stores and can learn multiple times past data, while ORB sees each training example only once.
- Being an offline learning approach, BORB collects the training examples into a training *set*. As such, the order of examples within this set is not respected when training on them, even though the online scenario explained in Section 3.1 is respected.
- When collecting new training examples over time, BORB is able to note if these examples correspond to previously seen training examples whose label has changed due to a late detection of a defect associated to them. Therefore, BORB can replace the old mislabelled clean examples by the new corresponding defect-inducing ones. ORB is able to learn the new defect-inducing example, but is unable to remove the old example which has already been learned.
- BORB is periodically retrained to enable offline learning models to be used, whereas ORB learns each training example separately.

Algorithm 2 presents BORB's pseudocode. For each new incoming software change ($x^{(t)}$) received at timestep t, the base learner clf, if already trained, provides a class prediction (line 10). Note that clf is not useful until τ contains at least one labeled software change from each class (clean and defect-inducing). Before that, all provided predictions are assigned to the clean class.

Different from online learning models, BORB stores all historical software changes in $\mathcal{X}$. As new class labels arrive (following the procedure described in Section 3 and using waiting period w), they are immediately used to create training examples corresponding to their respective software changes in $\mathcal{X}$ (line 15). These training examples are added to the training set τ. If a given new defect-inducing class label corresponds to a software change that was previously labeled as clean, the previous training example in τ is replaced by the new one with the defect-inducing label. The base learner is periodically reset whenever the modulo operation between the timestep (t) and the parameter rp is zero (line 16), and the training set τ is used to retrain it (lines 16 to 27).

BORB tackles the class imbalance problem at two different moments: in the test phase by picking a presumable adequate classifier prediction threshold

(lines 5 to 9) and in the training phase by means of an oversampling mechanism (lines 16 to 27). For the testing phase, BORB considers that classifiers usually make predictions based on a prediction threshold θ. The value of θ is typically set to 0.5. If the score given by the classifier is smaller than 0.5, class 0 is predicted. Otherwise, class 1 is predicted. However, this threshold can potentially be adjusted to help dealing with class imbalance. BORB does that in lines 5 to 9. In particular, $\vec{c}$ (line 7) stores the prediction scores over the last w_s software changes. The decision threshold θ to be adopted by the base learner is a quantile in $\vec{c}$ corresponding to a hyperparameter th. This hyperparameter represents the proportion of the predictions that is targeted to be defect-inducing predictions (th). E.g., if $th = 0.5$, $\theta = \text{median}(\vec{c})$. As JIT-SDP is a class imbalanced problem, the target proportion should normally be less than 0.5.

For the training phase, similar to ORB [7], BORB deals with class imbalance based on oversampling as follows. The oversampling rate is used to decide whether and by how much to oversample examples of a given class for training the base learner. The oversampling rate is adjusted based on the predictions given to the most recent test software changes. This enables adjustments on the base learners without having to wait for the labels of these software changes. In particular, a proportion ma of predictions given to the defect-inducing class over the most recent w_s software changes is determined (line 23). This proportion is compared to the same hyperparameter th used in the test phase. If ma indicates that BORB is predicting the defect-inducing class more/less often than the target proportion th, we need to oversample the clean/defect-inducing class, so that BORB focuses more on learning how to identify examples of this class. The idea is that if S (i.e., a sample of the training set) is skewed towards the defect-inducing class, the classifier should also incorporate this skewness in its predictions.

The iterations from lines 20 to 26 are responsible for updating the base learner based on the oversampling rate. The function *skewedSample* (line 21) retrieves the sample S containing n training examples from τ, based on the oversampling rate, which is determined according to the factors obf_0 and obf_1, as detailed in Algorithm 3. Therefore, many training iterations will be performed on different training sets S in order to make the base learner converge to a skew respecting th. The idea is that if ma (line 23) gets more distant from th, the obfs are adjusted so that S contains the necessary class imbalance to make the base learner accumulate new biased information such that at the last training iterations ma approaches th.

As an illustrative example of the impact of *skewedSample* function, consider using a Multilayer Perceptron (MLP) as a base learner and $th = 0.4$ (i.e., classes proportions (0.6:0.4)). If in the first epoch of the MLP the base learner average prediction for the last w_s test examples is 0.7, $th = 0.4$ and $ma = 0.7$ will be used to compute obf_0 and obf_1. The factors obf_0 and obf_1 will result in a new sample S containing training examples with the class proportions ($\frac{obf_0}{obf_0+obf_1}$: $1 - \frac{obf_0}{obf_0+obf_1}$), which will then be used for the second training epoch. Since in the first training epoch $ma > th$, in the second

Algorithm 3 *skewedSample* function

Input: n - the number of examples in the training set S, obf_0 and obf_1 - the absolute values for computing the desired proportions of clean and defect-inducing examples in S, $\tau_{(lab)}$

1: $S \leftarrow \emptyset$
2: **for** $i \leftarrow 0$ to n **do**
3: $r = \text{rand}(0, obf_0 + obf_1)$ ▷ r is a random number between 0 and $obf_0 + 0bf_1$
4: **if** $r < obf_0$ **then**
5: $S \leftarrow S \cup pickRandom(\tau, 0)$ ▷ S is incremented with a random example of the clean class from τ
6: **else**
7: $S \leftarrow S \cup pickRandom(\tau, 1)$ ▷ S is incremented with a random example of the defect-inducing class from τ
8: **end if**
9: **end for**

Return: S

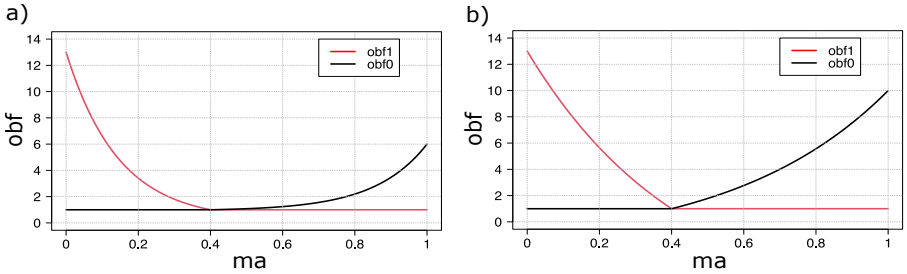


Fig. 1: Oversampling rate boosting function [7] for two different set of parameters. The x-axis is the average of the last W_s test examples while the y-axis depicts the resulting oversampling boosting factor (obf) according to Equations 2 and 3.

training epoch the proportion of examples from class 0 will be larger than the proportion of examples of class 1 (i.e., the oversampling rate for class 0 will be boosted). Eventually, repeating this process for many epochs will lead to the base learner's average predictions to be close to the target th.

As in ORB, Equations 2 and 3 [7] compute obf_0 and obf_1 (lines 24 and 25 of Alg. 1), respectively. Figure 1 presents the behaviour of these equations. Figure 1 a) shows the obf_0 and obf_1 curves generated by the parameters ($th = 0.4$, $l0 = 5$, $l1 = 12$, $m = 1000$ and $ma \in 0..1$) while in Fig. 1 b) the parameters $l0 = 9$ and $m = 10$. Due to different values for parameter m, in Fig. 1 a), when $ma \approx th$, obf_0 and obf_1 are less impacted than in Fig. 1 b). As in JIT-SDP the class 0 (clean class) is usually the majority class, it is advisable to use obf upper limits values (l_0) lower than the ones for defect-inducing class (l_1).

We investigated BORB for both WP and CP learning, respecting the online scenario introduced in Section 3. For WP learning, the JIT-SDP model is trained with data from the target project only. For CP learning, the JIT-SDP model is trained both with data from the target project (WP data) and from all other available projects (CP data). Hence, for CP learning, BORB

Table 1: An overview of the projects (adapted from [38])

Project	Total Changes	#Defect-inducing Changes	%Defect-inducing Changes	Median Defect Discovery Delay (days)	Time Period	Main Language
Tomcat	18960	5223	27.55	200.58	27-03-2006 - 06-12-2017	Java
JGroups	18434	3185	17.28	117.12	09-09-2003 - 05-12-2017	Java
Spring-Int	8750	2333	26.66	415.12	14-11-2007 - 16-01-2018	Java
Camel	30739	6360	20.69	27.73	19-03-2007 - 07-12-2017	Java
Brackets	17572	4143	23.58	14.69	07-12-2011 - 07-12-2017	JavaScript
Nova	48989	12430	25.37	88.56	28-05-2010 - 28-01-2018	Python
Fabric8	13483	2736	20.29	36.57	13-04-2011 - 06-12-2017	Java
Neutron	19689	4689	23.82	82.51	01-01-2011 - 27-12-2017	Python
Npm	7920	1407	17.77	111.51	29-09-2009 - 28-11-2017	JavaScript
BroadleafCommerce	15010	2531	16.86	42.58	19-12-2008 - 21-12-2017	Java

model is trained with both CP and WP data together similar to the All-in-One approach from [37, 38]. This means that any benefits of CP data mentioned in this study refer to the benefits obtained from combining both CP and WP data. This is reasonable in online scenarios, as both CP and WP data become available over time during the course of a project [38].

5 Datasets

We have used ten open source datasets extracted from open source GitHub repositories, which were made available by Cabral et al. [7] at `https://zenodo.org/record/2594681`. Table 1 shows details about these datasets. All datasets were extracted based on CommitGuru [34]. The change metrics include 14 metrics (input features) that can be divided into five groups: i) diffusion of the change, including input features NS (number of modified subsystems), ND (number of modified directories), NF (number of modified files), Entropy (distribution of modified code across each file), ii) size of the change, including input features LA (lines of code added), LD (lines of code deleted), LT lines of code in a file before the change), iii) purpose of the change, including input features FIX (whether or not the change is a defect fix), iv) history of the change, including input features NDEV (number of developers that changed the modified files), AGE (average time interval between the last and the current change), NUC (number of unique changes to the modified files) and v) experience of the developer that made the change, including input features EXP (developer experience), REXP (recent developer experience), SEXP (developer experience on a subsystem). These software change metrics have been shown to be adequate for JIT-SDP in previous work [22] and have been adopted in previous online JIT-SDP work [7, 37, 38].

6 Experimental Setup

This section explains the experimental setup for answering the RQs introduced in Section 1. To perform the analysis for RQ1, we compare the predictive performances of BORB-WP and ORB-WP [7] approaches; for RQ2, we compare

the predictive performances of BORB-WP, BORB-CP and ORB-CP [37, 38] approaches; and for RQ3, runtimes for BORB-WP, BORB-CP and ORB-CP are compared. Our analyses are based on various online and offline base learners, as listed in Section 6.1. For RQ1 and RQ2, we compared all approaches against a dummy classifier that predicts defect-inducing or clean uniformly at random. This is because being able to outperform a dummy classifier in terms of overall predictive performance means the JIT-SDP model was able to learn relevant JIT-SDP knowledge.

Given a certain project P, we are interested in using JIT-SDP to predict the software changes of P as defect-inducing or clean. Such predictions should respect chronological order according to the scenario explained in Section 3. Chronology is determined based on author timestamp, as recommended in [15].

When creating a predictive model for a given project P, WP approaches make use of only WP data from P for training. CP approaches make use of data from all projects for training, including P. The training procedure of all approaches at a given timestamp t ensures that only training examples that have already been labeled by timestamp t based on their chronology and waiting period are used for training, as explained in Section 3.1 [7]. Waiting period of 90 is used as in previous studies [7, 38] for open source data.

All approaches have been executed 30 times on each data set. A replication package can be found in the JIT-SDP-NN repository, `https://github.com/dinaldoap/jit-sdp-nn`. The datasets generated during and/or analysed during the current study are available in the JIT-SDP-DATA repository, `https://github.com/dinaldoap/jit-sdp-data`.

6.1 Base Learners

This section lists all the base learners that are investigated with the BORB and ORB approaches in this study. Altogether, our base learners were selected so as to: (1) cover a variety of different types of learning approaches for both offline and online learning (function-based, probabilistic and tree-based), as we wish to check what kind of online/offline model is most beneficial for JIT-SDP, (2) make the evaluation fair in the sense that we will select both online and offline approaches that are expected to achieve good results (in particular including Logistic Regression and Iterative Random Forest for fairness towards offline learning [10, 22, 26] and Oza Bagging of Hoeffding Trees and Naive Bayes for fairness towards online learning [7, 38, 40]) and (3) include the use of base learners that are the same as much as possible between online and offline learning (Logistic Regression and Multilayer Perceptron).

Overall, the following base learners were adopted by the ORB and BORB approaches in our experiments:

- ORB: Logistic Regression, Multilayer Perceptron, Naive Bayes and Oza Bagging of Hoeffding Trees.

– BORB: Logistic Regression, Multilayer Perceptron, Naive Bayes, Iterative Random Forest and Iterative Hoeffding Forest.

6.1.1 Offline Base Learners

– Logistic Regression (LR): Logistic regression is a well known offline linear classifier [25]. Its training requires iterating through all the training data multiple times (epochs), and it has been successfully used for offline JIT-SDP in previous work [10, 22]. LR approaches can be affected by multi-collinearity. To cope with that, the LR approach used in our experiments is regularized with elastic net, and the overall effect of elastic net is grouping correlated coefficients and selecting the groups that are relevant for the model.
– Multilayer Perceptron (MLP): MLP is an Artificial Neural Network that consists of three layers of interconnected nodes [17], being able to model any function. Training also requires iterating through all the training data multiple times (epochs) based on the backpropagation algorithm. It has been included here for being a universal approximator, able to model any function. MLP approaches can also be affected by multi-collinearity. To deal with that, the MLP adopted in our experiments is regularized with dropout. This avoids collinearity by disabling some input features on each step of backpropagation algorithm.
– Iterative Random Forest (IRF): IRF consists of an ensemble of CART decision trees [5]. It is similar to a Random Forest [4], but the decision trees are trained with different subsets of the training data to encourage more diversity. It was included here as previous work has shown that diversity is important in ensembles for JIT-SDP [26].
– Iterative Hoeffding Forest (IHF): IHF is the same approach as IRF, but using online Hoeffding trees as the base learners instead of CARTs. As the iterative ensemble approach itself is an offline learning approach, we classify this approach as an offline learning approach. We have adopted it with BORB in this study so that we can evaluate the benefits of the approach BORB itself compared with ORB, without being affected by the benefits of the offline decision tree over the online one. This evaluation can be conducted by comparing BORB-IHF against ORB-OHT.

6.1.2 Online Base Learners

– Logistic Regression (LR): despite Logistic regression being an offline algorithm, it is possible to set the number of epochs to one so that the algorithm becomes online. The downside of using logistic regression as an online learning algorithm is that the resulting model is likely to become weaker, i.e., to have poorer predictive performance.
– Multilayer Perceptron (MLP): similar to LR, it is also possible to set the number of epochs for training MLPs to one, so that MLPs become online

learning models. The downside is similar to that of LR, i.e., its predictive performance may considerably reduce when a single epoch is used.
- Naïve Bayes (NB): NB is a well known Bayesian classifier that can be trained through one pass over the training data. This approach is inherently online, as the equations used to update the model parameters can process each training example separately. It is included here because it has been successfully used in component-based software defect prediction [40].
- Oza Bagging of Hoeffding Trees (OHT): Oza Bagging is an online version of the Bagging ensemble learning algorithm. It requires a single pass through the training data to learn it. It is typically run with Hoeffding Trees, which are online decision trees suitable for large complex datasets [13]. Different from LR and MLP, it is not possible to make offline decision trees into online approaches by changing any of their hyperparameter values. Hoeffding Trees are a specific type of decision trees that can learn through a single pass through the training data. Due to its theoretical foundations on the Hoeffding bound, Hoeffding trees are able to produce online models with strong performance guarantees, reason why this approach is being adopted in this and in previous JIT-SDP work [7, 37, 38].

6.2 Performance Metrics

The metrics adopted for measuring predictive performance are Geometric Mean (G-Mean) of Recall0 and Recall1, where Recall0 is the recall on the clean class and Recall1 is the recall on the defect-inducing class. Different from biased metrics such as Matthews Correlation Coefficient, F1-Score, Accuracy, Precision and G-Mean of Precision and Recall, the G-Mean of $Recall_0$ and $Recall_1$ adopted in our work is a metric that is not biased by class imbalance [48], being suitable for class imbalanced problems such as JIT-SDP. For simplicity, we will refer to this metric simply as G-Mean from here onward. We have also chosen G-Mean instead of AUC because AUC incorporates several threshold values that are not meaningful in practice and makes comparison between approaches difficult, hence discouraged in the context of software defect prediction [36].

While computing the metrics in a prequential way, a fading factor is used to track changes in predictive performance over time as recommended for problems that may suffer concept drift [16]. As mentioned in our previous study [37], if the current example belongs to class i, $Recall_i^{(t)} = \theta Recall_i^{(t-1)} + (1-\theta)\mathbb{1}_{\hat{y}=i}$, where i is zero or one, t is the current time step, θ is a fading factor set to 0.99 as in [7], $\hat{y}$ is the predicted class, and $\mathbb{1}_{\hat{y}=i}$ is the indicator function, which evaluates to one if $\hat{y} = i$ and to zero otherwise. If the current example does not belong to class i, $Recall_i^{(t)} = Recall_i^{(t-1)}$. Also, $G\text{-}Mean^{(t)} = \sqrt{Recall_0^{(t)} \times Recall_1^{(t)}}$. It is worth noting that $Recall_0 = 1 - FalseAlarmRate$, i.e., false alarms are taken into account through Recall0 and G-Mean.

The performance metric used to measure the computational cost is the amount of time in seconds used to train and test the JIT-SDP models.

6.3 Statistical Tests and Effect Size

Scott-Knott procedure [33] is used to compare the performance obtained by all BORB and ORB approaches across datasets, ranking the models and separating them into subgroups. The use of statistical tests across datasets has been recommended by Demsar to reduce problems with multiple comparisons [11]. Each group of observations compared through the test corresponds to one learning approach run across all projects (data streams) as illustrated in points (1) and (2) of Figure 2. Therefore, given that we use 19 approaches (including the dummy approach) and 10 projects in our experiments, there are 19 groups with 10 observations each in the test. As recommended by Menzies et al. [32], this test uses non-parametric bootstrap sampling. This makes this a non-parametric test which is adequate for comparison across data sets [11]. Scott-Knott.A12 is used both to compare predictive performance in terms of G-Mean and computational cost in Seconds, but for the computation cost we remove the dummy approach, leading to 18 groups. This is because a comparison of computational cost against the dummy approach would be meaningless, as this approach does not spend any time on learning (there is no learning) and provides extremely fast predictions (it simply predicts randomly, rather than making predictions based on a predictive model). To rule out insignificant differences in performance, this test uses A12 effect size [41]. Approaches are placed in separate groups by Scott-Knott test only if the A12 size is significant [32]. We will refer to Scott-Knott based on Bootstrap sampling and A12 as Scott-Knott.BA12 in this paper. Smaller Scott-Knott.BA12 rankings are better rankings.

We also report the A12 effect sizes against the dummy approach for each learning approach on each dataset individually to support the analysis of predictive performance, as illustrated in Point (3) of Figure 2. Symbols [*], [s], [m] and [b] represent insignificant (A12 < 0.56), small (A12 ≥ 0.56), medium (A12 ≥ 0.64) and large (A12 ≥ 0.71) A12 effect size. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the corresponding WP approach.

6.4 Hyperparameter Tuning

Random search is used for hyperparameter tuning as suggested in [2] [28], which show that random search performed similar or better compared to grid search for hyperparameter optimisation. For each hyperparameter of each configuration of a classifier, a random value is chosen regarding the probability distribution specified (either uniform or log-uniform, depending on the hyperparameter being tuned). ORB and BORB (meta-models) are associated

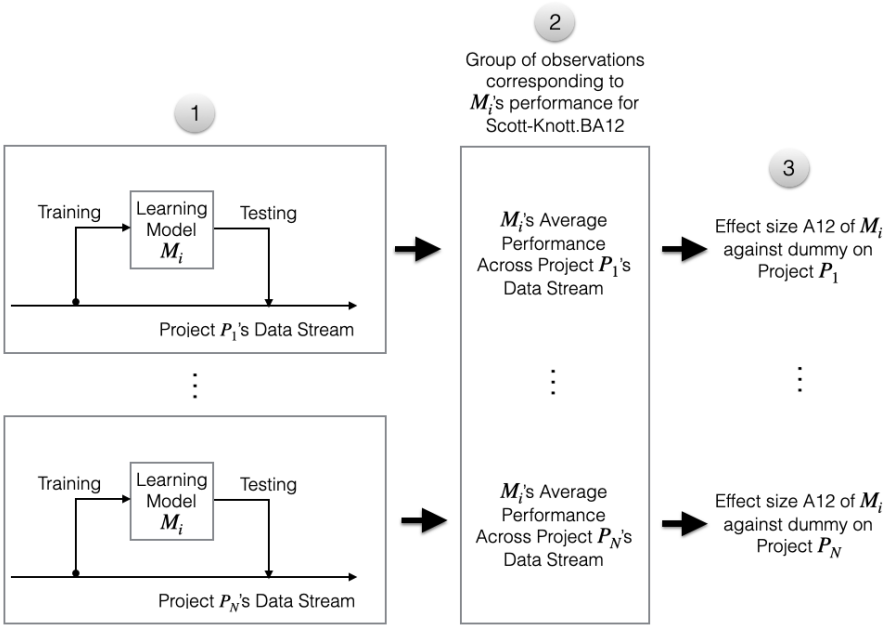


Fig. 2: Overview of Experiments. (1) Training and testing of a given learning model M_i on each Project P_j's data stream, where $1 \leq j \leq N$ and $N = 10$ is the number of projects. (2) Group of observations corresponding to learning approach M_i created for the Scott-Knott.BA12 test. (3) A12 effect sizes computed against a dummy approach.

with OHT, IHF, LR, MLP, NB and IRF (base learners). So, the overall configuration of the classifier includes the BORB, ORB and the base learner's hyperparameters. The first 3000 training examples from each data stream are used for hyperparameter tuning for both WP and CP approaches. For BORB and ORB approaches with each dataset, base learner and hyperparameter configuration, 3 executions have been performed for tuning purposes. For each dataset and classifier, 128 configurations were evaluated. More details of the investigated hyperparameter values are given in Table 1 in the appendix. It is worth noting that the application of log as a preprocessing step is considered as a hyperparameter choice when using MLP and LR as base models, as they can be affected by skewed distributions.

7 Experimental Results

Tables 2, 3, 4 and 5 present the average G-Mean with standard deviation and A12 effect size against the dummy classifier for the BORB and ORB approaches with different base learners, using WP and CP data. Table 6 presents the corresponding Scott-Knott.BA12 ranking of BORB and ORB approaches

with different base learners. Tables 7, 8, 9 and 10 present the average runtime. Table 11 presents the corresponding Scott-Knott.BA12 ranking. Section 7.1 focuses on the comparison between WP approaches to answer RQ1; section 7.2 focuses on the comparison between CP and WP approaches to answer RQ2; and section 7.3 focuses on the computational cost analysis of each approach to answer RQ3.

7.1 RQ1: Can offline learning help to improve predictive performance in online WP JIT-SDP scenarios? Which base learners usually perform best?

To answer RQ1, we compared the predictive performances of offline (BORB) and online (ORB) approaches with different base learners for WP data. As existing online WP JIT-SDP studies have never explored any other base learners except OHT, it is interesting to know whether using different base learners would improve the predictive performance not only of offine WP approaches, but also of online WP approaches.

From Table 6, we can see that offline WP approaches in general outperformed online WP approaches, being better ranked. In particular, BORB-MLP-WP, BORB-LR-WP and BORB-IRF-WP achieved similar predictive performance to each other, and better than the other BORB-WP and ORB-WP approaches. However, interestingly, when using the exact same base learner, NB for ORB and BORB, BORB-NB-WP did not outperform ORB-NB-WP. This suggests that BORB's adaptive resampling mechanism is not necessarily better than ORB's adaptive resampling mechanism, and that BORB's ability to enable offline base learners to be adopted is the likely reason for the generally better results obtained by the offline WP approaches.

When comparing BORB-MLP-WP against ORB-MLP-WP and BORB-LR-WP against ORB-LR-WP, it is thus clear that the single epoch used by the online base learners MLP and LR is the likely reason for the poorer predictive performance results obtained by ORB, rather than the different resampling mechanisms used by ORB and BORB. Such single epoch resulted in similar or worse predictive performance even than the dummy classifier, which is a very poor result.

When comparing BORB-IHF-WP and BORB-IRF-WP against ORB-OHT-WP, the offline IHF and IRF are also the likely the reason for the better predictive performance achieved by BORB. Nevertheless, the ranking of these tree-based ORB and BORB approaches is not far from each other – BORB-IHF-WP and BORB-IRF-WP were ranked second and ORB-OHT-WP was ranked third. This confirms our hypothesis mentioned in Section 1 in that OHT may be better suited for achieving good predictive performance in online JIT-SDP learning than MLP and LR.

When comparing the offline approach BORB-MLP-WP (ranked 2nd when considering all approaches investigated in this paper) against the online approach ORB-OHT-WP (ranked 3rd), we can see from Tables 2 and 3 that the absolute improvements in predictive performance obtained by BORB-MLP-

WP varied from 2.76% (for Nova) to 23.19% (for Spring-Integration), varying from small to moderate improvements and with median of with a median of 6.36%. Both these approaches performed better than the dummy approach with large effect size [b], except for Spring-Integration, where ORB-OHT-WP performed worse than the dummy with large effect size [-b].

From Fig. 3, it is visible that the best 3 BORB-WP approaches (MLP, LR and IRF as shown in blue, orange and green, respectively) performed on average very similar to each other, and comparatively better than the best ORB-WP approach (OHT as shown in black). Previous work [38] showed that WP approaches can suffer with low performance in the very beginning of a project, when there is lack of sufficient data. This initial period of low performance can be seen in most datasets for ORB-WP. BORB-WP approaches also suffered in such initial period, but was sometimes able to improve the G-Mean during the initial period (e.g., for Npm and Neutron).

Apart from the initial period, some other large performance drops can be observed in Camel, Npm and Spring-Integration (Fig. 3c, 3h and 3i) for ORB-WP. For these datasets, all 3 BORB-WP approaches managed to maintain stable performance during the drop periods. Hence, BORB-WP with MLP, LR and IRF are the best options compared to ORB-WP when considering the predictive performance.

It is worth noting that, if JIT-SDP often had concept drifts affecting the relationship between input features and the label (clean or defect inducing), retraining with all historical data as done by BORB-MLP-WP, BORB-LR-WP and BORB-IRF-WP would likely be detrimental to the predictive performance. This is because different portions of the training set would correspond to different relationships. The models would thus try to learn a mix of relationships, being unable to learn any of the individual relationships well enough. However, we have found in previous work [6, 37] that, despite sometimes happening, changes in such relationship are much less common than changes in the values of the input features. Therefore, retraining with historical data multiple times may offer some benefits, as shown in this section.

RQ1: Offline learning helps to improve average G-Mean in the online WP JIT-SDP scenario. BORB-WP with MLP, LR and IRF ranked the best, and performed better than the top ORB-WP approach, which used OHT. In particular, the top offline approach BORB-MLP-WP achieved up to 23.19% absolute improvement in G-mean against the top online approach ORB-OHT-WP.

7.2 RQ2: How beneficial is CP data to improve predictive performance of offline models compared to online models in online CP JIT-SDP scenarios?

Previous studies [37, 38] investigated the use of CP data for online JIT-SDP and found that, with the use of CP data, online learners are exposed to more data and are able to improve the predictive performance of the JIT-SDP model compared to using only WP data. These studies also showed that CP data was

Table 2: Average G-Mean for BORB-WP

Dataset	BORB-NB-WP	BORB-MLP-WP	BORB-LR-WP	BORB-IRF-WP	BORB-IHF-WP
Tomcat	29.87(0.57) [-b]	69.08(0.23) [b]	69.58(0.04) [b]	68.54(0.48) [b]	66.55(0.32) [b]
JGroups	52.4(0.41) [b]	64.81(0.22) [b]	62.78(0.08) [b]	63.2(0.31) [b]	60.88(0.41) [b]
Spring-Int	41.18(0.83) [-b]	64.27(0.46) [b]	65.55(0.14) [b]	63.84(1.2) [b]	49.94(1.57) [s]
Camel	48.29(0.19) [-b]	69.97(0.13) [b]	70.42(0.06) [b]	69.91(0.16) [b]	65.18(0.17) [b]
Brackets	61.21(0.08) [b]	72.09(0.1) [b]	71.32(0.05) [b]	70.98(0.09) [b]	66.39(0.37) [b]
Nova	62.31(0.52) [b]	80.62(0.07) [b]	79.79(0.03) [b]	79.9(0.11) [b]	78.74(0.07) [b]
Fabric8	55.32(0.22) [b]	62.8(0.53) [b]	64.46(0.08) [b]	65.3(0.48) [b]	63.75(0.3) [b]
Neutron	35.74(0.19) [-b]	83.39(0.1) [b]	82.12(0.03) [b]	81.54(0.24) [b]	82.14(0.16) [b]
Npm	44.6(0.22) [-b]	64.45(0.42) [b]	65.89(0.11) [b]	62.38(1) [b]	58.21(0.72) [b]
Broadleaf	54.5(0.25) [b]	70.73(0.21) [b]	70.13(0.1) [b]	68.77(0.73) [b]	65.57(0.26) [b]

Table 3: Average G-Mean for ORB-WP

Dataset	ORB-OHT-WP	ORB-NB-WP	ORB-MLP-WP	ORB-LR-WP
Tomcat	64.27(0.41) [b]	48.33(6.77) [*]	49.21(4.92) [-*]	46.68(0.59) [-b]
JGroups	60.89(0.48) [b]	55.47(1.11) [b]	46.6(4.85) [-b]	47.21(0.55) [-b]
Spring-Int	41.08(0.7) [-b]	50.24(1.92) [m]	39.71(8.32) [-b]	37.82(0.93) [-b]
Camel	53.17(0.8) [b]	51.9(1.86) [b]	48.64(4.47) [-s]	48.27(0.37) [-b]
Brackets	67.81(0.36) [b]	61.39(1.9) [b]	42.42(12.67) [-m]	44.7(0.57) [-b]
Nova	77.86(0.2) [b]	68.63(1.24) [b]	43.5(12.56) [-b]	46.31(0.32) [-b]
Fabric8	63.27(0.71) [b]	56.68(1.67) [b]	40.6(11.88) [-b]	42.92(0.88) [-b]
Neutron	77.03(0.47) [b]	39.99(5.4) [-b]	49.14(11.45) [-*]	49.52(0.51) [-b]
Npm	57.45(0.83) [b]	45.45(1.8) [-b]	48.77(5.02) [-s]	49.9(0.66) [*]
Broadleaf	64.25(0.83) [b]	55.7(2.32) [b]	46.61(5.92) [-b]	46.76(0.53) [-b]

Table 4: Average G-Mean for BORB-CP

Dataset	BORB-NB-CP	BORB-MLP-CP	BORB-LR-CP	BORB-IRF-CP	BORB-IHF-CP
Tomcat	46.84(0.67) [-b]	68.76(0.18) [b]	69.59(0.14) [b]	69.11(0.12) [b]	65.38(0.25) [b]
JGroups	43.83(0.62) [-b]	64.63(0.25) [b]	63.11(0.06) [b]	63.28(0.34) [b]	60.93(0.32) [b]
Spring-Int	44.13(1.39) [-b]	72.38(0.17) [b]	72.32(0.05) [b]	72.12(0.13) [b]	70.14(0.4) [b]
Camel	46.57(0.57) [-b]	70.42(0.11) [b]	69.96(0.03) [b]	69.51(0.07) [b]	66.74(0.19) [b]
Brackets	55.86(1.31) [b]	77.57(0.1) [b]	78.15(0.04) [b]	77.64(0.1) [b]	72.11(0.2) [b]
Nova	54.62(1.11) [b]	81.5(0.06) [b]	81.56(0.02) [b]	81.46(0.05) [b]	77.87(0.08) [b]
Fabric8	50.35(1.24) [s]	68.34(0.15) [b]	66.76(0.07) [b]	68.47(0.14) [b]	65.21(0.29) [b]
Neutron	70.82(1.77) [b]	82.8(0.08) [b]	83.49(0.07) [b]	83.34(0.07) [b]	80.77(0.19) [b]
Npm	48.29(1.37) [-b]	66.77(0.39) [b]	67.37(0.1) [b]	66.14(0.17) [b]	62.58(0.27) [b]
Broadleaf	50.02(0.99) [*]	71.26(0.14) [b]	71.54(0.05) [b]	71.04(0.12) [b]	70.38(0.25) [b]

Table 5: Average G-Mean for ORB-CP

Dataset	ORB-OHT-CP	ORB-NB-CP	ORB-MLP-CP	ORB-LR-CP
Tomcat	66.66(0.28) [b]	49.94(2.41) [-*]	48.16(3.87) [-m]	49.63(0.69) [-b]
JGroups	60.95(0.44) [b]	52.07(1.78) [b]	46.79(4.38) [-b]	47.6(0.61) [-b]
Spring-Int	70.36(0.61) [b]	52.2(3.87) [b]	48.3(4.36) [-s]	49.37(1.04) [-m]
Camel	69(0.25) [b]	55.66(1.53) [b]	47.54(5.64) [-b]	49.26(0.29) [-b]
Brackets	75.94(0.37) [b]	64.15(3.88) [b]	49.38(7.4) [-*]	48.55(0.43) [-b]
Nova	79.9(0.21) [b]	69.52(4.62) [b]	48.95(10.99) [-s]	49.49(0.23) [-b]
Fabric8	69.28(0.44) [b]	54.87(3.88) [b]	48.55(3.1) [-m]	49.77(1.02) [-s]
Neutron	81.94(0.25) [b]	79.04(3.35) [b]	51.53(8.11) [s]	49.55(0.43) [-b]
Npm	65.18(0.6) [b]	43.32(4.03) [-b]	47.41(3.76) [-b]	48.31(0.98) [-b]
Broadleaf	71.41(0.34) [b]	52.84(5.57) [b]	50.36(4.05) [-s]	48.24(0.88) [-b]

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach (which always gets 50% G-Mean). Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the corresponding Dummy approach.

Table 6: G-Mean ranking of approaches based on the Scott-Knott.BA12 test

Ranking	Approach				
1	BORB-MLP-CP	BORB-LR-CP	BORB-IRF-CP		
2	ORB-OHT-CP	BORB-MLP-WP	BORB-LR-WP	BORB-IRF-WP	BORB-IHF-CP
3	BORB-IHF-WP				
4	ORB-OHT-WP				
5	ORB-NB-CP				
6	ORB-NB-WP				
7	Dummy	BORB-NB-WP	ORB-LR-CP	BORB-NB-CP	ORB-MLP-CP
8	ORB-LR-WP	ORB-MLP-WP			

BORB approaches are shown in yellow, the Dummy approach is shown in red, and the ORB approaches are shown in white background. Scott-Knott.BA12 was run for all BORB and ORB-based WP and CP approaches together. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

helpful to maintain stable predictive performance during the periods when the WP models typically suffered sudden performance drops [37]. From section 7.1, we found that BORB's offline learners outperformed ORB's online learners with WP data. In particular, some offline models iterate over the training data several times, simulating the existence of a larger data set. It is unknown whether incorporating CP data with BORB would further improve predictive performance for offline models. Moreover, no other base learners were explored for online CP JIT-SDP except OHT. It is not known whether CP data would still be useful for JIT-SDP using other online base learners. Hence, it is important to investigate the use of CP data not only for offline models, but also for other online models than OHT.

To answer RQ2, we compare 4 approaches – ORB-WP, ORB-CP, BORB-WP and BORB-CP. According to the Scott-Knott.A12 test shown in Table 6, BORB-CP with MLP, LR and IRF are the best ranked approaches and outperformed all other BORB-WP, ORB-CP and ORB-WP approaches. Even though BORB-IHF-CP is also a BORB-CP approach, it did not rank best (instead it ranked second). Even though IHF is classified as an offline approach, it uses online Hoeffding Trees as base learners (Section 6.1). It is possible that using online Hoeffding Trees resulted into a weaker model for BORB.

When using the exact same base learner, NB, BORB-CP performed worse than ORB-CP. This corroborates the results presented in Section 7.1, suggesting that BORB's offline resampling mechanism is not necessarily better than that of ORB, and that its ability to enable the use of offline base learners is the likely reason for the typically better predictive performance achieved by BORB-CP approaches.

To address RQ2, we compare BORB-CP against BORB-WP approaches. We can see from Table 6 that BORB-MLP-CP (ranked first) outperformed BORB-MLP-WP (ranked second). Similarly, BORB-LR-CP also outperformed BORB-LR-WP, BORB-IRF-CP outperformed BORB-IRF-WP and BORB-IHF-CP outperformed BORB-IHF-WP. Only when using NB as the base learner, BORB-CP did not outperform BORB-CP, but both of these approaches are using online base models and performed worse than the dummy

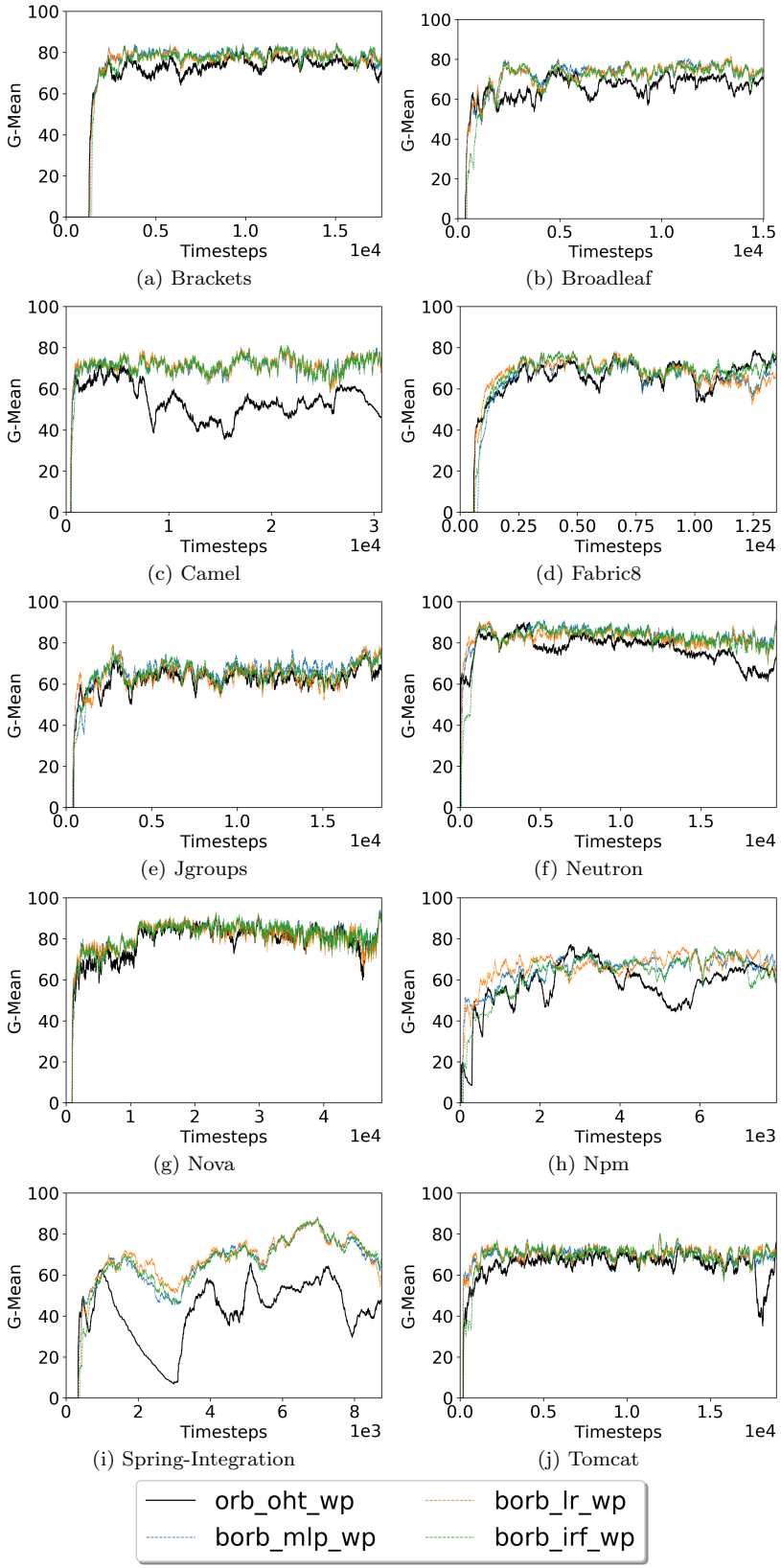


Fig. 3: G-Mean for all datasets through time for best ranked BORB and ORB approaches with WP data.

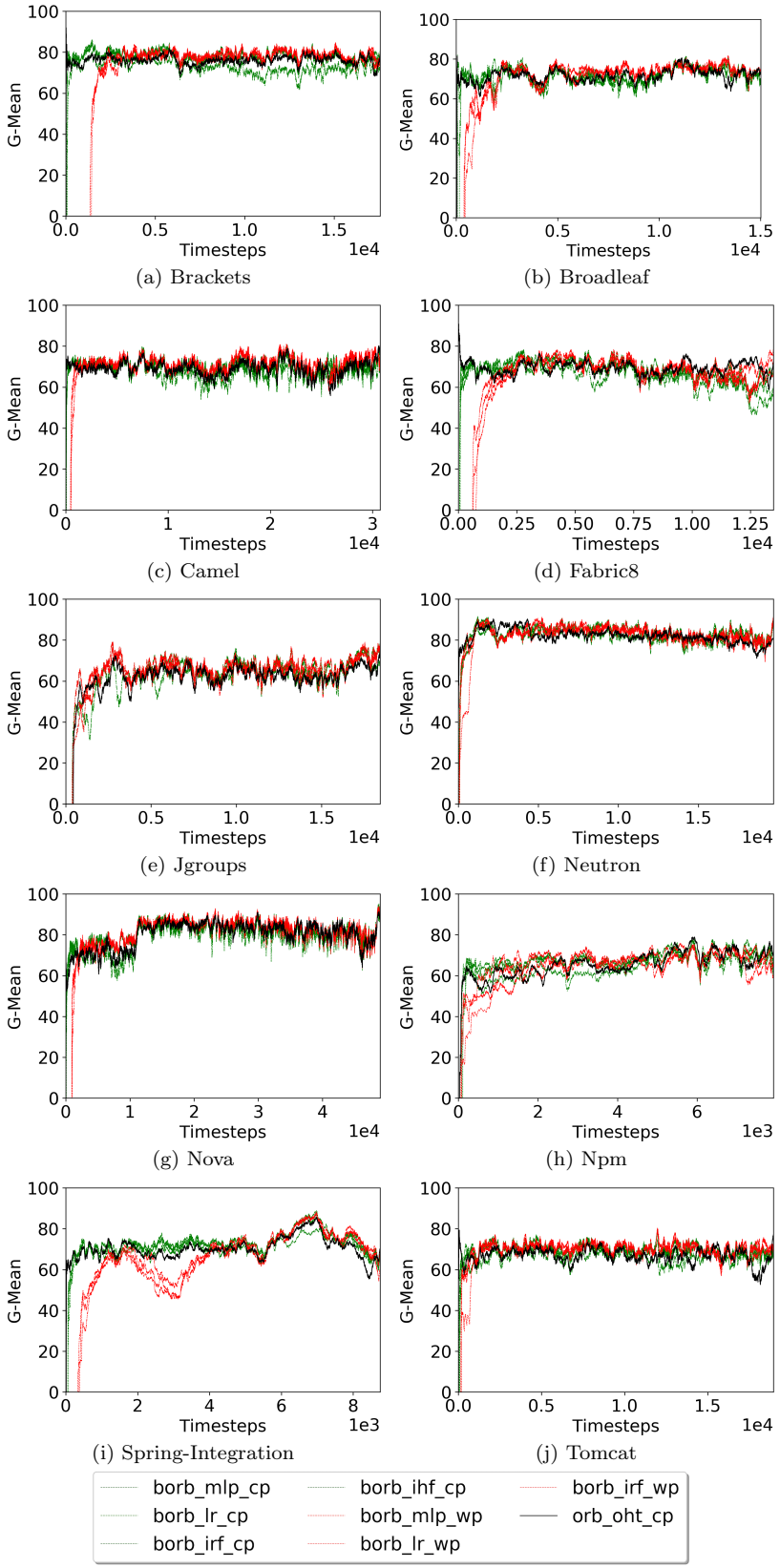


Fig. 4: G-Mean for all datasets through time for best ranked BORB and ORB approaches with WP and CP data.

classifier, meaning that the comparison between the two of them is not necessarily meaningful in this specific context. Therefore, these results show that CP data is helpful to improve predictive performance when using offline learning for JIT-SDP. However, the magnitude of the improvements in predictive performance were not very large. For instance BORB-MLP-CP (ranked 1st) approach had absolute improvements in G-Mean from 0.45% (for Camel) to 8.11% (for Spring-Integration) with a median of 2.32% against BORB-MLP-WP (ranked 2nd), and led to slightly worse G-Mean for some projects, as can be computed based on Tables 4 and 2.

We also compare ORB-CP against ORB-WP approaches. We can see from Table 6 that ORB-OHT-CP outperforemd ORB-OHT-WP, ORB-NB-CP outperformed ORB-NB-WP, ORB-LR-CP outperformed ORB-LR-WP and ORB-MLP-CP outperformed ORB-MLP-WP. Therefore, these results show that CP data is helpful to improve predictive performance when using online learning for JIT-SDP, for all base learners investigated. When comparing the best ORB-CP and ORB-WP approaches against each other (ORB-OHT-CP and ORB-OHT-WP), we can see that the absolute improvements in G-Mean varied from 0.06% (for JGroups) to 29.28% (for Spring-Integration) with a median of 6.59%. Therefore, CP data was more helpful for improving predictive performance in the context of online learning than offline learning.

Such larger increase in the competitiveness of the ORB approach when using CP data is also reflected in the magnitude of the differences in performance of the best BORB-CP approach (e.g., BORB-MLP-CP) against the best ORB-CP approach (ORB-OHT-CP). Even though BORB-MLP-CP was ranked better than ORB-OHT-CP, the absolute improvements in G-Mean varied from 0.86% (for Neutron) to 3.68% (for JGroups), being always small. Moreover, for some projects, BORB-MLP-CP obtained slightly worse G-Mean than ORB-OHT-CP. Therefore, even though BORB-CP can achieve better rank in terms of predictive performance than ORB-CP, the magnitudes of the differences in predictive performance are not large.

It is also worth noting that all best ranked BORB-CP approaches (BORB-MLP-CP, BORB-LR-CP and BORB-IRF-CP) outperformed the dummy classifier with large [b] effect size, for all projects. The best ranked ORB-CP approach (ORB-OHT-CP) also outperformed the dummy classifier with large [b] effect size for all projects. Therefore, the weakness of ORB-OHT-WP, which had performed worse than the dummy classifier for Spring-Integration, is overcome when using CP data.

From Fig.4, it is also visible that performance of best BORB-CP and ORB-CP were very similar. Both BORB-CP and ORB-CP were able to achieve better predictive performance during initial portion of the data streams than BORB-WP and ORB-WP. For Spring-Integration, BORB-CP managed to provide stable performance by eliminating the drops suffered by BORB-WP (Fig. 4i). These results suggest that CP data can be useful for both BORB and ORB during initial period of the project, and to help reducing drops in predictive performance over time for ORB.

RQ2: CP data was helpful to improve predictive performance for both offline and online learning approaches. It helped to improve predictive performance especially in the initial period of the projects for both BORB and ORB, and continued to help attenuating drops in predictive performance over time for ORB. Even though BORB-CP obtained better G-Mean rank than ORB-CP, the magnitude of the absolute differences in G-Mean between the best BORB-CP and the best ORB-CP approaches was not large. For instance, absolute improvements obtained by BORB-MLP-CP over ORB-OHT-CP were of up to only 3.68%.

7.3 RQ3: How high is the computational cost of offline learning in online scenarios compared to that of online learning models?

An ideal JIT-SDP approach should not be computationally too expensive as such approaches are not suitable to use in practice. Hence, while comparing between offline and online JIT-SDP approaches, it is important to consider computational cost (run time) along with the predictive performance. Typically, offline learners require multiple iterations of the training data leading to higher computational cost compared to online learners. Offline CP approaches could be even more computationally expensive than offline WP approaches as they require retraining with larger amount of data from several projects. An analysis of computational cost is required to understand how high these computational costs may be and whether they are feasible for adoption in practice.

Fig. 5a shows computational costs of BORB and ORB approaches for all datasets. We can see that offline (BORB) approaches have higher computational cost than their online (ORB) counterparts. This is also reflected by the Scott-Knott.BA12 tests shown in Table 11. For instance, ORB-NB-WP is better ranked than BORB-NB-WP, ORB-LR-WP is better ranked than BORB-LR-WP, ORB-MLP-WP is better ranked than BORB-MLP-WP. In particular, the better ranking obtained by ORB-NB-WP compared to BORB-NB-WP shows us that, even when using the exact same online base learner NB, ORB is still faster than BORB. The same is valid when comparing ORB-NB-CP against BORB-NB-CP. This means that the offline resampling and retraining process required by BORB is slower than ORB's procedures.

Moreover, the offline base learners MLP, LR, IRF and IHF adopted by BORB are also themselves generally slower than their online base learner counterparts. This is revealed by the magnitude of the differences in computational cost between BORB and ORB when using these base models, which is usually larger than the differences between BORB-NB and ORB-NB, as we can see from Figure 5. For instance, the magnitude of the difference in the computational cost between BORB-MLP-WP and ORB-MLP-WP is much larger than that between BORB-NB-WP and ORB-NB-WP. This is expected, as offline learning models frequently have to iterate through the dataset (or portions of

Table 7: Average run time in seconds for BORB-WP

Dataset	BORB-IHF-WP	BORB-IRF-WP	BORB-LR-WP	BORB-MLP-WP	BORB-NB-WP
Tomcat	910.58	454.93	315	242.39	332.7
JGroups	715.85	852.89	294.09	660.12	100.01
Spring-Int	131	234.67	31.65	464.56	27.38
Camel	1716.91	1878.08	1409.7	1246.01	91.71
Brackets	220.21	137.98	185.63	305.66	301.64
Nova	1891.89	1811.11	2497.66	1228.41	136.09
Fabric8	516.39	671.02	139.03	92.02	209.79
Neutron	1699.08	1068.19	784.8	652.13	353.53
Npm	583.67	188.23	134.03	409.15	111.06
Broadleaf	710.65	465.07	214.11	260.27	243.71

Table 8: Average run time in seconds for ORB-WP

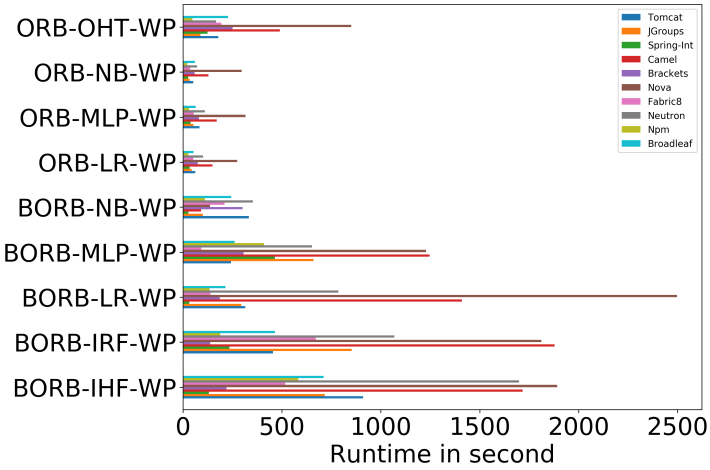
Dataset	ORB-LR-WP	ORB-MLP-WP	ORB-NB-WP	ORB-OHT-WP
Tomcat	61.94	83.7	52.48	178.4
JGroups	43.68	52.73	34.36	88.44
Spring-Int	33.66	38.77	25.82	124.14
Camel	148.73	170.75	128.25	489.43
Brackets	74.29	80.32	58.32	249.94
Nova	274.3	315.55	296.86	851
Fabric8	51.89	54.4	36.05	193.94
Neutron	101.13	110.94	71.44	167.34
Npm	26.97	30.93	21.9	47.63
Broadleaf	53.15	64.34	59.42	227.93

Table 9: Average run time in seconds for BORB-CP

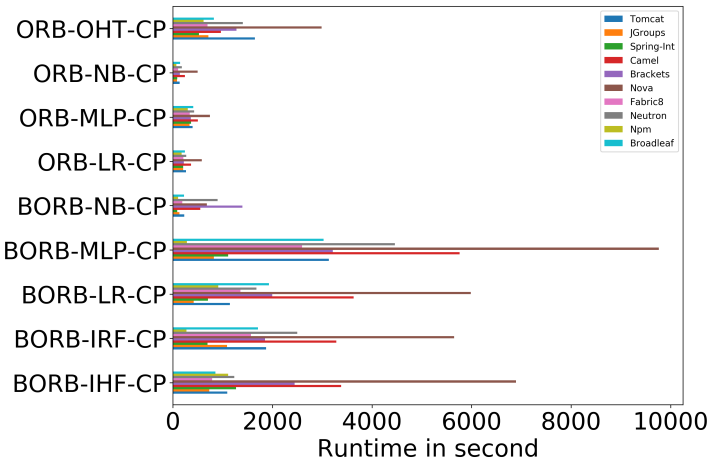
Dataset	BORB-IHF-CP	BORB-IRF-CP	BORB-LR-CP	BORB-MLP-CP	BORB-NB-CP
Tomcat	1092.86	1870.01	1146.12	3132.75	227.69
JGroups	728.93	1087.48	415.02	823.94	130.83
Spring-Int	1266.79	696.39	702.49	1109.44	86.17
Camel	3377.94	3280.28	3631.75	5758.18	549.3
Brackets	2445.16	1849.88	1994.94	3211.3	1396.35
Nova	6895.04	5650.16	5985.78	9761.95	679.11
Fabric8	788.67	1568.29	1356.3	2592.88	189.01
Neutron	1231.14	2496.1	1678.75	4460.94	896.24
Npm	1107.54	272.53	908.56	278.32	103.25
Broadleaf	852.48	1710.54	1929.85	3027.45	223.94

Table 10: Average run time in seconds for ORB-CP

Dataset	ORB-LR-CP	ORB-MLP-CP	ORB-NB-CP	ORB-OHT-CP
Tomcat	262.85	394.22	134.9	1646.8
JGroups	206.96	331.56	83.73	710.85
Spring-Int	205.72	362.32	85.34	523.09
Camel	364.98	499.95	240.91	964.85
Brackets	217.11	361.34	137.68	1275.75
Nova	578.34	744.04	494.33	2986.95
Fabric8	207.88	336.39	105.32	692.34
Neutron	267.73	425.42	177.64	1402.65
Npm	175.24	296.97	68.26	617.2
Broadleaf	239.91	407.53	144.37	823.43



(a) Runtime for BORB and ORB for WP data



(b) Runtime for BORB and ORB for CP data

Fig. 5: Computational Cost Analysis for BORB and ORB

it) several times to build the predictive model, whereas the online base learners require only one pass through the training examples.

We can also observe that all CP approaches have higher computational cost than their WP counterparts (note the different scale of the x-axis in Figures 5a and 5b). This is also confirmed by the Scott-Knott.BA12 results shown in Table 11. For instance, ORB-OHT-CP has higher computational cost than ORB-OHT-WP, whereas BORB-IRF-WP has higher computational cost than BORB-IRF-CP. This is also expected, as CP training sets are much larger than WP ones.

Table 11: Runtime ranking of approaches based on the Scott-Knott test

Ranking	**Approach**		
1	ORB-NB-WP	ORB-LR-WP	
2	ORB-MLP-WP		
3	ORB-NB-CP		
4	ORB-OHT-WP	BORB-NB-WP	
5	BORB-NB-CP	ORB-LR-CP	
6	BORB-LR-WP	ORB-MLP-CP	BORB-MLP-WP
7	BORB-IRF-WP		
8	BORB-IHF-WP		
9	ORB-OHT-CP		
10	BORB-IHF-CP	BORB-LR-CP	BORB-IRF-CP
11	BORB-MLP-CP		

Overall, this shows us that offline learning is in general slower than online learning, and that CP learning is in general slower than WP learning. In practice, one would be interested in adopting an approach that has low computational cost but high predictive performance. Therefore, we compared the computational cost of the offline and online models that obtained the best predictive performance.

The best offline approaches in terms of predictive performance are BORB-MLP-CP, BORB-LR-CP and BORB-IRF-CP (Table 6). As they are all ranked the same in terms of predictive performance, we pick the one with lowest computational cost for this comparison. As both BORB-LR-CP and BORB-IRF-CP have the same rank in terms of computational cost but have better rank then BORB-MLP-CP (Table 11), we randomly pick BORB-LR-CP for this analysis. The best online approach in terms of predictive performance was ORB-OHT-CP (Table 6).

ORB-OHT-CP was ranked 9th in terms of computational cost, whereas BORB-LR-CP was ranked 10th. The differences in computational cost varied from 179.4 seconds ($\approx$ 3 minutes for Spring-Integration) to 2998.83 seconds ($\approx$ 50 minutes for Nova) in total, as we can see from Tables 10 and 9. In other words, ORB-OHT-CP was from 1.2 to 3.8 times faster. Such differences in computational cost may be particularly relevant when conducting experiments to choose an approach for adoption. To give an example, for the most time consuming dataset Nova, ORB-OHT-CP required 2986.95 seconds ($\approx$ 50 min) for a single run. As such experiments require multiple runs to lead to more reliable conclusions, one may opt for performing 30 runs, which would take $\approx$ 25 hours, just for this dataset. When using BORB-LR-CP, this amount of time was approximately the double. If a company is performing experiments with their projects to double check which approach would be better in their context, they would need to perform similar experiments with several of their past projects. If they can narrow down the set of approaches investigated to include less computationally expensive ones, this could lead to significant

savings in computational costs. Moreover, in the future, if one proposes an approach to automatically tune hyperparameters over time, such approach may also rely on running multiple models concurrently, again resulting in a non-negligible computational cost.

That said, even though these are considerable computational costs when conducting experimental studies to run, tune and compare multiple approaches, when a given approach is chosen for adoption and we consider the duration of the projects in practice (e.g., 10.7 years for Spring-Integration and 6.99 years for Nova), this translates into a difference of only ≈ 0.05 to ≈ 1.18 seconds per day. Therefore, both BORB-LR-CP and ORB-OHT-CP are feasible for adoption in practice in terms of their computational cost per day, as illustrated in Table 12.

Even though ORB-OHT-CP is ranked worse in terms of predictive performance than BORB-LR-CP, the magnitudes of the differences in predictive performance are not high. In particular, BORB-LR-CP's predictive performance was better with absolute differences in G-Mean varying from 0.13% (for Broadleaf) to 2.93% (for Tomcat) with a median of just 1.96%. For one dataset, ORB-OHT-CP had higher G-Mean than BORB-LR-CP (Fabric8).

Similar results would have been achieved if we had compared BORB-MLP-CP against ORB-OHT-CP, but the magnitude of the differences in predictive performance would have been slightly larger (BORB-MLP-CP obtained absolute improvements of up to 3.68% over ORB-OHT-CP as shown in Section 7.2), and so would the differences in computational cost (ORB-OHT-CP runs up to 5.97 times faster than ORB-OHT-CP). BORB-MLP-CP's computational cost would also be feasible for adoption in practice, being up to 3.83 seconds per day.

Table 12: Computational Cost in Seconds for ORB-OHT-CP and BORB-MLP-CP

Dataset	Duration (year)	ORB-OHT-CP	Runtime Per Year (ORB)	Runtime Per Day (ORB)	BORB-LR-CP	Runtime Per Year (BORB)	Runtime Per Day (BORB)
Tomcat	6	1646.8	274.47	0.75	1146.12	191.02	0.52
JGroups	9.01	710.85	78.9	0.22	415.02	46.06	0.13
Spring-Int	10.7	523.09	48.89	0.13	702.49	65.65	0.18
Camel	6.65	964.85	145.09	0.4	3631.75	546.13	1.50
Brackets	14.25	1275.75	89.53	0.25	1994.94	140.00	0.38
Nova	6.99	2986.95	427.32	1.17	5985.78	856.33	2.35
Fabric8	7.68	692.34	90.15	0.25	1356.3	176.60	0.48
Neutron	8.17	1402.65	171.68	0.47	1678.75	205.48	0.56
Npm	10.18	617.2	60.63	0.17	908.56	89.25	0.24
Broadleaf	11.7	823.43	70.38	0.19	1929.85	164.94	0.45

RQ3: The online CP approach ORB-OHT-CP was up to 3.8 times (5.97 times) faster than an offline CP approach BORB-LR-CP (BORB-MLP-CP) that was top ranked in terms of G-Mean. Such differences in computational cost may be particularly relevant when tuning and comparing approaches to decide which ones to adopt in practice. However, once chosen, the computational cost of both approaches can be considered feasible in practice, as their average computational cost was up to 2.35 (3.83) second per day.

8 Threats to Validity

Internal validity: poor hyperparameter choices can affect the predictive performance of machine learning models. To mitigate this threat, random search was performed on a set of possible values for the hyperparameters of each approach and base learner based on the first 3000 training examples of the data streams. It is worth noting that this leads to an overlap between the examples involved in tuning and the examples used for testing in the beginning of the data streams. Therefore, for all approaches, the predictive performance in the beginning of the data streams is likely an overestimation of the predictive performance that these approaches can achieve in practice. Verification latency was also taken into account for all approaches respecting the chronology of the software changes. Besides, each of the approaches with each dataset was executed 30 times to mitigate the threats to internal validity.

Construct validity: the evaluation metrics used in this study were G-Mean, Recall0 and Recall1. These are widely used metrics appropriate for class imbalance learning [44] and were computed prequentially considering a fading factor recommended in [16], that allows the model to give more importance to the most recent data.

Statistical conclusion validity: Scott-Knott test with non-parametric bootstrap sampling and A12 effect size were used to address conclusion validity. This avoids concluding that approaches have differences even though the difference in the predictive performance is very small which is presented by insignificant effect size.

External validity: this study used 10 open source projects from GitHub repositories. These projects are based on different programming languages and have different characteristics (e.g. number of commits per day, starting date, number of modified files). The results obtained by this study may not generalise for projects with different characteristics to those used in our study. The conclusions about offline learning drawn by this study are based on the proposed approach BORB. Other offline learning approaches may lead to different conclusions. Similarly, other offline base learners than the ones used in our study may also lead to different conclusions.

9 Conclusion

This study investigated whether offline JIT-SDP can offer any benefits in terms of predictive performance when applied to online JIT-SDP scenarios compared to online JIT-SDP approaches, and whether such benefits may come at the cost of higher computational requirements. For that, we proposed a new offline approach called BORB that can apply adaptive resampling to deal with class imbalance in JIT-SDP when applied to online JIT-SDP scenarios. These approach's predictive performance and computational cost were compared against the existing online approach ORB when using various different base models on 10 open source projects.

Overall, our experiments suggest that, if one is focused on achieving the best possible predictive performance, it is worth considering offline learning through BORB using CP data as a possible choice, as it obtained slightly better predictive performance than ORB approaches while having an acceptable computational cost. If one is interested in saving computational cost, we recommend considering ORB using CP data with OHT as a possible choice, as it obtained better computational cost with just slightly worse predictive performance. Such saving in computational cost (and thus also in energy) may be relevant if multiple of such approaches are required to be run concurrently, e.g., when performing experimental studies to run, tune and choose among multiple models.

Future work can consider how to further improve predictive performance in JIT-SDP. In particular, the finding that offline models can be successfully applied to online JIT-SDP scenarios through BORB opens up the possibility of investigating other offline base learners such as deep learning approaches that may lead to even better predictive performance. However, even though the computational cost of the offline learning approaches adopted in the current study was feasible for adoption in practice, other more complex offline base learners such as deep learning may require a much higher computational cost, such that future work could also consider how to improve the computational cost of BORB. Future work could also evaluate BORB and ORB with additional projects. Finally, hyperparameter tuning in online JIT-SDP scenarios is still an open issue. Novel approaches for automatically tuning such hyperparameters are desirable and may benefit from faster JIT-SDP models such as ORB to be computationally feasible.

Data Availability Statements

A replication package can be found in the JIT-SDP-NN repository, `https://github.com/dinaldoap/jit-sdp-nn`. The datasets generated during and/or analysed during the current study are available in the JIT-SDP-DATA repository, `https://github.com/dinaldoap/jit-sdp-data`.

Declarations

Funding and/or Conflicts of interests/Competing interests:

The authors have no relevant financial or non-financial interests to disclose. The authors have no competing interests to declare that are relevant to the content of this article. All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript. The authors have no financial or proprietary interests in any material discussed in this article.

This work was partly supported by EPSRC Grant No. EP/R006660/2 through a School of Computer Science scholarship funded by the University of Birmingham in support of the grant.

References

1. Aggarwal, C.C., et al.: Data mining: the textbook, vol. 1. Springer (2015)
2. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of machine learning research **13**(2) (2012)
3. Bishop, C.M.: Pattern Recognition and Machine Learning, pp. 143–144. Springer, United States
4. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
5. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and regression trees, 1st edn. CRC Press (1984)
6. Cabral, G.G., Minku, L.L.: Towards reliable online just-in-time software defect prediction. IEEE Transactions on Software Engineering (2022)
7. Cabral, G.G., Minku, L.L., Shihab, E., Mujahid, S.: Class imbalance evolution and verification latency in just-in-time software defect prediction. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 666–676. IEEE (2019)
8. Catal, C., Diri, B.: A systematic review of software fault prediction studies. Expert systems with applications **36**(4), 7346–7354 (2009)
9. Catolino, G., Di Nucci, D., Ferrucci, F.: Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 99–110. IEEE (2019)
10. Chen, X., Zhao, Y., Wang, Q., Yuan, Z.: Multi: Multi-objective effort-aware just-in-time software defect prediction. Information and Software Technology **93**, 1–13 (2018)
11. Demšar, J.: Statistical comparisons of classifiers over multiple data sets. JMLR **7**, 1–30 (2006)
12. Ditzler, G., Roveri, M., Alippi, C., Polikar, R.: Learning in nonstationary environments: A survey. IEEE CIM **10**(4), 12–25 (2015)

13. Domingos, P., Hulten, G.: Mining high-speed data streams. In: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 71–80 (2000)
14. Eyolfson, J., Tan, L., Lam, P.: Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 153–162 (2011)
15. Flint, S., Chauhan, J., Dyer, R.: Escaping the time pit: Pitfalls and guidelines for using time-based git data. In: MSR, pp. 85–96 (2021)
16. Gama, J., Sebastiao, R., Rodrigues, P.P.: On evaluating stream learning algorithms. Machine learning **90**(3), 317–346 (2013)
17. Gardner, M.W., Dorling, S.: Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. Atmospheric environment **32**(14-15), 2627–2636 (1998)
18. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering **38**(6), 1276–1304 (2011)
19. Huang, Q., Xia, X., Lo, D.: Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In: IEEE International Conference on Software Maintenance and Evolution, pp. 159–170 (2017)
20. Huang, Q., Xia, X., Lo, D.: Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empirical Software Engineering (24), 2823–2862 (2019)
21. Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E.: Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering **21**(5), 2072–2106 (2016)
22. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2012)
23. Kim, S., Whitehead, E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on software engineering **34**(2), 181–196 (2008)
24. Kim, S., Zimmermann, T., Whitehead Jr. E. J.; Zeller, A.: Predicting faults from cached history. In: 29th International Conference on Software Engineering (ICSE'07) (2007)
25. Kleinbaum, D.G., Dietz, K., Gail, M., Klein, M., Klein, M.: Logistic regression. Springer (2002)
26. Li, W., Zhang, W., Jia, X., Huang, Z.: Effort-aware semi-supervised just-in-time defect prediction. Information and Software Technology **126**, 106364 (2020)
27. Li, Z., Jing, X.Y., Zhu, X.: Progress on approaches to software defect prediction. Iet Software **12**(3), 161–175 (2018)
28. Mantovani, R.G., Rossi, A.L., Vanschoren, J., Bischl, B., De Carvalho, A.C.: Effectiveness of random search in svm hyper-parameter tuning. In: 2015 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. Ieee (2015)

29. McCloskey, M., Cohen, N.J.: Catastrophic interference in connectionist networks: The sequential learning problem. In: Psychology of learning and motivation, vol. 24, pp. 109–165. Elsevier (1989)
30. McIntosh, S., Kamei, Y.: Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering **44**(5), 412–428 (2017)
31. McIntosh, S., Kamei, Y.: Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE TSE **44**(5), 412–428 (2018)
32. Menzies, T., Yang, Y., Mathew, G., Boehm, B., Hihn, J.: Negative results for software effort estimation. EMSE **22**(5), 2658–2683 (2017)
33. Mittas, N., Angelis, L.: Ranking and clustering software cost estimation models through a multiple comparisons algorithm. IEEE TSE **39**(4), 537–551 (2012)
34. Rosen, C., Grawi, B., Shihab, E.: Commitguru: analytics and risk prediction of software commits. In: FSE, pp. 966–969. ACM (2015)
35. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? ACM sigsoft software engineering notes **30**(4), 1–5 (2005)
36. Song, Q., Guo, Y., Shepperd, M.: A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Transactions on Software Engineering **45**(12), 1253–1269 (2018)
37. Tabassum, S., Minku, L.L., Feng, D.: Cross-project online just-in-time software defect prediction. IEEE Transactions on Software Engineering (2022 (in press)). DOI 10.1109/TSE.2022.3150153
38. Tabassum, S., Minku, L.L., Feng, D., Cabral, G.G., Song, L.: An investigation of cross-project learning in online just-in-time software defect prediction. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 554–565. IEEE (2020)
39. Tan, M., Tan, L., Dara, S., Mayeux, C.: Online defect prediction for imbalanced data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 99–108. IEEE (2015)
40. Turhan, B., Menzies, T., Bener, A.B., Stefano, J.D.: On the relative value of cross-company and within-company data for defect prediction. EMSE **14** (2009)
41. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)
42. Wang, S., Minku, L.L., Yao, X.: A learning framework for online class imbalance learning. In: 2013 IEEE Symposium on Computational Intelligence and Ensemble Learning (CIEL), pp. 36–45. IEEE (2013)
43. Wang, S., Minku, L.L., Yao, X.: Resampling-based ensemble methods for online class imbalance learning. IEEE Transactions on Knowledge and Data Engineering (TKDE) **27**(5), 1356–1368 (2015)
44. Wang, S., Minku, L.L., Yao, X.: A systematic study of online class imbalance learning with concept drift. IEEE transactions on neural networks and learning systems **29**(10), 4802–4821 (2018)

45. Yang, X., Lo, D., Xia, X., Sun, J.: Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology **87**, 206–220 (2017)
46. Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 17–26. IEEE (2015)
47. Zhu, K., Zhang, N., Ying, S., Zhu, D.: Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. IET Software (2020)
48. Zhu, Q.: On the performance of matthews correlation coefficient (MCC) for imbalanced dataset. Pattern Recognition Letters **136**, 71–80 (2020)