Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction

George G. Cabral*†, Leandro L. Minku*, Emad Shihab‡, Suhaib Mujahid‡
*School of Computer Science, University of Birmingham, UK
{G.Gomescabral, L.L.Minku}@cs.bham.ac.uk
†Federal Rural University of Pernambuco, Brazil
‡Department of Computer Science and Software Engineering, Concordia University, Canada
{eshihab, s_mujahi}@encs.concordia.ca

***Abstract*—Just-in-Time Software Defect Prediction (JIT-SDP) is an SDP approach that makes defect predictions at the software change level. Most existing JIT-SDP work assumes that the characteristics of the problem remain the same over time. However, JIT-SDP may suffer from class imbalance evolution. Specifically, the imbalance status of the problem (i.e., how much underrepresented the defect-inducing changes are) may be intensified or reduced over time. If occurring, this could render existing JIT-SDP approaches unsuitable, including those that rebuild classifiers over time using only recent data. This work thus provides the first investigation of whether class imbalance evolution poses a threat to JIT-SDP. This investigation is performed in a realistic scenario by taking into account verification latency – the often overlooked fact that labeled training examples arrive with a delay. Based on 10 GitHub projects, we show that JIT-SDP suffers from class imbalance evolution, significantly hindering the predictive performance of existing JIT-SDP approaches. Compared to state-of-the-art class imbalance evolution learning approaches, the predictive performance of JIT-SDP approaches was up to 97.2% lower in terms of g-mean. Hence, it is essential to tackle class imbalance evolution in JIT-SDP. We then propose a novel class imbalance evolution approach for the specific context of JIT-SDP. While maintaining top ranked g-means, this approach managed to produce up to 63.59% more balanced recalls on the defect-inducing and clean classes than state-of-the-art class imbalance evolution approaches. We thus recommend it to avoid overemphasizing one class over the other in JIT-SDP.**

***Index Terms*—Software defect prediction, class imbalance, verification latency, online learning, concept drift, ensembles**

I. Introduction

Reducing the number of software defects (and their high debugging cost) is a challenging problem, specially considering that software teams have limited testing resources [1], [2], and often face strong pressure towards rapid delivery [1], [3]. Therefore, machine learning approaches have been proposed for predicting defects in software source code [4]. Such Software Defect Prediction (SDP) approaches can potentially help testing and inspection effort to be more easily and wisely allocated, focusing more attention on software components that are likely to contain defects.

Just-in-Time (JIT) SDP is a specific type of SDP approach that makes predictions at the software change level. It identifies defect-inducing software changes as soon as they are implemented (i.e., "just-in-time"). As the changes are still fresh in the developers' minds, their inspection is much easier than at later stages. In addition, software changes typically involve few lines of code (i.e., they have fine code granularity) and can be straightforwardly assigned to the right developers for inspection, further facilitating their task. These are advantages over both (1) code debugging after a defect report is produced and (2) more conventional SDP, which is concerned with predicting defects in software components (e.g., files or packages) [5]. Therefore, JIT-SDP has attracted special interest from industry. For instance, Lucent [1], BlackBerry [6] and Cisco [7] have adopted this type of approach.

Most existing work on JIT-SDP assumes that past defect-inducing software changes are always similar to future ones. However, as recently shown by McIntosh and Kamei [8], the characteristics (input feature values) of defect-inducing software changes fluctuate during the lifecycle of a software. These fluctuations can negatively impact the predictive performance of classifiers trained on old data, requiring machine learning algorithms able to not only take chronology into account, but also learn and adapt over time. Algorithms that can learn new training examples separately over time are often referred to as *online learning* algorithms [9].

Besides fluctuations in the characteristics of defect-inducing software changes, JIT-SDP may also suffer from *class imbalance evolution*. This means that the imbalance status of the problem (i.e., how much underrepresented the defect-inducing software changes are in the training data) may be intensified or reduced over time. JIT-SDP is known to be a class imbalanced problem, where defect-inducing software changes are typically a minority compared to clean software changes [5], [7], [2]. However, existing work assumes that the class imbalance status of JIT-SDP is static, i.e., it does not evolve over time. If it actually does evolve over time, existing JIT-SDP approaches (e.g., [5], [2], [7]) are likely to become unsuitable (perform poorly) over time, because they would be assuming the wrong level of class imbalance. Even the recent approach of rebuilding classifiers using only recent data, recommended to cope with fluctuations in the characteristics of defect-inducing software changes [8], would struggle to obtain good predictive performance. Despite being able to track the level of class

Corresponding author: L. L. Minku.
This work was funded by EPSRC Grant No. EP/R006660/1 and supported by CNPq Universal Grant No. 408605/2016-2.

imbalance, this would be at the cost of discarding old training examples which may be vital for learning the minority class. Therefore, this is unlikely to be a suitable approach for coping with class imbalance evolution.

Given the potential threat posed by class imbalance evolution to the predictive performance of JIT-SDP classifiers, **the first aim of this work is to provide the first investigation of whether class imbalance *evolution* occurs and negatively impacts predictive performance of existing JIT-SDP approaches.** Our study, based on ten `GitHub` open source projects, reveals that class imbalance evolution is indeed an issue in JIT-SDP and can have a very detrimental effect on the predictive performance of JIT-SDP approaches from the literature. Their predictive performance in terms of g-mean was up to 97.2% worse than that achieved by machine learning algorithms prepared to tackle class imbalance evolution. Therefore, class imbalance evolution must be taken into account to improve the applicability of JIT-SDP approaches.

However, our study also reveals that the existing machine learning algorithms for class imbalance evolution [10] suffer from overemphasising one class (defect-inducing or clean) over the other in JIT-SDP. This can be very detrimental in practice. Overemphasising the clean class means that defect-inducing software changes may be missed. Overemphasising the defect-inducing class means that several changes are incorrectly flagged as defect-inducing, suggesting a too high number of clean software changes to be closely inspected by developers and thus reducing their trust in the approach. Therefore, **the second aim of this paper is to propose a novel JIT-SDP approach to address this problem.** Our proposed approach was successful in obtaining top-ranked g-means while reducing the difference in recalls between the defect-inducing and clean classes by up to 63.59%.

Our investigation is performed in a realistic scenario where not only new software changes are produced over time and arrive as incoming training examples, but also there is *verification latency*. This term refers to the fact that the labels associated to training examples arrive with delays [9]. We cannot know at commit time whether a new software change induces a defect or not. If we knew that, we would not need to provide a prediction of that, and JIT-SDP would be unnecessary. Instead, it takes time for defects to be found, meaning that it takes time to assign the label of "defect-inducing" to a given training example. And it takes time for one to gain confidence that a given change is clean. Therefore, we need to wait for a given amount of time (*waiting time*) before assigning the label "clean" to a given training example. Ignoring verification latency means training JIT-SDP models on data not yet available in practice, leading to invalid studies and over-optimistic estimations of predictive performance [7]. And yet, this issue is often overlooked by JIT-SDP studies, as alerted by Tan *et al.* [7]. Our study provides the first investigation of the extent to which verification latency occurs in JIT-SDP. We found that verification latency caused delays from 1 to 11.5 years (4210 days) in receiving the true label of defect-inducing software changes.

This paper is further organized as follows. Section II presents the research questions and summarises the novel contributions of this work. Section III presents related work. Section IV introduces the proposed approach. Section V details the investigated datasets. Section VI explains the experimental setup to investigate the RQs. Section VII presents the analyses to answer the RQs. Section VIII presents threats to validity. Section IX presents conclusions and implications of this work.

II. Research Questions and Novel Contributions

Overall, this work answers the following research questions:

RQ1 To what extent there is verification latency in JIT-SDP? What would be reasonable waiting times to use in JIT-SDP studies? This RQ informs the proposal and application of machine learning approaches to JIT-SDP in this and future studies, by investigating how long it typically takes for software changes to be found as defect-inducing.

RQ2 Does JIT-SDP suffer from class imbalance evolution? More specifically, do the ratios of defect-inducing and clean software changes evolve over time in JIT-SDP? How? This RQ investigates whether the topic of class imbalance evolution is really relevant in JIT-SDP.

RQ3 If class imbalance evolution does occur in JIT-SDP, what is its effect on the predictive performance of (1) existing JIT-SDP approaches and (2) machine learning algorithms specifically designed to cope with class imbalance evolution, when taking verification latency into account? This RQ reveals whether the predictive performance of existing JIT-SDP approaches is negatively affected by class imbalance evolution, and how well state-of-the-art machine learning algorithms for coping with class imbalance evolution perform in the context of JIT-SDP.

RQ4 How to improve the predictive performance of JIT-SDP, given class imbalance evolution and verification latency? This RQ proposes a novel JIT-SDP approach to better cope with class imbalance evolution in the presence of verification latency.

Overall, the main novel contributions of this work are that this is the first work to:

- show that class imbalance evolution occurs in JIT-SDP;
- reveal that this negatively affects the predictive performance of existing JIT-SDP approaches;
- investigate if state-of-the-art online class imbalance learning algorithms from the machine learning literature can help tackling this issue in JIT-SDP;
- propose a novel online class imbalance learning algorithm to improve upon the results achieved by these state-of-the-art algorithms in the context of JIT-SDP; and
- investigate to what extent verification latency occurs in JIT-SDP, enabling researchers and practitioners to make more informed choices when applying machine learning to JIT-SDP.

III. Related Work

We discuss four areas of SDP research that are most closely related to our paper, and contrast the prior work with ours. We

also discuss the machine learning literature on online class imbalance learning for tackling class imbalance evolution and verification latency.

A. JIT-SDP

One of the first studies on JIT-SDP was conducted by Kim *et al.* [11]. They used software change features like the terms in added and deleted deltas, terms in directory/file names, complexity metrics, etc., to classify changes as being defect-inducing or not.

Several other studies investigated the characteristics of defect-inducing software changes and potential metrics (input features) for predicting them, including the day of the week [12] or the time of the day [13] the change was committed, and metrics for identifying changes that require a lot of effort to fix [14]. Shihab *et al.* [6] studied risky (defect-inducing) changes – as deemed by the developers who committed them. They found that lines of code added, bugginess of the touched files (i.e., the ratio of bug fixing to total changes that touched a file), the number of bug reports linked to a commit and the developer experience are the top indicators of risky changes.

One of the largest JIT-SDP studies was conducted by Kamei *et al.* [5]. They used a variety of factors extracted from commits and bug reports, which were found as good indicators of defect-inducing software changes. They showed that the metrics used in their study yield high predictive performance for both open source and commercial projects. Therefore, we use the same metrics in this work, as provided by the tool `Commit Guru` (http://commit.guru). The dimensions used to create these metrics are further explained in Section V.

None of the papers above considered class imbalance evolution or verification latency.

B. Verification Latency in JIT-SDP

As explained in Section I, verification latency refers to the fact that the labels of training examples may arrive much later than their input features. Ignoring such delay means training JIT-SDP models on data not year available in practice, which is a serious threat to validity. Tan *et al.* [7] found that ignoring verification latency leads to over-optimistic estimates of the predictive performance. They proposed an approach that takes verification latency into account. It stores new batches of training examples over time, and uses all batches received so far for building a JIT-SDP classifier. Training examples are available to compose new batches only after a pre-defined waiting time has passed. This waiting time should reflect the time it takes for one to be confident enough that software changes are not defect-inducing. However, their study does not analyse how long it typically takes for defects to be found, and their proposed approach assumes that there is no class imbalance evolution. Different from their work, our study (1) investigates class imbalance evolution and its impact on the predictive performance of JIT-SDP classifiers over time, (2) proposes approaches to better deal with class imbalance evolution, and (3) investigates how long it typically takes for software changes to be revealed as defect-inducing.

C. Concept Drift in SDP

Concept drift is a change in the data generation process, affecting the underlying probabilities of the data [15]. They can be changes in (1) the relationship between input features describing an example and the label being predicted ($p(x|y)$), and/or (2) the ratio of examples of each of the classes being predicted ($p(y)$). In JIT-SDP, these correspond to changes in the defect generating process, and can occur due to the evolution or maturing process of a software project. For example, the development team may be initially focused in the GUI and then turn its effort to implement the business logic, leading to a concept drift affecting $p(x|y)$. Or, refactoring could potentially affect the rate of defect-inducing software changes, leading to concept drifts affecting $p(y)$. Class imbalance evolution corresponds to concept drifts affecting $p(y)$ in class imbalanced problems.

Very few studies investigated concept drift in SDP. Ekanayake *et al.* [16] studied four open source projects and showed that SDP classifiers' performance significantly varies over time, suggesting that SDP suffers from concept drift. This study did not investigate JIT-SDP. Recently, McIntosh and Kamei [8] examined the impact of concept drift affecting $p(x|y)$ in the specific context of JIT-SDP. They considered three open source projects and found that (i) JIT-SDP classifiers lose a significant amount of performance after one year and (ii) the important indicators of defect-inducing changes also vary over time. To tackle such concept drifts, they suggest re-building JIT-SDP classifiers on sliding windows containing only recent software changes. To the best of our knowledge, this is the first work to examine concept drift in JIT-SDP.

However, McIntosh and Kamei [8] did not investigate class imbalance evolution, which requires different strategies to avoid reduction in predictive performance [15]. Their suggested sliding windows strategies can be even detrimental to predictive performance in such scenario. This is because despite being able to track the level of class imbalance, this is at the cost of discarding potentially vital information for learning the minority class. Therefore, sliding windows are unlikely to be suitable for coping with class imbalance evolution. In addition, their work did not take verification latency into account, implicitly assuming that the labels of training examples are available immediately after commit time.

Tan *et al.* [7] investigated JIT-SDP in an updatable learning scenario, where additional training examples can be received over time. However, their updatable classifiers assume that all training examples come from the same underlying probability distribution, i.e., they assume that there is no concept drift. As explained by Ditzler *et al.* [9], learning in the presence of concept drift requires "approaches that can monitor and track the underlying changes, and adapt a model to accommodate those changes accordingly." The approaches investigated by Tan *et al.* [7] do not meet these requirements.

D. Class Imbalance Learning for SDP

SDP is well-known to be a class imbalanced problem. Many studies consider this issue, independent of it being their central

theme or not. Mahmood *et al.* [17] showed that the predictive performance of SDP classifiers (in terms of Mathews Correlation Coefficient) gets worse as the data get more imbalanced. Wang and Yao [18] provided a comprehensive study of different class imbalance learning techniques in the context of SDP, including resampling, threshold moving, and ensembles. Their study, based on performance measures such as balance, g-mean, and Area Under the ROC Curve (AUC), concluded that an ensemble approach called AdaBoost.NC yielded the best overall predictive performance. They also proposed a version of Adaboost.NC that is able to automatically tune its training parameters. Kamei et. al. [19] investigated the use of four resampling methods for fault prone module detection and showed that, when associated to linear discriminant analysis and logistic regression analysis, there is a performance improvement irrespective of the resampling method. Bennin et.al. [20] introduced a synthetic oversampling approach based on the chromosomal theory of inheritance that, according to their experiments, overcame four other resampling techniques. These studies did not investigate JIT-SDP.

In JIT-SDP, resampling is typically used to tackle class imbalance [5], [2], [7]. For instance, Kamei *et al.* [5], [2] applied an undersampling technique that randomly eliminates clean class examples to balance the number of training examples from both classes. Tan *et al.* [7] investigated the effect of different resampling techniques on JIT-SDP while taking the chronology of the data and verification latency into account, including oversampling techniques that replicate minority class examples. They concluded that resampling in general helps to improve predictive performance in terms of F1-measure. However, their approach adopts a fixed resampling rate, assuming that the imbalance ratio is fixed throughout time, i.e., assuming that there is no class imbalance evolution. Specifically, their parameter tuning procedure fixed the resampling rate to be used for a whole dataset to a single value, rather than enabling the resampling rate to dynamically adapt to the current imbalance level of the data.

None of the studies on class imbalance learning for SDP (including JIT-SDP) investigated the impact of class imbalance *evolution*, or techniques to handle class imbalance evolution. This paper is the first to provide such investigation.

E. Machine Learning To Tackle Class Imbalance Evolution

Class imbalance evolution started to be investigated only very recently by the machine learning community [15]. Wang *et al.* [10] proposed two online class imbalance learning approaches for coping with class imbalance evolution: Improved Undersampling Online Bagging (UOB) and Improved Oversampling Online Bagging (OOB). These approaches track the current imbalance ratio, i.e., the rate $\rho_c^{(t)}$ of examples belonging to each class $c \in \{0, 1\}$ as follows:

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

where t is the current time step; each time step corresponds to the presentation of a new training example to the algorithm; $(y^{(t)} == c)$ returns 1 if the training example at time t is of class c and 0 otherwise; and θ', $0 \ll \theta' \leq 1$, is a predefined parameter to tune the emphasis on the more recent data. Smaller θ' emphasizes the present, enabling $\rho_c^{(t)}$ to reflect changes in the imbalance ratio more quickly, but being potentially more affected by noise. Tracking changes in the imbalance ratio means tracking (but not yet coping with) class imbalance evolution.

To cope with class imbalance evolution, the rates $\rho_c^{(t)}$ are used to dynamically decide the resampling rate λ used by UOB and OOB to decide the number of times k that a given training example is to be presented to a given classifier. The number k is drawn from a $Poisson(\lambda)$ distribution. In UOB, examples of the majority class use $\lambda = p_{min}^{(t)}/p_{max}^{(t)}$, where min is the minority and max is the majority class. This results in frequently sampling them zero times (undersampling). Training examples of the minority class use $\lambda = 1$. In OOB, examples of the minority class have a resampling rate of $\lambda = p_{max}^{(t)}/p_{min}^{(t)}$. This results in k being frequently greater than 1 (oversampling). Training examples of the majority class use $\lambda = 1$. Both UOB and OOB maintain an Online Bagging ensemble [21] composed of n Hoeffding trees [22] as base classifiers. The pseudocode for OOB is in Algorithm 1, removing the statements in blue.

UOB and OOB are the state-of-the-art for dealing with class imbalance evolution [15], making them good candidates to tackle that in JIT-SDP. Based on UOB and OOB, this paper provides the first investigation of online class imbalance evolution learning approaches in the context of JIT-SDP.

F. Machine Learning to Tackle Verification Latency

Some work on learning data streams can be argued to take verification latency into account. However, they are designed for very specific learning scenarios. For example, Zhang *et al.* [23] assume that some training examples become available in a timely manner, whereas others never have their labels revealed. Dyer *et al.* [24] assume that labeled training examples are available only during an initial learning stage. After that, no further labeled training examples are provided. These studies proposed to use semi-supervised learning to cope with that. Pozzolo *et al.* [25] is a very recent work in the context of credit card fraud detection. It assumes that the system has prior knowledge of which training examples will have their labels arriving early and which examples are likely to have delayed labels. They propose to learn different classifiers with these two different types of training examples.

None of the learning scenarios above matches the case of JIT-SDP, where any example could receive its true label early (if a defect associated to it is quickly found), at the end of the waiting time (if the example is believed to be clean) or after the waiting time (if an example previously considered clean is found to actually be defect-inducing after the waiting time).

IV. Proposed Approach

Even though UOB and OOB are the state-of-the-art approaches to cope with class imbalance evolution, they have three potential problems in the context of JIT-SDP: (1) They

do not consider verification latency. (2) They assume that adjusting the resampling rate so that the class proportions in the training data get closer to (1:1) is enough to obtain a balanced predictive performance on different classes. If one of the classes is more difficult to learn, balancing their numbers of training examples may still lead to unbalanced predictive performances, where the recall on one class is still much worse than the recall on the other. As explained in Section I, unbalanced predictive performances are undesirable in JIT-SDP. (3) SDP frequently has noisy or outlier training examples, containing exactly the same input feature values but different labels [26]. The effect of noisy or outlier training examples of the defect-inducing class could be magnified when adopting oversampling, potentially leading to false alarms.

Section IV-A explains how to take verification latency into account, overcoming problem (1). Section IV-B proposes a novel approach to overcome problems (2) and (3) in the presence of verification latency.

A. A Framework for Verification Latency Classification

In practice, it is not possible to know, at commit time, whether a new software change induces a defect or not. So, we consider that each change is labeled as defect-inducing or clean only ω (waiting time) days after the commit time, or as soon as it is found to be defect-inducing, whichever is shorter. The waiting time ω is a parameter to be set by software managers. For instance, if it typically takes less than 90 days for a change to be found as defect-inducing, an appropriate value for ω would be 90 days. The reason is that, if no defect associated to a change has been found in 90 days from its commit, one can gain confidence that this change is clean. Once a change is labeled, it is immediately used to create a training example for learning. If a change that has already been labeled as clean is found to be defect-inducing after the waiting time, the training example corresponding to that change will be updated with the label defect-inducing and presented again for learning. This framework can be applied to any classifier.

B. Oversampling Rate Boosting (ORB)

This section proposes a novel approach (ORB) to overcome problems (2) and (3) discussed in the beginning of Section IV. To cope with (2), ORB builds upon OOB by monitoring the model predictions to support the adjustment of the resampling rate. If the predictions are considerably biased towards a given class c, the resampling rate of the opposite class c' should be boosted (increased). To cope with (3), a safety mechanism is adopted to prevent using potentially noisy (or outlier) examples of the defect-inducing class for training. Algorithm 1 depicts the proposed approach. ORB is run within the framework explained in Section IV-A.

When a new training example $d^{(t)} = (x^{(t)}, y^{(t)})$ is received at time step t (line 1), the moving average of the predictions ($ma^{(t)}$) is calculated over a time window of size w_s (lines 2 and 3). The current problem is a binary problem where $\hat{y} \in \{0,1\}$, with 0 representing the clean and 1 representing the defect-inducing class. Therefore, the moving average of the predictions enables us to detect any classification bias towards each of the classes. According to the severity of this bias, the resampling rate of one of the classes will be boosted.

Algorithm 1: ORB's training procedure. Statements in black correspond to statements used by OOB's training.

```
Input: Ensemble size n, incoming training examples d, parameters
       of the adjustment function (th, l0, l1, m), noise mechanism
       parameter o, decay factor θ', window size ws
1  for each training example d(t) = (x(t), y(t)), t ← 0 to ∞ do
2      Obtain the ensemble prediction ŷ(t) for x(t)
3      Compute the average ma(t) over the predictions on the most
         recent ws examples, including d(t)
4      Update the proportions ρ0(t) and ρ1(t) of each class using Eq. 1
5      for i ← 0 to n do
6          λ = 1
7          if y(t) == 1 and ρ1(t) < ρ0(t) then
8              λ = ρ0(t)/ρ1(t)
9          if y(t) == 0 and ρ0(t) < ρ1(t) then
10             λ = ρ1(t)/ρ0(t)
11         Set k ∼ Poisson(λ)
12         Calculate OBF(t)(ma(t), th, l0, l1, m) using Eq. 2 or
             Eq. 3
13         k = k · OBF(t)
14         Run noise safety mechanism with parameter o
           /* Depending on the noise safety
              mechanism outcome, update the ith
              Hoeffding tree with k copies of dt   */
15         Update(HTi, k, dt)
       end
   end
```

After calculating the moving average of the predictions, OOB's procedures are used to update the proportions of each class $\rho_0^{(t)}$ and $\rho_1^{(t)}$ and determine the resampling rate λ (lines 4 to 10). The resampling rate is used to determine the number of times k that the training example will be sampled for a given Hoeffding tree to learn (line 15). In OOB, k is directly taken from a $Poisson(\lambda)$ distribution (line 11). However, in ORB, the value taken from $Poisson(\lambda)$ is multiplied by a boosting factor $OBF^{(t)}$ (lines 12 to 13) before being used to resample the training example (line 15). This factor boosts (increases) the number of times that the training example is sampled.

The equations used to calculate the boosting factors for training examples of the clean and defect-inducing classes are shown below, respectively:

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

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

where P_0 and P_1 are the set of parameters for each function, containing the following parameters: m – determines the growth of the exponential function; th – stands for the threshold that indicates which class must be boosted; ma^t – the predictions moving average at time t; and l_0 and l_1 – control the maximum boosting factor values (i.e., the boosting

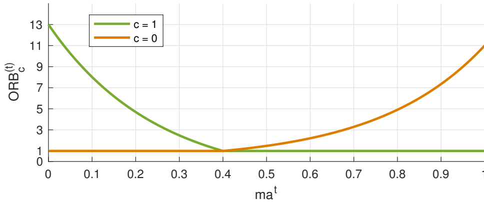

Fig. 1. Boosting factor functions with th = 0.4, l_0 = 10, l_1 = 12, m = 1.5.

TABLE I
STATISTICS OF THE STUDIED PROJECTS

Dataset	#commits	%defect-inducing	Period	Language
Fabric8	13,004	20%	12/2011 - 12/2017	Java
JGroups	18,317	17%	09/2003 - 12/2017	Java
Camel	30,517	20%	03/2007 - 12/2017	Java
Tomcat	18,877	28%	03/2006 - 12/2017	Java
Brackets	17,311	23%	12/2011 - 12/2017	JavaScript
Neutron	19,451	24%	12/2010 - 12/2017	Python
Spring-integration	8,692	27%	11/2007 - 01/2018	Java
Broadleaf	14,911	17%	11/2008 - 12/2017	Java
Nova	48,938	25%	08/2010 - 01/2018	Python
NPM	7,893	18%	09/2009 - 11/2017	JavaScript

factors varies from 1 to $1 + l_0$ and $1 + l_1$). The parameters th and ma^t must be scaled between 0 and 1.

The functions used to compute the boosting factors are illustrated in Fig. 1. The condition $ma^{(t)} \leq th$ implies that, at time t, less than $th\%$ of the commits are being classified as defect-inducing, so, the resampling rate of the defect-inducing class must be boosted. On the other hand, if $ma^{(t)} > th$, more than $th\%$ of the commits are being classified as defect-inducing. In this case, the resampling rate of the clean class is boosted. As the moving average approaches the limits (0 or 1), the function growth gets steeper. The steepness level is controlled by the parameter m. The parameter th, should be chosen to represent a reasonable imbalance rate in the classifier's predictions. Future work could investigate methods to automatically tune th.

ORB also contains a safety mechanism (line 14) to cope with noise on defect-inducing training examples. If such examples have exactly the same input features $x^{(t)}$ as more than o previous clean training examples, the algorithm skips to the next iteration of the outer loop without learning the current defect-inducing training example $d^{(t)}$.

V. Datasets

Our study uses the ten `GitHub` open source projects shown in Table I. These projects were randomly chosen among projects with more than 5 years of duration, rich history (>10k commits) and good defect-inducing changes ratio (∼20% overall). Their size ranges from 74k to over 1.3m lines of code (LOC), with development periods from 6 to over 14 years. To obtain defect-inducing software changes, we use `Commit Guru` [27], a tool that analyzes and provides change level analytics. `Commit Guru` applies the SZZ algorithm [12] to identify defect-inducing changes and their associated bug fixing commits. It provides a number of commit-level metrics related to five dimensions: (1) the size of the change, (2) the history of the files changed, (3) the diffusion of the change, (4) the experience of the developers making the modification, and (5) the purpose of the change. These dimensions have shown to perform well in JIT-SDP research [5], [2], [8], and lead to 14 metrics used as input features to describe software changes in this study. Further details on the specific metrics are omitted from this paper due to space constraints, and can be found in Kamei *et al.* [5]'s work.

VI. Experimental Setup

a) Setup for RQ1: An analysis of the time it takes for changes to be revealed as defect-inducing (defect discovery delay) will be provided based on statistics such as median, percentiles, minimum and maximum values. It will support the choice of waiting time (ω) explained in Section IV-A.

b) Setup for RQ2: The proportion of examples from each class over time will be analysed using Eq. 1 with $\theta' = 0.99$. This value was chosen for providing a good trade-off between tracking changes and not being too affected by noise, based on preliminary experiments. The investigation takes into account verification latency as explained in Section IV-A, using the waiting time of $\omega = 90$ days informed by RQ1. Possible causes for variations in the proportions of the classes are discussed.

c) Setup for RQ3: This RQ will be answered by comparing five approaches, taking verification latency into account as explained in Section IV-A. The approaches are OOB, UOB, OOB(FixedIR), OOB(FixedIR)* and OOB-SW. OOB and UOB [10] are two state-of-the-art online class imbalance learning approaches explained in Section III-E. OOB(FixedIR) uses a modified version of OOB that assumes that the imbalance ratio (and thus the resampling rate) of the problem is fixed over time. It thus corresponds to the resampling strategy typically used in the JIT-SDP literature [5], [2]. OOB(FixedIR)* is similar to OOB(FixedIR), but training examples of the defect-inducing class are only used for training at the end of the waiting time. This is the case even if their software changes are found to be defect-inducing earlier, as done in Tan *et al.* [7]'s JIT-SDP work. Finally, OOB-SW trains an OOB ensemble on a sliding window – the strategy recommended by McIntosh and Kamei [8]. The oversampling rate is set to n_0/n_1, where n_0 is the number of clean and n_1 is the number of defect-inducing changes within the window.

The evaluation metrics used for the comparison are the recalls on the clean (R_0) and defect-inducing (R_1) classes, and the g-mean ($\sqrt{Rec_0 \times Rec_1}$). These are the most commonly used metrics in the online class imbalance literature [15]. Different from precision and F-measure, they are robust to the class imbalance problem [28]. The absolute differences between R_0 and R_1 was also adopted, as large gaps between the recalls of different classes are undesirable in JIT-SDP, as explained in Section I. All metrics are computed in a prequential way, as recommended for online learning studies in the presence of concept drift [29]. The decay parameter θ used in the calculation of the metrics enables them to track the impact of concept drift on predictive performance. Similar to the parameter θ' used for RQ1, $\theta = 0.99$ was used. It is worth noting that, given the online concept drifting nature of the problem, AUC (an extensively used metric for offline binary problems) is not applicable.

A grid search based on the first 5,000 changes of each dataset, using g-mean as evaluation criterion, was conducted to choose the best parameters for each approach. Values in bold face correspond to the chosen values for the experiments. Some parameters are shared among the classifiers: (n) ensemble size = $\{10,\mathbf{20},30,40\}$; (θ') decay factor = $\{0.9,\mathbf{0.99},0.999\}$; and ($\omega$) waiting period = $\{\mathbf{90},180\}$. For OOB(FixedIR), the imbalance ratio was fixed at time step 500. For the OOB-SW, sliding windows of size **90** and 180 days were tested.

d) Setup for RQ4: ORB was compared against the approaches from RQ3 that obtained the best g-mean and $|R_0 - R_1|$. ORB has the same parameters of OOB plus (i) the parameters for exponential functions (Eqs. 2 and 3) and (ii) noise detection mechanism (n). The values tested for these parameters were: moving average window size = $\{50, \mathbf{100}, 200\}$; th = $\{0.3, \mathbf{0.4}, 0.5\}$; l_0 = $\{5, \mathbf{10}, 15\}$; l_1 = $\{6, \mathbf{12}, 18\}$; m = $\{\mathbf{1.5}, 2.0, e\}$; and n = $\{\mathbf{3}, 5, 7\}$. These values were chosen based on preliminary experiments. A sensitivity analysis showed that ORB is not very sensitive to parameters choice, and is available at http://doi.org/10.5281/zenodo.2555695.

For both RQ3 and RQ4, 30 executions of each approach were performed. The comparisons among the approaches were supported by the Scott-Knott multiple comparison procedure to separate the approaches into non-overlapping groups, regarding their overall predictive. This test was conducted considering the total number of experiments (i.e., 10 datasets times 30 executions for each classifier). Scott-Knott was recommended by Mittas and Angelis [30] because it can (1) separate different approaches into non-overlapping groups and (2) reduce the number of statistical tests performed, being a powerful test. As suggested by Menzies *et al.* [31], Scott-Knott was run with non-parametric bootstrap sampling, making the test non-parametric. In addition, Vargha and Delaney's non-parametric A12 effect size [32] was used to ensure that groups can only be split if medium or large effect sizes exist between them. Specifically, Scott-Knott only performed statistical tests to check whether groups should be separated if the A12 effect size was medium or large. If A12 effect size was not medium or large, groups were not separated. As suggested by Vargha and Delaney [32], A12 $\geq$ 0.56, $\geq$ 0.64 and $\geq$ 0.71 are considered small, medium and large, respectively. The use of effect size with Scott-Knott has also been recommended by Tantithamthavorn *et al.* [33], [34], even though they used Cohen's parametric effect size. The code provided by Tim Menzies (at https://github.com/txt/ase16/blob/master/doc/stats.md) was used to run the Scott-Knott procedure.

All approaches use Hoeffding trees [22] as base learners, as explained in Sections III-E and IV. A replication package is available at http://doi.org/10.5281/zenodo.2555695.

VII. Experimental Results

A. RQ1: Analysis of Verification Latency

We refer to the time taken for bug reports to be created and their corresponding defects fixed, making defect-inducing training examples available, as *defect discovery delay*. Fig. 2 depicts the defect discovery delay. The delays ranged from 1 to 4210 days (i.e., approximately 11.5 years). The medians delays varyied from 15 to 416.5 days, and were less than or close to 90 days (lower red dashed line) in 8 out of 10 projects (Fabric8, Jgroups, Camel, Brackets, Neutron, Broadleaf, Nova and NPM). Therefore, using a waiting time of 90 days implies that more than, or approximately, half of the defect-inducing changes would produce correctly-labeled examples for training before the end of the waiting time.

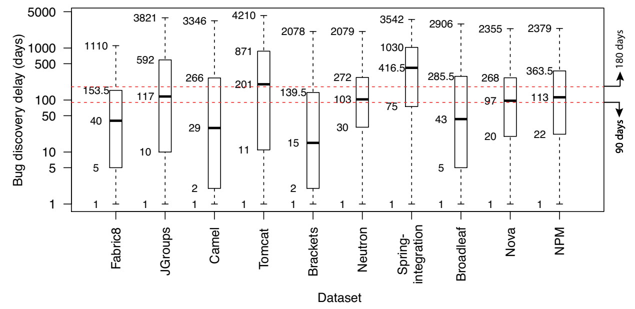

Fig. 2. Defect discovery delay (in days) for each software project, using a logarithmic y-axis.

Increasing the waiting time yields a higher number of correctly labeled defect-inducing changes to be used at training time. However, as the waiting time increases, the chances of a concept drift in $p(x|y)$ occurring before the example is available for training also increases. This means that the training examples are potentially obsolete and misleading when they arrive for training [8]. In fact, McIntosh and Kamei's [8] recommended using changes produced within the past 90 days, as older changes can be detrimental to predictive performance due to concept drift in $p(x|y)$. Therefore, a waiting period of 90 days can be argued to offer a good trade-off between mislabeled training examples and concept drift in $p(x|y)$.

RQ1: Verification latency is intrinsic to JIT-SDP. Defect discovery delays ranged from 1 day to over 11 years, and medians were typically close to, or lower than, 90 days. A waiting time of 90 days can be considered to provide a good trade-off between correct labeling and concept drift in $p(x|y)$.

B. RQ2: Analysis of Class Imbalance Evolution

Fig. 3 shows the evolution of the ratio of defect-inducing and clean software changes over time for each dataset. It represents the imbalance status as perceived by machine learning algorithms, taking verification latency into account as explained in Section IV-A. If a training example is considered as clean at the end of the waiting time and later on it is found to be defect-inducing, it will first be used to update the proportion of each class as a clean example, and then later on it will be used again as a defect-inducing example.

For Fabric8, Jgroups, Camel, Spring-Integration, Nova and NPM, there are several periods where the the ratio of defect-inducing software changes increases before reducing again. In Camel, for example, around time steps 11k, 15k, 18k and 26k, this ratio increases from around 15-20% to more than 50%. The increases observed at time steps 5k and 13k in Fabric8 coincide with major refactorings around timeframes

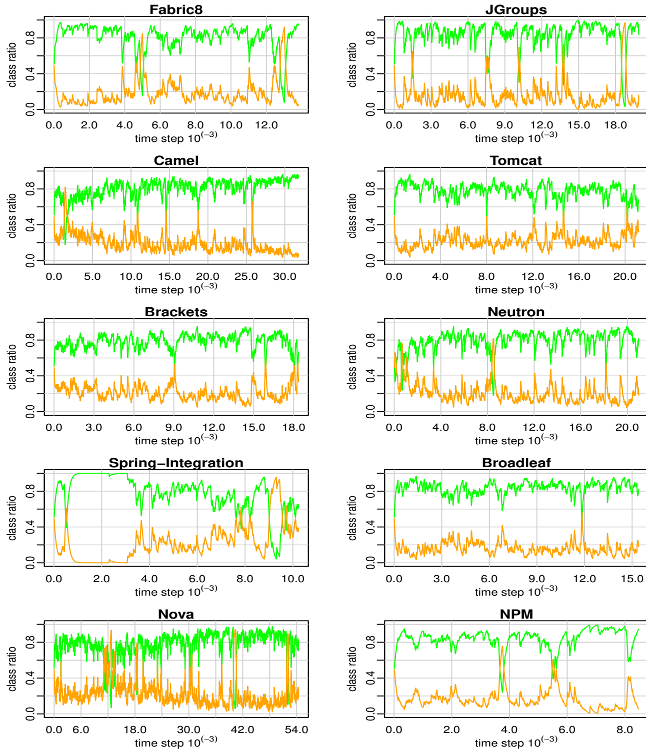

Fig. 3. Class imbalance evolution for all datasets, calculated based on Eq. 1 with $\theta' = 0.99$ and waiting time $\omega = 90$. Orange (dark grey) stands for the ratio of defect-inducing software changes whereas lime (light grey) stands for the ratio of clean software changes.

3.x and 4.x reported in their official website http://fabric8.io/gitbook/overview.html#history. We confirmed the time steps corresponding to refactorings based on the RefactoringMiner tool. This suggests that refactoring can indeed be linked to class imbalance evolution. Such large variations in the imbalance status of the problem are likely to disturb machine learning algorithms that are unable to adapt rapidly to them [15]. The remainder of the datasets also suffer from increases in the defect-inducing ratio.

According to Fig. 3, Spring-Integration experiences a period of time (from 1k to 3k) where the ratio of defect-inducing software changes becomes very small (less than 0.001%). This means that the problem becomes very highly imbalanced.As machine learning algorithms tend to struggle with more imbalanced SDP problems [17], classifiers unable to identify and tackle such class imbalance evolution may become unsuitable.

For some datasets, such as Camel, Brackets, Neutron and Nova, the ratio of defect-inducing software changes also presents a decreasing trend over time. For instance, for Camel, the ratio of defect-inducing examples started at around 40% and decreased to less than 10% by the end of the period analysed, having become more than four times smaller, and thus increasing the challenge posed by class imbalance [17]. There are two potential reasons behind such decreasing trends. First, as open source software matures, it may become less defective over time. And second, defects might be found more quickly in the initial stages of the project, causing the perceived rate of defect-inducing changes to be higher in the beginning. Independent of the reason, this decreasing trend shows that class imbalance evolution occurs continuously over time in some projects. Classifiers unable to continuously adapt to that are likely to become unsuitable [15].

RQ2: In short, JIT-SDP does present class imbalance evolution, including (1) steep peaks increasing the ratio of defect-inducing software changes, which cause the problem to become more balanced; (2) sudden decreases in such ratio, leading to very severe levels of class imbalance (defect-inducing ratio of 0.001%); and (3) decreasing trends in such ratio, causing the problem to become even 4+ times more imbalanced over prolonged periods of time.

C. RQ3: The Impact of Class Imbalance Evolution on Predictive Performance of Existing Approaches

Table II presents the overall predictive performances across time steps obtained by OOB, UOB, OOB(FixedIR), OOB(FixedIR)* and OOB-SW. ORB is also presented in this table, but is analysed in Section VII-D.

Regarding the approaches that adopt strategies recommended by the JIT-SDP literature:

OOB(FixedIR) obtained poor g-mean compared to the other approaches. Together with OOB(FixedIR)*, it was ranked worst in terms of g-mean by the Scott-Knott test across datasets. Compared to OOB, which was ranked second, OOB(FixedIR)'s g-mean was up to 70.34% lower (Tomcat), with large effect sizes in 9 out of 10 datasets. OOB(FixedIR) computes the oversampling rate based on the imbalance ratio of the beginning of the project (until time step 500). This is unlikely to be suitable throughout the whole project, given the class imbalance evolution presented in Section VII-B. Therefore, class imbalance evolution was detrimental to OOB(FixedIR).

OOB(FixedIR)* also fixes the oversampling rate based on the imbalance ratio computed until time step 500. However, changes found to be defect-inducing before the end of the waiting period are only used for computing the imbalance ratio at the end of the waiting period. This led to a lower ratio of defect-inducing examples at the beginning of the project, causing the approach to focus more on the clean class. As a result, it obtained worse R_1 than most other approaches. Compared to OOB, for instance, its R_1 was up to 99.82% lower (Fabric8), with large effect sizes for all datasets. Its g-mean was as a result also lower. Similar to OOB(FixedIR), fixing the oversampling rate based on values that are not appropriate throughout the whole project was detrimental, meaning that class imbalance evolution also negatively affects this approach.

OOB-SW obtained worse g-mean compared to OOB with large effect sizes in 8 out of 10 datasets. Its g-mean was up to 38.41% lower than OOB's (Broadleaf). In addition, OOB-SW was ranked worse than OOB in terms of $|R_0 - R_1|$ (i.e., group 4), based on the Scott-Knott test across datasets. OOB-SW's $|R_0 - R_1|$ was up to 52.97% higher than OOB's (Broadleaf). The effect sizes were large in 7 out of 10 datasets in favor of

TABLE II
OVERALL PREDICTIVE PERFORMANCE, EFFECT SIZES AND STATISTICAL TESTS TO COMPARE LEARNING APPROACHES

Dataset	Classifier	R_0	R_1	$\|R_0 - R_1\|$	G-Mean
Fabric8	OOB	50.24 (2.20)	74.44 (2.15)	28.50 (2.97)	59.03 (0.63)
	UOB	42.27 (5.94) [-b]	83.70 (4.68) [b]	43.90 (8.76) [-b]	57.07 (2.42) [-b]
	OOB(FixedIR)	62.64 (3.19) [b]	59.02 (2.63) [-b]	48.64 (2.88) [-b]	50.92 (0.98) [-b]
	OOB(FixedIR)*	99.95 (0.03) [b]	0.13 (0.17) [-b]	99.82 (0.20) [-b]	1.66 (1.47) [-b]
	OOB-SW	73.31 (0.11) [b]	46.35 (0.36) [-b]	47.73 (0.32) [-b]	50.74 (0.29) [-b]
	ORB	60.34 (1.75) [b]	68.35 (1.43) [-b]	20.59 (2.95) [b]	60.93 (0.87) [b]
JGroups	OOB	59.38 (1.40)	56.67 (1.52)	28.13 (1.45)	54.71 (0.50)
	UOB	73.77 (3.10) [b]	45.12 (3.02) [-b]	36.81 (3.66) [-b]	55.09 (0.71) [m]
	OOB(FixedIR)	53.14 (0.62) [-b]	58.70 (0.59) [b]	21.98 (0.67) [b]	53.38 (0.31) [-b]
	OOB(FixedIR)*	68.24 (1.70) [b]	47.96 (1.43) [-b]	30.19 (1.65) [-b]	53.08 (0.63) [-b]
	OOB-SW	81.49 (0.07) [b]	35.33 (0.26) [-b]	57.51 (0.31) [-b]	47.95 (0.25) [-b]
	ORB	62.65 (0.79) [b]	56.73 (1.23) [*]	17.79 (1.30) [b]	57.76 (0.96) [b]
Camel	OOB	57.06 (1.44)	73.99 (0.92)	25.47 (1.60)	62.90 (0.70)
	UOB	55.57 (2.53) [-b]	71.28 (2.34) [-b]	29.27 (2.72) [-b]	60.35 (1.05) [-b]
	OOB(FixedIR)	91.54 (0.54) [b]	20.97 (0.84) [-b]	76.82 (1.22) [-b]	38.24 (1.15) [-b]
	OOB(FixedIR)*	87.42 (0.88) [b]	33.85 (0.90) [-b]	58.62 (1.20) [-b]	50.81 (0.94) [-b]
	OOB-SW	71.67 (0.11) [b]	40.38 (0.27) [-b]	64.61 (0.31) [-b]	40.28 (0.41) [-b]
	ORB	60.74 (0.61) [b]	70.41 (0.77) [-b]	17.03 (0.87) [b]	63.63 (0.50) [b]
Tomcat	OOB	59.81 (2.41)	61.75 (1.79)	29.42 (2.29)	57.28 (0.88)
	UOB	68.47 (2.51) [b]	50.04 (2.57) [-b]	33.69 (2.85) [-b]	55.18 (1.05) [-b]
	OOB(FixedIR)	93.54 (1.15) [b]	6.52 (1.44) [-b]	87.08 (2.51) [-b]	16.99 (2.29) [-b]
	OOB(FixedIR)*	75.43 (1.06) [b]	45.06 (1.25) [-b]	42.59 (1.96) [-b]	53.04 (0.72) [-b]
	OOB-SW	65.19 (0.15) [b]	52.30 (0.27) [-b]	35.93 (0.25) [-b]	54.53 (0.19) [-b]
	ORB	59.43 (1.39) [-b]	64.37 (0.68) [b]	16.07 (1.52) [b]	60.18 (0.80) [b]
Brackets	OOB	49.10 (0.27)	89.48 (0.29)	41.90 (0.25)	63.93 (0.09)
	UOB	54.59 (1.52) [b]	83.09 (1.64) [-b]	32.98 (2.17) [b]	64.24 (0.44) [b]
	OOB(FixedIR)	58.54 (1.94) [b]	76.25 (3.09) [-b]	40.54 (2.35) [b]	61.17 (1.15) [-b]
	OOB(FixedIR)*	76.71 (1.39) [b]	60.59 (1.14) [-b]	27.29 (1.75) [b]	63.46 (0.91) [-b]
	OOB-SW	56.28 (0.10) [b]	79.82 (0.28) [-b]	42.80 (0.20) [-b]	61.67 (0.16) [-b]
	ORB	61.68 (1.07) [b]	77.14 (1.05) [-b]	36.00 (1.44) [b]	63.66 (0.48) [-b]
Neutron	OOB	69.71 (0.79)	91.89 (0.62)	23.97 (1.33)	79.31 (0.45)
	UOB	58.82 (1.60) [-b]	92.45 (0.34) [b]	38.40 (1.54) [-b]	70.72 (1.43) [-b]
	OOB(FixedIR)	94.81 (0.38) [b]	22.23 (0.56) [-b]	77.35 (0.62) [-b]	41.09 (0.69) [-b]
	OOB(FixedIR)*	82.72 (1.07) [b]	70.81 (1.71) [-b]	20.10 (2.02) [b]	75.59 (0.85) [-b]
	OOB-SW	73.82 (0.10) [b]	83.08 (0.36) [-b]	20.49 (0.32) [b]	76.73 (0.21) [-b]
	ORB	79.89 (1.63) [b]	81.12 (1.37) [-b]	13.98 (2.14) [b]	79.92 (0.46) [b]
Spring-Integration	OOB	62.47 (2.06)	53.74 (1.80)	47.27 (1.82)	48.11 (0.72)
	UOB	55.65 (6.14) [-b]	59.30 (3.15) [b]	37.57 (4.86) [b]	52.19 (2.78) [b]
	OOB(FixedIR)	55.39 (0.38) [-b]	52.86 (0.68) [-b]	48.29 (0.60) [-b]	44.75 (0.49) [-b]
	OOB(FixedIR)*	98.96 (0.19) [b]	1.59 (0.19) [-b]	97.37 (0.31) [-b]	9.47 (0.68) [-b]
	OOB-SW	45.55 (0.21) [-b]	79.87 (0.34) [b]	39.52 (0.30) [b]	56.11 (0.24) [b]
	ORB	74.32 (0.86) [b]	44.31 (1.24) [-b]	37.30 (1.73) [b]	52.20 (0.83) [b]
Broadleaf	OOB	59.24 (1.23)	68.32 (1.89)	33.40 (2.50)	60.07 (0.75)
	UOB	59.32 (4.45) [*]	62.69 (4.92) [-b]	43.26 (4.37) [-b]	55.45 (1.31) [-b]
	OOB(FixedIR)	67.47 (1.21) [b]	58.39 (1.72) [-b]	39.77 (1.72) [-b]	58.17 (0.88) [-b]
	OOB(FixedIR)*	89.05 (0.96) [b]	26.74 (2.09) [-b]	62.55 (2.78) [-b]	45.68 (1.59) [-b]
	OOB-SW	78.20 (0.10) [b]	34.73 (0.31) [-b]	71.02 (0.33) [-b]	37.00 (0.60) [-b]
	ORB	61.60 (1.48) [b]	67.00 (1.47) [-b]	19.17 (2.32) [b]	61.97 (0.76) [b]
Nova	OOB	68.53 (0.35)	86.27 (0.70)	24.34 (0.59)	75.41 (0.23)
	UOB	65.56 (0.63) [-b]	90.84 (0.89) [b]	27.59 (0.75) [-b]	75.93 (0.15) [b]
	OOB(FixedIR)	81.62 (0.20) [b]	42.68 (0.46) [-b]	65.17 (0.34) [-b]	39.07 (0.73) [-b]
	OOB(FixedIR)*	86.95 (0.64) [b]	55.90 (1.37) [-b]	36.96 (1.74) [-b]	66.29 (0.91) [-b]
	OOB-SW	66.41 (0.05) [-b]	85.90 (0.17) [-b]	33.66 (0.17) [-b]	72.85 (0.10) [-b]
	ORB	75.30 (2.26) [b]	80.13 (4.07) [-b]	20.31 (1.57) [b]	75.68 (0.96) [*]
NPM	OOB	37.91 (1.84)	74.88 (1.17)	49.67 (1.52)	46.17 (0.77)
	UOB	38.26 (2.67) [*]	72.82 (1.88) [-b]	48.89 (1.71) [b]	45.86 (0.94) [-b]
	OOB(FixedIR)	43.93 (1.52) [b]	67.66 (2.06) [-b]	47.74 (1.64) [b]	46.35 (1.08) [*]
	OOB(FixedIR)*	82.77 (1.31) [b]	26.22 (1.13) [-b]	59.00 (1.86) [-b]	39.85 (1.03) [-b]
	OOB-SW	55.55 (0.18) [b]	62.74 (0.54) [-b]	43.68 (0.31) [b]	50.53 (0.35) [b]
	ORB	55.24 (0.84) [b]	63.94 (1.31) [-b]	31.73 (1.70) [b]	54.26 (0.91) [b]
Averages	OOB	57.35 [3] (9.51)	**73.14** [1] (13.32)	33.21 [2] (9.61)	60.69 [2] (10.53)
	UOB	57.23 [3] (10.91)	**71.13** [1] (16.60)	37.24 [3] (6.71)	59.21 [2] (8.93)
	OOB(FixedIR)	70.26 [2] (18.67)	46.53 [4] (22.79)	55.34 [5] (20.48)	45.01 [4] (12.57)
	OOB(FixedIR)*	**84.82** [1] (9.97)	36.89 [5] (23.77)	53.45 [5] (27.78)	45.89 [4] (23.70)
	OOB-SW	66.75 [2] (11.32)	60.05 [3] (20.75)	45.69 [4] (15.16)	54.84 [3] (12.76)
	ORB	65.12 [2] (8.22)	67.35 [2] (11.17)	**23.00** [1] (8.63)	**63.02** [1] (8.70)

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against WFL-OOB. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than WFL-OOB. Cells in bold correspond to the approaches belonging to the best group according to Scott-Knott. The groups' rankings are in square brackets in the average rows, with smaller numbers indicating better ranks. A table using larger font is available at http://doi.org/10.5281/zenodo.2555695.

OOB. OOB-SW can track class imbalance evolution based on its sliding windows. However, its strategy to deal with concept drift discards potentially useful past information, not being adequate to tackle class imbalance evolution, as discussed in Section III-C. This is probably the reason for its worse g-mean.

Regarding the online class imbalance learning approaches prepared to cope with class imbalance evolution:

UOB was more competitive against OOB in terms of g-mean than OOB(FixedIR) and OOB(FixedIR)*, being ranked in the second group according to Scott-Knott. However, its $|R_0 - R_1|$ was still ranked worse than OOB (i.e., in group 3), based on the Scott-Knott test across datasets. Tackling class imbalance evolution seems to have helped UOB, but UOB discards training examples from the majority class. This may be a risky strategy since the discarded examples may contain useful information.

OOB was ranked in the second best group in terms of g-mean and $|R_0 - R_1|$ and in the first group in terms of R_1. However, based on the Scott-Knott test across datasets, its R_0 was worse than all the JIT-SDP approaches from the literature. Conversely, it overcame both FixedIR and OOB-SW approaches, regarding R_1. OOB demonstrated a better balance between the finals R_0 and R_1 averages than the JIT-SDP approaches from the literature. Nevertheless, the difference in recalls ($R_0 - R_1$) was still high (up to 49.67). Section VII-D analyses the approach ORB proposed to overcome this problem.

RQ3: Existing JIT-SDP approaches based on fixed class imbalance ratio struggled to obtain competitive predictive performance in the presence of class imbalance evolution, obtaining g-mean up to 97.2% worse than OOB's. OOB-SW tracks changes in the imbalance ratio, but discards historic data, which has shown to be detrimental, leading to g-mean of up to 38.41% lower than OOB's. OOB achieved competitive g-mean, but may not be suitable for adoption in practice due to its high $|R_0 - R_1|$.

D. RQ4: The Benefit of the Proposed Approach ORB

RQ4 is partly answered by our proposed approach ORB. In this section, we check whether ORB is really able to improve predictive performance against the approaches found to be the most competitive in terms of g-mean and $|R_0 - R_1|$ in Section VII-C. These are OOB and UOB. ORB's main goal is to improve $|R_0 - R_1|$. So, we are particularly interested in this performance metric.

Regarding g-mean, ORB was isolated ranked in the best group, according to the Scott-Knott tests across datasets. The magnitudes of the differences in g-mean between these two approaches are in favor of ORB in 9 out of 10 datasets, but were not large – at most 7.8% (Spring-Integration). Therefore, ORB and OOB perform quite similarly in terms of g-mean.

Regarding $|R_0 - R1|$, again, ORB ranked first according to Scott-Knott, having outperformed all other approaches including OOB and UOB. Its $|R_0 - R1|$ was up to 45.38% lower than OOB's (Tomcat) and up to 63.59% lower than UOB's (Neutron). Overall, ORB was successful in obtaining the top g-means while improving its $|R_0 - R_1|$.

Regarding R_0 and R_1 individually, ORB was ranked in the second best group for both of these metrics. It sometimes wins and sometimes looses against other approaches from the best group on single datasets. This is because, to obtain a better $|R_0 - R_1|$, sometimes R_1 has to be reduced a bit and sometimes increased. This prevents the recall on the defect-inducing class to be too high at the expense of a large number of false alarms, or the recall on the clean class being too high at the cost of missing too many defect-inducing changes.

Fig. 4 shows an example of R_0, R_1 and g-mean plotted over time for Neutron. As we can see, during a large period of time,

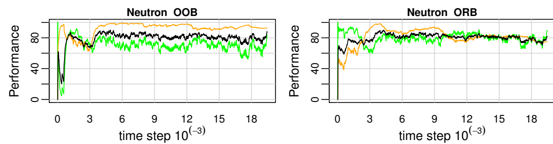

Fig. 4. R_0 (in lime / light grey), R_1 (in orange / dark grey) and g-mean (in black) over time for OOB and ORB on Neutron.

OOB obtained a very large R_1 of around 90%, while its R_0 was around 60%. So, even though OOB managed to retrieve most defect-inducing software changes in the project, around half of the changes flagged as defect-inducing were actually clean. Carefully inspecting all these changes that were actually clean would be costly for practitioners and would likely make them loose trust in the approach. ORB managed to reduce this problem, producing more balanced recalls.

RQ4: Our proposed approach ORB adjusts the resampling rate based on the ratio of examples predicted as being of the defect-inducing class. This enabled it to significantly reduce the problem of over-emphasising one class over the other, while maintaining top g-means. ORB is therefore successful in improving JIT-SDP predictive performance over state-of-the-art machine learning algorithms for class imbalance evolution. In particular, ORB's $|R_0 - R_1|$ was up to 45.38% better than OOB's and up to 63.59% better than UOB's.

VIII. Threats to Validity

Internal validity: when using machine learning approaches, poor parameter choices can highly influence the results [35], [36]. We mitigate this threat by using a grid search based on the first 5,000 changes of each dataset, as detailed in Section VI. This strategy has shown to be effective and easily automated.. The parameter values investigated included values that were used in previous work that investigated online class imbalance learning in the machine learning literature [10]. The set of values P_0 and P_1, used in Equations 3 and 2, were chosen based on preliminary experiments. A sensitive analysis has shown that ORB is not too sensitive to these parameters.

Construct validity: as explained in Section VI the evaluation metrics used in this work were R_0, R_1, $|R_0 - R_1|$ and g-mean. They were calculated prequentially with fading factors, as recommended for online learning studies [29]. These are the most commonly used metrics in the online class imbalance literature [15].

Statistical conclusion validity: Bootstrap-based Scott-Knott test and A12 effect size were used to address conclusion validity. The benefits of using them is explained in Section VI. Our work assumes that a given software change can become a training example once 90 days have passed from its commit, as explained in Section IV-A. However, projects that are not changed very often may require longer time gaps to reflect the confidence on the label of committed software changes.

External validity: this study was based on ten open source projects from the `GitHub` repository, as explained in Section V. All of these projects are currently and continuously being developed and maintained. They cover a variety of different characteristics, such as starting date, number of modified files per commit and number of commits per day. The programming language was Java in most cases. The results obtained in this study may not generalize to other projects (specially proprietary projects) or to different problems than JIT-SDP. We would be keen to perform additional case studies with proprietary projects in the future.

IX. Conclusions and Implications

We conducted the first work to investigate and deal with class imbalance evolution in JIT-SDP. We show that class imbalance evolution is a significant issue in JIT-SDP. The imbalance status can vary between fairly balanced and extremely imbalanced during the course of a project (RQ2). Existing JIT-SDP approaches did not perform well under this scenario, obtaining g-means up to 97.2% lower than those obtained by OOB, an online class imbalance learning approach for coping with class imbalance evolution (RQ3). Despite obtaining better g-mean, OOB still often favored the performance on one class too much in detriment of the other (RQ3). Our proposed approach ORB was able to improve on that, being top ranked both in terms of g-mean and differences in recalls. In particular, its difference in recals was up to 45.38% lower than OOB's (RQ4). We also provided an analysis of the typical defect discovery delay in the GitHub projects used in this study, showing to what extent verification latency occurs in JIT-SDP. The analysis suggests that a waiting time of 90 days is adequate for those projects (RQ1).

Our work has implications to both practice and future research. Regarding implications to practice:

1) Based on the results from RQ2 and RQ3, it is important for practitioners to apply online learning algorithms able to tackle class imbalance evolution if they wish to perform JIT-SDP. Otherwise, they risk using classifiers that, over time, will get a high rate of false alarms (i.e., low recall on clean changes) or miss a substantial amount of defect-inducing software changes (i.e., low recall on defect-inducing changes).

2) Based on RQ3, even the sliding window strategy recommended in the JIT-SDP literature for dealing with concept drift in $p(x|y)$ is not enough to cope with class imbalance evolution. Therefore, simply re-building classifiers from scratch over time is not enough to obtain good predictive performance over time.

3) Based on RQ4, practitioners adopting ORB could potentially be less overloaded by false alarms at the same time as not missing too many defect-inducing changes w.r.t. other approaches in existing literature. Future studies within their company's environment would enable to check whether such findings generalize to their company.

4) Based on RQ1, our study further emphasizes that, when deciding which JIT-SDP to adopt, it is important to investigate them taking verification latency into account.

Future work includes: (i) quantitative and qualitative studies of ORB with industry using proprietary data; (ii) investigation of other base classifiers than Hoeffding trees; (iii) and techniques to automatically tune parameters of machine learning approaches over time, including the waiting period.

References

[1] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[2] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering (EMSE)*, vol. 21, no. 5, pp. 2072–2106, 2015.

[3] N. Nan and D. E. Harter, "Impact of budget and schedule pressure on software development cycle time and effort," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 5, pp. 624–637, 2009.

[4] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 6, pp. 1276–1304, 2012.

[5] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 757–773, 2013.

[6] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[7] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015, pp. 99–108.

[8] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 5, pp. 412–428, 2018.

[9] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, "Learning in non-stationary environments: A survey," *IEEE Computational Intelligence Magazine*, vol. 10, no. 4, pp. 12–25, 2015.

[10] S. Wang, L. L. Minku, and X. Yao, "Resampling-based ensemble methods for online class imbalance learning," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 27, no. 5, pp. 1356–1368, 2015.

[11] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 17th International Workshop on Mining Software Repositories*, ser. MSR '05, 2005, pp. 1–5.

[13] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, 2011, pp. 153–162.

[14] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *Empirical Software Engineering Journal (EMSE)*, vol. 21, no. 2, pp. 605–641, 2016.

[15] S. Wang, L. L. Minku, and X. Yao, "A systematic study of online class imbalance learning with concept drift," *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2018.

[16] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," in *Proceedings of the 6th Working Conference on Mining Software Repositories (MSR)*, 2009, pp. 51–60.

[17] Z. Mahmood, D. Bowes, P. Lane, and T. Hall, "What is the impact of imbalance on software defect prediction performance?" in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2015, pp. 4.1–4.4.

[18] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability (TR)*, vol. 62, no. 2, pp. 434–443, 2013.

[19] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007, pp. 196–204.

[20] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 6, pp. 534–550, June 2018.

[21] N. C. Oza, "Online bagging and boosting," in *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 3, 2005, pp. 2340–2345.

[22] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2000, pp. 71–80.

[23] P. Zhang, X. Zhu, J. Tan, and L. Guo, "Classifier and cluster ensembles for mining concept drifting data streams," in *Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM)*, 2010, pp. 1175–1180.

[24] K. B. Dyer, R. Capo, and R. Polikar, "Compose: A semi-supervised learning framework for initially labeled non-stationary streaming data," *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, vol. 25, no. 1, pp. 12–26, 2013.

[25] A. D. Pozzolo, G. Boracchi, O. Caelen, C. Alippi, and G. Bontempi, "Credit card fraud detection: a realistic modeling and a novel learning strategy," *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, vol. 29, no. 8, pp. 3784–3797, 2018.

[26] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 9, pp. 1208–1215, 2013.

[27] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*, 2015, pp. 966–969.

[28] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 21, no. 9, pp. 1263–1284, 2009.

[29] J. Gama, R. Sebastião, and P. P. Rodrigues, "On evaluating stream learning algorithms," *Machine Learning*, vol. 90, no. 3, pp. 317–346, 2013.

[30] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 4, pp. 537–551, 2013.

[31] T. Menzies, Y. Yang, G. Mathew, B. Boehm, and J. Hihn, "Negative results for software effort estimation," *Empirical Software Engineering (EMSE)*, vol. 22, no. 5, pp. 2658–2683, 2017.

[32] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[33] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 1, pp. 1–18, 2017.

[34] ——, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–1, 2018.

[35] T. Menzies and M. Shepperd, "Special issue on repeatable results in software engineering prediction," *Empirical Software Engineering (EMSE)*, vol. 17, pp. 1–17, 2012.

[36] L. Song, L. L. Minku, and X. Yao, "The impact of parameter tuning on software effort estimation using learning machines," in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering (PROMISE)*, 2013, pp. 1–10.