Towards Reliable Online Just-in-time Software Defect Prediction

George Cabral, *Member, IEEE,* Leandro L. Minku, *Senior Member, IEEE*

***Abstract*—Throughout its development period, a software project experiences different phases, comprises modules with different complexities and is touched by many different developers. Hence, it is natural that problems such as Just-in-Time Software Defect Prediction (JIT-SDP) are affected by changes in the defect generating process (concept drifts), potentially hindering predictive performance. JIT-SDP also suffers from delays in receiving the labels of training examples (verification latency), potentially exacerbating the challenges posed by concept drift and further hindering predictive performance. However, little is known about what types of concept drift affect JIT-SDP and how they affect JIT-SDP classifiers in view of verification latency. This work performs the first detailed analysis of that. Among others, it reveals that different types of concept drift together with verification latency significantly impair the stability of the predictive performance of existing JIT-SDP approaches, drastically affecting their reliability over time. Based on the findings, a new JIT-SDP approach is proposed, aimed at providing higher and more stable predictive performance (i.e., reliable) over time. Experiments based on ten GitHub open source projects show that our approach was capable of produce significantly more stable predictive performances in all investigated datasets while maintaining or improving the predictive performance obtained by state-of-art methods.**

***Index Terms*—Just-in-time Software Defect Prediction, Online Learning, Concept Drift, Verification Latency, Class Imbalance Learning.**

I. Introduction

Defects in software are one of the main threats to software development companies, increasing software costs and potentially damaging companies' reputations. A report created by The Consortium for IT Software Quality (CISQ)[1] revealed that poor software quality costed \$ 2.8 trillions in 2018 only in US. This same report states that developers introduce on average 100 to 150 defects for every thousand of lines of code, among which 10% can be considered serious. However, reducing the number of software defects is challenging, especially considering that the presence of defects can be associated to multiple uncontrollable factors such as changes in the development team and time to market, and exacerbated by limited testing resources.

To help with improving software quality, methods for automatically detecting defect-inducing software changes based on Machine Learning have been proposed [1], [2], [3]. These methods are frequently referred to as Just-In-Time Software Defect Prediction (JIT-SDP) [4], [5], [6] methods. They can alert a software developer of potential defects associated to their software change at commit time (i.e., just-in-time), when the change is still fresh in the developer's mind and easier/cheaper to be inspected. In this way, software testing and inspection can be done in a more cost-effective way, contributing towards better software quality.

A key problem in most JIT-SDP studies is that they do not respect the chronology of the available training data when building JIT-SDP classifiers, i.e., their experiments may use data that, in practice, would be unavailable to train the classifier. As pointed out by Tan et. al. [7], this leads to overestimations of predictive performance, giving the impression that JIT-SDP classifiers perform better than they would do in real scenarios.

The reason for the overestimations of predictive performance is twofold. First, JIT-SDP is affected by concept drifts [8], [6], i.e., changes in the defect generating process. These changes mean that JIT-SDP classifiers trained on data generated at certain periods of time may be unsuitable for later periods of time [6], because the training data generated at earlier periods may not share the same underlying defect generating process as the software changes produced at later periods. Therefore, training a classifier with data from the future could cause this classifier to perform well on future commits, when in practice it would not have performed so well due to the unavailability of data similar to the future data for training.

Second, JIT-SDP suffers from verification latency [8]. This refers to the fact that the label associated to a software change arrives with a delay, rather than arriving at commit time. In particular, a software change can only be labelled as defect-inducing once a defect associated to it is found. And, it can only be labelled as clean once enough time has passed for one to be confident that this software change is not associated to any defects. This issue has been overlooked by most existing work [9], [5], [2]. When associated with concept drift, it means that labelled training data coming from the same defect generating process as the software changes being currently predicted may be unavailable in practice, making it difficult to adapt to concept drift and further hindering predictive performance. The negative effects of that remain unnoticed if verification latency is overlooked in an experimental study.

Existing studies that take chronology into account reveal that the predictive performance of JIT-SDP classifiers gets worse [6] and fluctuates over time [8] potentially as a result

G. Cabral is with the Department of Computing, Federal Rural University of Pernambuco, BR, george.gcabral@ufrpe.br

L. L. Minku is with the School of Computer Science, The University of Birmingham, UK, L.L.Minku@bham.ac.uk

This work was supported by EPSRC Grants No. EP/R006660/2 and EP/P005578/1.

[1]https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf

of concept drift and exacerbated by verification latency. At any given point in time, JIT-SDP classifiers may be performing very well or failing dramatically [8]. This is a serious impediment for wider adoption in practice, as it renders JIT-SDP classifiers unreliable even if the overall average predictive performance across time would seem acceptable. Stability of predictive performance is thus an important issue to be considered when dealing with JIT-SDP. A *reliable* JIT-SDP method should perform *consistently well over time*, i.e., it should obtain stable and high predictive performance over time.

Very few studies so far have considered concept drift and verification latency in JIT-SDP [6], [7], [8], [10] and none of them provide a detailed understanding of how different types of concept drift affect predictive performance in JIT-SDP in the presence of verification latency. None of them have proposed methods to overcome the challenges posed by verification latency in view of concept drift either. A detailed understanding is essential for the proposal of strategies able to respond to concept drifts in realistic scenarios that take chronology and verification latency into account, creating more reliable JIT-SDP classifiers. Our paper aims at providing such understanding and proposing a novel approach to improve the reliability of JIT-SDP classifiers.

We focus on *online* JIT-SDP, where new training examples are produced over time and used to update classifiers. This enables the adoption of strategies to potentially cope with concept drift. With that in mind, this paper answers the following Research Questions (RQs):

- **(RQ1:)** Which types of concept drift occur in online JIT-SDP and how they affect the predictive performance of existing online JIT-SDP methods in view of verification latency? In particular, how do they affect the reliability of existing online JIT-SDP methods?
- **(RQ2:)** In view of the knowledge provided by RQ1, how to improve the reliability of online JIT-SDP classifiers, so that they perform more consistently well over time? How well such new method performs compared with existing methods?

To answer RQ1, we carefully analyse the online JIT-SDP classifiers and their predictive performance over time taking verification latency into account, showing that there is evidence supporting the existence of different types of concept drift in JIT-SDP. We then highlight the strengths and weaknesses of existing online JIT-SDP methods for tackling these concept drifts, paying close attention to their reliability over time. To answer RQ2, we propose a novel method able to use the predictions given by the JIT-SDP classifier to decide how to adapt to concept drift. The use of the predictions rather than the true labels of the software changes means that adaptation can commence even before the true labels of such changes are revealed. The method also makes use of unlabelled data to more efficiently recover from concept drifts. Our study based on ten open source data sets shows that our proposed method achieves better robustness to concept drift, leading to more reliable JIT-SDP classifiers, i.e., classifiers able to perform more consistently well through time than existing online JIT-SDP methods.

Our paper provides the following novel contributions:

- the first detailed analysis of concept drift in online JIT-SDP, revealing what types of concept drift are present;
- a detailed understanding of when and why existing online JIT-SDP methods are unable to cope with certain types of concept drift, which hinders their reliability;
- the first detailed analysis of the stability of online JIT-SDP's predictive performance over time, taking verification latency into account;
- a novel online method (Prediction-Based Sampling Adjustment - PBSA) to cope with different types of concept drift in the presence of verification latency in JIT-SDP, leading to more reliable JIT-SDP classifiers.

This paper is further organized as follows: Section II provides basic definitions for online JIT-SDP. Section III presents related work; Section IV introduces the datasets used in our study; Section V thoroughly discusses the types of concept drifts present in JIT-SDP and the effectiveness of state-of-art methods for tackling them (RQ1). Section VI proposes our new method to improve reliability and compares it to existing methods (RQ2). Section VII presents threats to validity. Section VIII presents conclusions and implications of this work.

II. Problem Formulation

Definition 1 (commit time step) A sequential index representing the order of arrival of software changes in terms of their commit time. Each software change is a test example requiring a prediction at commit time.

Definition 2 (test example) A test example $\vec{x}_i$ is a software change represented by a vector of features $\vec{x}_i$ that needs to be predicted as clean or defect-inducing.

In this work, we adopt the software change features proposed by Kamei et al. [5], as they have been vastly adopted by JIT-SDP literature [9], [6], [3], [2], [8] and shown to be suitable for practical scenarios [8], [10]. These features are: (1) NS - number of modified subsystems; (2) ND - number of modified directories; (3) - NF - number of modified files; (4) Entropy - distribution of modified code across each file; (5) LA - lines of code added; (6) LD - lines of code deleted; (7) LT - lines of code in a file before the change; (8) FIX - flag indicating if the change is a defect fix; (9) NDEV - number of developers that touched the files; (10) AGE - average time interval between last and current change; (11) NUC - number of unique last changes to the files; (12) EXP - developer experience; (13) REXP - recent developer experience; and (14) SEXP - developer experience on a subsystem.

Definition 3 (training time step) A sequential index representing the order of arrival of training examples. Each training example is used for updating the JIT-SDP classifier as soon as it becomes available.

It is important not to confuse time step with Unix timestamp. A Unix timestamp is a moment in time measured in terms of number of seconds elapsed since the Unix Epoch on January 1st, 1970 at UTC, whereas a time step is a sequential index representing order of arrival.

Definition 4 (training example) A training example $s_i = (\vec{x}_i, y_i)$ is a software change represented by a vector of features

$\vec{x}_i$ and its respective class label y_i, where i is the training time step. The features are the same as the ones described in the definition of test example. The class label can be either clean (represented by 0) or defect-inducing (represented by 1).

Definition 5 (verification latency) Verification latency is the delay for obtaining the label (clean or defect-inducing) of a software change. Consider a software change committed at a Unix timestamp u_i. The label corresponding to this software change can only become available at a Unix timestamp u_k (s.t. $u_k > u_i$). This is because, at commit time, it is unknown whether this software change truly will or will not induce defects. Indeed this is the reason why software changes have to be *predicted* as clean or defect-inducing at commit time in JIT-SDP. Therefore, verification latency is in practice inherent to the JIT-SDP problem.

To take this into account, we adopt the framework proposed by Cabral et al. [8], which produces labels based on the cases below, where w is a parameter called *waiting time*:

- No defect is found to be induced by the software change during w days after its commit – it will be labeled as clean once w days have passed, producing a clean training example;
- A defect is found to be induced by the software change in $t < w$ days after its commit – it will be labeled as defect-inducing once t days have passed, producing a defect-inducing training example; and
- A defect is found to be induced by the software change after $t > w$ days from its commit – it will be first labeled as clean once w days have passed and used to produce a clean training example, and then it will be labeled as defect-inducing once t days have passed and used to produce a defect-inducing training example.

Definition 6 (class imbalance) A problem where the number of training examples of a given class is much smaller than the number of training examples of another class is referred to as a class imbalanced problem.

JIT-SDP is typically a class imbalanced problem where the defect-inducing class is a minority. For instance, the proportion of examples of the defect-inducing class of the data sets used in this study is shown in Table I, which is presented in Section IV. If not treated, class imbalance can cause classifiers to over-emphasize the majority class in detriment of the minority class [11].

Definition 7 (online learning) Consider a data stream composed of training examples ordered by the time they were produced $\mathcal{S} = \{(\vec{x}_i, y_i)\}_{i=1}^{\infty}$, where i is the training time step. Online learning maintains a classifier $\hat{f}$ that is updated whenever a new training example $(\vec{x}_i, y_i) \in \mathcal{S}$ becomes available. Such update may or may not require access to previous training examples $(\vec{x}_j, y_j)$, where $j < i$, depending on the machine learning algorithm being used.

Whenever a prediction is required for a new software change, the most up-to-date classifier $\hat{f}$ is used. This up-to-date classifier is the one trained on all training examples $(\vec{x}_i, y_i)$ produced *before* the prediction is required. Therefore, chronology is always respected, i.e., predictions are never made by using classifiers trained on data that would not yet have been available in practice. Therefore, online learning reflects practical JIT-SDP environments.

Definition 8 (concept) A concept is a joint probability distribution ($p_t(\vec{x}, y) = p_t(y|\vec{x})p_t(\vec{x}) = p_t(\vec{x}|y)p_t(y)$) underlying a machine learning problem at a given time step t. In JIT-SDP, this can be seen as the status of the defect generating process underlying commit time step t. It represents the underlying function that captures the relationship between features and classes, the chances of observing examples from each class and the chances of observing each feature value.

Definition 9 (concept drift) A concept drift takes place when the concept changes over the time, i.e., $p_t(\vec{x}, y) \neq p_{t+\Delta}(\vec{x}, y)$, where $\Delta \neq 0$. There are different types of concept drift, based on the component of the joint probability distribution that they affect [11]:

1) **Changes in the proportions of examples of each class** [8]. These proportions are represented by the prior probabilities of the classes, i.e., $p(y)$. In a class imbalanced problem, such changes are referred to as class imbalance evolution [8].
2) **Evolving probability of observing different feature values given the class**, i.e., $p(\vec{x}|y)$. Such evolution can be of one or both of the following types:
   - Changes in how likely an example is to belong to a given class (defect-inducing or clean) given its feature values, i.e., in the posterior probabilities $p(y|\vec{x})$. These concept drifts mean that changes described by features that would typically be associated to the clean class may now be associated to the defect-inducing class, or vice-versa.
   - Changes in the frequency of observing different feature values, i.e., changes in the probability distribution of the features $p(\vec{x})$. These concept drifts mean that the typical values of the features vary over time.

One of the typical side-effects of concept drifts is that they cause drops in the predictive performance of classifiers, resulting in unstable predictive performance over time [12]. Fluctuations have been observed in previous JIT-SDP work that consider chronology or online scenarios [8], [10], [6]. However, to be adopted in practice, a JIT-SDP classifier must attempt to minimize these fluctuations (i.e., yield a stable predictive performance) while maximizing its predictive performance over time. Combining a high and stable predictive performance over time produces a more reliable classifier.

Definition 10 (reliability) We define reliability of a classifier as a combination of a high and stable predictive performance. Consider a 2-dimensional space formed by the standard deviation and the average of the predictive performance through time, respectively. To maximize reliability, one has to minimize the first dimension while maximizing the second.

III. Related Work

A. JIT-SDP

Many studies have been conducted investigating different aspects of the JIT-SDP problem, such as effort-aware JIT-SDP

[4], [5], local vs. global JIT-SDP classifiers [3] and cross-project JIT-SDP [13], [9].

Effort-aware JIT-SDP [4], [5] takes into account the workload generated to the software quality assurance team. This task involves maximizing the classifier accuracy while minimizing the software quality assurance team's effort. The effort is often proportional to the number of lines of code modified by a change [5]. Chen et al. [2] formalized JIT-SDP as a multi-objective optimization problem to conduct Effort-aware JIT-SDP. They used the number of bugs found and their corresponding lines of code changed as objectives to be maximized and minimized, respectively [14]. Both performance objectives were optimized over six open source datasets, however, the results still suggests room for improvements w.r.t. the performance on the clean class.

Yang et al. [3] evaluated the effectiveness of local [15], [16] vs. global JIT-SDP methods. Local methods cluster the whole training data into regions composed of similar training examples and then create separate classifiers to learn training examples from each region. Global methods create a single classifier that learns all training examples. The study conducted on six open source datasets found that global methods produce better JIT-SDP classifiers.

Other studies investigated the use of cross-project data in JIT-SDP [9], [13]. Kamei et al. [9] were the first ones to investigate cross-project data in JIT-SDP. They used 11 open source projects and among their findings, they confirm that the cross-project approach can be useful for projects with limited historical data. Catolino et al. investigated 14 mobile projects, with number of software changes ranging from 193 to 13067. Among their findings is the fact that the features LA, LD, ND, NF, NUC and NDEV better contribute to identify defect-inducing software changes. In cross-project JIT-SDP, training data includes data generated by projects other than the project of interest. It was initially believed to be particularly beneficial in the initial phase of a project, when there is not enough within-project data to build a suitable within-project classifier. More recent work has found that it can also be useful for prolonged periods of time as will be discussed in Section III-C.

Most studies use resampling strategies to cope with class-imbalance in JIT-SDP [2], [13], [9], [3]. In particular, undersampling of training examples of the clean class is frequently used to avoid the learning algorithm over-emphasizing this class in detriment of the defect-inducing class [2], [9], [3].

All studies above investigate JIT-SDP in an offline learning scenario where a JIT-SDP classifier is built based on a pre-existing training set, and then is applied on a given project. These studies ignore the chronology of the data. Few studies take chronology into account; these will be discussed in Sections III-B, III-C and III-D.

B. Analysis of Concept Drift in JIT-SDP

A big challenge in studying concept drift is that, for real world problems, we have no access to the true underlying statistical distribution. Therefore, concept drifts must be inferred based on the data being received over time and in their respective classifiers. McIntosh and Kamei [6] investigated the presence of concept drift in JIT-SDP. They showed that (i) JIT classifiers lose a large proportion of predictive performance after one year; (ii) the predictive importance of most families of features fluctuates over time, suggesting that the properties of defect-inducing changes tend to evolve as projects age. Both findings are strong evidence that concept drift occurs. However, this paper does not investigate which types of concept drift occur in JIT-SDP, which is important for proposing novel JIT-SDP methods that are able to better handle concept drift. It does not consider verification latency in the analysis either.

Cabral et.al. [8] analyzed the proportion of examples of the clean and defect-inducing class over time based on an exponential smoothing function of the class labels. They found that such proportion varies over time, i.e., there is class imbalance evolution. It means that the severity of class imbalance varies over time, and sometimes the defect-inducing class can even become a minority, hindering the predictive performance of existing JIT-SDP methods. They also found that existing JIT-SDP methods that discard old data (i.e., sliding windows [6]) are detrimental to the predictive performance compared to methods that adopt other mechanisms to deal with concept drift. This suggests that old training examples do not always become detrimental to predictive performance. In particular, there may be recurrent concepts in JIT-SDP, i.e., concepts that are valid for certain periods of time and reappear again at later periods. However, their study has not analyzed other types of concept drift than drifts affecting the proportion of clean and defect-inducing examples.

The literature provides high level evidences of concept drift in JIT-SDP, notwithstanding, a deeper understanding of which specific types of concepts drifts occur and how they affect JIT-SDP is necessary to guide the development of more effective JIT-SDP classifiers.

C. Dealing with Concept Drift in JIT-SDP

To deal with concept drift in JIT-SDP, McIntosh and Kamei [6] suggested the use of the most recent window of training examples (i.e., sliding windows) to build JIT-SDP classifiers. Wang et.al. [17] introduced two general purpose online resampling methods for tackling class imbalance evolution, namely improved Oversampling Online Bagging (OOB) and improved Undersampling Online Bagging (UOB). They relied on the assumption that making the class proportions even by oversampling examples of the majority class and undersampling examples of the minority class, respectively, is sufficient for solving this issue. These methods were investigated in the context of JIT-SDP by Cabral et al. [8]. They showed that this strategy is insufficient for JIT-SDP, as large gaps between the recall on the clean and defect-inducing classes may still occur when adopting this method. They proposed Oversampling Rate Boosting (ORB) [8], which further adjusts the resampling rate based on the level of imbalance in the *predictions* given by a classifier. They found that this leads to an advantage in predictive performance over OOB, UOB [17] and Sliding Window [6] methods in JIT-SDP. However, their proposal carries a weakness of resampling only the most recent labeled change. As there is verification latency in JIT-SDP, this most recently labeled change may already be outdated and unrepresentative of the current concept. Moreover, OOB, UOB and ORB only have strategies to deal with concept drifts affecting the proportion of examples of each class, and do not attempt, even indirectly, to deal with other types of concept drift in JIT-SDP.

More recently, Tabassum et al. [10] showed that cross-project data generate better classifiers throughout the whole development cycle (not only in the beginning of the project) when considering online JIT-SDP. Such data can also alleviate drops in predictive performance that may be caused by concept drifts. However, their study focused on investigating cross-project learning and has not investigated the issue of concept drift itself in JIT-SDP.

D. *Verification Latency in JIT-SDP*

To the best of our knowledge, Tan et al. [7] was the first work to alert the software engineering community about the verification latency problem in JIT-SDP. They showed that ignoring chronology and verification latency leads to overoptimistic estimations of the predictive performance in JIT-SDP. They recommended the use of a method that can be updated with incoming chunks of training examples to learn JIT-SDP classifiers over time while taking verification latency into account. However, their study assumes that the delay to receive the labels from all examples from both classes is fixed, which has more recently been shown to be unrealistic in JIT-SDP [8]. In addition, they do not propose strategies to deal with concept drift, or strategies to overcome the challenges posed by verification latency.

Cabral et al. [8] showed that the delay for assigning the correct label of a training example in JIT-SDP may vary from days to years, which may negatively impact the classifier's predictive performance when there is concept drift. However, they have not proposed any strategy to improve classifiers' predictive performance in the presence of verification latency. Tabassum et al. [10] also took verification latency into account, but did not propose methods to overcome this issue.

Overall, even tough existing studies are aware of verification latency, none of them proposed strategies to overcome the low predictive performance that may be caused by verification latency when there is concept drift.

IV. Datasets

We have used the same ten `GitHub` open source projects as in previous work [8]. This number of projects is in line with other existing studies in the area [7], [9]. These projects were chosen among projects with more than 5 years of duration, rich history (at least around $\sim$8k software changes) and good defect-inducing changes ratio ($\sim$20% overall).

The defect-inducing software changes were obtained by using the tool `Commit Guru` [18].

The most commonly used features in the literature (presented in Section II) are retrieved by `Commit Guru`. These features have shown to perform well in JIT-SDP research [5], [9], [6]. `Commit Guru` was also configured to retrieve the defect discovery delay, i.e., the time taken (verification latency) to discover the label of a defect-inducing change. This was set as the difference between (i) the change time stamp and (ii) the time stamp of its associated fix change.

TABLE I: Datasets' statistics

Dataset	Number of changes	Defect-inducing proportion	Period	Median defect discovery delay (in days)
Fabric8	13,003	20%	04/2011 - 05/2017	40
JGroups	18,316	17%	09/2003 - 11/2017	117
Camel	30,517	20%	03/2007 - 11/2017	29
Tomcat	18,877	28%	03/2006 - 12/2017	201
Brackets	17,310	23%	12/2011 - 08/2017	139.5
Neutron	19,450	24%	01/2011 - 11/2017	103
Spring Integration	8,691	27%	11/2007 - 10/2017	416.5
Broadleaf	14,911	17%	12/2008 - 09/2017	43
Nova	48,937	25%	05/2010 - 01/2018	97
NPM	7,892	18%	09/2009 - 09/2017	113

Table I shows specific information for each dataset. The median of the defect discovery delay suggests that the projects Tomcat (201 days) and Spring-Integration (416.5 days) are highly affected by verification latency. In theory, the larger the verification latency, the longer the delay in receiving training examples of the minority class. This poses an extra challenge to JIT-SDP. Not only JIT-SDP has relatively few examples of the defect-inducing class, but also these examples may already be obsolete (due to concept drift) when they arrive. If an obsolete defect-inducing example is used to update a JIT-SDP classifier, it might add irrelevant, or even noisy, information.

V. (RQ1:) Analysis of Concept Drift in JIT-SDP And Its Effect On the Reliability Of Existing JIT-SDP Methods

A. *Experimental Setup*

To answer RQ1, besides investigating the types of concept drift outlined in Section II that may be present in JIT-SDP, we will also analyze how these types of concept drift affect predictive performance (and in particular reliability) of existing JIT-SDP methods. Previous work [6], [8] has already analyzed class imbalance evolution in JIT-SDP and shown that JIT-SDP methods that assume a fixed proportion of examples of each class $p(y)$ over time fail to achieve acceptable predictive performance over time. Therefore, this section will concentrate mainly on concept drifts affecting $p(y|\vec{x})$ and $p(\vec{x})$.

The datasets introduced in Section IV and the following methods will be investigated, using Hoeffding Trees [19] as base learners: i) Oversampling-based Online Bagging (OOB) [17]; ii) Undersampling-based Online Bagging (UOB) [17]; iii) Oversampling-based Online Bagging Sliding Windows (OOB-SW) [6]; and iv) Oversampling Rate Boosting (ORB) [8]. These methods were chosen because they form the state-of-the-art in online within-project JIT-SDP [8]. The analysis presented in Section V-B is a detailed analysis to identify the factors responsible for sudden, significant drops in predictive performance – among them, concept drifts of type $p(y|\vec{x})$ and $p(\vec{x})$. It will thus concentrate on two datasets (Camel and Tomcat) that are representative of different types of concept drift, and ORB, which is the method that obtained the best within-project JIT-SDP predictive performance in non-stationary environments so far [8]. Section V-C will provide an overall discussion of all four methods. All methods fully respect the chronology of the data, including verification latency. Therefore, our entire analysis respects chronology and takes verification latency into account. We focus on within-project JIT-SDP as the use of cross-project data would prevent the identification of concept drifts, given that cross-project approaches use a mix of data that comes from potentially different defect generating processes to train JIT-SDP classifiers.

*1) **Performance Metrics**:* The evaluation metrics used for assessing the classifiers' predictive performances are the recalls on the clean ($rec(0)$) and defect-inducing ($rec(1)$) classes, the average of the absolute differences between recalls for each time step ($|rec(0) - rec(1)|$) and the g-mean ($\sqrt{rec(0) \times rec(1)}$). The metrics $rec(0)$, $rec(1)$ and g-mean were adopted because they were recommended as unbiased metrics for evaluating predictive performance in class imbalance learning studies [20], different from other metrics such as precision and F1-score [21], [20]. This is particularly important in online class imbalance learning [17], where the imbalance ratio may vary over time. The metric $|rec(0) - rec(1)|$ enables us to quantify the similarity of the recalls on both classes and was also adopted in Cabral et al. [8]'s study. Small values for this metric assure that the strategy adopted for coping with class imbalance is effective, i.e., the improved predictive performance on the minority class is not at the cost of a worse performance on the majority class. A large value indicates a high bias towards one of the classes, making the classifier unreliable to practitioners. It is important to note that $rec(0) = 1 - FalseAlarmsRate$, where $FalseAlarmsRate$ is the ratio of clean software changes that have been classified as defect-inducing. Therefore, $FalseAlarmsRate$ is also taken into account by the evaluation metrics used in this study. We have not adopted Area Under the ROC Curve (AUC) because it incorporates several threshold values that are not meaningful in practice, having been recently discouraged in the context of software defect prediction [22].

The recalls are computed in a prequential way and with a fading factor, as recommended for online learning studies in the presence of concept drift [23]. This enables tracking changes in the predictive performance over time. As in previous work [8], [10], [17], [11], the fading factor was $\theta = 0.99$. Too large (small) θ causes the prequential performance to vary wildly (have almost imperceptible variations) over time. The value of $\theta = 0.99$ enables a good trade-off between tracking changes in performance while preventing wild variations.

*2) **Parameters Choice**:* For tuning the classifiers' parameters, a grid search based on the execution of each dataset up to the commit time step 5000, using g-mean as evaluation criterion, was conducted based on the following values, where values in bold face are the ones most often included in the parameter configuration that led to top g-mean across datasets and were thus chosen for the experiments[2]: ensemble size (n) = {10,**20**,30,40}; decay factor (θ') = {0.9,**0.99**,0.999}; and waiting period (ω) = {**90**,180}. For OOB(FixedIR), the imbalance ratio was fixed at commit time step 500. For the OOB-SW, sliding windows of size **90** and 180 days were tested. The values tested for the ORB parameters were: moving average window size = {50, **100**, 200}; th = {0.3, **0.4**, 0.5}; l_0 = {5, **10**, 15}; l_1 = {6, **12**, 18}; m = {**1.5**, 2.0, e}; and n = {**3**, 5, 7}. Finally, thirty runs are conducted with different random seeds using the chosen parameters for each method and dataset.

[2]Some parameters are shared among the methods but were tuned separately.

The comparisons among the methods were supported by the Scott-Knott multiple comparison procedure to separate the methods into non-overlapping groups, regarding each overall performance metric, as suggested by Menzies et al. [24]. This test was conducted considering the total number of experiments (i.e., 10 datasets times 30 executions for each method). Vargha and Delaney's nonparametric A12 effect size [25] was used to ensure that groups can only be split by the Scott-Knott test if the effect sizes between them are not insignificant. Therefore, we will refer to this Scott-Knott procedure as Scott-Knott-A12. As suggested in previous work [25], [24], A12 $<$0.56, $\geq$ 0.56, $\geq$ 0.64 and $\geq$ 0.71 are considered insignificant, small, medium and large, respectively.

B. Investigating Concept Drifts in JIT-SDP

Previously, Cabral et al. [8] have shown that concept drifts in $p(y)$ occur in JIT-SDP, requiring strategies to tackle it. In this section, we thus concentrate mainly on investigating whether concept drifts in $p(y|\vec{x})$ and $p(\vec{x})$ (which may or may not happen simultaneously with changes in $p(y)$) also occur, and whether this would require coping strategies that are unavailable in existing JIT-SDP methods. Such analysis is presented in Sections V-B1 and V-B2.

Both Sections V-B1 and V-B2 will make use of two rules extracted from Hoeffding Trees generated by the ORB approach as case studies. Figure 1 shows the periods of time associated to the creation and evolution of these rules. Note that the number of test examples is different from the number of training examples in each quarter. This is because verification latency causes test examples to become available for training only at a later date. Equation 1 depicts the rule investigated in Fig. 1-a) which is part of a Hoeffding Decision Tree generated for the Camel dataset:

$$\begin{aligned} &EXP \leq 677.31 \wedge ND > 1.18 \wedge NDEV \leq 10.27 \wedge \\ &REXP \leq 960.1 \implies DEFECT - INDUCING \end{aligned} \quad (1)$$

We note the following behavior for this rule:

1) Blue region (quarters 1 to almost the end of 24th): the parent of this rule accumulated data gathered from around 24 quarters to finally create it. Throughout this period, a total of (70 clean, 42 defect-inducing) changes were used to create this rule (sum of values in brackets corresponding to training examples in the blue region of the plot). Despite the smaller number of defect-inducing changes used to create this rule, ORB's oversampling mechanism assigned the defect-inducing class to this rule as a strategy to deal with class imbalance.
2) Green region (end of quarter 24 to quarter 25): after its creation, this rule was further reinforced by being trained on (2 clean, 5 defect-inducing) training examples. Note

that, in quarter 25, it was used to predict only 5 examples (3 clean, 2 defect-inducing), i.e., 60% error rate.

3) White region, quarter 26: in order to conduct our analysis of the presence of concept drift, we stopped training this rule from this quarter onward, so that it remained unaltered during the performance drop observed during quarter 27. This enables us to check whether a rule that was created in the past remains suitable over time.
4) Red region, quarter 27: this rule was exceptionally activated to predict 200 software changes (167 clean, 33 defect-inducing), yielding a false alarm rate of 83.5%.

Equation 2 presents the rule generated in the period depicted in Fig. 1-b) which is part of a Hoeffding Tree generated by ORB for the Tomcat dataset:

$$NDEV > 3.1 \wedge LD \leq 2679.4 \wedge ND > 1.82 \wedge LA \leq 2062 \wedge \\ NDEV \leq 8.91 \implies DEFECT-INDUCING \tag{2}$$

We note the following behavior for this rule:

1) Blue region (almost whole quarter 1): the rule was created based on (1 clean, 12 defect-inducing) examples in a time interval smaller than one quarter.
2) Green region (small part from the end of quarter 1 to 23): after its creation, this rule was triggered to predict only one example (the clean test example in the 2nd quarter) and then remained without being used for predictions for 21 quarters (around 5 years, from quarters 3 to 23). However, in this period, this rule remained being reinforced by the presentation of (2 clean, 105 defect-inducing) training examples. The significantly higher number of software changes used for training but not for testing is explained by the verification latency. In this case, the software changes corresponding to the defect-inducing training examples were committed before time step 4000, when the underlying defect generating process was still generating software changes that matched this rule.
3) White region, quarters 24 to 27: for the same reason as the training for the Camel dataset was stopped during quarter 26, the Tomcat training was stopped from quarters 24 to 27.
4) Red region, quarter 28: despite this rule having remained unused for prediction for 21 quarters, it suddenly started being activated again in quarter 28 by an exceptionally high number of changes (95 clean, 16 defect-inducing), yielding to a false alarm rate of 85.6%.

We can see that these rules have been trained on a large number of examples (119 for Camel and 120 for Tomcat, as summarized in Table II). Therefore, they are unlikely to have been generated simply as a result of noise. For Tomcat, the training examples strongly support the defect-inducing class. For Camel, the defect-inducing training examples used to create the Camel rule consisted of approximately 40% of the total number of examples used to train this rule. Given that the overall percentage of defect-inducing examples in Camel is 20%, it is likely that this rule should indeed support the defect-inducing class. Therefore, it is likely that these rules do represent part of the true defect generating process that was active during the time period corresponding to when the software changes corresponding to these training examples were committed.

We can also see that, in both cases and especially for Tomcat, these defect-inducing rules faced a period of time when they were not triggered very often by test examples (green and white regions), and later on started to be triggered very often by clean software changes (red regions), leading to a high rate of false alarms. The fact that these rules became unsuitable over time is related to concept drift in $p(y|\vec{x})$, and is further discussed in Section V-B1. The fact that these rules are triggered often by test examples during some periods but not during others is related to concept drifts in $p(\vec{x})$, and is discussed in Section V-B2.

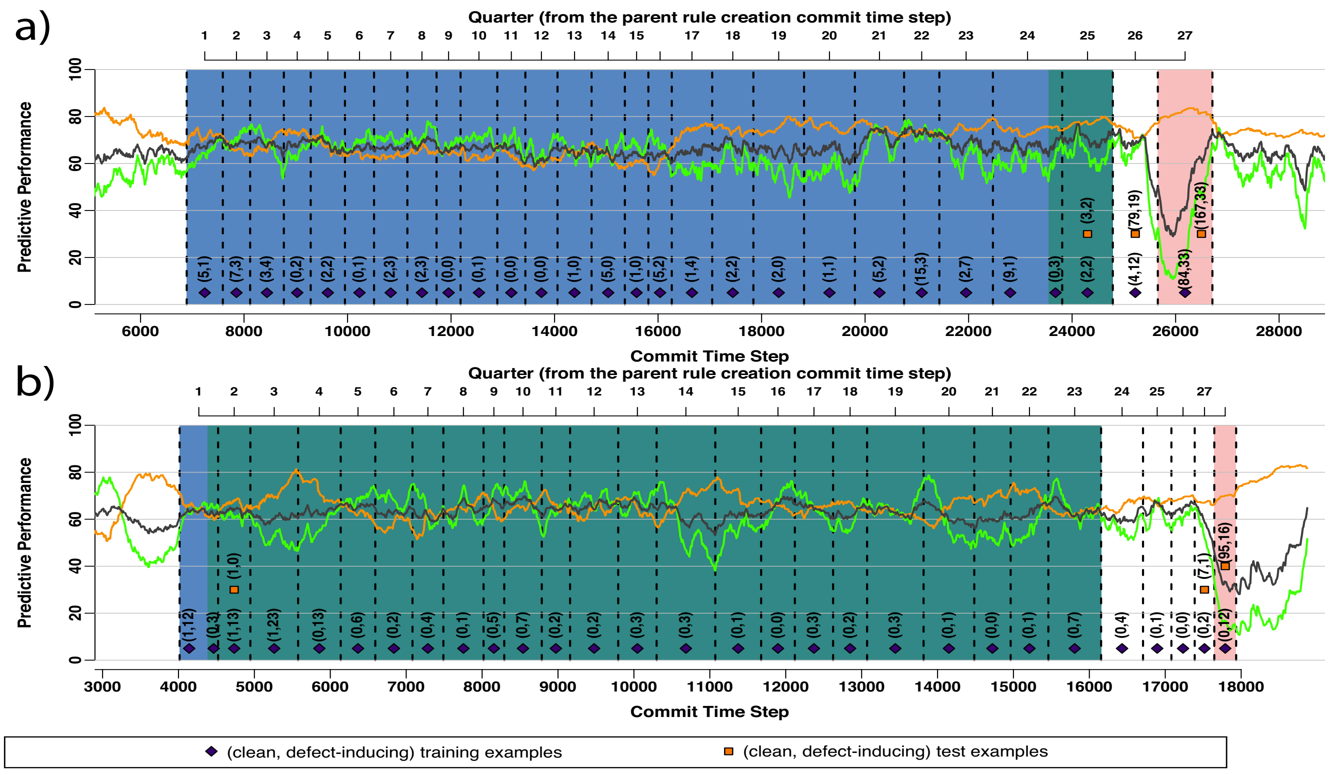
1) $p(\vec{x}|y)$ Concept Drift Affecting $p(y|\vec{x})$: A concept drift of type $p(y|\vec{x})$ means that a class previously assigned to a specific region of the feature space may have changed. This type of concept drift cannot be proved in real world problems since the true underlying data distribution is unavailable. However, certain statistics obtained from the data produced by the distribution can strongly suggest the presence of this type of concept drift. We will use such information to support our analysis in this section.

As explained in the beginning of Section V-B, both the defect-inducing rules presented in Equations 1 and 2 were supported by a large number of training examples, as shown in Table II. Nevertheless, later on these rules started being triggered very often by clean test examples and played an important role in the severe drops in g-mean observed in the 27th quarter for Camel and 28th quarter for Tomcat (red regions in Fig. 1). The overall test accuracy during these periods is shown in Table II. Therefore, even though these rules were initially suitable, they became unsuitable.

It is worth noting that overfitting can also potentially lead to poor predictive performance on test examples, but it is unlikely that the test predictive performance would suddenly and drastically drop as observed in the last quarters of Figure 1 as a result of overfitting. Therefore, the above mentioned cases present areas of the feature space genuinely associated to the defect-inducing class in the past, that became representative of the clean class. This strongly suggests that there was a drift affecting $p(y|\vec{x})$.

None of the existing JIT-SDP methods has strategies to tackle this type of concept drift satisfactorily, including the online learning methods. In particular, OOB, UOB and ORB do not have any strategy to delete obsolete rules from the classifier. Therefore, they depend on a very large number of training examples representing the new concept to arrive before adaptation is successfully completed, yielding such big drops in predictive performance as those shown in Figure 1, which can compromise the reliability of JIT-SDP classifiers. Despite having the ability to remove old rules, Sliding Window methods [6] have already been shown to be inadequate as each classifier is trained on a limited number of changes, hindering predictive performance [8]. Therefore, none of the existing JIT-SDP methods can adequately deal with projects affected by concept drifts in $p(y|\vec{x})$.

2) $p(\vec{x}|y)$ Concept Drift Affecting $p(\vec{x})$: Despite having been created based on a good number of training examples, the



Blue regions (time interval producing training examples used to create the rule), green regions (time interval where the rule is used both for prediction and for further training) and red regions (time interval the rule is used only for predictions) corresponding to the rules depicted in Equations 1 and 2, for Camel and Tomcat, respectively. Vertical dashed line intervals consist of a period of a quarter of a year. The green, orange and black lines represent ORB [8]'s $rec(0)$, $rec(1)$ and g-mean, respectively. Values in brackets (clean,defective) provide the total number of software changes satisfying the rule in a quarter according to their labeling and commit times for the training and testing examples, respectively.

Fig. 1: Predictive performance of the JIT-SDP classifier and training / test information corresponding to the rules presented in Equations 1 and 2 belonging to the classifier, for Camel (a) and Tomcat (b), respectively.

TABLE II: Examples used for training/testing two different rules before and after the large performance drop depicted in the red region of Figure 1. Numbers in brackets represent the number of used clean and defect-inducing changes.

rule	#training examples for creation	#training examples for reinforcement	total #training examples	test accuracy$^{(1)}$	test accuracy$^{(2)}$
a - Camel	(70|42)	(2|5)	(72|47)	40% (3|2)	16.5% (167|33)
b - Tomcat	(1|12)	(2|105)	(3|117)	0% (1|0)	14.4% (95|16)

The rules were created at time steps 23749 and 4498, for camel and tomcat, respectively.
(1) a - camel, from commit time step 23750 to 24787 (0.28 years); and b - tomcat, from commit time step 4499 to 16155 (5.46 years), corresponding to the green region in Fig. 1.
(2) a - camel, from commit time step 25656 to 26711 (0.25 years); and b - tomcat, from commit time step 17646 to 17934 (0.25 years), corresponding to the red region in Fig. 1.

rules presented in Equations 1 and 2 and discussed in Section V-B1 were not triggered very often by test examples right after their creation (green regions in Fig. 1). In total, they were triggered $3+2=5$ times for Camel (over a 0.28 years period - 25th quarter) and $1+0=1$ times for Tomcat (over a 5.46 years period - 2nd quarter). Hence, even though they became poor rules right after their creation (see test accuracy$^{(1)}$ in Table II), this did not lead to an overall poor predictive performance of the classifier since they were not triggered very often by test examples.

However, as discussed in the beginning of Section V-B, these rules suddenly started being triggered very often by test examples during later periods (red regions, and white region for camel, in Fig. 1). In total, they were triggered $167+33=200$ times for Camel and $95+16=111$ times for Tomcat (over 0.3 years for both cases). As feature values that were previously not commonly observed now became more common, this means that there was a concept drift affecting $p(\vec{x})$. In particular, we can observe at least two sudden drifts in $p(\vec{x})$ associated to these periods of time in Figure 2 (red dashed lines). As the rules in Equations 1 and 2 became unsuitable due to the concept drift affecting $p(y|\vec{x})$ (as explained in Section V-B1), the fact that the concept drift in $p(\vec{x})$ caused this rule to be triggered more often by test examples contributed to the large performance drops observed in Figures 1-a) and 1-b).

This problem inevitably affects any method that does not

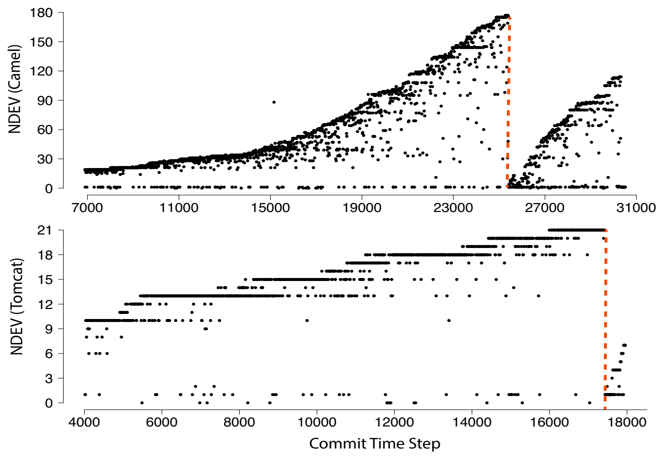


Fig. 2: NDEV feature values during periods discussed for the rules in Equations 1 and 2.

have strategies to eliminate or deactivate old rules, such as ORB, OOB and UOB. Depending on the window size, the sliding window method recommended by McIntosh and Kamei [6] would be unaffected by old rules, as it would eliminate them. However, due to verification latency, it would still take time to learn new rules corresponding to the new region of the feature space where software changes started to appear. This is because training examples corresponding to these software changes arrive with a delay. Therefore, this method would still take time to recover from concept drifts in $p(\vec{x})$.

Besides activating rules that became unsuitable due to drifts affecting $p(y|\vec{x})$, sudden concept drifts affecting $p(\vec{x})$ may also activate rules that have been generated as a result of noise or rules that over-generalize to areas of the space that were not seen at training time, potentially leading to further performance drops.

In addition to sudden drifts, there are also many gradual drifts in $p(\vec{x})$. For instance, it is reasonable that as software matures, the experience of the developers (EXP), the number of developers who have changed the modified files in the past (NDEV) and the number of prior changes to the modified files (NUC) increase. As shown in Figure 3, gradual increases/decreases of the feature values can be observed over time for several features of the analyzed datasets. This type of concept drift thus seems to be quite common in JIT-SDP, potentially more common than drifts in $p(y|\vec{x})$.

A side effect of this type of concept drift is that newly created rules can become obsolete very quickly, since they represent areas of the input space that are not visited anymore in the near future. For instance, Figure 1-a) shows that the rule from Equation 1 became obsolete soon after it was created. In particular, this rule was used only once right after its creation and remained unused for 5.46 years (Table II).

Such gradual feature evolution emphasizes the importance of adopting online learning methods that update classifiers as soon as labeled examples become available, given that the prediction task often requires new rules to describe new areas of the feature space. It also highlights the challenges posed by verification latency, as training examples may already be unrepresentative of the feature values of current software

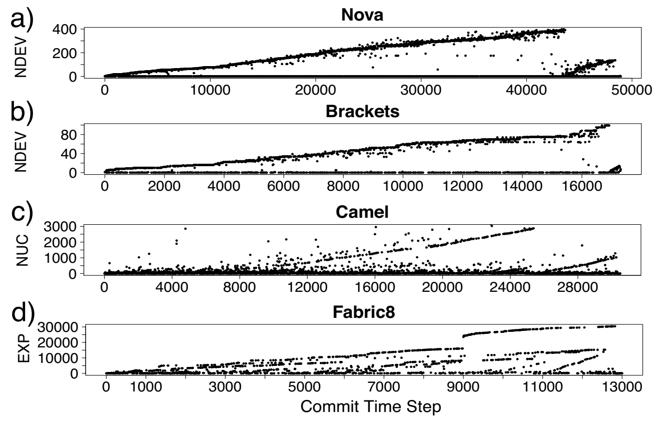


Fig. 3: Example of evolution through time for some features values and datasets.

changes by the time they are generated. For instance, in Tomcat, the green region of Fig. 1 shows the presence of many training examples but almost no test example. This means that, when those training examples became available, the regions of the feature space that they covered were not much activated by test examples anymore.

C. *Predictive Performance Reliability*

As discussed in Sections V-B1 and V-B2, the various types of concept drift, associated to the weaknesses of resampling strategies, are linked to large performance drops and, consequently, large predictive performance variations over time. Therefore, we expect that existing JIT-SDP methods will suffer from lack of reliability. They may not perform consistently well over time, i.e., their predictive performances may not be high and stable. This section thus analyses the reliability of OOB, UOB, OOB-SW and ORB.

Previous work has already compared the average predictive performance of OOB, UOB, OOB-SW and ORB [8]. In particular, it identified that ORB outperformed the other approaches in terms of g-mean and $|rec(0)-rec(1)|$ based on Scott-Knott-A12. In the current paper, we expand that analysis to discuss reliability, including the stability of predictive performance over time. The average predictive performance obtained in Cabral et al. [8]'s work is reproduced in Table III of this paper. The results of our additional statistical test corroborate those from Cabral et al. [8], confirming that ORB was better ranked than OOB, UOB and OOB-SW in terms of g-mean and $|rec(0) - rec(1)|$, and that even though OOB and UOB achieved top average $rec(1)$ this was at the cost of a low average $rec(0)$.

Achieving a good result in terms of $|rec(0) - rec(1)|$ is particularly important so that (1) improvements in the ability to identify defect-inducing changes are not at the cost of a large number of false alarms, which would reduce practitioners' trust in the method, and (2) a good $rec(0)$ does not come at the cost of missing a large number of defect-inducing changes, which would render the method useless. However, we hereby note that even ORB still obtained a $|rec(0) - rec(1)|$ larger than 20 in half of the datasets (JGroups, Brackets, Spring Integration, Nova and NPM). This means that, while one of the

TABLE III: Performance results for literature methods.

Dataset	Classifier	rec(0)	rec(1)	|rec(0)-rec(1)|	g-mean
Fabric	OOB	50.24 [-b]	74.45 [b]	28.50 [-b]	59.04 [-b]
	UOB	42.28 [-b]	83.70 [b]	43.91 [-b]	57.07 [-b]
	OOB-SW	73.32 [b]	46.36 [-b]	47.73 [-b]	50.74 [-b]
	ORB	60.35	68.36	20.59	60.93
Jgroups	OOB	59.38 [-b]	56.67 [-b]	28.13 [-b]	54.71 [-b]
	UOB	73.78 [b]	45.12 [-b]	36.81 [-b]	55.09 [-b]
	OOB-SW	81.50 [b]	35.33 [-b]	57.51 [-b]	47.95 [-b]
	ORB	62.65	56.73	17.79	57.76
Camel	OOB	57.06 [-b]	73.99 [b]	25.47 [-b]	62.90 [-b]
	UOB	55.57 [-b]	71.28 [s]	29.27 [-b]	60.35 [-b]
	OOB-SW	71.67 [b]	40.38 [-b]	64.61 [-b]	40.29 [-b]
	ORB	60.74	70.41	17.03	63.63
Tomcat	OOB	59.82 [*]	61.75 [-b]	29.42 [-b]	57.28 [-b]
	UOB	68.48 [b]	50.04 [-b]	33.69 [-b]	55.18 [-b]
	OOB-SW	65.20 [b]	52.30 [-b]	35.93 [-b]	54.53 [-b]
	ORB	59.43	64.37	16.08	60.18
Brackets	OOB	49.11 [-b]	89.49 [b]	41.90 [-b]	63.94 [m]
	UOB	54.59 [-b]	83.10 [b]	32.98 [b]	64.24 [b]
	OOB-SW	56.29 [-b]	79.82 [b]	42.80 [-b]	61.68 [-b]
	ORB	61.68	77.15	36.01	63.66
Neutron	OOB	69.71 [-b]	91.89 [b]	23.97 [-b]	79.32 [-b]
	UOB	58.83 [-b]	92.45 [b]	38.41 [-b]	70.73 [-b]
	OOB-SW	73.83 [-b]	83.08 [b]	20.49 [-b]	76.74 [-b]
	ORB	79.89	81.12	13.98	79.93
Spring Integration	OOB	62.48 [-b]	53.74 [b]	47.28 [-b]	48.12 [-b]
	UOB	55.65 [-b]	59.31 [b]	37.58 [b]	52.19 [*]
	OOB-SW	45.55 [-b]	79.88 [b]	39.52 [-b]	56.12 [b]
	ORB	74.33	44.31	37.30	52.20
Broadleaf	OOB	59.25 [-b]	68.33 [m]	33.40 [-b]	60.07 [-b]
	UOB	59.32 [-b]	62.69 [-b]	43.26 [-b]	55.46 [-b]
	OOB-SW	78.21 [b]	34.73 [-b]	71.02 [-b]	37.00 [-b]
	ORB	61.60	67.00	19.17	61.97
Nova	OOB	68.54 [-b]	86.27 [b]	24.34 [-b]	75.41 [-b]
	UOB	65.56 [-b]	90.84 [b]	27.60 [-b]	75.94 [m]
	OOB-SW	66.41 [-b]	85.90 [b]	33.66 [-b]	72.85 [-b]
	ORB	75.44	79.78	20.28	75.57
NPM	OOB	37.92 [-b]	74.89 [b]	49.68 [-b]	46.17 [-b]
	UOB	38.27 [-b]	72.83 [b]	48.90 [-b]	45.87 [-b]
	OOB-SW	55.56 [s]	62.75 [-b]	43.68 [-b]	50.53 [-b]
	ORB	55.25	63.95	31.74	54.26
Ranking	OOB	2	1	2	2
	UOB	2	1	3	2
	OOB-SW	1	3	4	3
	ORB	1	2	1	1

Table adapted from [8]. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and big A12 effect size against ORB. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than ORB. The groups' rankings with smaller numbers indicate better ranks according to Scott-Knott-A12 test [26].

TABLE IV: Standard deviations through time for all classifiers.

Dataset	Classifier	rec(0)	rec(1)	|rec(0)-rec(1)|	g-mean
Fabric	OOB	16.79 [-s]	11.60 [b]	20.33 [b]	11.96 [b]
	UOB	16.41 [*]	10.29 [b]	19.63 [b]	12.51 [b]
	OOB-SW	25.34 [-b]	27.14 [-b]	30.46 [-b]	16.53 [-m]
	ORB	15.31	18.35	22.75	16.08
Jgroups	OOB	22.24 [-b]	15.26 [-b]	20.22 [-b]	11.82 [-b]
	UOB	18.32 [-b]	14.61 [-b]	19.58 [-b]	10.61 [s]
	OOB-SW	22.58 [-b]	22.09 [-b]	25.58 [-b]	12.77 [-b]
	ORB	14.09	13.36	17.36	10.71
Camel	OOB	18.84 [-b]	11.48 [b]	21.15 [-b]	10.21 [b]
	UOB	19.98 [-b]	13.58 [-b]	20.75 [-b]	9.55 [b]
	OOB-SW	31.06 [-b]	33.81 [-b]	29.26 [-b]	17.28 [-b]
	ORB	14.58	11.87	19.45	10.80
Tomcat	OOB	20.07 [-b]	18.02 [-b]	21.49 [-b]	10.72 [-b]
	UOB	19.83 [-b]	16.89 [-b]	21.35 [-b]	9.77 [s]
	OOB-SW	23.91 [-b]	17.99 [-b]	21.71 [-b]	9.28 [b]
	ORB	15.48	10.00	18.04	9.66
Brackets	OOB	14.80 [b]	9.47 [b]	18.01 [b]	15.37 [b]
	UOB	18.09 [-b]	13.69 [b]	23.98 [-s]	15.55 [b]
	OOB-SW	22.77 [-b]	22.88 [b]	24.54 [-b]	16.43 [b]
	ORB	16.25	25.29	23.81	18.57
Neutron	OOB	12.00 [-b]	8.65 [b]	13.20 [-b]	9.66 [-b]
	UOB	21.76 [-b]	12.32 [m]	21.25 [-b]	18.64 [-b]
	OOB-SW	15.22 [-b]	13.78 [-b]	17.60 [-b]	11.57 [-b]
	ORB	7.42	12.51	11.34	6.14
Spring Integration	OOB	30.72 [-b]	28.71 [-b]	30.97 [-b]	19.21 [-b]
	UOB	25.14 [-b]	20.39 [b]	21.78 [b]	12.94 [b]
	OOB-SW	24.10 [-b]	14.69 [b]	29.34 [-m]	16.29 [b]
	ORB	18.74	21.39	28.91	17.57
Broadleaf	OOB	23.67 [-b]	16.05 [-b]	20.08 [-b]	11.41 [b]
	UOB	25.06 [-b]	23.03 [-b]	20.04 [-m]	10.18 [b]
	OOB-SW	28.78 [-b]	35.66 [-b]	29.43 [-b]	19.65 [-b]
	ORB	15.01	14.49	18.74	12.64
Nova	OOB	16.20 [-b]	12.17 [b]	16.47 [b]	12.54 [b]
	UOB	14.84 [b]	9.33 [b]	15.66 [b]	12.64 [b]
	OOB-SW	17.93 [-b]	19.49 [-b]	17.83 [s]	14.58 [-b]
	ORB	15.13	15.57	17.93	13.96
NPM	OOB	27.86 [-b]	18.41 [b]	29.56 [-b]	16.67 [-b]
	UOB	28.60 [-b]	19.00 [b]	28.69 [-b]	16.36 [-b]
	OOB-SW	32.84 [-b]	21.57 [-b]	29.90 [-b]	18.15 [-b]
	ORB	22.87	19.98	26.96	14.80
Ranking	OOB	2	1	1	1
	UOB	2	1	1	1
	OOB-SW	3	3	2	2
	ORB	1	2	1	1

Symbols [*], [s], [m] and [b] represent insignificant, small, medium and big A12 effect size against ORB. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than ORB. The groups' rankings with smaller numbers indicate better ranks according to Scott-Knott-A12 test [26].

recalls on these datasets may have been typically very good, the other was more than 20 units worse, which is undesirable in practice, affecting ORB's reliability. Other methods with higher $|rec(0) - rec(1)|$ would be even more unreliable.

Next, we analyze the stability of the predictive performance over time. Aggregating the performance metrics across time through averaging can overlook considerable periods of time with poor performance. Therefore, it is important to check how the predictive performance of JIT-SDP methods varies through time.

Hence, when designing a classifier, a visual analysis of the performance through time is important to understand the reliability of a JIT-SDP method.

Table IV shows the standard deviations of the predictive performances through time. The standard deviations can roughly summarize a method's stability. As shown by Scott-Knott-A12, ORB's ranking in terms of standard deviation of g-mean and $|rec(0) - rec(1)|$ were not better than that of OOB and UOB, despite being better than that of OOB-SW. Therefore, despite achieving a better ranking in terms of the average g-mean and $|rec(0) - rec(1)|$, ORB did not really lead to ranking improvements over OOB and UOB in terms of the stability of such metrics.

The analysis above shows that the better ranked average g-mean and $|rec(0) - rec(1)|$ results achieved by ORB compared with OOB, UOB and OOB-SW do not mean that its difference in recalls was good enough through time. There were variations of the performance metrics over time that caused ORB to present significant drops in predictive performance over time. Such lack of stability in the predictive performance over time means that, at any given point in time, a classifier may be performing very well or failing dramatically, hindering their adoption in practice. Therefore, existing JIT-SDP methods still need to be improved.

Considering Table III, ORB obtained the best overall performance (i.e., based on the Scott-Knott-A12 analysis for metrics gmean and $|rec(0) - rec(1)|$). However, in order to have a better understanding on how these results can indicate whether or not these classifiers may be adopted in practice, it is important to analyze the results from Table III together with the variation of the $|rec(0) - rec(1)|$ through time in Fig. 4. This is because a top rank in terms of $|rec(0) - rec(1)|$

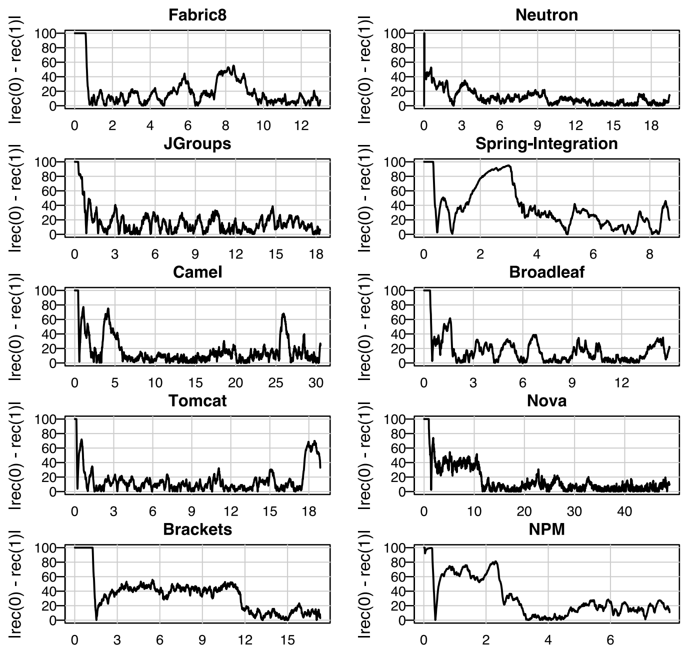


Fig. 4: Absolute difference between the recalls ($|rec(0) - rec(1)|$) trough time for ORB. The x axis corresponds to the commit timestep $\times 10^{-3}$.

does not mean that such $|rec(0) - rec(1)|$ is good enough. In particular, aggregating the performance metrics across time into a single metric value can hide considerable periods of time with poor performance. Fig. 4 shows many periods where the classifier is not reliable due to high values in $|rec(0)-rec(1)|$. Hypothetically, a $|rec(0)-rec(1)|$ higher than 30 may indicate that the classifier has an accuracy of 80% for clean commits and 50% for defect-inducing ones. In Fig. 4 the timestep periods Fabric8 from ± 7500 to ± 11000, Tomcat from ± 17800 to ± 19000, Camel from ± 1500 to ± 12000 and Nova from ± 1000 to ± 12000 are some of the representative examples of periods where the classifier is unreliable.

RQ1: Section V-B reveals that JIT-SDP suffers from various types of concept drift. Due to verification latency, most types of concept drift cannot be diagnosed in time for an effective reaction, leading to variations and drops in predictive performance over time, which negatively affect the reliability of JIT-SDP methods. Section V-C confirms that reliability issues are present in all datasets investigated in this study.

VI. (RQ2) Improving the Reliability of Online JIT-SDP Classifiers

The findings from RQ1 (Section V) show that new methods to improve JIT-SDP's reliability are desirable, in view of issues caused by concept drift and verification latency. The current section answers RQ2 by building on the knowledge obtained through RQ1 to propose a new JIT-SDP method aimed at improving reliability. Section VI-A briefly explains how the findings from RQ1 led us to the design of an approach able to detect and react earlier to concept drifts in JIT-SDP. Section VI-B explains the proposed approach, called Prediction-Based Sampling Adjustment (PBSA), in detail.

A. Leveraging the Findings from RQ1 to Improve JIT-SDP

Section V thoroughly discusses the different types of concept drifts affecting JIT-SDP and makes explicit the weaknesses of the existing methods regarding these concept drifts. Among the reasons for their poor performance is the fact that they monitor concept drifts relying on the true classes labels, but, due to verification latency, that is is not a suitable strategy for JIT-SDP. Therefore, we need a timely strategy to detect these drifts.

In this work, all observed changes in $p(y|\vec{x})$ turned defect-inducing areas in the hyperspace into clean areas and occurred at the same time as changes in $p(\vec{x})$ that caused a high number of software changes fall in these areas abruptly. This situation leads to a shift in the predictions made by the JIT-SDP classifier. Specifically, a much higher proportion of software changes may start being predicted as defect-inducing when such concept drifts occur. Such shift involves only the predictions made by the JIT-SDP classifier. Detecting it would not require us to wait for the arrival of the true class labels corresponding to these software changes. Therefore, keeping track of the proportions of predictions of each class is a potential way to detect this type of concept drift in JIT-SDP.

Our observations from RQ1 show that changes in $p(\vec{x})$ are likely to be very common in JIT-SDP. It is possible that they occur together with 3 different situations:

- An area in the input space may be associated to an incorrect label as a consequence of an overgeneralization of the classifier. If this area suddenly becomes very populated due to a change in $p(\vec{x})$, predictions based on such incorrect label are likely to cause a shift in the predictions made by the JIT-SDP classifier, causing it to either predict the clean or the defect-inducing class more often than expected. Such concept drift could thus also be detected by tracking the proportions of predictions of each class.
- An area in the input space may be associated to an incorrect label as consequence of a rule created from noise, leading to the same previously mentioned situation.
- An area in the input space may have been previously labeled as consequence of noise or overgeneralization, but this label may actually be correct. A change in $p(\vec{x})$ causing this area to become very populated would not lead to a shift in the predictions made by the JIT-SDP classifier. Monitoring concept drifts by tracking the proportions of predictions of each class would thus be unlikely to detect such change in $p(\vec{x})$. However, that is not a problem, because this change is not detrimental to the predictive performance of the JIT-SDP classifier.

Therefore, our proposed approach will track the predictions given by the classifier in order to achieve earlier detection and reaction to concept drift in JIT-SDP.

B. Prediction-Based Sampling Adjustment (PBSA)

Given that JIT-SDP is a binary classification problem, a plausible strategy to track the proportion of predictions of each class is to monitor the moving average of predictions:

$$ma_{\hat{y}} = \frac{\sum_{i=t-ws+1}^{t} \hat{y}_i}{ws}. \quad (3)$$

where ws is the size of the sliding window, t is the current commit time step, and $\hat{y}_i$ is the prediction of the classifier to the software change produced at commit time step i (0 for the clean class and 1 for the defect-inducing class).

As JIT-SDP is a class imbalanced problem, monitoring $ma_{\hat{y}}$ means monitoring the class imbalance ratio of the *predictions* provided by the classifier. This monitoring can be performed in real time (without delay) so that a significant deviation from an expected classifier's behaviour (in this case, an expected class imbalance ratio in the predictions) might provide an earlier indication of the need for adjusting the classifier.

Given that JIT-SDP is a class imbalanced problem, our approach is based on the following assumption:

Assumption 1: Given that JIT-SDP is a class imbalanced problem, the moving average of the predictions ($ma_{\hat{y}}$) must be skewed towards the majority class (i.e., the clean class).

PBSA manipulates the training procedure to maintain $ma_{\hat{y}}$ within a given acceptable/expected interval around a target value th for the moving average. If $ma_{\hat{y}}$ moves outside this interval, this indicates a suspected concept drift, requiring adjustments to avoid drops in predictive performance. Such adjustments are made through a concept drift recovery mechanism that is explained later in this section.

The target value th is set by taking Assumption 1 into account. In particular, given that the clean class (0) is a majority, th should be smaller than 0.5. In a perfect scenario, th should match the true class imbalance ratio exactly. However, this is unrealistic given that there will always be some errors in the predictions. Therefore, th should be set closer to 0.5 than the true imbalance ratio, so that we allow an accepted level of error in the majority class for the sake of increasing $rec(1)$. For example, given a true class imbalance ratio of 3:7, setting a target $ma_{\hat{y}}$ to 40% implies in accepting an error of at least 10% in the majority class for the sake of increasing $rec(1)$.

The interval around th is set based on a parameter p which represents the percentage of allowed deviation from th. The lower boundary of the interval is set to th_1 = th - (th * p) and the upper boundary to th_0 = th + ((1 - th) * p). Figure 5 shows an example of $ma_{\hat{y}}$ over time for the Tomcat dataset, where $th = 0.4$ and $p = 0.2$, resulting in lower and upper boundaries of 0.32 and 0.52, respectively.

To detect meaningful deviations from the interval, let $\overrightarrow{ma}_{\hat{y}}$ be a vector containing the last ws moving averages of the predictions $ma_{\hat{y}}$. PBSA only triggers the concept drift recovery mechanism if $avg(\overrightarrow{ma}_{\hat{y}})$ is placed outside the boundaries of the interval.

The **concept drift recovery mechanism** consists in performing a recovery training with recent examples from the "opposite" class from the biased class. In particular:

- If the skew is towards predicting the clean class, the recovery training is performed with examples from the defect-inducing class sampled from a defect-inducing pool containing all training examples labeled as defect-inducing seen so far, sorted by their training time steps.

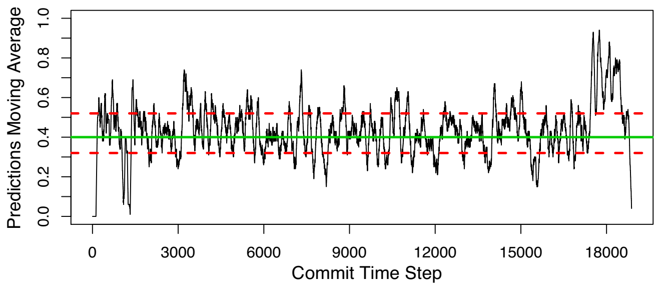


Fig. 5: Moving average of the predictions (ws = 100) for the Tomcat dataset. Green line represents target moving average (th) and red dashed lines represent boundaries of the acceptable interval.

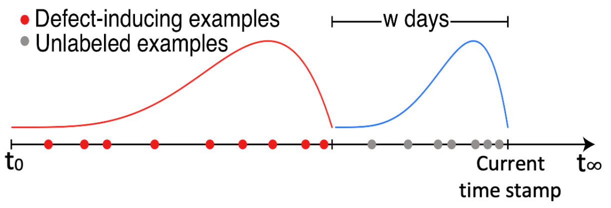


Fig. 6: PDF curves for picking an unlabeled example (blue curve) and for picking a defect-inducing example (red curve) for retraining the classifier.

- If the skew is towards predicting the defect-inducing class, the recovery training is performed with examples sampled from an unlabeled pool, by (pseudo-)labeling them as clean training examples. The unlabeled pool stores all committed software changes that are currently unlabeled. It is worth noting that only software changes produced up to w (waiting time) days ago are unlabeled (see Definition 5 in Section II). Clean examples older than w days are not used as part of the recovery training because (1) the model has already been trained on plenty of examples of the clean class older than w days (as this is overall a majority class) and (2) the most recent examples may be representing a new concept still not learned well enough by the tree, such that training on them may improve predictive performance on the current concept. Later on, once the real label of such examples is revealed, they will be used again for training.

In the second case, as the clean class is a majority, it is more likely that the examples sampled from the unlabelled pool belong to the clean class. Therefore, the number of mislabelled examples will typically be smaller than the number of correctly labelled examples. Such mislabelling is thus likely to be a smaller problem when compared to the benefits of training with very recent software changes[3], which can help to speed up adaptation to concept drifts. This is a key difference between the proposed method and existing ones such as ORB [8], OOB and UOB [17], which need to rely solely on labelled examples affected by verification latency for training. Besides that, since

[3]These changes will be much more recent than labelled changes due to verification latency.

existing methods such as ORB, OOB and UOB only resample the last training example rather than sampling from a pool, they are also more sensitive to noise. This is because if this last training example is noisy, replicating it several times will hinder predictive performance.

Examples are repeatedly sampled from the corresponding pool based on a probability density function and used for recovery training until the average of the predictions for the ws most recently committed software changes becomes larger than the target moving average value th in the first case, and smaller than th in the second case.

The probability density function was designed to prioritize more recent examples, as defined by Equation 4 and illustrated in Figure 6. This is important since prioritizing recent examples helps to recover from concept drifts. The probability density function associated to the defect-inducing changes has a longer tail, because training examples of this class arrive less frequently, i.e., this class is typically a minority class in JIT-SDP.

$$f(x, \alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)} \tag{4}$$

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{5}$$

Equation 4 is the probability density function of a Beta distribution. It returns the relative likelihood of sampling an example at location x of the pool, where x are the positions of the examples in the pool, but scaled between 0 and 1. The parameters α and β define the shape of the distribution and Γ (Eq. 5) stands for the Gamma function. In practice, an example at a given position x is sampled from the pool based on a random number r, respecting the distribution defined by Eq. 4, if x is the closest position to r among all examples in the pool.

After being triggered, PBSA prevents the concept drift recovering mechanism from being triggered again for the next at commit time steps. This is because, despite the use of strategies to speed up recovery from concept drift, such recovery will take some time to be reflected by the moving average $\vec{ma}_y$. This could cause PBSA to unnecessarily trigger the recovery mechanism again for dealing with the same concept drift, resulting in overfitting.

Algorithm 1 depicts PBSA. As input, it receives: th - a target moving average of the predictions; p - the acceptable percentage of deviation of $avg(\overrightarrow{ma_{\hat{y}}})$ from th; w - the waiting time for receiving the true labels (in days); ws - the window size for computing the moving averages; and at - number of commit time steps before the concept drift recovery mechanism can be triggered again. $Defs$ and $Unlabs$ are the pools of defect-inducing and unlabeled software changes, respectively. $Labs$ is an array of labeled examples. New training examples are inserted into $Labs$ when (i) a software change is found to be defect-inducing or (ii) when an unlabeled change is labeled as clean as a result of the waiting time w (Line 12). The examples in $Labs$ are used for training in Line 13, based on any existing online base learning algorithm. In other words, the JIT-SDP model is continuously updated at each iteration of the loop using any new labeled training examples that may have arrived since the previous iteration. Once the examples in $Labs$ are used for training, they are discarded (if they belong to the clean class) or moved to $Defs$ (if they belong to the defect-inducing class). The array $\overrightarrow{ma_{\hat{y}}}$ contains the last ws values of $ma_{\hat{y}}$. If the condition in Lines 15 or 24 is true, the concept drift recovery mechanism is triggered and will then be ignored for at commit time steps as aforementioned. Note that in lines 19 and 28, the classifier is trained without using any resampling strategy, different from line 13, when it may inherit the resampling strategy from the base learner.

In essence, PBSA monitors the classifiers' outputs in order to react to concept drift. It can be considered as a special resampling strategy to react to concept drifts in the presence of class imbalance and verification latency.

Algorithm 1: PBSA

```
   input : $th$,$p$,$ws$,$at$
1  $th_1$ = $th$ - ($th$ * p)
2  $th_0$ = $th$ + ((1 - $th$) * p)
3  $Defs$ = ∅
4  $Unlabs$ = ∅
5  $Labs$ = ∅
6  $\overrightarrow{ma_{\hat{y}}}$ = ∅
7  $lastRecovTS = 0$
8  while receiving new software changes - $\vec{c}_i$ do
9      $\hat{y} = predict(\vec{c}_i)$
10     $update(\overrightarrow{ma_{\hat{y}}}, \hat{y})$
11     $update(Unlabs, \vec{c}_i)$
12     $update(Labs)$
13     $trainOn(Labs)$
14     $update(Defs)$
15     if $avg(\overrightarrow{ma_{\hat{y}}}) > th_0$ & $lastRecovTS < i - at$ then
16         while $ma_{\hat{y}} > th$ do
17             $r$ = $B(\alpha, \beta).randSample()$
18             $idx$ = $r \times size(Unlabs) + 1$
19             $trainNoSampling(Unlabs[idx])$
20             $update(ma_{\hat{y}})$
21         end
22         $lastRecovTS = i$
23     end
24     if $avg(\overrightarrow{ma_{\hat{y}}}) < th_1$ & $lastRecovTS < i - at$ then
25         while $ma_{\hat{y}} < th$ do
26             $r$ = $B(\alpha, \beta).randSample()$
27             $idx$ = $r \times size(Defs) + 1$
28             $trainNoSampling(Defs[idx])$
29             $update(ma_{\hat{y}})$
30         end
31         $lastRecovTS = i$
32     end
33 end
```

C. Experimental Setup

*1) **Performance Metrics:*** The objective of the experiments is to evaluate PBSA's reliability, which is defined in Section

II. It requires an evaluation of the magnitude of the predictive performance and its stability over time.

As in Section V-A and in previous work [8], the performance metrics used for the comparison are the recalls on the clean ($rec(0)$) and defect-inducing ($rec(1)$) classes, the g-mean ($\sqrt{rec(0)} \times \sqrt{rec(1)}$) and the difference between the recalls ($|rec(0) - rec(1)|$), calculated in a prequential way and with a fading factor $\theta = 0.99$ [8], [10], [17], [11]. The Scott-Knott-A12 test and A12 effect sizes with respect to ORB are also used to support the analysis.

In addition, the number of months Ξ_l where the difference in recalls $|rec(0) - rec(1)|$ surpasses a threshold l will also be analyzed.

*2) **Parameters Choice:*** The proposed method can be used with any online base learner. For this work, given its top performance when compared with existing methods [8], ORB was adopted as base learner for PBSA. ORB's parameters were assigned as in previous work [8]. In addition to ORB's parameters, the proposed method incorporates four parameters, th, p, ws and at, as shown in Algorithm 1. The tested values for these parameters were th = {0.2, **0.3**, 0.4, 0.5, 0.6} and p = {0.15, **0.25**, 0.35}. Values in bold represent the values used in the experiments. These values were chosen based on a grid search carried out on a smaller number of executions (5 repetitions) for each dataset up to the commit time step 5000. A single unique set of parameter values was chosen based on the values that most often led to top g-mean across all datasets. This is because, in practice, the best parameter values may change over time and it is impossible to know which values would lead to the best results before observing the whole data stream of labeled examples [4]. Since they were not shown to be crucial parameters based on a preliminary analysis, $ws = 100$ and $at = 30$ were used. Equation 5 also relies on parameters, nevertheless, these parameters were also not crucial, and thus fixed to $\alpha = 5$ and $\beta = 2$. Based on the chosen parameters, 30 runs were performed for PBSA on each full data stream.

*3) **Open Science and Reproducibility:*** The code implementing PBSA, the scripts used to run the experiments, the datasets used in the experiments and csv files with the results of all runs are available at http://doi.org/10.5281/zenodo.6548768.

D. Analysis of the Average Predictive Performances

Cabral and Minku [8] showed that ORB overcame state-of-art methods in terms of g-mean and $|rec(0) - rec(1)|$. Those methods were also further analyzed and discussed in Section V. Therefore, this section will focus on comparing the proposed method against ORB.

Table V shows the predictive performance results for ORB and PBSA. Figure 3 in the supplementary material contains plots to facilitate visualization of the g-mean and $|rec(0) - rec(1)|$ obtained by these approaches. Table 1 in the supplementary material also contains a comparison that includes the other approaches OOB, UOB and OOB-SW. The analysis shows that the proposed method obtained a larger number of significantly better results in all metrics but $rec(1)$. The difference between the recalls, $|rec(0) - rec(1)|$, is a particularly important performance metric, as explained in Section V. Table V shows that PBSA reduced the overall $|rec(0) - rec(1)|$ by a maximum of 47% and a minimum of 2.7% in comparison with ORB. However, for Neutron, PBSA performed worse in terms of this metric. PBSA operates by maintaining the detection rate close to a pre-defined value (i.e., th). Specifically for Neutron, further experiments shown that a smaller value for th, such as 0.2, produces classifiers with much better and stabler performances.

TABLE V: Performance results for ORB and PBSA methods.

Dataset	Classifier	rec(0)	rec(1)	|rec(0) - rec(1)|	g-mean
Fabric8	ORB	60.35	68.36	20.59	60.93
	PBSA	66.37[b]	61.42[-b]	14.46[b]	61.20[s]
Jgroups	ORB	62.65	56.73	17.79	57.76
	PBSA	65.64[b]	52.94[-b]	16.32[b]	57.68[-b]
Camel	ORB	60.74	70.41	17.03	63.63
	PBSA	68.60[b]	66.99[-b]	11.67[b]	66.72[b]
Tomcat	ORB	59.43	64.37	16.08	60.18
	PBSA	66.33[b]	58.04[-b]	14.62[b]	61.19[b]
Brackets	ORB	61.68	77.15	36.01	63.66
	PBSA	65.56[b]	74.25[-b]	24.93[b]	65.49[b]
Neutron	ORB	79.89	81.12	13.98	79.93
	PBSA	74.96[-b]	86.44[b]	19.53[-b]	79.88[-b]
Spring Integration	ORB	74.33	44.31	37.30	52.20
	PBSA	75.61[b]	43.58[-b]	36.31[b]	52.16[-b]
Broadleaf	ORB	61.60	67.00	19.17	61.97
	PBSA	66.48[b]	62.55[-b]	12.41[b]	62.65[b]
Nova	ORB	75.44	79.78	20.28	75.57
	PBSA	75.15[-b]	82.93[-b]	13.63[b]	77.72[b]
NPM	ORB	55.25	63.95	31.74	54.26
	PBSA	61.83[b]	56.11[-b]	16.57[b]	56.87[b]
Wins	ORB	2	**8**	1	3
	PBSA	**8**	2	**9**	**6**
p-values		2.2e-16	2.2e-16	2.2e-16	2.2e-16

Table V also shows that PBSA predominantly obtained a better $rec(0)$ than ORB (by a maximum of 13% and a minimum of -0.4%, except for the Neutron dataset), in detriment of $rec(1)$. Still, the magnitude of the improvement in $rec(0)$ is mostly larger than the respective deterioration in $rec(1)$. It is worth mentioning that the ORB's worse performance in terms of $rec(0)$ means a high rate of false alarms. This may add a significant, and unnecessary, workload to the test team, besides reducing practitioners' trust in the method. For instance, software developers at Facebook prefer a conservative lower bound on fix detection [27] when using the tools Infer [28] and Sapienz [29]. In a scenario where the cost of analyzing the source code is high, the recommendation is to prioritize $rec(0)$. Other authors [30], [31] also emphasize the problem of wasting effort on a large number false alarms. For safety critical applications, $rec(1)$ may be prioritized. Table 1 in the supplementary material expands the comparisons of the proposed method against methods present in Section V.

Figure 1 discussed in Section V shows two large performance drops in datasets Camel and Tomcat. As discussed in that section, these drops were caused by, among others, concept drifts in $p(y|x)$. Table VI presents the performance of ORB and PBSA for these specific time intervals (commit time step 25500 to 27000 for Camel and 17400 to 18878 for Tomcat). In these periods, PBSA enhanced ORB's g-mean by

[4]Note that, after the whole data stream of labeled examples is observed, predictions to its software changes would not be necessary anymore. Therefore, in practice, it would be unreasonable to choose the best parameter values only after observing the whole data stream.

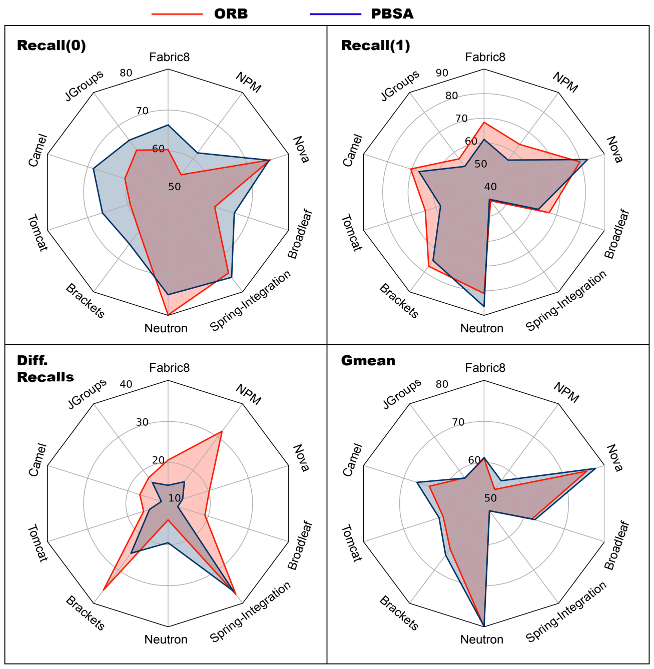


Fig. 7: Performance comparison for the average results of each metrics for ORB and PBSA classifiers.

11% and 27% for Camel and Tomcat, respectively. Regarding $|rec(0) - rec(1)|$, the achieved reductions were 32% and 52% over ORB, respectively. This substantial improvement may be explained by the concept drift recovering mechanism present in PBSA.

TABLE VI: PBSA and ORB predictive performances during the large drops in Camel and Tomcat datasets depicted in Figure 1 around timesteps 26000 and 18000, respectively.

Dataset	Classifier	rec(0)	rec(1)	|rec(0) - rec(1)|	gmean
Camel	ORB	40.48 (2.61)	**80.52** (1.95)	40.34 (3.54)	55.00 (2.18)
	PBSA	**49.46** (2.39)	76.78 (1.38)	**27.32** (3.17)	**61.28** (1.49)
Tomcat	ORB	25.20 (2.35)	**76.92** (2.39)	51.82 (3.96)	42.03 (1.91)
	PBSA	**43.62** (2.73)	66.94 (2.13)	**24.93** (3.69)	**53.41** (1.60)

Standard deviations are shown in brackets. Values in bold are statistically better according the paired Wilcoxon Signed-rank test.

RQ2.1: The proposed PBSA achieved top ranked average predictive performance in terms of all metrics but $rec(1)$ compared to the state of the art. Improvements were larger especially in terms of $|rec(0) - rec(1)|$, where PBSA's performance was from 2.7% (minimum) to 47.8% (maximum) better, except for Neutron. PBSA outperformed ORB in terms of $rec(0)$ by a maximum of 13% and, in the worst case, was outperformed by 0.4% (except for Neutron), demonstrating that PBSA throws less false alarms than ORB. Improvements were particularly large during periods affected by concept drifts in $p(y|\vec{x})$.

E. Analysis of the Stability of the Predictive Performance

This section will further examine (i) the standard deviation of the predictive performance through time and (ii) periods of large class imbalance in the predictions Ξs.

*1) **Classifiers' Stability**:* Table VII presents the standard deviations through time of the performance metrics for all methods. According to paired Wincoxon Signed-rank test, PBSA obtained a much larger number of better results than ORB for all metrics. PBSA overcame ORB in terms of $rec(0)$ in 9 out of 10 results with a maximum of 42%, in terms of $rec(1)$ in 8 out of 10 results with a maximum of 34%, in terms of $|rec(0) - rec(1)|$ in 9 out of 10 results with a maximum of 36% and in terms of g-mean in 7 out of 10 results with a maximum of 22%. The A12 effect size of the difference in standard deviations was almost always large. The aforementioned improvement in stability can be visually assessed in Fig. 8. For almost all metrics and datasets, PBSA obtained higher stability (i.e., smaller standard deviations). For the datasets Neutron and Spring-Integration, ORB slightly outperformed PBSA. In Table 2 in the supplementary material, the PBSA stability is further compared to the methods in literature.

TABLE VII: Performance stability for ORB and PBSA methods.

Dataset	Classifier	rec(0)	rec(1)	|rec(0) - rec(1)|	g-mean
Fabric8	ORB	15.31	18.35	22.75	16.08
	PBSA	11.30[b]	16.89[b]	20.52[b]	15.32[b]
Jgroups	ORB	14.09	13.36	17.36	10.71
	PBSA	11.45[b]	11.28[b]	16.22[b]	9.66[b]
Camel	ORB	14.58	11.87	19.45	10.80
	PBSA	8.86[b]	11.39[b]	13.57[b]	9.42[b]
Tomcat	ORB	15.48	10.00	18.04	9.66
	PBSA	10.98[b]	8.93[b]	12.37[b]	7.4[b]9
Brackets	ORB	16.25	25.29	23.81	18.57
	PBSA	11.27[b]	22.35[b]	22.77[b]	18.98[-b]
Neutron	ORB	7.42	12.51	11.34	6.14
	PBSA	8.58[-b]	13.07[*]	9.91[b]	6.81[-b]
Spring Integration	ORB	18.74	21.39	28.91	17.57
	PBSA	17.38[b]	23.34[-b]	32.12[-b]	18.33[-b]
Broadleaf	ORB	15.01	14.49	18.74	12.64
	PBSA	10.27[b]	12.52[b]	16.09[b]	11.64[b]
Nova	ORB	15.13	15.57	17.93	13.96
	PBSA	9.15[b]	15.16[b]	14.50[b]	13.46[b]
NPM	ORB	22.87	19.98	26.96	14.80
	PBSA	13.28[b]	13.20[b]	17.20[b]	11.88[b]
Wins	ORB	1	1	1	3
	PBSA	**9**	**8**	**9**	**7**
p-values		2.2e-16	2.2e-16	2.2e-16	2.2e-16

*2) **Large Class Imbalance in the Predictions Periods** (Ξ):* According to Table V, for some datasets such as Camel, Tomcat, Nova and NPM, when comparing PBSA to ORB, it is possible to notice considerable improvements on the performance stability considering $|rec(0) - rec(1)|$. Moreover, as presented in Fig. 4, many Ξs present in ORB (e.g., in Fabric8 (commit time step 4.7k to 6.5k and 6.5k to 10k), in Camel (commit time step 3k to 6k and 25.5k to 29k) and in NPM (commit time step 0.4k to 3.4k)) were totally or partially eliminated by PBSA. These results further illustrate the improvements in stability obtained by PBSA, which had a positive effect on the standard deviations presented in Table VII. For Brackets, PBSA obtained a long Ξ, but $|rec(0) - rec(1)|$ was considerably decreased in comparison to ORB. Nevertheless, new Ξs were introduced by PBSA, however, these new Ξs tended to be less accentuated (i.e., smaller l) than the ones present in ORB.

Figure 9 presents the average and standard deviation of the amount of time (in months) comprised in Ξ_l with l longer than month periods in x axis. It illustrates how detrimental

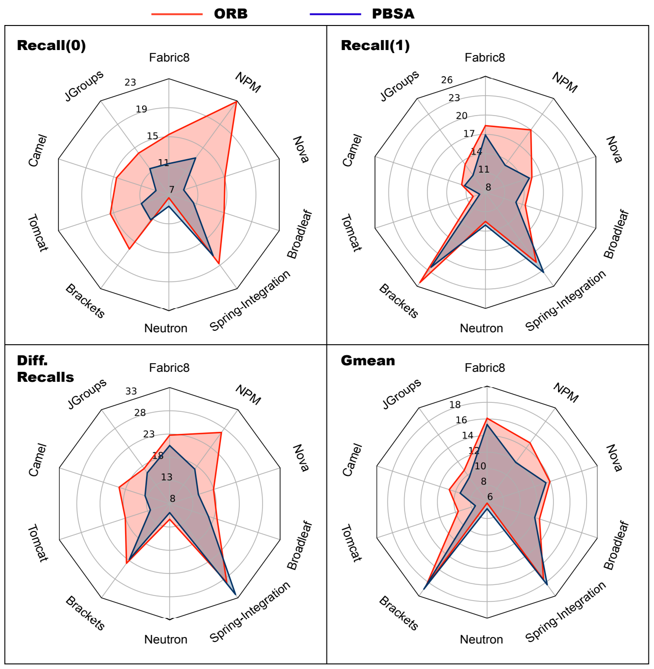


Fig. 8: Stability comparison for the average standard deviations through time of each metrics for ORB and PBSA classifiers.

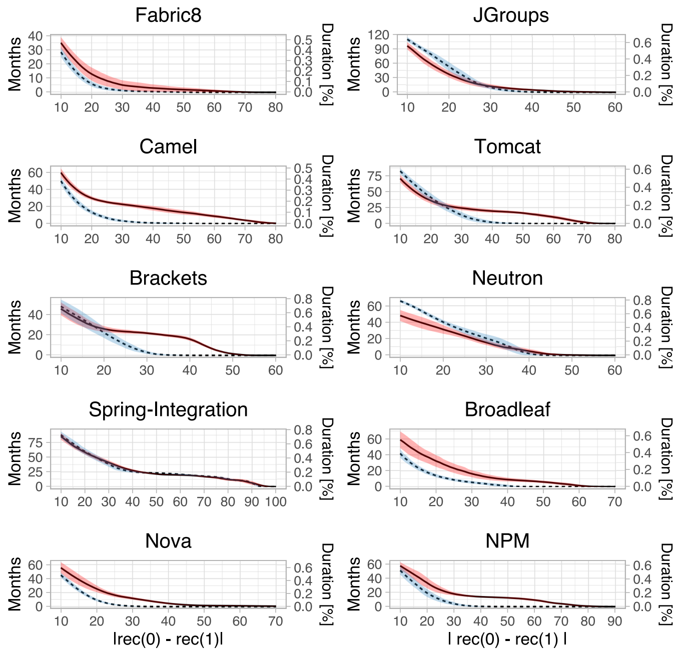


Fig. 9: Average number of months comprised in Ξ_l with l in the x axis. Blue area corresponds to the PBSA whereas the red one corresponds to ORB.

increases in $|rec(0) - rec(1)|$ resulting from poor stability can be in PBSA and ORB. In addition, the secondary y axis shows the percentage of the project duration corresponding to the number of months shown in the main y axis. For example, for Fabric8, ORB spent an average of around 35 months in Ξ_l with l larger than 10, which corresponds to almost 50% of the project duration. The larger the number of months (or percentage), the longer the Ξs are. Longer Ξs associated to larger magnitudes l are particularly detrimental.

According to Fig. 9, the PBSA performed better than ORB for minimizing Ξs, in 7 out of 10 datasets. ORB overcame PBSA in two datasets (JGroups and Neutron). For the Spring-integration dataset, both methods performed poorly.

RQ2.2: Given the class imbalance and concept drift challenges faced by JIT-SDP, creating a classifier able to keep a stable performance throughout the whole project is, so far, virtually impossible. Nevertheless, based on a realistic analysis, the proposed PBSA was able to produce a more stable performance in comparison to the state of the art (e.g., up to 42%, 34%, 36% and 22% stabler than ORB in terms of the standard deviation of $rec(0)$, $rec(1)$, $|rec(0)-rec(1)|$ and g-mean through time, respectively). This better stability associated to the top average predictive performance results shown in RQ2.1 render PBSA the most reliable JIT-SDP classifier to date.

VII. Treats to Validity

Internal Validity: as with any real world data stream problem, it is impossible to prove that concept drift really did occur in JIT-SDP, since we have no access to the true underlying probability distribution. To mitigate this threat, we have collected strong evidence of the different types of concept drifts based on statistics collected from the data and information on the models generated from such data. There is a non-zero chance that rules such as those in Eq. 1 and 2 became inadequate as a result of overfitting instead of concept drift. Nevertheless, as explained in Section V-B, there is a good number of training examples supporting these rules, mitigating this threat. Another threat is the sensitivity of machine learning methods to their parameters. To mitigate this problem, a grid search based on the first 5,000 examples of each dataset was performed. In addition, preliminary experiments have shown that the proposed method is not too sensitive to the parameters α and β.

Construct Validity: The rules presented in Section V-B have been used as proxies for the posterior probability distribution $p(y|\vec{x})$ associated to a portion of the input space. We have also used the values of the input features of the software changes as a proxy for the distribution $p(\vec{x})$. There may be inaccuracies resulting from noise. The predictive performance was assessed by the recalls for each class and their geometric mean, which are insensitive to class imbalance, and by the difference between the recalls, which enables us to assess how well the methods are dealing with class imbalance. These metrics were computed using fading factors as suggested by Gama et al. [12], enabling us to track changes in predictive performance over time. In addition, the stability of the classifier was measured by the overall standard deviation for each metric and experiment. *External Validity*: This work was conducted on ten open source GitHub projects. As with other machine learning studies, the results may not generalize to other projects.

VIII. Conclusions and Future Work

JIT-SDP has shown to be a problem affected by all several types of concept drift. In the projects analyzed in this study, gradual drifts in $p(x)$ and $p(x|y)$ usually happened along the whole project while drifts in $p(y|x)$ abruptly appeared in occasions where the projects were already mature. These concept drifts have shown to be very detrimental to the predictive performance, particularly, due to their combination with the verification latency problem. These issues suggest the use of online methods able to learn labeled training examples as soon as they become available. Furthermore, the analysis also suggests that strategies able to employ unlabeled data to train the classifier may be necessary for improving predictive performance through time. Making these issues explicit contributes to better cope with the development of new JIT-SDP methods.

Based on these observations, we proposed a new method called PBSA that makes use of information about the predictions given to unlabeled software changes in order to trigger adaptation to suspected concept drifts. This approach was able to obtain a more stable and high (i.e., reliable) JIT-SDP classifier. Improvements in reliability are crucial for the adoption of JIT-SDP classifiers in practice.

Future work includes the proposal of online methods to automatically tune parameters of JIT-SDP approaches over time, experiments with other projects (including proprietary projects) and base learners, and investigation of whether feature normalisation strategies could help dealing with concept drift by addressing feature trends observed over time.

References

[1] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *QRS*, 2015, pp. 17–26.

[2] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effort-aware just-in-time software defect prediction," *IST*, vol. 93, pp. 1 – 13, 2018.

[3] X. Yang, H. Yu, G. Fan, K. Shi, and L. Chen, "Local versus global models for just-in-time software defect prediction," *Scientific Programming*, vol. 2019, pp. 1–13, 06 2019.

[4] L. Qiao and Y. Wang, "Effort-aware and just-in-time defect prediction with neural network," *PLOS ONE*, vol. 14, p. e0211359, 02 2019.

[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 TSE*, vol. 39, no. 6, pp. 757–773, 2013.

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

[7] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *ICSE*, 2015, pp. 99–108.

[8] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *ICSE*, 2019, pp. 666–676.

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

[10] S. Tabassum, L. Minku, D. Feng, G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *ICSE*, 2020, pp. 1–1.

[11] S. Wang, L. L. Minku, and X. Yao, "A systematic study of online class imbalance learning with concept drift," *IEEE TNNLS*, 2018.

[12] J. Gama, I. Zliobaite, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Computing Surveys*, vol. 46, no. 4, 2014.

[13] G. Catolino, D. Di Nucci, and F. Ferrucci, "Cross-project just-in-time bug prediction for mobile apps: An empirical assessment," in *MOBILESoft*, 2019, p. 99–110.

[14] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *CSMR*, 2010, pp. 107–116.

[15] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies, "Class level fault prediction using software clustering," in *ASE*, 2013, pp. 640–645.

[16] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE TSE*, vol. 39, no. 6, pp. 822–834, 2013.

[17] S. Wang, L. L. Minku, and X. Yao, "Resampling-based ensemble methods for online class imbalance learning," *IEEE TKDE*, vol. 27, no. 5, pp. 1356–1368, 2015.

[18] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *FSE*, 2015, pp. 966–969.

[19] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *ACM SIGKDD*, 2001, pp. 97–106.

[20] "The impact of class imbalance in classification performance metrics based on the binary confusion matrix," *Pattern Recognition*, vol. 91, pp. 216–231, 2019.

[21] "Facing imbalanced data recommendations for the use of performance metrics," in *ACII*, 2013.

[22] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1253–1269, 2018.

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

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

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

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

[27] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *SCAM*, 2018, pp. 1–23.

[28] "A tool to detect bugs in java and c/c++/objective-c code before it ships," https://fbinfer.com/, accessed: 2020-04-16.

[29] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with sapienz at facebook," 08 2018.

[30] D. Bowes, S. Counsell, T. Hall, J. Petric, and T. Shippey, "Getting defect prediction into industrial practice: the elff tool," in *ISSREW*, 2017, pp. 44–47.

[31] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 372–381.

George G. Cabral received his PhD degree from the Federal University of Pernambuco (Brazil) in 2014. He conducted his postdoc at the University of Birmingham (UK) in 2019. Currently, he is an adjunct professor at the Department of Computing at the Federal Rural University of Pernambuco (Brazil). He serves as reviewer for reputable journals such as Neurocomputing and Applied Software Computing. He has systematically served as committee member in prestigious conferences such as ICSE and ASE. His research interests include Novelty Detection, One-class Classification, Artificial Neural Networks, Data Mining, Class Imbalance, Software Defect Prediction, Concept Drift, Online Learning, etc.

Leandro L. Minku is an Associate Professor at the School of Computer Science, University of Birmingham (UK). Prior to that, he was a Lecturer in Computer Science at the University of Leicester (UK), and a Research Fellow at the University of Birmingham (UK). He received the PhD degree in Computer Science from the University of Birmingham (UK) in 2010. Dr. Minku's main research interests are machine learning for software engineering, machine learning for non-stationary environments / data stream mining, class imbalanced learning, ensembles of learning machines and search-based software engineering. Among other roles, Dr. Minku is Associate Editor-in-Chief for Neurocomputing, Senior Editor for IEEE TNNLS, and Associate Editor for EMSE and JSS.