A Practical Human Labeling Method for Online Just-in-Time Software Defect Prediction

Liyan Song
songly@sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Leandro Lei Minku*
l.l.minku@bham.ac.uk
The University of Birmingham
Birmingham, UK

Cong Teng
12132358@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Xin Yao*
xiny@sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

ABSTRACT

Just-in-Time Software Defect Prediction (JIT-SDP) can be seen as an online learning problem where additional software changes produced over time may be labeled and used to create training examples. These training examples form a data stream that can be used to update JIT-SDP models in an attempt to avoid models becoming obsolete and poorly performing. However, labeling procedures adopted in existing online JIT-SDP studies implicitly assume that practitioners would not inspect software changes upon a defect-inducing prediction, delaying the production of training examples. This is inconsistent with a real-world scenario where practitioners would adopt JIT-SDP models and inspect certain software changes predicted as defect-inducing to check whether they really induce defects. Such inspection means that some software changes would be labeled much earlier than assumed in existing work, potentially leading to different JIT-SDP models and performance results. This paper aims at formulating a more practical human labeling procedure that takes into account the adoption of JIT-SDP models during the software development process. It then analyses whether and to what extent it would impact the predictive performance of JIT-SDP models. We also propose a new method to target the labeling of software changes with the aim of saving human inspection effort. Experiments based on 14 GitHub projects revealed that adopting a more realistic labeling procedure led to significantly higher predictive performance than when delaying the labeling process, meaning that existing work may have been underestimating the performance of JIT-SDP. In addition, our proposed method to target the labeling process was able to reduce human effort while maintaining predictive performance by recommending practitioners to inspect software changes that are more likely to induce defects. We encourage the adoption of more realistic human labeling methods in research studies to obtain an evaluation of JIT-SDP predictive performance that is closer to reality.

CCS CONCEPTS

• Software and its engineering → Risk management; **Software defect analysis**; **• Computing methodologies → Online learning settings**; **Classification and regression trees**; **Bagging**.

KEYWORDS

Just-in-time software defect prediction, online learning, verification latency, waiting time, human labeling, human inspection

ACM Reference Format:
Liyan Song, Leandro Lei Minku, Cong Teng, and Xin Yao. 2023. A Practical Human Labeling Method for Online Just-in-Time Software Defect Prediction . In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616307

1 INTRODUCTION

Just-in-Time Software Defect Prediction (JIT-SDP) is a type of SDP specialized at the code change level with the aim of predicting whether or not a software change is defect-inducing at commit time (just-in-time) [19, 20]. It is of practical relevance as a decision supporting tool for improving software quality by automatically alerting developers of potential defects at a very early stage, as soon as a software change is produced, and at a relatively fine granularity compared to module-based SDP. As such, it has been attracting increasing interest from both academia [53] and industry [29, 38, 47]. From the machine learning viewpoint, JIT-SDP can be seen as a binary classification problem for which models are constructed based on training examples labeled as *defect-inducing* (class 1) or *clean* (class 0) that can then be used to predict whether or not new software changes would induce defects.

In practice, training examples corresponding to software changes arrive sequentially in order over time and thus JIT-SDP should be taken as an online learning task, where JIT-SDP models are updated with new incoming training examples [4, 27, 44]. Existing literature has revealed that updating JIT-SDP models with such training examples (so that the models can capture the latest data generation status) can lead to better predictive performance compared to models built

*Liyan Song, Cong Teng, and Xin Yao (Corresponding Author) are with Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China.
Leandro Lei Minku (Corresponding Author) is with School of Computer Science, the University of Birmingham, Edgbaston, Birmingham, UK

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616307

with obsolete training examples [27]. Updating JIT-SDP models over time is particularly important given the existence of *concept drift*, which are changes taking place in the code defect generation process. Concept drift has been shown to occur in JIT-SDP [3, 4] and may significantly deteriorate predictive performance of JIT-SDP models if they are not updated over time [27].

Dealing with concept drift in JIT-SDP is nevertheless particularly challenging as the actual labels of software changes only become available long after their commit time, an issue referred to as *verification latency* in machine learning [9, 11, 24, 42]. This can delay the model updates, potentially affecting the ability of such models to react to concept drift. In particular, a training example can only be labeled as a clean software change after enough time has passed from its commit time for one to be confident on its clean status, or as a defect-inducing training example when a defect is found to be associated to this change, whichever is shorter [4]. Such labeling process is referred to as the *waiting time* method and has been adopted in recent online JIT-SDP studies [3, 4, 41].

However, such labeling process implicitly assumes that practitioners would not inspect software changes upon a defect-inducing prediction. This is inconsistent with a real-world scenario where developers adopt JIT-SDP models during the software development process. Specifically, when a JIT-SDP model predicts a new software change to be defect-inducing, the developer who has just completed the change may inspect the change to check whether or not it really induces any defect. Ignoring such inspection means assuming that practitioners would always completely ignore the warnings given by the JIT-SDP model and wait until defects are found much later, when the defect is more difficult to be fixed. In other words, it means completely ignoring the main point of using JIT-SDP models in practice, which is to inspect software changes predicted as defect-inducing at (or close to) commit time, in an attempt to eliminate their defects when the code change is still fresh in the developers' minds. This may in turn negatively impact the evaluation of predictive performance of JIT-SDP models, as it results in a delay in the labeling process that is larger than the delay that would likely occur in practice, potentially preventing JIT-SDP models from reacting to concept drift in a more timely manner.

Therefore, a more realistic labeling procedure that takes into account the adoption of JIT-SDP models during the software development process and the resulting human inspection of software changes predicted as defect-inducing should also be taken as an important part of the JIT-SDP process. This paper aims at formulating such labeling procedure and investigating whether and to what extent it would impact the predictive performance of JIT-SDP models. We refer to this labeling procedure as Immediate **Hum**an **La**beling for Software Changes Predicted as Defect-Inducing (HumLa). Together with the delayed waiting time method, it forms a labeling process that is closer to the reality that would be adopted in practice when JIT-SDP models are used during the software development process. It is noteworthy that the term "immediate" in our paper does not mean that the developers has to give up all other work that they had planned in their schedule to immediately inspect the software change. What we mean is that the developer will label the change as part of the process of inspecting this change at an early stage (which is inherent from the adoption of JIT-SDP), and this is a more immediate labeling than the waiting time procedure. Moreover, the process of inspecting a change predicted as defect-inducing is the same as the labeling process. In other words, if a developer opts for inspecting a change predicted as defect inducing as part of the adoption of JIT-SDP in their project/organization, labeling this software change does not increase this effort further. That said, inspecting all software changes that are predicted as defect-inducing as part of the adoption of JIT-SDP could result in high inspection/labeling effort. So, we also propose a human labeling method to save human labeling costs by helping practitioners to target their inspection/labeling effort towards specific defect-predicted changes, called **E**ffort-**Co**nservative **Hum**an **La**beling (ECo-Humla).

Our study answers the following Research Questions (RQs):

- RQ1 How to formulate a JIT-SDP labeling method with immediate human labeling of changes predicted as defect-inducing?
  - RQ1.1 How does this labeling method affect the JIT-SDP predictive performance compared to the delayed waiting time method for JIT-SDP?
  - RQ1.2 To what extent the quality of human labels impacts the predictive performance when using this method?
  - RQ1.3 How does the amount of human effort affect the predictive performance when using this method?
- RQ2 How can we target the labeling process towards specific software changes to reduce the amount of required human inspection effort?
  - RQ2.1 How does this labeling method affect the JIT-SDP predictive performance?
  - RQ2.2 How much human effort can this method save?
  - RQ2.3 How helpful is this method in encouraging defects to be found when saving effort?

The main contributions of this paper are listed below:

- We are the first to formulate the immediate human labeling method (HumLa) into JIT-SDP, being closer to the reality that would be adopted in practice (RQ1).
- We show based on experiments with 14 datasets that this practical labeling method can significantly benefit predictive performance of JIT-SDP when the human label quality and quantity are above a given threshold (RQ1.1-1.3). Studies that do not take such labeling process into account may thus be underestimating the predictive performance of JIT-SDP models that would be obtained in practice.
- We propose an effort-conservative human labeling method (ECo-HumLa) to prioritize the most confidently defect-predicted software changes for practitioners to inspect (RQ2).
- We show experimentally that ECo-HumLa can substantially reduce human effort by around 50% while maintaining JIT-SDP predictive performance (RQ2.1-2.2). ECo-HumLa encourages a higher number of defects to be found through inspection than a baseline effort reduction method (RQ2.3).

2 RELATED WORK

2.1 JIT-SDP

Early studies usually modeled JIT-SDP as an offline learning task, assuming that all training examples are available beforehand and no further adjustment or evaluation of the JIT-SDP models would happen over time. A landmark study is that of Kamei et al., who summarized 14 features extracted from commits and bug reports for

JIT-SDP model construction and showed them to be good indicators for yielding high predictive performance in JIT-SDP [19]. Many later studies were conducted based on these features [17, 28]. Various learning machines have been investigated for (JIT-)SDP, among which tree-based methods were shown to have potential in yielding good performance [13, 17, 18, 25, 45, 50]. Techniques such as oversampling and undersampling [33] have also been adopted to deal with the class imbalance issue usually suffered by JIT-SDP, where the defect-inducing (clean) class is typically the minority (majority).

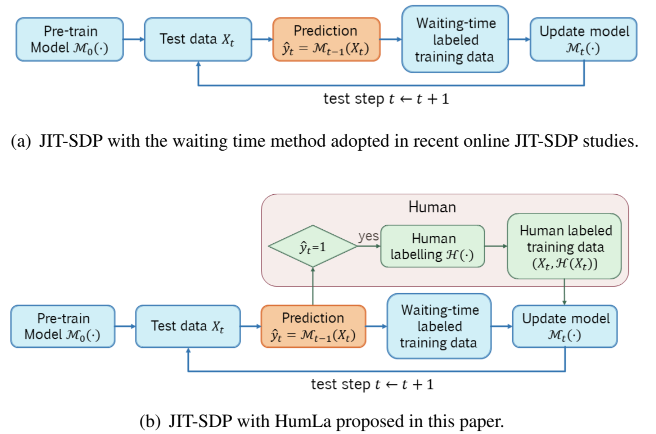
However, studies on offline JIT-SDP disregard the chronology of software changes that arrive sequentially over time in practice. Tan et al. showed that overlooking chronology can result in deceptively higher predictive performance than the prediction models could achieve in practice, posing serious threats to validity in JIT-SDP studies [44]. Later studies [3, 4, 27] further found that predictive performance of JIT-SDP models can deteriorate over time as a result of concept drift. Therefore, JIT-SDP approaches being able to learn over time and reduce drops in predictive performance that are potentially caused by concept drift have been proposed [3, 4, 43, 44].

2.2 Chronology-Preserving Labeling Procedures

When taking chronology into account in JIT-SDP, one also needs to consider the issue of verification latency when labeling training examples, as explained in Section 1. In particular, Tan et al. [44] adopted a labeling procedure where a full batch of software changes is labeled after a pre-defined amount of time passes, so that there is an increased chance that defects associated to software changes would have been found to produce defect-inducing training examples. However, this procedure does not consider that defect-inducing training examples could be produced as soon as defects are found to be associated to them, which potentially happens before the end of this pre-defined period of time. Therefore, this labeling procedure can unnecessarily delay the training process of JIT-SDP models, potentially slowing down reaction to concept drift.

Cabral et al. [4] proposed a method that overcomes this issue. It makes use of a pre-defined parameter called *waiting time* that defines how long one would wait after a software change is committed to label it as clean. If no defect is found within this waiting time, a clean training example is produced at the end of the waiting time. If a defect is found to be associated to this software change within the waiting time, a defect-inducing training example is produced at the moment when this is found. If a defect is found after the waiting time for a change that had been previously labeled as clean, a new defect-inducing training example is produced for this change.

Such waiting time strategy has been used as a labeling method for an online JIT-SDP learning procedure as illustrated in Figure 1(a). Consider an initial JIT-SDP model $\mathcal{M}_0(\cdot)$ that has been produced with existing data. When a new software change X_t is committed at test time step t, where $X_t \in \mathbb{R}^d$ denotes a d-dimensional feature vector representing the software change, the latest model $\mathcal{M}_{t-1}(\cdot)$ is adopted to predict whether X_t would be *defect-inducing* (class 1) or *clean* (class 0), formulated as $\widehat{y}_t = \mathcal{M}_{t-1}(X_t)$. Once that is done, new training examples are produced based on the waiting time method until a new software change arrives to be predicted. Such training examples are used to produce an updated JIT-SDP model $\mathcal{M}_t(\cdot)$. When the new change arrives to be predicted, the procedure



(a) JIT-SDP with the waiting time method adopted in recent online JIT-SDP studies.

(b) JIT-SDP with HumLa proposed in this paper.

Figure 1: JIT-SDP with different labeling methods.

is repeated for test time step $t \longleftarrow t+1$. This iterative test-then-train process based on the waiting time method continues throughout the process of JIT-SDP. Several other studies have adopted this procedure with the waiting time method [3, 40, 41].

However, all previous studies have overlooked the fact that practitioners would be inspecting software changes predicted as defect-inducing at commit time when JIT-SDP is adopted in practice, to check if they are really associated to a defect. Even though a waiting time is necessary for labeling software changes predicted as clean by a JIT-SDP model, changes predicted as defect-inducing would likely be labeled much earlier. Therefore, this more immediate human labeling procedure should be taken as an important part of the JIT-SDP process to formulate a labeling procedure that is closer to the reality that would be adopted in practice.

3 IMMEDIATE HUMAN LABELING IN JIT-SDP

3.1 HumLa

Figure 1(b) illustrates the procedure of Immediate **Hum**an **La**beling for Software Changes Predicted as Defect-Inducing (HumLa), formulated to answer RQ1. In HumLa, if a software change is predicted as defect-inducing, this change will be more immediately labeled by humans; whereas a change predicted as clean is labeled following the waiting time labeling procedure [4] explained in Section 2.2.

In particular, when a test example $X_t \in \mathbb{R}^d$ is predicted as defect-inducing (class 1) at test time step t, formulated as

$$[\widehat{y}_t = \mathcal{M}_{t-1}(X_t)] == 1,$$

the developer who has just produced the code change is requested to immediately inspect the code while it is still fresh in their mind, and to decide whether or not this change really contains any defect. This process produces a training label $\mathcal{H}(X_t)$ immediately after t, where $\mathcal{H}(\cdot)$ represents the practical human labeling procedure. Meanwhile, the waiting time labeling procedure is still used to label any software changes that have been previously predicted as clean and that are now ready to be labeled due to the end of their waiting time. When X_t is predicted as clean (class 0), formulated as

$$[\widehat{y}_t = \mathcal{M}_{t-1}(X_t)] == 0,$$

there is no need for developers to immediately inspect this change. Therefore, only the waiting time procedure is used.

The immediate training example $(X_t, \mathcal{H}(X_t))$ that is possibly created by HumLa, and any delayed training example(s) that were possibly created by the waiting time method at test step t are all used to update the JIT-SDP model following their labeling chronological order. The latest model available before a new software change needs to be predicted is denoted as $\mathcal{M}_t(\cdot)$. It is worth noting that when no software change has been labeled before a new software change arrives to be predicted, the model $\mathcal{M}_t(\cdot)$ remains the same as the one in the prior test step as $\mathcal{M}_{t-1}(\cdot)$, i.e., $\mathcal{M}_t(\cdot) = \mathcal{M}_{t-1}(\cdot)$. When a new software change arrives to be predicted, the time step t is incremented as $t \leftarrow t + 1$, such that the most up-to-date model used for its prediction is now referred to as $\mathcal{M}_{t-1}(\cdot)$.

The HumLa procedure makes the modeling of JIT-SDP closer to the reality. As more immediate human labeled examples could better capture recent concepts of the defect-generation process in JIT-SDP, this labeling procedure could be potentially beneficial to predictive performance especially when concept drift occurs and as long as the quality of human labeling is not poor. The benefit of HumLa to predictive performance is investigated in RQ1.1. However, two issues need to be considered when investigating HumLa. The first is that developers may make mistakes when labeling the defect-predicted examples. Therefore, it is important to investigate the impact of such human label noise on predictive performance of JIT-SDP when adopting HumLa. This is done in RQ1.2 and is further explained in Section 3.1.1. The second issue is that developers may consider the effort to inspect and label *all* changes predicted as defect-inducing as too high. To save resources and reduce human effort, developers may want to inspect only part of, not all, defect-predicted examples. Therefore, it is of particular interest to investigate the extent to which the amount of human effort would affect predictive performance when adopting HumLa. This is done in RQ1.3 and is further explained in Section 3.1.2.

It is worth noting that, since only defect-predicted software changes would be immediately inspected by humans, the current underlying data generating process would likely be represented mostly by examples of the defect-inducing class. However, since the defect-inducing class is usually the minority in JIT-SDP [44], such distribution bias towards this minority class would probably not be detrimental to predictive performance of JIT-SDP models compared to the benefits gained by the more recent training examples produced by HumLa [3]. Indeed, our experimental studies in Section 6 show that HumLa can typically lead to significant improvement in predictive performance.

3.1.1 Human Label Noise.

When adopting HumLa, defect-predicted examples may be mislabeled as clean by humans, who may fail to find the defect induced by the software change. When developers do find a defect-predicted software change to induce a defect, it is assured that the change should really be defect-inducing, i.e., such label would be unlikely to be noisy. Therefore, mislabeling of defect-predicted examples is one-sided [40]. That being said, a change that is truly defect-inducing may be manually mislabeled as clean, but a change that is truly clean would be highly unlikely to be manually mislabeled as defect-inducing.

Therefore, the practical human labeling process $\mathcal{H}(X)$ of a software change described by the feature vector X and predicted as defect-inducing by the most up-to-date model can be formulated as

$$\mathcal{H}(X) = \begin{cases} 0, & \text{if } y = 0 \\ 0, & \text{if } y = 1 \text{ \& with the probability } \alpha \\ 1, & \text{if } y = 1 \text{ \& with the probability } 1 - \alpha \end{cases} \quad (1)$$

where y is the true label of X, $\alpha \in [0, 1]$ denotes a pre-defined human label noise corresponding to the probability that X would be mislabeled by human. This formulation of $\mathcal{H}(\cdot)$ will be used to analyze the impact of different amounts of one-sided human label noise $\alpha \in \{0, 0.1, \cdots, 0.9, 1.0\}$ on the predictive performance of JIT-SDP when adopting HumLa to answer RQ1.2 in Section 6.1.2. In this paper hereafter, we refer to such formulation as *HumLa at α-human noise*, whereas for the sake of simplicity, HumLa at 0-human noise is adopted as the default setting unless otherwise specified.

3.1.2 Human Effort of HumLa.

In this section, we formulate HumLa considering that some software changes predicted as defect-inducing would not be inspected by developers to save effort. Given a random variable following the uniform distribution $\iota \sim U[0, 1]$, we decide whether humans would inspect and label a software change X predicted as defect-inducing based on the following

$$\mathbb{1}(X) = \begin{cases} 1, & \text{if } \iota \leq \beta \\ 0, & \text{if } \iota > \beta \end{cases} \quad (2)$$

where $\beta \in [0, 1]$ denotes a pre-defined *human labeling percentage* corresponding to the percentage of defect-predicted examples that developers opt for inspecting over the total number of defect-predicted examples, and $\mathbb{1}(\cdot)$ is the indicator function deciding whether (value 1) or not (value 0) the developer opts for inspecting this change to produce a training example with human label $\mathcal{H}(X)$ as in Eq. (1). Larger β allows for more defect-predicted examples to be inspected by developers, usually requiring more human effort.

We will analyze the impact of different amounts of human effort in terms of the labeling percentage $\beta \in \{1, 0.9, \cdots, 0.1\}$ on predictive performance of JIT-SDP for answering RQ1.3 in Section 6.1.3. In this paper hereafter, we refer to it as *HumLa at β-human effort*, and for the sake of simplicity, HumLa at 100%-human effort is adopted as the default setting, unless otherwise specified. It is also worth noting that JIT-SDP at 0%-human effort is equivalent to the waiting time method.

3.2 ECo-HumLa

This section proposes an **E**ffort-**Co**nservative **Hum**an **La**beling (ECo-Humla) method for RQ2 to save human effort in labeling defect-predicted examples while maintaining predictive performance of JIT-SDP. As software changes to be labeled correspond to those that are inspected by practitioners at commit time in an attempt to find potential defects at an early stage, it is also important for ECo-HumLa to prioritize the labeling of software changes that are more likely to contain defects.

Given a software change X, JIT-SDP predicts whether or not this software change would be defect-inducing (class 1) or clean (class 0) based on the latest model $\mathcal{M}(\cdot)$. Besides the class prediction as $\widehat{y} = \mathcal{M}(X)$, many predictive models can also provide prediction probabilities. In the case of ensembles of models, the prediction probabilities can be computed as the mean predicted class probabilities of the base learners in the ensemble [30]. We use $c^1(X)$ and $c^0(X)$ to denote the prediction probability that X belongs to

class 1 and class 0, respectively, where $0 \le c^0(X), c^1(X) \le 1$ and $c^1(X) + c^0(X) = 1$. A predicted label is typically determined as

$$\widehat{y} = \begin{cases} 1 & \text{if } c^1(X) > c^0(X) \\ 0 & \text{if } c^1(X) \le c^0(X). \end{cases}$$

Based on these notations, we define the *prediction confidence* of test example X based on the JIT-SDP model $\mathcal{M}(\cdot)$ as

$$\rho(X) = |c^1(X) - c^0(X)|, \tag{3}$$

where $\rho(\cdot) \in [0, 1]$ and $|\cdot|$ denotes the absolute value. This metric can measure how much confidence the model has in predicting the software change X, and larger ρ indicates higher confidence upon this prediction. For JIT-SDP, a defect-predicted example with larger ρ means that this change has higher chances of inducing a defect and thus the developer would be strongly recommended to inspect this change at commit time, producing an immediately labeled example.

Given a random variable uniformly distributed as $\iota \sim U[0, 1]$, practitioners can heuristically decide whether to inspect and label a defect-predicted example X according to a probability equal to the prediction confidence $\rho(X)$. This ECo-HumLa process can be formulated as

$$\mathbb{1}(\rho(X)) = \begin{cases} 1, & \text{if } \iota \le \rho(X) \\ 0, & \text{if } \iota > \rho(X) \end{cases} \tag{4}$$

where $\mathbb{1}(\cdot)$ is the indicator function deciding whether (value 1) or not (value 0) practitioners opt for inspecting the software change to produce an immediately labeled training example. It is worth noting that, similar to HumLa, ECo-HumLa only deals with defect-predicted examples. There is no need for humans to immediately inspect clean-predicted examples, as this would lead to a large amount of effort for inspecting changes that are unlikely to induce defects.

Based on this formulation, the higher the confidence $\rho(\cdot)$, the more strongly practitioners are encouraged to inspect and label the software change, and the more likely practitioners are to really label it. However, as this procedure is stochastic, there is still some chance that practitioners would (would not) label a given change that was predicted as defect-inducing with low (high) confidence. ECo-HumLa could also be used in a deterministic way by setting a fixed threshold to replace τ in Eq. (4). However, we encourage the use of this stochastic process to avoid the strict assumption that practitioners would have to label all software changes above a certain threshold and cannot label any of the changes below the threshold.

We illustrate how to conduct the ECo-HumLa procedure as follows. Given test example X_1 and X_2, suppose that the model produces prediction probability $c^0(X_1) = 0.68$ and $c^1(X_1) = 0.32$ for X_1 and $c^0(X_2) = 0.32$ and $c^1(X_2) = 0.68$ for X_2, individually. As $c^1(X_1) < c^0(X_1)$, we have $\widehat{y}_1 = 0$ and it is unnecessary for humans to inspect X_1. As $c^1(X_2) > c^0(X_2)$, we have $\widehat{y}_2 = 1$ and its prediction confidence is further computed as $\rho(X_2) = |0.68 - 0.32| = 0.36$. This means that X_2 has the probability of 36% to be inspected by human to immediately obtain its training label.

The procedure of ECo-HumLa is consistent with the real-world process where practitioners would favor inspecting the *most* confident predictions as defect-inducing to find as many defects as possible while saving inspection effort. However, it relies on the assumption that the most confident defect predictions correspond to software changes that are more likely to be defect-inducing, so that the number of defects that practitioners would miss to find at commit time is not large. It is also unclear how much the reduction of effort is obtained through this procedure and how it would affect the predictive performance of the resulting JIT-SDP models. These three points will be investigated as part of the experiments to answer RQ2.1, RQ2.2 and RQ2.3, respectively.

Another possible setup for ECo-HumLa could be the opposite, where practitioners would be recommended to prioritize inspecting those *least* confident predictions, so that the most informative training samples [1, 26] for the JIT-SDP model can be produced during the human labeling process. This could possibly contribute the most to the performance improvement of the JIT-SDP model. We have confirmed this conjecture through experiments which have shown that human labeling the least confident predictions did outperform the proposed ECo-HumLa in terms of achieving significantly better predictive performance and conserving much more human effort. However, the least confident defect-inducing predictions may be more likely to correspond to software changes that are actually clean than the more confident defect-inducing predictions. If we recommend developers to prioritize inspecting (and thus labeling) these changes, they would waste effort in inspecting changes that may be clean, and miss several defects by not inspecting the changes that are more likely defect-inducing. Therefore, while such alternative setup makes sense from a model predictive performance perspective, it would be unsuitable as a practical approach to JIT-SDP.

4 DATASETS

This paper uses 14 GitHub open source projects as in previous work [40, 41] to investigate the proposed human labeling methods for JIT-SDP, as summarized in Table 1 of the supplementary material. Twelve metrics have been used as input features following previous work [19], as explained in Section 1 of the supplementary material.

The `Commit Guru` [37] tool was used to collect the data with input features and labels of software changes. The tool is based on the SZZ algorithm [39] to decide actual labels of software changes, which are defect-inducing (class 1) or clean (class 0). As SZZ is known to lead to label noise [16, 20, 34–36], we have conducted a manual inspection of a random sample of changes of each project to investigate its data quality. Four experts with at least 4 years of programming experience have been asked to work in pairs to label these changes. Each pair was asked to discuss each of the software changes to decide on their labels, leading to two sets of human-generated labels (Pair 1 and Pair 2). Following existing work [16, 27], human annotators were asked to label software changes as defect-fixing or non-defect-fixing, instead of labeling software changes as defect-inducing or clean directly. This is because it would be extremely time-consuming, if even possible at all, for a software developer who had not worked on a given project to manually check whether a software change is defect-inducing or clean directly on this project based on (e.g.) codes and/or commit messages. Fixes are used by SZZ to identify defect-inducing software changes. Therefore, a high level of noise in the SZZ identification of changes as defect-fixing and non-defect-fixing means a high level of noise in the SZZ labels of defect-inducing and clean. Note that noise arising from `git blame` within `Commit Guru` is not included in this manual inspection process and could lead to additional noise [2,

Table 1: Kappa scores

Dataset	(1) SZZ vs Pair 1	(2) SZZ vs Pair2	Average of (1) and (2)	(3) Pair 1 vs Pair2
Brackets	0.5384	0.4722	0.5053	0.7083
Broadleaf	0.7101	0.6522	0.6812	0.7867
Camel	0.4376	0.5604	0.4990	0.6330
Fabric8	0.6418	0.8056	0.7237	0.6620
jGroups	0.5070	0.5303	0.5187	0.9045
Nova	0.5550	0.5639	0.5594	0.7085
Tomcat	0.5740	0.6840	0.6290	0.7451
Corefx	0.7573	0.6291	0.6932	0.6835
Django	0.7791	0.8739	0.8265	0.7387
Rails	0.4324	0.5556	0.4940	0.7258
Rust	0.5935	0.4804	0.5370	0.4804
Tensorflow	0.8584	0.8052	0.8318	0.7973
VScode	0.8750	0.8095	0.8423	0.8750
wp-Calypso	0.8751	0.7746	0.8249	0.7101
Median	0.6176	0.6407	0.6551	0.7180

35]. A sample of 100 changes (50 defect-fixing and 50 non-defect-fixing) was drawn from each project and sorted in random order to circumvent bias during the human annotation process. If the commit message did not contain an issue ID within it, then only the commit message itself was checked to determine whether the software change is defect-fixing or not. Otherwise, both the commit message and the issue were used. In particular, if a commit message is addressing an issue identified by a given ID corresponding to a bug, this change was labeled as a defect-fixing change.

The Kappa scores [7] (1) between SZZ and Pair 1, (2) between SZZ and Pair 2, and (3) between Pair 1 and Pair 2 are shown in Table 1. Following Hall et al. [15], we interpret Kappa score as follows: $[-1, 0]$ less than chance agreement; [0.01, 0.20] slight agreement; [0.21, 0.40] fair agreement; [0.41, 0.60] moderate agreement; [0.61, 0.80] substantial agreement; and [0.81, 0.99] almost perfect agreement. We can see that the Kappa scores (1) and (2) indicate at least moderate agreement for all datasets. In four datasets (Broadleaf, Fabric8, Tomcat, and Corefx), the average of (1) and (2) indicates a substantial agreement and in other four (Django, Tensorflow, VScode, and wp-Calypso) it indicates almost perfect agreement. Moreover, the median Kappa scores across datasets between human annotators and SZZ (average of (1) and (2)), and between human annotators themselves (3) both indicate a substantial agreement, showing that the level of agreement between human annotators and SZZ is in line with the level of agreement between humans themselves. This suggests that the labels provided by SZZ were in general unlikely to be worse than the labels given by humans.

Kappa scores indicate the level of label noise associated to correctly distinguishing defect-fixes from non-defect-fixes, which are in turn used to identify defect-inducing changes by SZZ. However, if a given fix has not yet been implemented, SZZ would be unable to link this fix to a defect-inducing change, even if its ability to distinguish defect-fixes from non-defect-fixes is perfect. A previous study [41] showed that, if we use the first 10k changes of the projects in our study, there is at least an estimated 99% confidence level that the fixes corresponding to these changes have already been reported. Therefore, we use the first 10k software changes of each project in the experiments to increase data quality.

5 EXPERIMENTAL SETUP

To investigate the impact of the labeling processes that take into account the human labeling conducted through inspection of defect-predicted software changes, the predictive performance of JIT-SDP models with and without immediate human labeling will be compared. Our labeling approaches can be adopted with different machine learning algorithms. In our experiments, **O**versampling-based **Da**ta **S**treaming bagging with **C**onfidence (ODaSC) using Hoeffding trees [40] are adopted whenever JIT-SDP models need to be created/updated. This is a recent online learning algorithm for JIT-SDP. Being an online algorithm, it updates JIT-SDP models using each training example individually upon arrival, and then discards it, without the need for retraining on past examples. ODaSC is chosen for being the state-of-the-art online JIT-SDP model. It deals with label noise resulting from verification latency by estimating the confidence in the labels assigned to training examples [40]. It was shown to be more robust to noise in JIT-SDP than OOB [40], which in turn was shown to be better than sliding window approaches [4].

We use Geometric Mean of Recall 0 and Recall 1 (G-Mean) [23] to evaluate predictive performance of JIT-SDP models with and without immediate human labeling. Different from accuracy or precision, G-mean is known to be robust against class imbalance, which is particularly important for studies suffering from class imbalance evolution such as JIT-SDP [4, 41, 48]. We have also adopted Matthews Correlation Coefficient (MCC) [6, 54] as it has become popular in the area of JIT-SDP [21, 22, 55]. We use tp to denote true positives (the number of defect-inducing software changes that are predicted correctly), fn to denote false negatives (the number of defect-inducing software changes that are erroneously predicted as clean), tn to denote true negatives (the number of clean software changes that are predicted correctly) and fp to denote false positives (the number of clean software changes that are erroneously predicted as defect-inducing). Based on them, G-Mean $= \sqrt{\frac{tp}{tp+fn} \cdot \frac{tn}{tn+fp}} \in [0, 1]$. As the false positive rate is defined as $1 - tn/(tn + fp)$, G-mean takes into account the trade-off between true positives and false positives; larger G-Mean means better performance. MCC $= \frac{tp \cdot tn - fp \cdot fn}{\sqrt{(tp+fp)\cdot(tp+fn)\cdot(tn+fp)\cdot(tn+fn)}} \in [-1, 1]$ takes all 4 elements of the confusion matrix into consideration and thus provides high scores only if the predictions return good rates for all 4 entries of the confusion matrix [5, 6]. G-Mean and MCC are computed in a sequential way based on a *fading factor* to track the changes in predictive performance over time as recommended in the online learning scenario [14]. Fading factor $\theta \in [0, 1]$ controls how much emphasis one would like to place on past evaluation examples compared to the new one and larger/smaller values for θ places more emphasis on the past/present model performance status. The fading factor $\theta = 0.99$ is used in this paper following previous JIT-SDP studies [4, 40, 43], enabling a good trade-off between rapidly tracking performance changes and preventing wild variations. When computing the average predictive performance, an average of the sequential performance was used.

We used a grid search to choose the best parameter setting for each project. ODaSC has three parameters: the ensemble size (# Hoeffding trees) $\in \{5, 10, 20, 30, 40\}$, the decay factor of class imbalance $\in \{0.9, 0.95, 0.99, 0.999\}$ and the resampling threshold $\in \{0.8, 0.9\}$.

We adopt the parameter settings that can achieve the best average G-Means across 30 runs based on the first 500 software changes in the data stream of each project. Hoeffding trees use the default parameters provided in the python package *scikit-multiflow* [30], following previous JIT-SDP studies [3, 40, 41].

To investigate to what extent the proposed ECo-HumLa can encourage defects to be found when saving human effort, we need to formulate two additional metrics. The first metric is *human recall-1* that is the ratio of defect-inducing changes ($y = 1$) that were predicted as defect-inducing ($\widehat{y} = 1$) and that developers were asked to label ($\mathcal{H}(X)$). Given a human labeling process, this metric can be formulated as

$$R1_{\mathcal{H}} = \frac{\#[\widehat{y} = 1, \mathcal{H}(X), y = 1]}{\#[y = 1]}, \quad (5)$$

where $\#(\cdot)$ denotes the number of software changes satisfying the inside condition(s). We can see that $R1_{\mathcal{H}}$ is the ratio of defect-inducing changes that we asked developers to inspect / label. A higher value means that developers are being asked to inspect more software changes that are really defect-inducing, potentially uncovering more defects. The second metric is *human false alarm* that is the ratio of clean software changes ($y = 0$) that were predicted as defect-inducing and that developers were asked to label. Given a human labeling process $\mathcal{H}(\cdot)$, this metric can be formulated as

$$FalseAlarm_{\mathcal{H}} = \frac{\#[\widehat{y} = 1, \mathcal{H}(X), y = 0]}{\#[y = 0]}. \quad (6)$$

We can see that $FalseAlarm_{\mathcal{H}}$ is the ratio of clean changes that we asked human to inspect / label. A higher value means that more human inspection effort is wasted.

Predictive performance of online learning methods with the best parameter settings is evaluated based on the rest 500~10,000 software changes of the project. Comparisons between JIT-SDP with vs without the practical human labeling method are conducted based on the mean predictive performance across 100 runs to account of ODaSC's stochasticity. We report the results corresponding to the waiting time of 15 days, following previous related studies [40]. In particular, this waiting time was found to lead to JIT-SDP models with better predictive performance than other values as they offer a better trade-off between one-sided label noise and the ability to tackle concept drift [40, 41]. Friedman tests [10] will be performed for statistical comparisons between more than two JIT-SDP methods across datasets. The null hypothesis (H0) states that all methods perform similar; the alternative hypothesis states that they have statistically significant difference. Given the rejection of H0, we will further perform pairwise comparisons using the Conover post hoc tests. Two-tailed pairwise Wilcoxon signed rank tests at significance level 0.05 [10] will be performed whenever statistical comparisons between two methods across datasets are needed.

To analyze human effort, in addition to considering the human labeling percentage mentioned in Section 3.1.2, we will also analyze its corresponding code churn, which is a popular metric of human inspection effort in the defect prediction literature [19, 31, 51, 52]. The code churn of a software change is defined as $(LA + LD)/2$, where LA is the number of lines added and LD is the number of lines deleted by the software change.

Table 2: RQ1.1 – Performance comparisons between HumLa at 0%-human noise and 100%-human effort (the default setup) and the waiting time method for each dataset in terms of G-Mean and MCC, respectively.

Dataset	G-Mean			MCC		
	Waiting time	HumLa	Imp%	Waiting time	HumLa	Imp%
Bracket	0.643	0.639 [-b]	-0.48	0.300	0.297 [-m]	-0.97
Broadleaf	0.607	0.663 [b]	9.34	0.292	0.347 [b]	18.63
Camel	0.669	0.681 [b]	1.82	0.356	0.378 [b]	6.30
Fabric8	0.653	0.661 [b]	1.13	0.320	0.333 [b]	4.25
jGroup	0.568	0.600 [b]	5.68	0.193	0.242 [b]	25.68
Nova	0.682	0.688 [b]	0.85	0.380	0.391 [b]	3.01
Tomcat	0.613	0.638 [b]	4.05	0.282	0.309 [b]	9.36
Corefx	0.639	0.636 [-s]	-0.50	0.362	0.360 [-*]	-0.41
Django	0.690	0.698 [b]	1.21	0.413	0.430 [b]	3.93
Rails	0.562	0.623 [b]	10.77	0.214	0.296 [b]	38.54
Rust	0.584	0.586 [*]	0.32	0.250	0.248 [-*]	-0.75
Tensorflow	0.691	0.678 [-b]	-1.87	0.389	0.394 [b]	1.27
VScode	0.527	0.527 [*]	0.12	0.276	0.302 [b]	9.43
wp-Calypso	0.551	0.622 [b]	12.98	0.256	0.293 [b]	14.68
ave-rank	1.75	1.25	-	1.79	1.2	-

"Imp%" denotes the improvement ratio of HumLa over the waiting time (control) method. Symbols [*], [s], [m] and [b] denote insignificant, small, medium, and large A12 [46], respectively. Presence/absence of the sign "-" in A12 means that HumLa was worse/better than the waiting time method. The A12 effect size of differences in performance for each dataset was typically large. The last row reports the average ranks across datasets for the two methods, which were found to be statistically significantly different both in terms of G-Mean and MCC. Smaller ranks are better ranks.

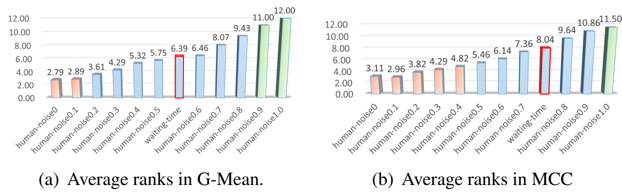
6 EXPERIMENTAL RESULTS

6.1 RQ1: JIT-SDP with HumLa

We complete our answer to RQ1 in this section by investigating the impact of HumLa on predictive performance of JIT-SDP compared to the waiting time method, and with respect to different human label quality and levels of human effort.

6.1.1 RQ1.1 – Predictive Performance. Table 2 shows the performance comparison between HumLa and the waiting time method. Performance tables using other metrics are reported in the supplementary material for space consideration. We can see that HumLa leads to significant difference in predictive performance in terms of both G-Mean and MCC across datasets. Two-tailed pairwise Wilcoxon signed rank tests found significant difference with p-values 0.0208 and 0.0016 in terms of G-Mean and MCC, respectively. This means that a more practical labeling procedure would lead to significant differences in predictive performance compared to the less practical waiting time method. Thus, it is important to evaluate JIT-SDP methods with a labeling procedure that is closer to the reality.

We can also see from Table 2 that HumLa usually improves predictive performance of JIT-SDP compared to the waiting time method in most datasets except for Bracket with a negative improvement ratio -0.48%, Corefx with -0.50% and Tensorflow with -1.87% in terms of G-Mean, and Bracket with -0.97%, Corefx with -0.41% and Rust with -0.75% in terms of MCC, respectively. While such negative effects are of a small magnitude, the benefit to predictive performance of JIT-SDP can be substantial: the improvement ratios are 9.34% (18.63%) in Broadleaf, 10.77% (38.54%) in Rails and 12.98% (14.68) in wp-Calypso in terms of G-Mean (MCC). This is in line with the statistical test, which found significant benefit to the performance. Such improvements also mean that exiting studies may be underestimating predictive performance of JIT-SDP methods.



(a) Average ranks in G-Mean.

(b) Average ranks in MCC

Figure 2: RQ1.2 – Predictive performance of HumLa at different amounts of human noise. Bar plots report average ranks of JIT-SDP methods across datasets. The waiting time method is chosen as the control method (framed in red). Methods significantly better (worse) than the control method are filled in light orange (green).

6.1.2 RQ1.2 – HumLa at Different Human Noise.

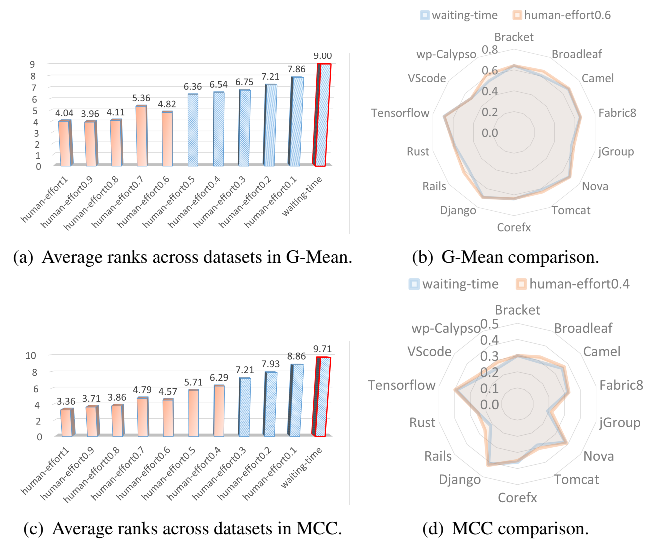
When labeling software changes predicted as defect-inducing through HumLa, humans may mislabel some software changes, leading to noisy training examples. Different persons and organizations may have different human annotation error rates. Determining the typical human annotation error rate associated to software engineers is not possible as the ground truth labels are unknown. In particular, neither humans nor algorithms such as SZZ are currently able to provide fully reliable ground truth labels for this purpose. Therefore, we provide a detailed analysis to investigate the impact of various amounts of human annotation error rates, which we refer to as human noise in this section for brevity.

Figure 2 shows average Friedman ranks of HumLa in terms of G-Mean and MCC at different amounts of human noise against the waiting time method across datasets. Plots of the performance obtained over time and tables of overall performance using several metrics are reported in the supplementary material for space considerations. Friedman tests with the significance level 0.05 reject H0 with p-values 8.61E-19 and 2.12E-17 in terms of G-Mean and MCC, respectively, meaning that different amounts of human noise leads to significant difference in predictive performance of JIT-SDP. The waiting time method is then chosen as the control method to conduct Conover post-hoc tests, whose results are also illustrated in Figure 2.

We can see that HumLa at 0%-human noise can significantly improve predictive performance of JIT-SDP compared to the waiting time method; when the human label noise is no worse than 10% (40%) in terms of for G-Mean (MCC), such beneficial impact produced by HumLa would be significant. Therefore, even when there is human label noise, adopting a labeling procedure closer to reality can have positive impact on predictive performance of JIT-SDP compared to adopting the waiting time method. In other words, existing work adopting waiting time may be underestimating the predictive performance that can be achieved in practice especially when adopting MCC as a performance metric. In addition, HumLa can obtain similar or better ranking than the waiting time method so long as the human label noise is no higher than 80% (90%) in G-Mean (MCC). This means that producing labels earlier leads to similar or better predictive performance, unless the amount of noise in the human labels is extremely high. Therefore, delaying the production of labels for changes predicted as defect-inducing is not recommended.

6.1.3 RQ1.3 – HumLa at Different Human Effort.

Figure 3 shows performance comparisons between HumLa at different amounts of human effort in terms of G-Mean and MCC, respectively. Performance tables using other metrics are reported in the supplementary material for space consideration. Friedman tests at the significance level 0.05 run across datasets reject H0 with p-values 1.576E-05 and 3.159E-12 in terms of G-Mean and MCC, respectively. Therefore, different amounts of human effort lead to significant difference in predictive performance of JIT-SDP. Given the rejection of H0, the waiting time method (HumLa at 0%-human effort) is chosen as the control method to conduct Conover post-hoc tests, whose results are illustrated in Figures 3(a) and 3(c).



(a) Average ranks across datasets in G-Mean.

(b) G-Mean comparison.

(c) Average ranks across datasets in MCC.

(d) MCC comparison.

Figure 3: RQ1.3 – Predictive performance of HumLa at different amounts of human effort. Bar plots report average ranks of each method across datasets. The waiting time method is equivalent to HumLa at 0-human effort and is chosen as the control method (framed in red). Methods that perform significantly better than the control method are filled in light orange. No method was worse than the control. Radar plots show performance comparisons of HumLa at a particular human effort against the waiting time method for each dataset.

We can see from these figures that larger human effort is in general beneficial to the predictive performance. In addition, when practitioners randomly label 60% (40%) defect-predicted test examples, HumLa significantly improves predictive performance of JIT-SDP compared to the waiting time method in terms of G-Mean (MCC). Indeed, so long as the amount of human effort is no less than 60% (40%) in terms of G-Mean (MCC), HumLa would have significant benefit to predictive performance compared to the control method. Moreover, HumLa achieves similar ranking as the waiting time method when using only 10% of the effort in terms of human labeling percentage. As the key difference between HumLa and the waiting time method is the earlier labeling of training examples, the positive impact of HumLa on predictive performance is due to the ability to update JIT-SDP models earlier. This in turn may enable JIT-SDP to react to concept drift faster, even when the amount of immediately labeled data is not large. From Figures 3(b) and 3(d), we can see that HumLa at 60%(40%)-human effort can indeed usually attain performance improvement compared to the control method in most datasets, whereas for the datasets where there is no improvement, the magnitude of the differences in performance is very small

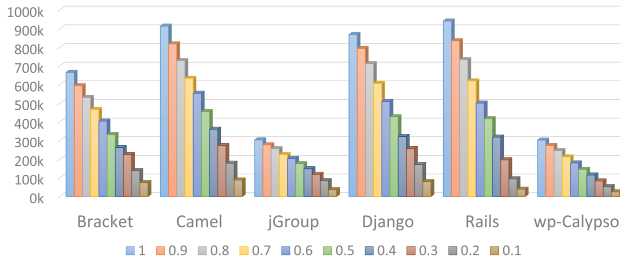


Figure 4: RQ1.3 – Relationship between the human labeling percentage and the cumulative code churn on some random datasets for HumLa at varying amounts of human effort. Results of other datasets showed the same pattern and were omitted for the space reason.

(-0.02% for Bracket, -0.35% for Corefx and -0.70% for Tensorflow in terms of G-Mean, and -2.95% for Corefx in terms of MCC).

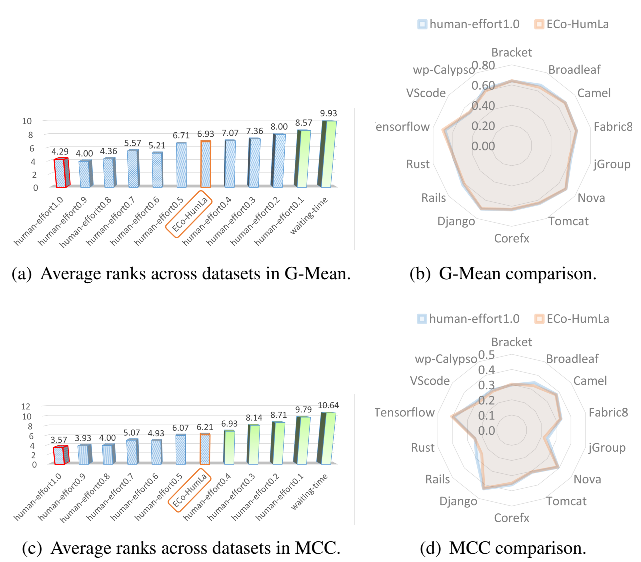
Figure 4 shows the relationship between cumulative code churn and the human labeling percentage on 6 randomly picked datasets. Plots of other datasets showed the same pattern and were reported in the supplementary material for space restrictions. We can see that higher human labeling percentage almost always accounts for larger value of the cumulative code churn, showing a good correlation between these two metrics. Therefore, given a project for which practitioners may be interested in creating a JIT-SDP model, reducing the inspection rate is really likely to correlate with a decrease in churn for this project. Table 3 contains the code churn values for all datasets. We can see that, for example, HumLa at 40%- and 60%-human effort would reduce around up 40% and 60% code churns, respectively, being consistent to the human labeling percentages.

These results are particularly relevant and of practical significance as they indicate that when practitioners allow for a relatively low cost of human effort in randomly labeling a small portion (e.g., 10%) of defect-predicted test examples, HumLa already begins to have beneficial impact; when HumLa allows for a moderately large human effort such as 60%, it would perform significant better performance compared to the waiting time method.

Answer to RQ1: HumLa would not only model the practical JIT-SDP process to be closer to the reality but also leads to significant difference in predictive performance compared to the less practical waiting time method. Even when there is human label noise, adopting HumLa can usually have positive impact on predictive performance of JIT-SDP, indicating that existing work adopting the waiting time method is likely underestimating the predictive performance that can be achieved in practice especially in terms of MCC. HumLa allowing for higher amounts of human effort generally attain better performance and when the allowance for human effort is beyond a moderately large value such as 60%, HumLa would produce significantly better performance compared to the waiting time method.

6.2 RQ2: JIT-SDP with ECo-HumLa

This section completes our answer to RQ2 to evaluate the proposed ECo-HumLa with respect to how well it can save human effort while maintaining predictive performance and avoiding a large number of defect-inducing software changes to be missed by practitioners.



(a) Average ranks across datasets in G-Mean. (b) G-Mean comparison.

(c) Average ranks across datasets in MCC. (d) MCC comparison.

Figure 5: RQ2.1 – Performance comparisons between ECo-HumLa vs HumLa at different amounts of human effort. Bar plots report average ranks of each method across datasets. HumLa at 100%-human effort is chosen as the control method (framed in red) and methods that perform significantly inferior to the control method are filled in light green. The proposed ECo-HumLa is framed in orange to facilitate visualization. Radar plots show performance comparisons between ECo-HumLa and HumLa at 100%-human effort for each dataset.

6.2.1 RQ2.1 – Retained Performance. Figure 5 shows performance comparisons between ECo-HumLa and HumLa at different amounts of human effort in terms of G-Mean and MCC. Performance values using several metrics are reported in the supplementary material for space consideration. Friedman tests at significance level 0.05 reject H0 with p-values 2.65E-05 and 5.10E-10 in terms of G-Mean and MCC, respectively, meaning that ECo-HumLa and HumLa with different amounts of effort achieve significantly different performance. Given the rejection of H0, HumLa at 100%-human effort is chosen as the control method to conduct Conover post-hoc test, whose results are illustrated in Figures 5(a) and 5(c).

We can see from these figures that statistical tests found no significant difference between the performance of ECo-HumLa and HumLa at 100%-human effort across datasets, in terms of both G-Mean and MCC. This indicates the capability of ECo-HumLa in retaining predictive performance compared to HumLa at 100%-human effort, despite labeling less software changes. We can also see that ECo-HumLa's ranking is in between HumLa at 40%~50%-human effort in terms of both G-Mean and MCC. Further post-hoc comparison using ECo-HumLa as the control method cannot find significant difference between these three methods, indicating them to have similar performance when compared across datasets.

Figures 5(b) and 5(d) show that ECo-HumLa's performance values are usually below those achieved by HumLa at 100%-human effort in most datasets as one would expect, but the inferiority ratio is usually of small magnitude: in terms of G-Mean, the inferiority ratio is below 3.6% in Broadleaf and wp-Calypso; in terms of MCC, most of the inferiority ratios are around below 5% except for 9.21%

Table 3: RQ2.2 – Saved human effort of the proposed ECo-HumLa against HumLa at different amounts of human effort in terms of the human labeling percentage and the cumulative code churn (in kilo).

Dataset	Human effort (in kilo) of HumLa 100%	90%	80%	70%	60%	Eco-HumLa auto	human%	Human effort (in kilo) of HumLa 50%	40%	30%	20%	10%
Bracket	664.8[b]	593.2	530.4	466.0	403.9	371.0	55.51%	331.5	261.1	223.2	137.9	74.3
Broadleaf	2715.4[b]	2418.2	2171.9	1884.0	1612.6	2162.6	46.60%	1326.4	1002.0	777.2	506.9	258.2
Camel	913.3[b]	818.0	727.8	633.0	553.6	578.2	54.57%	455.0	360.5	271.8	177.9	87.6
Fabric8	2099.6[b]	1941.0	1733.7	1528.7	1297.5	1165.2	46.85%	1071.2	844.6	651.0	403.9	214.5
jGroup	302.8[b]	276.2	255.3	224.4	203.8	200.0	45.55%	174.8	147.8	118.6	83.4	35.6
Nova	612.7[b]	555.1	488.5	434.7	380.4	332.7	62.09%	338.5	267.1	199.3	134.2	73.1
Tomcat	564.9[b]	510.0	450.8	386.8	329.3	373.2	58.01%	273.5	214.7	156.0	104.1	50.5
Corefx	4486.7[b]	4129.2	3631.0	3382.0	2904.6	3305.1	51.95%	2384.0	1913.4	1450.3	985.9	512.4
Django	867.6[b]	793.3	711.8	605.3	509.1	499.8	71.28%	426.5	321.5	255.6	171.3	79.6
Rails	940.6[b]	834.8	733.1	619.4	500.5	359.1	47.07%	416.2	317.9	195.5	93.6	38.5
Rust	547.8[b]	479.8	418.9	356.6	323.1	234.2	40.73%	289.7	241.7	179.0	130.7	55.0
Tensorflow	1140.3[b]	1106.2	1058.1	979.3	902.2	900.4	55.94%	796.3	655.1	506.2	328.5	164.5
VScode	286.1[b]	262.6	232.6	206.7	170.2	203.4	48.64%	144.4	123.8	83.4	57.5	35.6
wp-Calypso	301.9[b]	273.5	245.8	211.6	178.9	187.8	51.41%	145.3	114.4	82.6	52.1	24.0
median	766.2	693.3	621.1	535.7	452.2	372.1	51.68%	377.3	292.5	211.3	136.1	73.7

For HumLa at 100%-human effort, the A12 effect size [46] of differences in saved human effort for each dataset was always large ([b]) compared to Eco-HumLa as the control method. Effect sizes for other levels of human effort are in the supplementary material.

in jGroup and 14.31% in Rails. This is in line with the results of the above mentioned Conover tests, which found no difference between ECo-HumLa and HumLa at 100% human effort across datasets.

An important question is then how much effort ECo-HumLa can save compared to these other methods that obtained similar performance. This is investigated in Section 6.2.2.

6.2.2 RQ2.2 – Saved Human Effort.

Table 3 shows that the median human effort of ECo-HumLa across datasets is 372.1k in terms of the cumulative code churn. Compared to the median cumulative code churn 766.2k of HumLa at 100%-human effort, ECo-HumLa can generally conserve around 50%-human effort in checking the lines of code being changed, demonstrating the effectiveness of ECo-HumLa in saving human effort. We can also see that the median human labeling percentage of ECo-HumLa across datasets is 51.68%, in-between HumLa at 50%~60%-human effort, also showing that ECo-HumLa saves around half of human effort.

Combining with the observation in Section 6.2.1, we can conclude that ECo-HumLa achieved similar performance of HumLa at 100%-human effort while requiring only around half of the inspection effort in terms of human labeling percentage and code churn. Therefore, we would recommend practitioners to inspect and label all defect-predicted software changes when the human cost is allowable, as the inspection used to label defect-predicted examples is the same as the inspection needed to find and fix any defects that the software change may induce. Therefore, inspecting these software changes may help practitioners to find more defects at an early stage. However, if the cost of inspecting all defect-predicted software changes is too high, inspecting only around half of the defect-predicted software changes through ECo-HumLa will not lead to worse predictive performance of the resulting JIT-SDP models. In this sense, it would be acceptable to reduce labeling effort through ECo-HumLa.

6.2.3 RQ2.3 – Finding Defects.

Based on the previous sections, ECo-HumLa requires similar effort and leads to similar predictive performance as HumLa at 50%-human effort. However, ECo-HumLa was designed to direct practitioners' inspection / labeling effort towards software changes that are more likely to contain defects, so that the reduction in effort does not come at the cost of practitioners missing a too large number of defects in the code. So, a question remains on whether ECo-HumLa can really encourage a larger number of defects to be found than HumLa at 50%-human effort (RQ2.3).

Table 4: RQ2.3 – Recall 1 and False alarm for humans between ECo-HumLa vs HumLa at 50%-human effort that perform similarly to ECo-HumLa at the cost of similar amount of human effort as found in RQ2.2.

Dataset	Recall 1 for humans ECo-HumLa	HumLa-50%	False alarm for humans ECo-HumLa	HumLa-50%
Bracket	0.4039	0.3338 [-b]	0.1631	0.1787 [-b]
Broadleaf	0.2854	0.2793 [-s]	0.0885	0.1168 [-b]
Camel	0.4253	0.3612 [-b]	0.1815	0.1900 [-b]
Fabric8	0.3488	0.3380 [-m]	0.1434	0.1704 [-b]
jGroup	0.2995	0.2544 [-b]	0.1159	0.1527 [-b]
Nova	0.4246	0.3330 [-b]	0.1573	0.1366 [b]
Tomcat	0.3684	0.3044 [-b]	0.1639	0.1567 [b]
Corefx	0.2932	0.2322 [-b]	0.0734	0.0788 [-m]
Django	0.4745	0.3116 [-b]	0.1026	0.1016 [s]
Rails	0.3038	0.3203 [b]	0.1417	0.1906 [-b]
Rust	0.2208	0.2489 [b]	0.0851	0.1190 [-b]
Tensorflow	0.4446	0.3886 [-b]	0.1627	0.1902 [-b]
VScode	0.1924	0.1538 [-b]	0.0551	0.0511 [b]
wp-Calypso	0.2555	0.2531 [-*]	0.0910	0.1075 [-b]
ave-rank	1.14	1.86	1.29	1.71

Symbols [*], [s], [m] and [b] denote insignificant, small, medium, and large A12 [46], respectively. Presence/absence of the sign "-" in A12 means that HumLa at 50%-human effort was worse/better than ECo-HumLa. The A12 effect size of differences in performance for each dataset were typically large. The last row reports the average ranks across datasets for the two methods, which were found to be statistically significantly different both in terms of Recall 1 and False Alarms. Smaller ranks are better ranks.

Table 4 reports the human recall 1 that is formulated in Eq. (5) and the human false alarm that is formulated in Eq. (6) between Eco-HumLa and HumLa at 50%-human effort. We can see that in terms of the human recall 1, practitioners can usually detect more defects (with the improvement ratio of up to 52.27% for Django and around 26% for Nova, Corefx, and VScode) in software changes when they opt for ECo-HumLa compared to HumLa at 50%-human effort, and such superiority is statistically significant according to two-tailed pairwise Wilcoxon signed rank tests at significance level 0.05 (p-value 4.03E-03). In the meantime, practitioners can have lower (better) human false alarm (with the improvement ratio of around 25% for Broadleaf, jGroup, Rails and Rust) compared to HumLa at 50%-human effort, and such superiority is significant according to two-tailed pairwise Wilcoxon signed rank tests at significance level 0.05 (p-value 0.0166). Such results are of practical significance as it means that even though ECo-HumLa can get similar predictive performance at similar human cost compared to HumLa at 50%-human effort, this targeted human labeling approach can help practitioners inspect software changes that are more likely truly defect-inducing, encouraging them to find defects at an early stage.

Answer to RQ2: The proposed ECo-HumLa can save around 50%-human effort in terms of both the human labeling percentage and the cumulative code churn while still retaining predictive performance of JIT-SDP compared to HumLa at 100%-human effort. Moreover, when adopting ECo-HumLa, practitioners would be encouraged to find more detects and achieve a lower false alarm rate than when randomly deciding which changes to inspect based on HumLa at 50%-human effort.

7 THREATS TO VALIDITY

Internal Validity. One potential issue for HumLa is that developers may take a while to decide actual labels of defect-predicted examples, resulting in verification latency of the "immediate" human labeling procedure. However, such time delay would be negligible compared to that in the waiting time labeling procedure. Also, as such human delay is disregarded, the HumLa labeling procedure will not produce training examples exactly in chronological order. Nevertheless, it still leads to a more realistic labeling and model training procedures than the ones adopted in existing literature when the true time taken to label defect-inducing software changes is unknown. Future work could further investigate the length and impact of such time delay. Similar to previous work that has adopted the same datasets [4, 40, 41], other threats to internal validity include various types of noise arising from SZZ [16, 20, 34–36] including noise stemming from the `git blame` command used in `Commit Guru` that were not captured by our manual Kappa analysis in Section 4. The data were collected based on the original SZZ algorithm [39], which was shown to lead to JIT-SDP models of similar predictive performance to models created based on the most recent SZZ algorithm [12], which applies the largest number of noise filters in comparison to several other SZZ variants [8, 32, 49]. We also adopted the waiting time that was shown to lead to a good trade-off between label noise and the obsolescence of the trained models [40, 41]. To mitigate threats related to the randomness of ODaSC, our results are based on 100 runs of each method on each dataset.

Construct Validity. G-Mean, Recall 0 and Recall 1 are unbiased metrics suitable for class imbalanced problems such as JIT-SDP. We have also adopted MCC as a popular metric in the SDP literature which uses all entries of the confusion matrix. A fading factor was adopted to enable tracking the fluctuations in predictive performance over time, as recommended for online learning problems [14].

External Validity. We have investigated 14 open source projects, covering a wide range of data characteristics as explained in Section 4. However, as with any study involving machine learning, experimental results may not generalize well to other projects. We investigate the proposed methods based on ODaSC with Hoeffding trees, which have been previously opted for online JIT-SDP [40, 41]. Other online machine learning models could lead to different results. However, it is worth noting that machine learning approaches are in general expected to struggle more to adapt to a concept drift if there is no labeled data coming from the new underlying data distribution than if they had access to such labeled data. Our labeling procedures can thus act as enablers for learning approaches to adapt to concept drift more promptly, possibly benefiting predictive performance of other learning algorithms. When investigating the impact of human effort within (Eco-)HumLa, zero human noise was assumed, facilitating a focused analysis of human effort without being affected by human noise. In practice, a non-zero and unknown level of human noise would be associated with (Eco-)HumLa, which could lead to different conclusions. Following standard practice in the JIT-SDP literature, we have used the main branch of the open source repositories to collect software changes and their labels. Therefore, only software changes that have been accepted in the main branch (possibly after code review) have been used in our evaluation. The results may not generalize to predicting rejected changes.

8 CONCLUSIONS

We have conducted the first study on how to consider the effect of adopting JIT-SDP during the software development process in the labeling procedure of software changes for online JIT-SDP, leading to the HumLa procedure. The impact of HumLa on the predictive performance of JIT-SDP was investigated at different levels of noise and human effort. We have also proposed ECo-HumLa to save human effort by targeting the inspection process towards software changes predicted as defect-inducing with higher confidence.

Our experiments showed that adopting a labeling procedure closer to reality leads to a significant impact on the predictive performance of JIT-SDP, with generally better predictive performance than the delayed labeling method with waiting time even when human labeling contained a certain level of noise. As an implication to research, this shows the importance of adopting labeling methods such as HumLa that are closer to what would be adopted in practice, when conducting studies to evaluate JIT-SDP models. As an implication to practice, it shows the importance of not delaying the inspection of software changes to achieve better performing JIT-SDP models.

We also showed that it is possible to save around 50% of inspection effort through ECo-HumLa while maintaining predictive performance compared to HumLa at 100%-human effort and encouraging a larger number of defects to be found than when saving effort through HumLa at 50%-human effort. As an implication to research, this shows that effort-aware strategies can be designed to work in an online manner, encouraging further research on online effort-aware JIT-SDP. Researchers may also analyse the predictive performance of their models under different labeling efforts through ECo-HumLa. As an implication to practice, these results show that if practitioners are unable to inspect all defect-predicted software changes due to the inspection effort, we recommend to target the inspection effort based on the confidence of the predictions through ECo-HumLa rather than randomly deciding which changes to inspect. If they can afford the higher effort or are dealing with safety-critical systems, it is still recommended to use HumLa to find more defects.

Future work includes investigating HumLa and ECo-HumLa with other JIT-SDP models, datasets, and input features such as high-level latent representations of a deep neural network; proposing novel effort-aware online JIT-SDP approaches to further improve on ECo-HumLa; analyzing the verification latency of the human inspection process; investigating (ECo-)HumLa with JIT-SDP models for predicting pre-code review changes; investigating the impact of `git blame` used by SZZ in the label quality of the investigated datasets; and investigating the impact of various levels of human effort under various levels of human noise for (Eco-)HumLa.

DATA AVAILABILITY

A replication package is available at doi: 10.5281/zenodo.8272293. The source code is available under a GNU GPL v3.0 license.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (NSFC) under Grant Nos. 62002148 and 62250710682, the Program for Guangdong Introducing Innovative and Enterpreneurial Teams under Grant No. 2017ZT07X386, Guangdong Provincial Key Laboratory under Grant No. 2020B121201001 and Research Institute of Trustworthy Autonomous Systems (RITAS).

REFERENCES

[1] Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul Fieguth, Xiaochun Cao, Abbas Khosravi, U. Rajendra Acharya, Vladimir Makarenkov, and Saeid Nahavandi. 2021. A Review of Uncertainty Quantification in Deep Learning: Techniques, Applications and Challenges. *Information Fusion* 76 (2021), 243–297.

[2] Peter Bludau and Alexander Pretschner. 2022. PR-SZZ: How pull requests can support the tracing of defects in software repositories. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, IEEE, Munich, Germany, 1–12.

[3] George Gomes Cabral and Leandro L. Minku. 2023. Towards Reliable Online Just-in-time Software Defect Prediction. *IEEE Transactions on Software Engineering* 49, 3 (2023), 1342–1358. https://doi.org/10.1109/TSE.2022.3175789

[4] George G. Cabral, Leandro L. Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction. In *International Conference on Software Engineering*. Monteal, Canada, 666–676.

[5] Davide Chicco and Giuseppe Jurman. 2020. The Advantages of the Matthews Correlation Coefficient (MCC) over F1 Score and Accuracy in Binary Classification Evaluation. *BMC Genomics* 21 (2020).

[6] Davide Chicco, Matthijs J. Warrens, and Giuseppe Jurman. 2021. The Matthews Correlation Coefficient (MCC) is More Informative Than Cohen's Kappa and Brier Score in Binary Classification Assessment. *IEEE Access* 9 (2021), 78368–78381.

[7] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[8] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uira Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2017), 641–657.

[9] Andrea Dal Pozzolo, Giacomo Boracchi, Olivier Caelen, Cesare Alippi, and Gianluca Bontempi. 2018. Credit Card Fraud Detection: A Realistic Modeling and a Novel Learning Strategy. *IEEE Transactions on Neural Networks and Learning Systems* 29, 8 (2018), 3784–3797.

[10] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.

[11] Karl Dyer, Robert Capo, and Robi Polikar. 2014. COMPOSE: A Semisupervised Learning Framework for Initially Labeled Nonstationary Streaming Data. *IEEE Transactions on Neural Networks and Learning Systems* 25 (2014), 12–26.

[12] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. 2021. The Impact of Mislabeled Changes by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1559–1586.

[13] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An Empirical Study of Just-in-Time Defect Prediction Using Cross-Project Models. In *Working Conference on Mining Software Repositories* (Hyderabad, India). 172–181.

[14] Joao Gama, Raquel Sebastiao, and Pedro Pereira Rodrigues. 2013. On Evaluating Stream Learning Algorithms. *Journal of Machine Learning* 90, 3 (2013), 317–346.

[15] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. 2014. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology* 23, 4 (2014).

[16] Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. 2022. Problems with SZZ and Features: An Empirical Study of the State of Practice of Defect Prediction Data Collection. *Empirical Software Engineering* 27, 2 (2022).

[17] Yasutaka Kamei, Takafumi Fukushima, Shane Mcintosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. 2016. Studying Just-in-Time Defect Prediction Using Cross-project Models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[18] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E Hassan. 2010. Revisiting Common Bug Prediction Findings Using Effort-aware Models. In *IEEE International Conference on Software Maintenance*. 1–10.

[19] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773.

[20] Sunghun Kim, E. James Whitehead, and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[21] Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. 2020. The Impact of Context Metrics on Just-In-Time Defect Prediction. *Empirical Software Engineering* 25 (2020), 890–939.

[22] Masanari Kondo, Yutaro Kashiwa, Yasutaka Kamei, and Osamu Mizuno. 2022 (in press). An Empirical Study of Issue-Link Algorithms: Which Issue-Link Algorithms Should We Use? *Empirical Software Engineering* 27, 6 (2022 (in press)). https://doi.org/10.1007/s10664-022-10120-x

[23] Miroslav Kubat, Robert Holte, and Stan Matwin. 1997. Learning When Negative Examples Abound. In *European Conference on Machine Learning*. 146–153.

[24] Ludmila I. Kuncheva and J. Salvador Sánchez. 2008. Nearest Neighbour Classifiers for Streaming Data with Delayed Labelling. In *IEEE International Conference on Data Mining*. 869–874.

[25] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[26] Minlong Lin, Ke Tang, and Xin Yao. 2013. Dynamic Sampling Approach to Training Neural Networks for Multiclass Imbalance Classification. *IEEE Transactions on Neural Networks and Learning Systems* 24, 4 (2013), 647–660.

[27] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering* 44, 5 (2018), 412–428.

[28] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. 2016. Studying High Impact Fix-Inducing Changes. *Empirical Software Engineering Journal* 21, 2 (2016), 605–641.

[29] Audris Mockus and David M. Weiss. 2000. Predicting Risk of Software Change. *Bell Labs Technical Journal* 5, 2 (2000), 169–180.

[30] Jacob Montiel, Jesse Read, Albert Bifet, and Talel Abdessalem. 2018. Scikit-Multiflow: A Multi-output Streaming Framework. *Journal of Machine Learning Research* 19, 72 (2018), 1–5.

[31] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *International Conference on Software Engineering (ICSE)*. 284–292.

[32] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *International Conference on Software Analysis, Evolution and Reengineering*. 380–390.

[33] Hien M. Nguyen, Eric W. Cooper, and Katsuari Kamei. 2011. Online Learning from Imbalanced Data Streams. In *International Conference of Soft Computing and Pattern Recognition*. 347–352.

[34] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How different are different diff algorithms in Git? Use–histogram for code changes. *Empirical Software Engineering* 25 (2020), 790–823.

[35] Christophe Rezk, Yasutaka Kamei, and Shane McIntosh. 2022. The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3297–3309.

[36] Gema Rodríguez-Pérez, Meiyappan Nagappan, and Gregorio Robles. 2022. Watch Out for Extrinsic Bugs! A Case Study of Their Impact in Just-In-Time Bug Prediction Models on the OpenStack Project. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1400–1416.

[37] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit Guru: analytics and risk prediction of software commits. In *International Symposium on the Foundations of Software Engineering*. 966–969.

[38] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An Industrial Study on the Risk of Software Changes. In *International Symposium on the Foundations of Software Engineering*. 1–11.

[39] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[40] Liyan Song, Shuxian Li, Leandro L. Minku, and Xin Yao. 2022. A Novel Data Stream Learning Approach to Tackle One-Sided Label Noise From Verification Latency. In *International Joint Conference on Neural Networks (IJCNN)*. 1–8.

[41] Liyan Song and Leandro L. Minku. 2023. A Procedure to Continuously Evaluate Predictive Performance of Just-In-Time Software Defect Prediction Models During Software Development. *IEEE Transactions on Software Engineering* 49, 2 (2023), 646–666. https://doi.org/10.1109/TSE.2022.3158831

[42] Vinicius M.A. Souza, Diego F. Silva, Gustavo E.A.P.A. Batista, and João Gama. 2015. Classification of Evolving Data Streams with Infinitely Delayed Labels. In *International Conference on Machine Learning and Applications (ICMLA)*. 214–219.

[43] Sadia Tabassum, Leandro L. Minku, Danyi Feng, George G. Cabral, and Liyan Song. 2020. An Investigation of Cross-Project Learning in Online Just-in-Time Software Defect Prediction. In *International Conference on Software Engineering*. 554–565.

[44] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. In *International Conference on Software Engineering*. 99–108.

[45] Alexander Tarvo, Nachiappan Nagappan, Thomas Zimmermann, Thirumalesh Bhat, and Jacek Czerwonka. 2013. Predicting Risk of Pre-Release Code Changes with Checkinmentor. In *International Symposium on Software Reliability Engineering*. 128–137.

[46] Andras Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[47] Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2020. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1241–1266.

[48] Shuo Wang, Leandro L. Minku, and Xin Yao. 2018. A Systematic Study of Online Class Imbalance Learning With Concept Drift. *IEEE Transaction on Neural Networks and Learning Systems* 29, 10 (2018), 4802–4821.

[49] Chadd Williams and Jaime Spacco. 2008. SZZ Revisited: Verifying When Changes Induce Fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*. 32–36.

[50] Limin Yang, Xiangxue Li, and Yu Yu. 2017. Vuldigger: A Just-In-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes. In *IEEE Global Communications Conference*. 1–7.

[51] Xingguang Yang, Huiqun Yu, Guisheng Fan, Kai Shi, Liqiong Chen, and Emiliano Tramontana. 2019. Local versus Global Models for Just-In-Time Software Defect Prediction. *Scientific Programming* 2019 (2019), 1–13.

[52] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-Aware Just-in-Time Defect Prediction: Simple Unsupervised Models Could Be Better than Supervised Models. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 157–168.

[53] Y. Zhao, K. Damevski, and H. Chen. 2023. A Systematic Survey of Just-in-Time Software Defect Prediction. *Comput. Surveys* 55, 10 (2023), 201.1–201.35.

[54] Qiuming Zhu. 2020. On the Performance of Matthews Correlation Coefficient (MCC) for Imbalanced Dataset. *Pattern Recognition Letters* 136 (2020), 71–80.

[55] Xiaoyan Zhu, Chenyu Yan, E James Whitehead Jr, Binbin Niu, Lei Zhu, and Long Pan. 2022. Just-In-Time Defect Prediction for Software Hunks. *Software: Practice and Experience* 52, 1 (2022), 130–153.

Received 2023-02-02; accepted 2023-07-27