An Investigation of Cross-Project Learning in Online Just-In-Time Software Defect Prediction

Sadia Tabassum
sxt901@cs.bham.ac.uk
University of Birmingham, UK

Leandro L. Minku*
L.L.Minku@cs.bham.ac.uk
University of Birmingham, UK

Danyi Feng
danyi@ouchteam.com
Xiliu Tech, China

George G. Cabral
george.gcabral@ufrpe.br
Federal Rural University of Pernambuco, Brazil

Liyan Song
L.Song.1@cs.bham.ac.uk
University of Birmingham, UK

ABSTRACT

Just-In-Time Software Defect Prediction (JIT-SDP) is concerned with predicting whether software changes are defect-inducing or clean based on machine learning classifiers. Building such classifiers requires a sufficient amount of training data that is not available at the beginning of a software project. Cross-Project (CP) JIT-SDP can overcome this issue by using data from other projects to build the classifier, achieving similar (not better) predictive performance to classifiers trained on Within-Project (WP) data. However, such approaches have never been investigated in realistic online learning scenarios, where WP software changes arrive continuously over time and can be used to update the classifiers. It is unknown to what extent CP data can be helpful in such situation. In particular, it is unknown whether CP data are only useful during the very initial phase of the project when there is little WP data, or whether they could be helpful for extended periods of time. This work thus provides the first investigation of when and to what extent CP data are useful for JIT-SDP in a realistic online learning scenario. For that, we develop three different CP JIT-SDP approaches that can operate in online mode and be updated with both incoming CP and WP training examples over time. We also collect 2048 commits from three software repositories being developed by a software company over the course of 9 to 10 months, and use 19,8468 commits from 10 active open source GitHub projects being developed over the course of 6 to 14 years. The study shows that training classifiers with incoming CP+WP data can lead to improvements in G-mean of up to 53.90% compared to classifiers using only WP data at the initial stage of the projects. For the open source projects, which have been running for longer periods of time, using CP data to supplement WP data also helped the classifiers to reduce or prevent large drops in predictive performance that may occur over time, leading to up to around 40% better G-Mean during such periods. Such use of CP data was shown to be beneficial even after a large number of WP data were received, leading to overall G-means up to 18.5% better than those of WP classifiers.

*Corresponding author.

,

KEYWORDS

Software defect prediction, cross-project learning, transfer learning, online learning, verification latency, concept drift, class imbalance

1 INTRODUCTION

The primary objective of software quality assurance activities is to reduce the number of defects in software products [20]. It is a challenging problem considering the limitation of budget and time allocation for such activities. Software Defect Prediction (SDP) helps to reduce the time and effort required for testing software products. Different machine learning approaches have been proposed for SDP [11]. Many studies have focused on identifying defect prone components (e.g., modules or files) [10]. Recent studies in this area have been increasingly focusing on identifying defect-inducing software changes. This is known as Just-In-Time Software Defect Prediction (JIT-SDP) [16]. Advantages of JIT-SDP over component level SDP include [16]: (1) prediction made at an early stage, facilitating code inspection, (2) finer granularity of the predictions, making it easier to find defects, and (3) straightforward allocation of developers to inspect the code.

Similar to SDP at the module/file level, JIT-SDP classifiers require sufficient amount of training data which is not available during the initial phase of a project. To overcome this problem, previous work has proposed Cross-Project (CP) JIT-SDP, where historical data from other projects are used to train the classifier [15]. Existing CP JIT-SDP work [15] assumes an offline learning scenario, where classifiers are built based on a pre-existing training set and never updated anymore. This means that the classifier is never updated with Within-Project (WP) data. However, in practice, JIT-SDP is an online learning problem [2], where both additional CP and WP training examples arrive over time.

The role of CP data in such online learning scenario is unclear. CP JIT-SDP has never been investigated in online mode before. In particular, it is unknown whether CP data is only helpful at the very early stages of the project when there is little WP data, or it brings a prolonged benefit to the predictive performance of the classifier. For instance, it may be that CP approaches using an augmented training data stream formed by both WP and CP examples lead to increased predictive performance even at later stages of the project, given that classifiers are built using more data than WP classifiers trained only with WP data. Or, it may be that CP data cause such approaches to obtain worse predictive performance than WP classifiers once enough WP data is used for training. Besides, prediction quality can fluctuate due to variations (concept drifts) in the underlying defect generating process [19]. Concept drift can cause the predictive performance of JIT-SDP classifiers to drop [19]. The use of CP training data could potentially help to handle concept drift. This is specially the case considering that JIT-SDP is a class imbalance problem, where the number of defect-inducing software changes is typically much smaller than that of clean changes. In such type of problem, it would take a lot of time to collect new WP defect-inducing examples to recover well from concept drift. CP training examples could potentially help to recover from concept drift more quickly.

This paper thus aims at investigating when and to what extent CP JIT-SDP data can be helpful in a realistic online learning scenario. It answers the following Research Questions (RQs):

RQ1 Can CP data help to improve predictive performance in the initial phase of a project, when there is no or little WP training data available? For how long and to what extent?
RQ2 Can CP data help to prevent sudden drops in predictive performance, which may be caused by concept drift? To what extent?
RQ3 What is the overall effect of using CP and WP data together on the predictive performance throughout the development of software projects? In particular, do classifiers trained on both CP and WP data improve, deteriorate or retain similar overall predictive performance to WP classifiers?

It is worth noting that, in online scenarios, both WP and CP training data arrive over time as their collection can be automated. If incoming CP data is used for training a CP classifier, there is no reason to exclude incoming WP data from the training process of this classifier, as such WP data should not hurt its predictive performance. Therefore, we use the term CP when referring to online learning approaches that make use of both incoming CP and WP data for training. This means that, when we refer to the benefits of CP data, we do not mean the benefits of CP data in isolation, but the benefits of CP data used along with incoming WP data. We use the term WP when referring to approaches that only use WP data for training.

To answer the above research questions, this paper investigates three CP online learning approaches: (1) a single online learning classifier trained on incoming WP and CP training examples, (2) an online learning ensemble where each classifier is trained on incoming training examples from a different project, and (3) a single online learning classifier that filters out CP training examples that are likely to be very different from the recent WP examples. These approaches are enhancements of the approaches used in the JIT-SDP literature [15], so that they can operate in online mode. They are compared against online WP approaches. Our experiments based on 13 software repositories show that the first and third approaches are helpful to improve predictive performance in JIT-SDP compared to WP classifiers, while the second is not.

The contributions of this work are the following:

- This paper provides the first investigation of CP JIT-SDP in a realistic online learning scenario.
- We show how to adapt CP JIT-SDP approaches so that they can be used in an online learning scenario.
- We reveal that the use of CP data combined with WP data can improve overall predictive performance (rather than just achieving similar predictive performance) compared to WP learning in JIT-SDP.
- We show that CP data can be helpful for prolonged periods of time, rather than only in the beginning of the learning period as assumed in previous work.
- We show that CP data can reduce the negative effect of sudden predictive performance drops in the classifier, resulting in more reliable predictions over time.
- We show that it is better to use both CP and WP data together to build a single classifier, rather than creating different classifiers using disjoint subsets of the data.

This paper is further organized as follows. Section 2 presents related work. Section 3 introduces our online JIT-SDP approaches. Section 4 presents the details of the investigated datasets. Section 5 explains the experimental setup for answering the RQs. Section 6 explains the results of the experiments. Section 7 presents threats to validity. Section 8 presents the conclusions and implications of this work.

2 RELATED WORK

There are many SDP studies [11, 18, 32], including recent studies investigating class imbalance techniques [1], automated feature engineering [17], ensemble learning [34], among others. In this section, we discuss three main research areas of SDP that are closely related to this work: CP SDP at the component level (Section 2.1), CP JIT-SDP (Section 2.2) and Online JIT-SDP (Section 2.3).

2.1 CP SDP at the Component Level

There have been several studies on CP SDP at the component level. An initial study provided guidelines for choosing training projects [35]. They proposed an approach to identify factors that influence CP prediction success, such as data and process factors. Another study [13] showed that carefully selected CP training data may provide better prediction results than training data from the same project. Peters et al. [12] focused on selecting suitable CP training data based on the similarities between the distribution of the test and potential training data. In particular, they used similarity measuring and feature subset selection to remove irrelevant training data. Canfora et al. [3] proposed a multi-objective approach for CP defect prediction. They attempted to achieve a compromise between amount of code to inspect and number of defect-prone artifacts. This approach performed better than WP models. Panichella et al. [24] analysed the equivalence of different defect predictors and proposed a combined approach CODEP (COmbined DEfect Predictor) that uses machine learning to combine different and complementary classifiers. This combination performs better than the stand-alone CP technique. Nam et al. [23] applied Transfer Component Analysis (TCA) to CP SDP. TCA is a transfer learning approach that maps the data to a common latent space where CP and WP data are similar to each other. They also proposed a new approach called TCA+, which selects suitable normalisation options for TCA. Other studies [14, 26, 27] consider class imbalance learning for CP SDP. For instance, Ryu et al. [26] proposed an approach that uses similarity weight drawn from distributional characteristics and the asymmetric misclassification cost to balance imbalanced distributions.

Overall, these studies demonstrate that data distributional characteristics are important for CP SDP. In particular, they proposed approaches to select CP data that are similar to WP data, or to map CP and WP data into a latent space where they are similar. However, none of these studies were in the context of JIT-SDP or online SDP.

2.2 CP JIT-SDP

The first CP JIT-SDP study was done by Kamei et al. [15]. They carried out an empirical evaluation of the JIT-SDP performance by using data from 11 open source projects. They investigated five CP JIT-SDP approaches based on project similarity, three variations of data merging approaches, and ensemble approaches where each model was trained on data from a different project. All approaches were based on random forests as base learners. They found that simple merging of all CP data into a single training set and ensemble approaches obtained similar predictive performance to that of WP models. Different from SDP at the component level, other more complex approaches, including similarity-based approaches, did not offer any advantage compared to these.

Another study [4] investigated CP JIT-SDP in mobile platforms using 14 apps and 42,543 commits extracted from the Commit.Guru platform [25]. They compared CP performance of four different well-known classifiers and four ensemble techniques. Naive Bayes performed best compared to other classifiers and some ensemble techniques.

Chen et al. [5] considered JIT-SDP as a multi-objective problem to maximise the number of identified defect-inducing changes while minimising the effort required to fix the defects. They proposed a multi-objective optimization-based supervised method called MULTI to build logistic regression-based JIT-SDP models. They used six open source projects. MULTI was evaluated on three different model performance evaluation scenarios (cross-validation, cross-project-validation, and timewise-cross-validation) against 43 state-of-the-art supervised and unsupervised methods. They found that it can perform significantly better compared to WP methods.

Despite showing that CP JIT-SDP can obtain promising results compared to WP JIT-SDP, none of the studies above considered a realistic online learning scenario.

2.3 Online JIT-SDP

Few studies explored JIT-SDP in online mode. McIntosh et al. [19] performed a longitudinal case study of 37,524 changes from the rapidly evolving QT and OPENSTACK systems and found that fluctuations in the properties of fix-inducing changes can impact the performance of JIT models. They showed that JIT models typically lose power after one year. Hence, the JIT model should be updated with more recent data.

Tan et al. [29] investigated JIT-SDP in a scenario where new batches of training examples arrive over time and can be used for updating the predictive models, using one proprietary and six open source projects. They considered the fact that the labels of training data may arrive much later than the commit time. This is known as verification latency in the machine learning literature [7]. They used resampling techniques to deal with the class imbalanced data issue and updatable classification to learn over time. However, their approach assumes that there is no concept drift, i.e., that the defect generating process does not suffer variations over time.

Cabral et al. [2] proposed a method called Oversampling Online Bagging (ORB) to tackle class imbalance evolution in an online JIT-SDP scenario taking verification latency into account. Class imbalance evolution is a type of concept drift where the proportion of defect-inducing and clean examples fluctuates over time. ORB has an automatically adjustable resampling rate to tackle class imbalance evolution, being able to improve predictive performance over JIT-SDP approaches that assume a fixed level of class imbalance.

None of the online JIT-SDP studies investigated CP JIT-SDP.

3 ONLINE CP JIT-SDP APPROACHES

In this section, we modify and enhance three CP JIT-SDP approaches adopted in [15] to enable them to be applied to online JIT-SDP. All our algorithms fully respect chronology. In particular, they never use future CP/WP training examples, future knowledge about labels (i.e., defect-inducing or clean), or test examples, for training a model used for testing the present.

Training examples are generated using the online procedure recommended by Cabral et al. [2] to take verification latency into account for all approaches studied in this paper. A software change becomes a training example either when a defect is found to be associated to it, or once a pre-defined waiting period w has passed, whichever is earlier. This waiting period represents the amount of time that it takes for one to be confident that the change in question is clean. In other words, if no defect is found to be associated to the software change during the waiting period, a training example of the clean class is created to represent this software change. Otherwise, a training example of the defect-inducing class is created immediately after the defect is found. If, after the waiting period, a defect is found to be associated to a change that was previously considered clean, a new defect-inducing training example is created for it. Training examples are used to update the classifier as soon as they are created.

It is worth noting that, for all our CP approaches, classifiers can be trained not only with CP and WP training examples that are made available over time after the first WP commit, but also with CP training examples produced before the first WP commit.

3.1 All-in-One Approach

Existing offline JIT-SDP work [15] assume that CP classifiers are created only with CP data. Unlike offline approaches, the All-in-One online approach can use both CP and WP data for training. All incoming CP and WP training data are considered as part of a single data stream of training examples, which are used to train a single online classifier as soon as they are produced. The data stream is in chronological order, i.e., the training examples are sorted based on the unix timestamp of their creation. Algorithm 1 shows the pseudocode for the All-in-One approach.

The predictive model is initialised as an empty model that always predicts "clean". When a new incoming change x_p^t is received at

Algorithm 1 All-in-One Approach

```
1: $S$ = stream of incoming changes from several projects, $b$ = index
   identifying the target project, $w$ = waiting period
2: Initialise predictive model $m$
3: for each incoming change $x_p^t \in S$ do // $x_p^t$ is a change arriving
   from project $p$ at timestamp $t$
4:     if $p = b$ then
5:         $\hat{y} \leftarrow$ predict($m$, $x_p^t$)
6:     end if
7:     store $x_p^t$ in a queue $WFL\text{-}Q$      // $WFL\text{-}Q$ is a queue of
   incoming examples waiting to be used for training
8:     for each item $q^i$ in $WFL\text{-}Q$ do
9:         if a defect was linked to $q^i$ at a timestamp $\leq t$ then
10:            create a defect-inducing $training\_example$ for $q^i$
11:            train($m$, $training\_example$)
12:            remove $q^i$ from $WFL\text{-}Q$
13:        else
14:            if $q^i$ is older than $w$ then
15:                create a clean $training\_example$ for $q^i$
16:                train($m$, $training\_example$)
17:                remove $q^i$ from $WFL\text{-}Q$
18:                store $training\_example$ in $CL\text{-}H$    // $CL\text{-}H$ is a
   hash of clean training examples
19:            end if
20:        end if
21:    end for
22:    if a defect was linked to a $training\_example$ in $CL\text{-}H$ at a
   timestamp $\leq t$ then
23:        Swap the label of $training\_example$ to defect-inducing
24:        train($m$, $training\_example$)
25:        remove $training\_example$ from $CL\text{-}H$
26:    end if
27: end for
```

time step t (line 3), the algorithm first checks if this change belongs to the target project, i.e., to the project whose changes are being predicted (line 4). If it does, a prediction is provided for this change. After that, x_p^t is stored in a queue for a pre-defined waiting period (line 7). All changes in this queue are checked to see whether they can be used for training (line 8 to 21). If a defect is found to be associated to a given change in the queue during the waiting period (lines 9 to 12), a defect-inducing training example is created to represent this change used for training. If a defect is not found by the end of the waiting period of a given change (lines 13 to 20), a clean training example is created for this change and used for training. After that the change is removed from the queue. The algorithm also checks whether there is any past change found to be defect-inducing, but that was previously considered as a clean training example (lines 22 to 26). If there is, the classifier is trained using that change as a defect-inducing training example.

The key difference between the proposed approach and the data merging approaches used by Kamei et al. [15] is that, in [15], the learning was offline (without taking into account incoming training examples and verification latency) and they used only CP data for training. In the proposed approach, the learning is online, takes verification latency into account, and the classifier is trained on both CP and WP data whose labels are produced before the current timestamp.

3.2 Ensemble Approach

The Ensemble approach uses an ensemble of classifiers rather than a single classifier. A separate classifier is built from each project's separate training data stream (e.g., for 10 projects there will be 10 different classifiers). This includes both CP and WP data streams. Each change belonging to the target project is then predicted by all the classifiers, and the mean of the predicted probabilities retrieved by the classifiers is calculated. This mean is used to predict whether the change is clean or defect-inducing. The pseudocode for the Ensemble approach is similar to that of the All-in-One approach, and can be found in the supplementary material [28]. As with the All-in-One approach, the chronological order of the training examples is always respected.

In the previous offline ensemble approach [15], Kamei et al. investigated both simple voting ensembles (where equal weight is given to each classifier) and weighted voting ensembles (where more weight is given to classifiers trained on projects that are more similar to the target project). They showed that weighted voting did not offer any advantage over simple voting. Hence, our online ensemble approach uses simple voting. The key differences between our approach and the approach used by Kamei et al. [15] are that our approach is online, and our ensemble contains a classifier built from WP training examples that have been labelled up to the current timestamp, rather than using only CP data classifiers.

3.3 Filtering Approach

Even though filtering did not improve predictive performance in offline JIT-SDP [15], filtering strategies have shown to be very beneficial in offline SDP at the component level [30]. Therefore, we investigate whether filtering out software changes that are dissimilar to the target changes could be useful in online JIT-SDP.

We proposed the following Filtering approach for online JIT-SDP. First, a fixed-size window of most recent incoming WP training examples is maintained. Whenever a CP training example arrives, it is compared with the training examples in the WP window. As with the All-in-One and Ensemble approaches, the chronological order of the training examples is always respected. Euclidean distances between the input features of the CP training example and each of the WP training examples in the window that have the *same label* as the CP training example are calculated to check how similar the CP training example is to recent WP examples. It is important to use only training examples with the same label to compute the distance. If the labels had been ignored, the approach would consider that a CP clean training example described by similar features as a WP defect-inducing training example are similar training examples. However, they are different due to the different label.

The average of the smallest K distances is calculated. If this average distance is equal to or lower than a maximum threshold, the CP training example is allowed to train the classifier. Discarded

Algorithm 2 Filtering Approach

```
1: $S$ = stream of incoming changes from several projects, $b$ = index identifying the test project, $w$ = waiting period, windowSize = size of the $WP\text{-}Q$ sliding window, $K$ = number of top short distances to be used, $maxDist$ = distance threshold for similarity, $cpqSize$ = maximum size of the queue $CP\text{-}Q$ of dissimilar CP instances to be re-checked for similarity in the future
2: Initialise predictive model $m$
3: for each incoming change $x_p^t \in S$ do // $x_p^t$ is a change arriving from project $p$ at timestamp $t$
4:     if avgDist($training\_example$, $WP\text{-}Q$,$K$) $\leq$ $maxDist$ for any $training\_example$ in $CP\text{-}Q$ then
5:         train(m, $training\_example$)
6:         remove $training\_example$ from $CP\text{-}Q$
7:     end if
8:     if $p = b$ then
9:         $\hat{y}$ = predict($m$, $x_p^t$)
10:    end if
11:    store $x_p^t$ in a queue $WFL\text{-}Q$          // $WFL\text{-}Q$ is a queue of incoming examples waiting to be used for training
12:    for each item $q^i$ in $WFL\text{-}Q$ do
13:        if a defect was linked to $q^i$ at a timestamp $\leq t$ then
14:            Create a defect-inducing $training\_example$ for $q^i$
15:        else
16:            if $q^i$ is older than $w$ then
17:                Create a clean $training\_example$ for $q^i$
18:                store $q^i$ in $CL\text{-}H$      // $CL\text{-}H$ is a hash of clean training examples
19:            end if
20:        end if
21:        if a $training\_example$ was created for $q^i$ then
22:            if avgDist($training\_example$, $WP\text{-}Q$,$K$) $\leq$ $maxDist$ or this is a WP change then
23:                train($m$, $training\_example$)
24:            else
25:                Add $training\_example$ to $CP\text{-}Q$
26:            end if
27:            Remove $q^i$ from $WFL\text{-}Q$
28:            Slide $WP\text{-}Q$ if $training\_example$ is WP
29:        end if
30:    end for
31:    if a defect was linked to a $training\_example$ in $CL\text{-}H$ before time $t$ then
32:        swap label of $training\_example$ to defect-inducing
33:        remove $training\_example$ from $CL\text{-}H$
34:        if avgDist($training\_example$,$WP\text{-}Q$,K) $\leq$ $maxDist$ then
35:            train($m$, $training\_example$)
36:        else
37:            add $training\_example$ to $CP\text{-}Q$
38:        end if
39:    end if
40: end for
```

CP training examples are kept in a fixed-sized queue. This queue is checked in every iteration to see whether any old discarded CP training example has now become suitable for training. This can be useful in case concept drift causes such discarded examples to become relevant.

Algorithm 2 shows the pseudocode for the Filtering approach. The predictive model is initialised as an empty model that always predicts "clean". When a new incoming change x_p^t is received at time step t (line 3), the algorithm checks whether there are any old CP training examples that were previously not used for training due to their dissimilarity to WP examples, but that are now suitable for training due to their similarity to the current WP sliding window (line 4 to 7). Then, a prediction is given if the change x_p^t belongs to target project (line 8 to 10). The change x_p^t is then stored in a queue (line 11), waiting to be labelled. All changes in this queue are checked to see whether they can be labelled (lines 12 to 30). If they can, corresponding training examples are created and used for training only if they are similar enough to the WP sliding window (lines 21 to 23). If they are not similar enough and are CP examples, they are stored in the queue $CP\text{-}Q$ of discarded CP examples for possible future use (line 25). The sliding window is updated (slided) if the training example is WP (line 28). The algorithm also checks whether any change that was previously considered as clean has now been associated to a defect and uses it for training, but only if it is similar enough to the WP sliding window (line 31 to 39).

4 DATASETS

We have extracted data from three proprietary software development project repositories from a Chinese software development company for the purpose of this study. We have also used ten existing datasets extracted from open source GitHub projects, which were made available by Cabral et al. [2] at https://zenodo.org/record/2594681. Some information on the datasets is shown in Table 1. All datasets were extracted based on Commit Guru [25]. The change metrics include 14 metrics that can be grouped into five types of metrics: i) diffusion of the change, ii) size of the change, iii) purpose of the change, iv) history of the change, and v) experience of the developer that made the change. These change metrics have been shown to be adequate for JIT-SDP in previous work [16].

It is worth noting that there is evidence of concept drift that can be attributed to software engineering in the datasets. For instance, in Tomcat, the number of developers changing the modified files associated to a commit increases as the project matures during the first 16,000 commits, then drops. Upon dropping, new changes are usually clean, different from old ones with similar change metrics.

To collect the proprietary data for this study, all commit messages from the three repositories were extracted using the git log command. A Chinese language native speaker knowledgeable of programming was asked to read the commit messages to identify keywords that can be used to identify *corrective* commits. Keywords, representative commit messages of corrective and non-corrective cases, and commit messages for which the native speaker was uncertain about were stored in a separate file. This gave a total of 57 commit messages. The second author of this paper then went through the file providing his independent classification of the commits as corrective or non-corrective, by using Google Translate to translate the commit messages. Unclear Google translations were

Table 1: An overview of the projects

Project	Total Changes	# Defect-inducing Changes	% Defect-inducing Changes	Median Defect Discovery Delay (days)	Time Period	Main Language	Project Type
Tomcat	18907	5207	27.54	200.5798	27-03-2006 - 06-12-2017	Java	Open source [2]
JGroups	18325	3153	17.21	116.1565	09-09-2003 - 05-12-2017	Java	Open source [2]
Spring-integration	8750	2333	26.66	415.1201	14-11-2007 - 16-01-2018	Java	Open source [2]
Camel	30575	6255	20.46	28.1947	19-03-2007 - 07-12-2017	Java	Open source [2]
Brackets	17364	4047	23.31	14.454	07-12-2011 - 07-12-2017	JavaScript	Open source [2]
Nova	48989	12430	25.37	88.5615	28-05-2010 - 28-01-2018	Python	Open source [2]
Fabric8	13106	2589	19.75	39.1833	13-04-2011 - 06-12-2017	Java	Open source [2]
Neutron	19522	4607	23.6	82.5097	01-01-2011 - 27-12-2017	Python	Open source [2]
Npm	7920	1407	17.77	111.514	29-09-2009 - 28-11-2017	JavaScript	Open source [2]
BroadleafCommerce	15010	2531	16.86	42.5818	19-12-2008 - 21-12-2017	Java	Open source [2]
C1	1030	confidential	confidential	35.81	06-09-2018 - 17-07-2019	JavaScript	Proprietary
C2	601	confidential	confidential	1.98	16-10-2018 - 19-07-2019	JavaScript and 3D Studio	Proprietary
C3	417	confidential	confidential	2.97	05-09-2018 - 19-07-2019	Python	Proprietary

discussed with the native speaker. After that, the native speaker and the second author met to discuss all 57 commit messages. Commit messages for which there was still some doubt after this discussion were further discussed with the Company, who confirmed whether they were corrective or non-corrective. The native speaker and second author then agreed on a list of keywords to identify commits, which was presented to the Company. The Company confirmed that the list of keywords was adequate for their projects.

The list of keywords was used as input for Commit Guru to identify corrective commits, which were then used to identify which changes are defect-inducing or clean [25], to generate the data sets. As a data quality assurance procedure, all commit messages considered by Commit Guru as corrective and a sample of non-corrective commit messages were double checked by the native speaker, giving a total of 1,230 commit messages. There was a disagreement between Commit Guru's classification and the native speaker in only 3 cases, and the native speaker was unsure of the correct classification in 7 cases. Therefore, Commit Guru's classification of commits as corrective and non-corrective was deemed appropriate.

5 EXPERIMENTAL SETUP

The RQs introduced in Section 1 will be answered by comparing the predictive performance of the All-in-One, Ensemble and Filtering approaches against WP learning. The analysis done for RQ1, RQ2 and RQ3 will concentrate on the predictive performance (1) in the beginning of the projects, (2) during periods of time where we can observe sudden drops in predictive performance of the WP approach, and (3) average across time steps, respectively. We define a time step as a sequential number indicating the order of WP commits. Each WP commit requires a WP software change to be predicted as defect-inducing or clean. Due to the poor results obtained by the Ensemble approach on the open source data, this approach was not run for the proprietary data.

Given an open source project repository, all other 9 open repositories are considered as the CP data. Given a proprietary repository, two cases were considered: (1) the other 2 proprietary repositories are the CP data, and (2) all other 12 repositories are the CP data.

JIT-SDP is a class imbalance problem [2, 15, 16, 29], and the open source data used in this study are known to be class imbalanced [2]. Therefore, learning approaches need to use online learning classifiers that can deal with this issue. Two state-of-the-art approaches for online class imbalanced learning are Improved Oversampling Online Bagging (OOB) and Improved Undersampling Online Bagging (UOB) [33]. Also, Cabral et al. [2] proposed a new approach called Oversampling Rate Boosting (ORB), which improves the predictive performance further for JIT-SDP. The three approaches are ensembles of Hoeffding Trees [8]. These are online learning base classifiers, which are updated incrementally with each new training example, preserving old knowledge without requiring storage of old examples. Previously, OOB and UOB achieved similar performance to JIT-SDP [2], and ORB performed the best. Thus, only OOB and ORB are selected as the base classifiers in this study.

To evaluate the performance, we adopt recall on the clean class (Recall0), recall on the defect-inducing class (Recall1) and Geometric Mean of Recall0 and Recall1 (G-Mean). They were computed prequentially and using a fading factor to enable tracking changes in predictive performance over time, as recommended for problems that may suffer concept drift [9]. If the current example belongs to class i, $Recall_i^{(t)} = \theta Recall_i^{(t-1)} + (1-\theta)\mathbb{1}_{\hat{y}=i}$, where i is zero or one, t is the current time step, θ is a fading factor set to 0.99 as in [2], $\hat{y}$ is the predicted class, and $\mathbb{1}_{\hat{y}=i}$ is the indicator function, which evaluates to one if $\hat{y} = i$ and to zero otherwise. If the current example does not belong to class i, $Recall_i^{(t)} = Recall_i^{(t-1)}$. Also, $G\text{-}Mean^{(t)} = \sqrt{Recall_0^{(t)} \times Recall_1^{(t)}}$. It is worth noting that $Recall0 = 1 - FalseAlarmRate$, and so false alarms are taken into account through both Recall0 and G-Mean. These metrics were chosen because they are the most recently recommended for online class imbalance learning [33].

The predictive performances obtained during the initial phase of the projects, and the overall predictive performances calculated using all time steps will be compared across data sets using the Scott-Knott procedure [22], which ranks the models and separates them into clusters. This test is used to select the best subgroup among different models. Non-parametric bootstrap sampling is used to make the test non-parametric, as recommended by Menzies et al. [21]. As explained by Demsar [6], non-parametric tests are adequate for comparison across data sets. In addition, the Scott-Knott test adopted in this paper uses A12 effect size [31] to rule out any small differences in performance. Specifically, Scott-Knott only performed statistical tests to check whether groups should be separated if the A12 effect size was medium or large, as recommended in [21]. If the A12 effect size was not medium or large, groups were *not* separated. We will refer to Scott-Knott based on Bootstrap sampling and A12 as Scott-Knott.BA12. We have also included the A12 effect sizes for each dataset individually to support the analysis.

The parameters for the Filtering approach were chosen by performing grid search on the initial portion (1000 commits) of the datasets using the following set of values, where bold values were selected: windowSize= {**500**,600,700,1000}, K= {**50**,100,200}, maxDist= {0.6,**0.7**,0.8} and cpqSize= {**500**, 1000}. Parameters of OOB and ORB were kept to the same values as in [2] for open source datasets, as they have already been tuned for these datasets. Ensemble sizes and decay factors were further tuned for the proprietary datasets. The waiting period was 90 for the open source datasets as in [2] and 30 for the proprietary datasets, due to their lower defect discovery delay (see Table 1). Thirty executions of each approach with each of the base classifiers have been performed on each dataset.

6 EXPERIMENTAL RESULTS

6.1 RQ1: Initial Phase of the Project

We define the initial phase of the open source projects as the period of time ranging from the first time step until the time step where the G-Mean of the WP approach reaches the value of its average G-Mean across time steps. It represents the time it takes for the G-Mean of this approach to reach its typical values for a given project. For the proprietary data, the total number of time steps is too small to use the average G-Mean across time steps for this purpose. In particular, had longer periods of time been observed, the G-Mean values would be likely to improve further, given the trends in G-Mean at the end of the period analyzed for these projects (see Fig. 3, which will be discussed later in this section). Therefore, instead of using the average G-Mean across all time steps, we have used the average G-Mean across the last 40 time steps to determine the initial phase. Table 2 shows the number of time steps of the initial phase of all projects, as well as the average G-Mean of each approach, the effect size A12 against the corresponding WP approach, and the Scott-Knott.BA12 results during this period.

The initial phase of the open source projects lasted from 461 to 6271 time steps with a median of 1418, and from 900 to 6271 time steps with a median of 1553 when using OOB and ORB, respectively (Table 2). The average G-Mean of the All-in-One and Filtering approaches was frequently higher during the initial phase of the open source projects than that of the WP approach (up to 53.90% higher, for Brackets using All-in-One-ORB). This is further illustrated by Figs. 1 and 2, which show the G-Means across time steps. The G-Means of the WP classifiers were lower than those of All-in-One and Filtering in the initial phase of all plots, except for Figs. 1b, 1c and 2b, where the G-Means were similar. The superiority of All-in-One and Filtering is confirmed by Scott-Knott.BA12, which shows that these approaches were better ranked than the others in terms of G-Mean. A12 effect sizes against WP learning for individual datasets were typically large. The Ensemble approach sometimes achieved better G-Means than the WP approach at the very beginning of the projects, but performed worse than the other CP approaches during most of the initial phase (Table 2). These results are also supported Scott-Knott.BA12, which shows that the Ensemble approach was better ranked than the WP approach, but worse ranked than All-in-One and Filtering in terms of G-Mean.

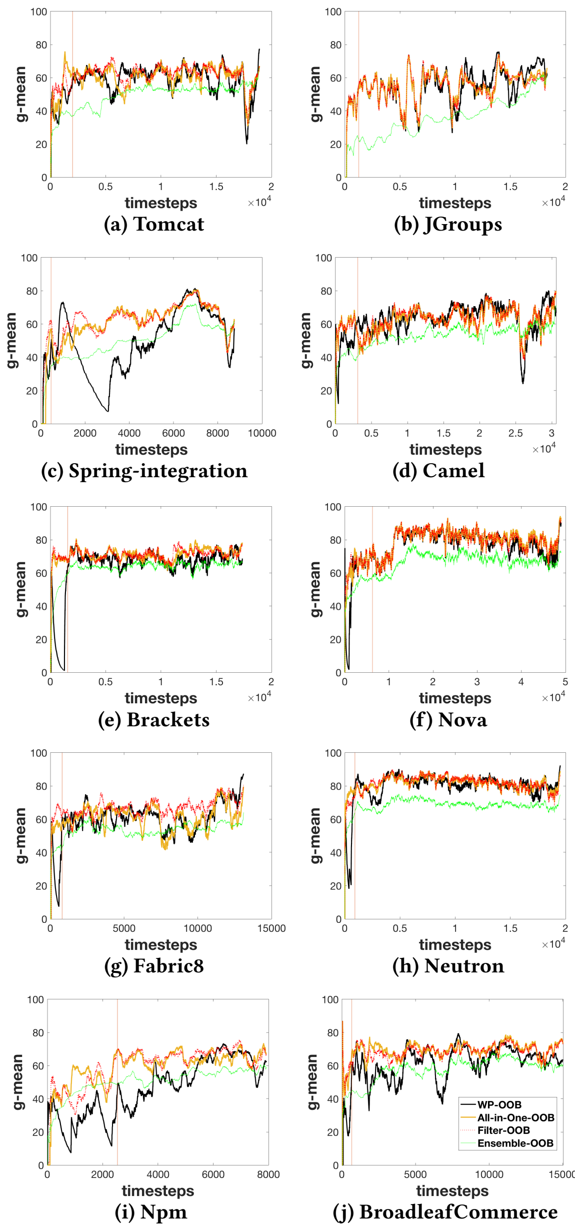

Figure 1: G-Mean for all datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project, shown in Table 2.

Given the promising results of the All-in-One and Filtering approaches, we investigated them further in the context of the proprietary data. All-in-One was investigated in two different ways: a) combining both open source and proprietary training data and b) only with proprietary data. The initial phase of the proprietary projects was typically much smaller than that of the open source projects, lasting from 81 to 581 time steps and from 82 to 581 time steps for OOB and ORB, respectively (Table 2). The G-Means were also much lower than for the (longer) initial phase of the open source projects. CP approaches helped to improve average G-Mean for OOB-based approaches, which is confirmed by the

Table 2: Number of initial time steps of the initial phase, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase

	OOB					ORB				
Open Source Data	**Initial Time Steps**	**WP**	**All-in-One**	**Filtering**	**Ensemble**	**Initial Time Steps**	**WP**	**All-in-One**	**Filtering**	**Ensemble**
Tomcat	2006	43.61	51.94[b]	**52.49**[b]	33.66[-b]	926	40.44	45.82[b]	48.12[b]	35.32[-b]
JGroups	1268	38.6	38.28[-s]	**39.01**[s]	16.64[-b]	1412	30.71	31.47[m]	33.47[b]	15.14[-b]
Spring-integration	461	27.3	24.27[-b]	38.69[b]	19.48[-b]	900	25.70	60.47[b]	**63.43**[b]	42.17[b]
Camel	3112	46.81	57.74[b]	57.06[b]	39.42[-b]	5111	49.68	**57.98**[b]	57.63[b]	41.38[-b]
Brackets	1569	21.36	64.88[b]	66.17[b]	46.83[b]	1721	13.49	**67.39**[b]	67.20[b]	50.45[b]
Nova	6271	55.42	63.15[b]	64.39[b]	51.66[-b]	6271	52.80	**66.43**[b]	65.0[b]	51.84[-s]
Fabric8	795	27.01	51.5[b]	58.63[b]	40.3[b]	1613	29.97	**63.33**[b]	62.89[b]	45.19[b]
Neutron	917	44.83	73.55[b]	67.29[b]	55.06[b]	3304	71.78	**75.09**[b]	74.84[s]	71.04[-b]
Npm	2536	26.91	**48.36**[b]	45.75[b]	42.01[b]	1494	30.12	40.07[b]	43.48[b]	43.52[b]
BroadleafCommerce	677	26.36	50.31[b]	**53.16**[b]	37.25[b]	950	27.68	51.81[b]	52.40[b]	41.68[b]
Ranking		4	2	1	3		4	1	1	3
Proprietary Data	**Initial Time Steps**	**WP**	**All-in-One (combined)**	**All-in-One (proprietary)**	**Filtering (combined)**	**Initial Time Steps**	**WP**	**All-in-One (combined)**	**All-in-One (proprietary)**	**Filtering (combined)**
C1	347	18.93	38.31[b]	17.28[-b]	**39.66**[b]	289	15.93	11.48[-b]	11.57[-b]	11.49[-b]
C2	581	25.03	40.75[b]	32.95[b]	40.23[b]	581	24.48	43.04[b]	36.03[b]	**44.01**[b]
C3	81	7.96	13.67[b]	**35.98**[b]	13.42[b]	82	8.48	0[-b]	0[-b]	0[-b]
Ranking		2	1	1	1		2	2	2	2

Top G-Means for each dataset are in bold. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the corresponding WP approach (WP-OOB or WP-ORB). Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the corresponding WP approach. Scott-Knott.BA12 was run for all OOB- and ORB-based approaches together. For each performance metric, one test was run for the open source, and one test was run for the proprietary data results. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

Scott-Knott.BA12 results and further illustrated in Fig. 3. For instance, in Fig. 3a, the WP approach obtained very low G-Mean of 8% around time step 290, while All-in-One (combined) and Filtering obtained a higher G-Mean of around 40%. Interestingly, All-in-One (combined) led to better results than All-in-One (proprietary) for C1 and C2 when using OOB and for C2 when using ORB, indicating that open source data can sometimes help to improve JIT-SDP predictions on proprietary data during the initial period.

However, the use of CP data for the proprietary projects was less helpful when using ORB-based approaches (see Table 2; plots in supplementary material [28]). It is possible that the whole period of time analyzed for these projects belongs to the initial phase, and that the last time step of the initial phase could not be precisely identified due to the lack of information on the typical G-Means that would be obtained by the WP approaches in prolonged periods of time, as was done for the open source data. As shown in Section 6.3, the G-Means obtained for ORB-based combined CP approaches improve when considering the whole period of the projects.

RQ1: CP data was helpful in the initial phase of the project when there was no or little WP training data available, in particular when using All-in-One and Filtering approaches for the open source projects and OOB-based approaches for the proprietary projects. This initial phase lasted from 461 to 6271 and from 81 to 581 time steps for the open source and proprietary projects, respectively. Improvements in average G-Mean were up to 53.90%, avoiding extremely low G-Means.

6.2 RQ2: Periods with Sudden Drops in WP Classifier's Predictive Performance

In some datasets, after the initial phase, the WP approach suffered periods of large drops in G-Means. Some clear cases can be observed from time steps 1000 to 3000 in Fig. 1c, near time step 25,000 in Fig. 1d, around time step 7000 in Fig. 1j, around time step 18,000 in Fig. 2a, from time step 1000 to 3000 in Fig. 2c, and around time step 2100 in Fig. 2i. The CP approaches frequently managed to reduce or sometimes even eliminate such drops.

For example, in Fig. 2c, we can see that from time steps 1000 to 3000, the WP classifier had a large fall of performance reducing the G-Mean to around 20%. During this period, All-in-One and Filtering managed to maintain a G-Mean of around 60%. The Ensemble approach also managed to avoid the drop in G-Mean, but did not perform so well as All-in-One and Filtering.

WP classifiers may suffer such drops in performance due to changes in the characteristics of WP training data over time. However, in CP learning, training data comes from different projects. Some of the CP training data may have similar distribution as the target project currently has, helping to reduce the negative effect of differences in the distribution over time. This is a potential reason for CP data to be helpful in case of sudden performance drops.

Interestingly, the Filtering approach managed to achieve a more stable G-Mean than All-in-One for Fabric8. This suggests that even though CP data may prevent performance drops resulting from changes in characteristics of the data, it might introduce other performance drops due to the use of too dissimilar CP data. Experiments with additional projects are needed to confirm that.

For the proprietary data, the period of time analysed was not long enough to identify large sudden drops in predictive performance after the initial phase. Hence, to understand whether CP data is helpful to prevent sudden drops in performance for proprietary data, future work on other proprietary projects should be performed.

RQ2: CP approaches frequently help to reduce or even prevent sudden drops in performance compared to WP approaches. In particular, the All-in-One and Filtering approaches obtained up to around 40% better G-Mean than the WP approach during such periods.

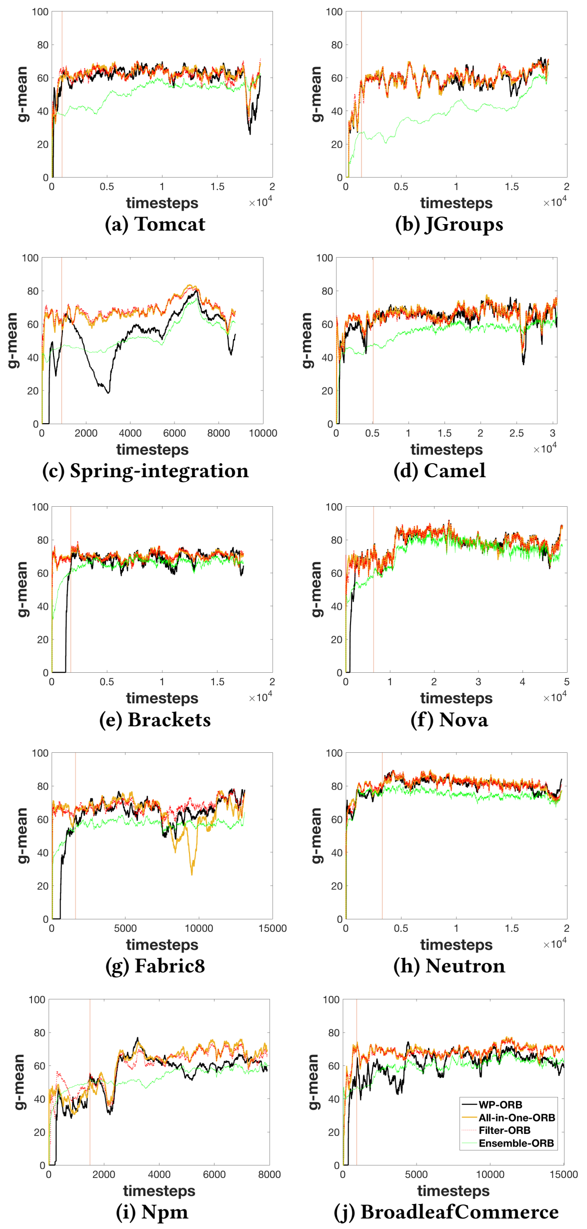

Figure 2: G-Mean for all datasets through time using ORB. The vertical red bar indicates the last time step of the initial phase of the project, shown in Table 2.

6.3 RQ3: Overall Predictive Performance

According to the Scott-Knott.BA12 test to rank the overall G-Mean of all OOB- and ORB-based approaches across open source datasets (Table 3), All-in-One-OOB, All-in-One-ORB, Filtering-OOB and Filtering-ORB ranked best, WP-ORB ranked second, WP-OOB ranked third, Ensemble-ORB ranked fourth and Ensemble-OOB ranked worst.

Table 3 shows that All-in-One-OOB's G-Means were up to 13.43% better (for Npm) and All-in-One-ORB's were up to 16.04% better (for Spring-integration). The improvements in average G-Mean

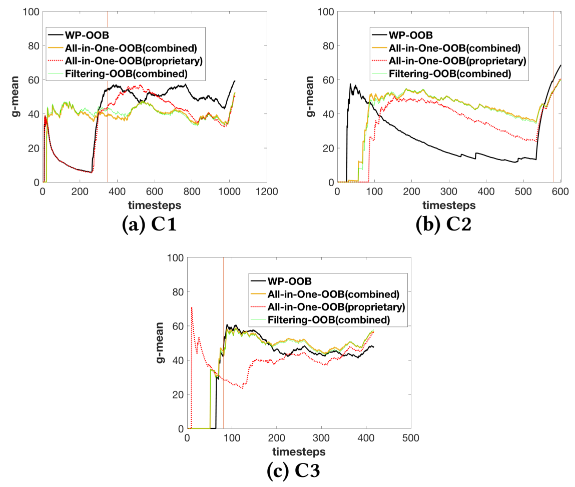

Figure 3: G-Mean for proprietary datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project, shown in Table 2.

for the open source data when using All-in-One-OOB compared with WP-OOB had large effect size in 8 out of 10 datasets, and the improvements when using All-in-One-ORB compared with WP-ORB had large effect size in all 10 datasets.

Filtering performed similarly to All-in-One. Table 3 shows that Filtering-OOB's G-Means were up to 13.97% better than WP-OOB's (for spring-integration), and Filtering-ORB's G-Means were up to 16.88% better than WP-ORB's (for spring-integration). The improvements in average G-Means when using All-in-One-OOB compared with WP-OOB had large effect size in 8 out of 10 datasets, and the improvements when using All-in-One-ORB compared with WP-ORB had large effect size in all 10 datasets. Therefore, even though Filtering improved the results over the WP approach, filtering instances dissimilar to the target project does not have a major impact on the performance of the classifier compared to All-in-One. As All-in-One merges all data together to build one classifier, it is possible that classifier performance mainly depends on the amount and potentially variety of training data rather than CP project similarity.

From that we can see that merging CP and WP data together to train a classifier (through All-In-One or Filtering) improved overall predictive performance in terms of G-Mean, rather than maintaining similar predictive performance as in offline [15] scenarios.

Although Ensemble approaches performed well in offline mode [15], our study shows that ensembles of classifiers trained on separate projects did not perform well in a realistic online scenario. Ensemble-OOB's G-Means were up to 19.25% worse than WP's (for JGroups), and Ensemble-ORB's were up to 19.85% worse than WP's (for JGroups). A further analysis of the results obtained by each classifier within the ensemble reveals that their individual G-Means were not high, which resulted in the poor G-Mean of the ensemble as a whole. This could be due to lack of enough training data for the individual classifiers, given that each classifier in the ensemble is trained with data from one project. This again suggests that the larger amount of varied data was crucial to improve predictive performance in online JIT-SDP. It also explains why the Ensemble approach worked in offline mode but not in online mode.

Table 3: Overall predictive performance, A12 effect sizes and Scott-Knott.BA12 statistical tests to compare learning approaches

Dataset	Approach	Recall0	Recall1	G-Mean	Dataset	Approach	Recall0	Recall1	G-Mean
Tomcat	WP-OOB	58.25(1.82)	63.59(1.53)	57.78(0.61)	Tomcat	WP-ORB	58.41(1.7)	64.49(1.19)	59.78(0.82)
	All-in-One-OOB	61.07(1.32)[b]	63.76(1.13)[-*]	59.85(0.47)[b]		All-in-One-ORB	61.98(1.17)[b]	63.38(0.73)[-b]	**61.72**(0.8)[b]
	Filtering-OOB	66.99(1.22)[b]	60.05(0.97)[-b]	61.57(0.52)[b]		Filtering-ORB	61.07(0.88)[b]	63.31(0.72)[-b]	61.25(0.47)[b]
	Ensemble-OOB	63.78(0.7)[b]	43.06(0.73)[-b]	49.12(0.2)[-b]		Ensemble-ORB	61.75(1.21)[b]	47.67(1.06)[-b]	50.82(0.34)[-b]
JGroups	WP-OOB	57.55(1.68)	59.01(1.51)	54.76(0.68)	JGroups	WP-ORB	61.18(0.7)	56.93(1.26)	57.2(0.79)
	All-in-One-OOB	64.54(1.58)[b]	52.85(1.46)[-b]	54.77(0.64)[*]		All-in-One-ORB	62.77(0.93)[b]	56.67(0.95)[*]	57.92(0.65)[b]
	Filtering-OOB	65.94(1.32)[b]	51.59(1.22)[-b]	54.68(0.42)[-s]		Filtering-ORB	63.05(1.06)[b]	57.21(1.24)[s]	**58.29**(0.67)[b]
	Ensemble-OOB	84.03(0.4)[b]	18.6(0.36)[-b]	35.51(0.29)[-b]		Ensemble-ORB	83.72(0.32)[b]	20.1(0.41)[-b]	37.35(0.32)[-b]
Spring-integration	WP-OOB	61.13(1.64)	56.09(1.31)	48.41(0.7)	Spring-integration	WP-ORB	71.92(0.98)	45.83(1.29)	51.73(0.77)
	All-in-One-OOB	65.32(0.61)[b]	61.57(0.86)[b]	60.62(0.29)[b]		All-in-One-ORB	67.51(0.84)[-b]	69.48(1.03)[b]	67.77(0.74)[b]
	Filtering-OOB	69.82(0.75)[b]	59.96(1.19)[b]	62.38(0.68)[b]		Filtering-ORB	68.57(0.82)[-b]	69.93(0.81)[b]	**68.61**(0.57)[b]
	Ensemble-OOB	73.09(0.57)[b]	37.6(0.63)[-b]	49.4(0.27)[-b]		Ensemble-ORB	74.61(0.47)[b]	39.98(0.66)[-b]	52.4(0.53)[b]
Camel	WP-OOB	56.04(1.23)	74.62(0.95)	62.45(0.57)	Camel	WP-ORB	59.59(1.16)	70.11(1.13)	62.92(0.96)
	All-in-One-OOB	53.89(1.13)[-b]	75.29(0.71)[m]	62.16(0.61)[-m]		All-in-One-ORB	59.12(0.51)[-m]	73.1(0.55)[b]	**65.0**(0.44)[b]
	Filtering-OOB	55.52(0.81)[-s]	74.29(0.71)[-s]	62.66(0.55)[s]		Filtering-ORB	58.58(0.61)[-b]	72.7(0.52)[b]	64.48(0.42)[b]
	Ensemble-OOB	55.25(0.87)[-m]	55.17(0.77)[-b]	52.63(0.24)[-b]		Ensemble-ORB	61.46(0.86)[b]	52.33(1.11)[-b]	54.26(0.5)[-b]
Brackets	WP-OOB	49.16(0.27)	89.6(0.23)	64.03(0.11)	Brackets	WP-ORB	61.31(0.73)	76.66(1.41)	63.31(0.48)
	All-in-One-OOB	65.47(0.99)[b]	79.68(1.3)[-b]	70.85(0.47)[b]		All-in-One-ORB	66.9(1.01)[b]	74.94(1.68)[-b]	69.98(0.86)[b]
	Filtering-OOB	69.15(0.83)[b]	75.44(1.28)[-b]	**70.91**(0.39)[b]		Filtering-ORB	66.34(0.9)[b]	75.35(1.31)[-b]	69.78(0.73)[b]
	Ensemble-OOB	64.89(0.76)[b]	62.81(1.11)[-b]	62.68(0.47)[-b]		Ensemble-ORB	61.7(1.07)[m]	69.14(1.21)[-b]	64.71(0.74)[b]
Nova	WP-OOB	68.18(0.24)	86.96(0.52)	75.55(0.2)	Nova	WP-ORB	74.25(1.47)	80.57(2.6)	75.42(0.64)
	All-in-One-OOB	70.04(0.32)[b]	88.9(0.53)[b]	**78.25**(0.2)[b]		All-in-One-ORB	74.15(0.87)[-*]	81.95(1.25)[b]	77.28(0.32)[b]
	Filtering-OOB	70.91(0.3)[b]	87.24(0.37)[s]	78.01(0.17)[b]		Filtering-ORB	72.72(0.21)[-b]	82.85(0.53)[b]	76.83(0.34)[b]
	Ensemble-OOB	75.88(0.78)[b]	57.23(1.59)[-b]	65.5(0.6)[-b]		Ensemble-ORB	76.76(0.74)[b]	68.42(1.77)[-b]	71.7(0.58)[-b]
Fabric8	WP-OOB	50.56(2.68)	75.72(2.17)	59.75(0.99)	Fabric8	WP-ORB	61.32(2.13)	67.83(1.3)	61.42(1.02)
	All-in-One-OOB	55.91(0.99)[b]	70.76(1.43)[-b]	60.92(0.47)[b]		All-in-One-ORB	57.39(1.41)[-b]	73.78(1.4)[b]	63.7(1.02)[b]
	Filtering-OOB	61.94(1.86)[b]	73.25(1.55)[-b]	66.55(0.62)[b]		Filtering-ORB	64.14(0.61)[b]	71.8(1.07)[b]	**67.42**(0.41)[b]
	Ensemble-OOB	48.84(1.2)[-m]	61.42(1.37)[-b]	53.27(0.35)[-b]		Ensemble-ORB	49.61(1.22)[-b]	65.68(1.2)[-b]	55.91(0.58)[-b]
Neutron	WP-OOB	70.03(0.69)	91.81(0.6)	79.43(0.36)	Neutron	WP-ORB	80.47(1.68)	79.52(1.85)	79.52(0.57)
	All-in-One-OOB	73.24(0.41)[b]	91.39(0.49)[-m]	81.48(0.23)[b]		All-in-One-ORB	75.34(0.56)[-b]	87.53(0.88)[b]	80.72(0.39)[b]
	Filtering-OOB	74.89(0.44)[b]	89.63(0.53)[-b]	**81.51**(0.25)[b]		Filtering-ORB	73.96(0.73)[-b]	88.94(1.02)[b]	80.62(0.46)[b]
	Ensemble-OOB	78.58(1.02)[b]	59.78(3.26)[-b]	68.28(1.39)[-b]		Ensemble-ORB	77.01(1.52)[-b]	72.02(3.35)[-b]	74.28(1.08)[-b]
Npm	WP-OOB	36.99(2.15)	75.74(1.5)	45.91(0.89)	Npm	WP-ORB	54.93(1.15)	63.67(1.3)	53.81(0.94)
	All-in-One-OOB	54.8(1.58)[b]	68.69(2.0)[-b]	59.34(0.6)[b]		All-in-One-ORB	50.95(1.54)[-b]	76.91(1.89)[b]	**60.03**(1.09)[b]
	Filtering-OOB	55.39(1.58)[b]	68.59(1.46)[-b]	59.68(0.53)[b]		Filtering-ORB	51.23(3.95)[-b]	73.26(3.88)[b]	59.15(1.91)[b]
	Ensemble-OOB	54.22(1.0)[b]	51.89(1.03)[-b]	50.42(0.28)[b]		Ensemble-ORB	52.48(1.23)[-b]	56.11(0.95)[-b]	52.41(0.57)[-b]
BroadleafCommerce	WP-OOB	58.44(1.55)	69.82(2.09)	60.41(0.82)	BroadleafCommerce	WP-ORB	59.43(2.91)	65.93(2.96)	59.72(1.26)
	All-in-One-OOB	67.28(1.34)[b]	72.11(1.51)[b]	**69.01**(0.57)[b]		All-in-One-ORB	66.78(0.84)[b]	70.12(1.0)[b]	67.85(0.57)[b]
	Filtering-OOB	67.41(0.93)[b]	70.53(0.96)[s]	68.39(0.42)[b]		Filtering-ORB	66.83(0.63)[b]	69.2(1.45)[b]	67.51(0.84)[b]
	Ensemble-OOB	61.96(0.88)[b]	56.48(1.07)[-b]	57.31(0.17)[-b]		Ensemble-ORB	61.57(1.18)[b]	59.13(1.44)[-b]	58.94(0.44)[-m]
Ranking	WP-OOB	2	1	3	Ranking	WP-ORB	1	2	2
	All-in-One-OOB	1	1	1		All-in-One-ORB	1	1	1
	Filtering-OOB	1	1	1		Filtering-ORB	1	1	1
	Ensemble-OOB	1	4	5		Ensemble-ORB	1	3	4
C1	WP-OOB	60.93(5.53)	38.32(4.22)	**40.56**(0.97)	c1	WP-ORB	58.15(4.91)	40.56(4.66)	33.04(2.3)
	All-in-One-OOB (combined)	30.42[-b](1.81)	67.12[b](1.91)	39.45[-b](1.45)		All-in-One-ORB (combined)	46.46[-b](4.49)	47.98[b](3.89)	33.29[*](2.54)
	All-in-One-OOB (proprietary)	24.52[-b](3.5)	74.84[b](3.04)	35.76[-b](1.71)		All-in-One-ORB (proprietary)	34.44[-b](0.49)	59.36[b](0.47)	26.2[-b](0.71)
	Filtering-OOB	30.97(2.22)	67.77(1.96)	39.93[-m](1.35)		Filtering-ORB	43.63[-b](4.81)	51.16[b](4.16)	31.68[-s](2.81)
C2	WP-OOB	16.49(0.17)	87.11(0.12)	26.36(0.46)	c2	WP-ORB	16.3(0.11)	87.14(0.04)	25.81(0.32)
	All-in-One-OOB (combined)	44.47[b](1.1)	58.6[-b](2.23)	41.3[b](1.01)		All-in-One-ORB (combined)	44.58[b](9.89)	56.03[-b](7.71)	43.53[b](3.37)
	All-in-One-OOB (proprietary)	41.12[b](1.1)	54.58[-b](1.85)	33.77[b](0.83)		All-in-One-ORB (proprietary)	26.65[b](5.99)	76.45[-b](6.74)	36.86[b](3.7)
	Filtering-OOB	45.88(2.71)	56.64(2.83)	40.79[b](0.97)		Filtering-ORB	52.54[b](7.32)	48.5[-b](8.29)	**44.31**[b](3.71)
C3	WP-OOB	52.05(4.13)	48.31(3.48)	40.28(1.05)	c3	WP-ORB	49.64(4.43)	51.2(7.06)	39.79(1.14)
	All-in-One-OOB (combined)	48.12[-b](1.52)	61.14[b](0.88)	**43.53**[b](0.88)		All-in-One-ORB (combined)	64.04[b](1.01)	48.98[-s](1.46)	42.85[b](0.79)
	All-in-One-OOB (proprietary)	23.51[-b](2.41)	78.5[b](1.87)	38.97[-b](1.75)		All-in-One-ORB (proprietary)	57.95[b](0.83)	50.52[-s](1.08)	39.78[*](0.61)
	Filtering-OOB	47.36(1.33)	61.12(1.47)	43.05[b](0.63)		Filtering-ORB	62.97[b](1.01)	48.07[-m](1.54)	41.76[b](0.73)
Ranking	WP-OOB	1	3	2	Ranking	WP-ORB	2	2	3
	All-in-One-OOB (combined)	2	2	1		All-in-One-ORB (combined)	2	3	1
	All-in-One-OOB (proprietary)	3	1	2		All-in-One-ORB (proprietary)	2	2	2
	Filtering-OOB (combined)	2	2	1		Filtering-ORB (combined)	1	3	1

Top G-Means for each dataset are in bold. Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the corresponding WP approach (WP-OOB or WP-ORB). Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the corresponding WP approach. Scott-Knott.BA12 was run for all OOB- and ORB-based approaches together. For each performance metric, one test was run for the open source, and one test was run for the proprietary data results. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

Specifically, studies in offline scenarios ignore the chronology of the projects. When the target and other projects have an overlap in their development period, offline CP approaches train their individual classifiers with a considerably larger amount of data that would still not have been available for training in practice, leading to overoptimistic estimates of predictive performance.

In terms of recalls for the open source datasets, WP-OOB performed generally poorly in terms of Recall0, while Ensemble-OOB, Ensemble-ORB and WP-ORB performed generally poorly in terms of Recall1. As a result, these approaches ranked worse than the others on these performance metrics. The recalls of the approaches across data sets are influenced by trade-offs between Recall0 and Recall1, resulting in several approaches obtaining the same best rank across datasets. This is because a given approach sometimes performs better in terms of Recall0 and sometimes in terms of Recall1, resulting in a the same rank among approaches across datasets. However, given the G-Mean results, which combine Recall0 and Recall1, the trade-offs between recalls obtained by All-In-One and Filtering were better than those obtained by WP and Ensemble.

All-in-One (combined) and Filtering (combined) were the best ranked in terms of G-Mean for the proprietary projects, according to Scott-Knott.BA12 (Table 3). Effect sizes varied from insignificant to large. In particular, All-in-One-OOB (combined) obtained G-means up to 14.94% better (for C2) than those of WP-OOB, and Filtering (combined) obtained G-means up to 18.5% better (for C2) than those of WP-ORB.

Interestingly, both All-in-One-OOB and All-in-One-ORB using only the proprietary CP data did not perform so well, and were ranked second in terms of G-Mean, together with WP-OOB. WP-ORB was the worst ranked approach in terms of G-Mean. These results again show that open source data can be helpful for proprietary JIT-SDP predictions. They also suggest again that the number and variety of training examples used for training a classifier is a key factor for obtaining better G-Means, as the All-in-One approach using only the proprietary CP data was trained on less CP data. However, the G-Mean values obtained when predicting the proprietary data were in general much lower than when predicting the open source data, even when using all the open source data as CP data. This suggests that having a good amount of WP data is also important. The importance of the number of WP training examples is also supported by the (lower) overall G-Mean during the initial phase of the open source projects (Table 2) against the (higher) overall G-Mean across the whole open source projects (Table 3). So, both open source and proprietary projects need a good number of WP training examples to perform well.

In terms of recalls, the results for the proprietary data were varied. All-in-One-OOB trained only with proprietary CP data ranked best in terms of Recall1, but worst in terms of Recall0. WP-OOB and Filter-ORB ranked best in terms of Recall0, but were in the worst group in terms of Recall1.

RQ3: Merging CP and WP data together (All-in-One or Filtering) to train a classifier achieved up to 16.88% higher overall G-Mean than WP classifiers for the open source, and up to 18.5% higher overall G-Mean for the proprietary data in an online scenario. There was no evidence that filtering out training examples dissimilar to the recent software changes of the target project is helpful to improve overall predictive performance. Building separate classifiers from individual projects (Ensemble approach) was detrimental.

7 THREATS TO VALIDITY

Internal validity: each approach for each dataset with each classifiers has been executed 30 times to mitigate threats to internal validity. Also, results can be influenced by poor parameter choices. To mitigate this threat, a grid search was performed on a set of possible values for each parameter based on an initial portion of the data stream (see Section 5). Besides, all approaches take verification latency into account and fully respect chronology.

Construct validity: The evaluation metrics used in this work are G-Mean, Recall0, and Recall1. These are widely used metrics appropriate for class imbalance learning [33]. Predictive performance is calculated in a prequential way with fading factor to discount older information across time, so that plots of predictive performance reflect the variations in predictive performance observed over time.

Statistical conclusion validity: Scott-Knott test was run with non-parametric bootstrap sampling considering A12 effect size to avoid concluding that there is a difference in predictive performance when this difference is likely to be irrelevant due to low effect size.

External validity: This concerns with generalisation of the findings. This study used 10 open source projects and 3 proprietary projects of various characteristics such as programming language, starting date, number of commits per day, etc. The results may not be generalised for other types of projects.

8 CONCLUSION

This study investigated CP learning for JIT-SDP in a realistic online learning scenario, using both open source and proprietary data. In offline learning, existing CP approaches for JIT-SDP did not perform better than WP approaches [15]. In online learning, we showed that CP approaches trained with incoming CP and WP data can help to improve predictive performance over WP approaches trained only with WP data. The All-in-One and Filtering CP approaches were particularly helpful during the initial phase of the project when there is not enough WP data available (RQ1), leading to up to 53.90% improvements in G-Mean. These approaches also helped to reduce sudden drops in performance of the predictive model (RQ2) after the initial phase of the project, achieving up to around 40% better G-Mean during such periods of time. They also improved overall predictive performance (RQ3) compared to the WP approach, obtaining up to 18.5% higher overall G-Mean.

Even though the ensemble approach was shown to perform well in offline learning [15], it was the worst approach when considering a realistic online learning scenario, obtaining average G-Means that were even lower than those of the WP approach. This indicates that splitting data from different projects may not be appropriate in online scenario. On the other hand, training a single model combining CP and WP together (All-in-One) significantly improved performance, hence is more suitable in online JIT-SDP. Our results indicate that both the number of CP and WP training examples is important for achieving good predictive performance in JIT-SDP. Filtering out very different CP examples did not improve the models performance significantly compared to the All-in-One approach.

Our work has practical implications which are described below:

- In online JIT-SDP, if practitioners use CP data along with WP data, this can prevent the performance of the model to become very low at the initial phase of a project which often occurs due to lack of sufficient training data. This will enable practitioners to use JIT-SDP earlier during the development of a project (RQ1).
- WP models can suffer performance drops which cause them to be unsuitable during certain periods of time. These drops mean that, at any given point in time, models may be performing very well or very poorly, being unreliable for practitioners. Using CP data along with WP data can overcome this issue by helping to prevent or reduce such drops, enabling practitioners to more continuously use JIT-SDP throughout the lifetime of the project (RQ2).
- The combined use of both WP and CP data through All-in-One and Filtering improves overall predictive performance of JIT-SDP compared to WP classifiers (RQ3). Our study indicates the importance of the amount of training data. Practitioners should consider collecting large amounts of both CP and WP training data when adopting JIT-SDP.

Future work includes incorporating additional longer running industrial projects and additional open source projects, investigation of different CP approaches for online JIT-SDP, and investigation of methods for automatically adjusting the hyperparameters of the approaches over time.

Acknowledgements: This work was funded by EPSRC Grant No. EP/R006660/2.

REFERENCES

[1] A. Agrawal and T. Menzies. 2018. Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction. In *Proceedings of the 40th International Conference on Software Engineering*. 1050–1061.

[2] 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 *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 666–676.

[3] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2013. Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 252–261.

[4] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. 2019. Cross-Project Just-in-Time Bug Prediction for Mobile Apps: An Empirical Assessment. In *6th IEEE/ACM International Conference on Mobile Software Engineering and Systems*.

[5] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology* 93 (2018), 1–13.

[6] J. Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *JMLR* 7 (2006), 1–30.

[7] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. 2015. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine* 10, 4 (2015), 12–25.

[8] Pedro Domingos and Geoff Hulten. 2000. Mining High-speed Data Streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '00)*. ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/347090.347107

[9] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. 2013. On evaluating stream learning algorithms. *Machine learning* 90, 3 (2013), 317–346.

[10] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Trans. Softw. Eng.* 31, 10 (Oct. 2005), 897–910. https://doi.org/10.1109/TSE.2005.112

[11] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[12] Zhimin He, Fayola Peters, Tim Menzies, and Ye Yang. 2013. Learning from open-source projects: An empirical study on defect prediction. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 45–54.

[13] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. 2012. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering* 19, 2 (2012), 167–199.

[14] Xiao-Yuan Jing, Fei Wu, Xiwei Dong, and Baowen Xu. 2016. An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems. *IEEE Transactions on Software Engineering* 43, 4 (2016), 321–339.

[15] 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. https://doi.org/10.1007/s10664-015-9400-x

[16] 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* (2013). https://doi.org/10.1109/TSE.2012.70

[17] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. 2017. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 318–328.

[18] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[19] 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.

[20] Thilo Mende and Rainer Koschke. 2010. Effort-aware defect prediction models. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 107–116.

[21] Tim Menzies, Ye Yang, George Mathew, Barry Boehm, and Jairus Hihn. 2017. Negative results for software effort estimation. *Empirical Software Engineering* 22, 5 (2017), 2658–2683.

[22] Nikolaos Mittas and Lefteris Angelis. 2012. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Transactions on software engineering* 39, 4 (2012), 537–551.

[23] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 382–391.

[24] Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2014. Cross-project defect prediction models: L'union fait la force. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 164–173.

[25] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 966–969.

[26] Duksan Ryu, Okjoo Choi, and Jongmoon Baik. 2016. Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering* 21, 1 (2016), 43–71.

[27] Duksan Ryu, Jong-In Jang, and Jongmoon Baik. 2017. A transfer cost-sensitive boosting approach for cross-project defect prediction. *Software Quality Journal* 25, 1 (2017), 235–272.

[28] 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 – Supplementary Material. http://www.cs.bham.ac.uk/~minkull/publications/TabassumICSE2020-supplement.pdf

[29] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 99–108.

[30] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14, 5 (2009), 540–578.

[31] András 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.

[32] Romi Satria Wahono. 2015. A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. *Journal of Software Engineering* 1, 1 (2015), 1–16.

[33] Shuo Wang, Leandro L Minku, and Xin Yao. 2018. A systematic study of online class imbalance learning with concept drift. *IEEE transactions on neural networks and learning systems* 29, 10 (2018), 4802–4821.

[34] Tiejian Wang, Zhiwu Zhang, Xiaoyuan Jing, and Liqiang Zhang. 2016. Multiple kernel ensemble learning for software defect prediction. *Automated Software Engineering* 23, 4 (2016), 569–590.

[35] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 91–100.