Commit-Time Defect Prediction Using One-Class Classification

Mohammed A. Shehab[a], Wael Khreich[b], Abdelwahab Hamou-Lhadj[a,*] and Issam Sedki[a]

[a]Concordia University, 1455 De Maisonneuve Blvd. W., Montreal, H3G 1M8, Canada
[b]American University of Beirut, Beirut, Lebanon

ARTICLE INFO

Keywords:
Just-In-Time Software Defect Prediction (JIT-SDP)
One-Class Classification
Machine Learning
Software Maintenance and Evolution
Software Reliability

ABSTRACT

Existing Just-In-Time Software Defect Prediction methods suffer from the data imbalance problem, where the majority class (normal commits) significantly outnumbers the minority class (buggy commits). This results in a higher probability of misclassification. Various data balancing techniques have been proposed to address this challenge with varying degrees of success. In this study, we propose an approach that rely on One-Class Classification (OCC) to train models using data from the majority class only. This eliminates the need for data balancing. We compare the accuracy of three OCC algorithms - One-class SVM, Isolation Forest, and One-class k-NN - to their binary counterparts - SVM, Random Forest, and k-NN - on 34 software projects. Our results show that the data imbalance ratio (the proportion of normal to buggy commits) plays a crucial role in determining the optimal classification approach. We found that for projects with medium to high imbalance ratio, OCC algorithms outperform binary classifiers with and without data balancing, using cross and time-sensitive validation approaches. Furthermore, we found that OCC methods require fewer features for projects with medium to high IR, reducing the computational overhead of training and response time while providing a better understanding of the data and algorithm behaviour.

1. Introduction

Software Defect Prediction (SDP) approaches have shown to be useful in improving the quality of software systems while reducing the cost Nayrolles and Hamou-Lhadj (2018), Kamei, Shihab, Adams, Hassan, Mockus, Sinha and Ubayashi (2013). SDP models are built using historical software data and various machine learning techniques. The models are then deployed to recognize potential defects in a newly written code Nayrolles and Hamou-Lhadj (2018). Recently, there has been a growing interest in Just-in-Time SDP (JIT-SDP) techniques where the prediction of bugs is performed at the level of code commits, i.e., before the changes reach the central code repository. The common practice is to build a model using one or multiple machine learning algorithms of historical commits (both normal and buggy commit) that can later be used to detect whether a new commit is buggy or not. A buggy commit is a commit that is identified by the models as a potentially bug-introducing commit. Unlike traditional SDP techniques, which operate on the entire source code, JIT-SDP approaches can be used to provide quick feedback to developers on the code changes they make. Developers can decide to make fixes while the changes are fresh in their minds. Furthermore, JIT-SDP methods can be embedded in code versioning systems, eliminating the need for external tools Kamei et al. (2013); Wang, Liu and Tan (2016); Nayrolles and Hamou-Lhadj (2018).

Existing JIT-SDP studies rely on a variety of machine learning (ML) techniques including decision trees Song, Jia, Shepperd, Ying and Liu (2011); Nayrolles and Hamou-Lhadj (2018), Bayesian approaches Catolino, Di Nucci and Ferrucci (2019), Neural Networks Kiehn, Pan and Camci (2019), and Deep Learning Wang and Yao (2009); Hoang, Khanh Dam, Kamei, Lo and Ubayashi (2019). These approaches face a significant challenge due to the imbalance data problem Song, Guo and Shepperd (2019); Cabral, Minku, Shihab and Mujahid (2019) where one class (the minority class) has a much lower proportion in the training data as compared to the majority class. Class imbalance affects the performance of a classifier because it creates a bias towards the majority class, leading to errors when predicting the minority class Lomio, Pascarella, Palomba and Lenarduzzi (2022); Hoang, Kang, Lo and Lawall (2020). The class imbalance problem is particularly relevant to JIT-SDP because the buggy commits are considerably fewer than the normal ones Lomio et al. (2022); Song et al. (2019). To address this, existing studies treat the problem as a binary (i.e., two-class) classification problem and use some sort of data balancing approaches such as Over-Sampling (OS), Synthetic Minority Oversampling (SMOTE), or Under-Sampling (US) so as not to undermine the minority class Song et al. (2019). A comprehensive study of the use of data balancing approaches in the context of JIT-SDP research is presented by Song et al. Song et al. (2019). However, data balancing methods have many limitations such as increasing noise when over-sampling or, worse, dropping important patterns when under-samplingBruce and Bruce (2017). Additionally, some classifiers are sensitive to the data distribution that affects the re-sampling process. Investigating which sampling method works best for a specific classifier takes time and effort Bishop (2006); Wang, Minku and Yao (2018).

In this paper, we investigate the use of One-Class Classification (OCC) algorithms in JIT-SDP, where we train a

mohammed.shehab@concordia.ca (M.A. Shehab);
wk47@aub.edu.lb (W. Khreich);
wahab.hamou-lhadj@concordia.ca (A. Hamou-Lhadj);
issam.sedki@concordia.ca (I. Sedki)
ORCID(s):

model on the majority class only (in our case the normal commits). We used two approaches to build and evaluate models Cross-Validation (CV) and Time-sensitive Validation (TV). The model is then used to predict buggy commits (the minority class). By doing so, we completely eliminate the need for data-balancing approaches. In addition, we only require the presence of normal data to train the model. Our approach is inspired by the area of anomaly detection where the common practice is to build machine learning models using the normal behavior of the system and then use these models to detect any deviations from normalcy Khreich, Khosravifar, Hamou-Lhadj and Talhi (2017); Islam, Khreich and Hamou-Lhadj (2018). In this work, we compare the performance of three different OCC algorithms to their binary counterparts on 34 datasets (a total of 259,925 commits) with various levels of class imbalance ratios. These algorithms are trained using the features described by Kamei et al. Kamei et al. (2013). More specifically, we compare the performance of the following OCC algorithms, Isolation Forest (IOF) Hariri, Kind and Brunner (2021), One-Class k-Nearest Neighbors (OC-k-NN) Yousef, Jung, Showe and Showe (2008), and One-Class Support Vector Machine (OC-SVM) Khreich et al. (2017) to their binary classification counterparts, Random Forest (RF), k-Nearest Neighbor (k-NN), and Support Vector Machine (SVM) with and without data balancing.

However, it is worth noting that our study deliberately excluded Deep learning models such as DeepJIT, DBN-JIT, and CC2Vec due to their utilization of different features, specifically semantic and syntactic elements Hoang et al. (2019); Zeng, Zhang, Zhang and Zhang (2021). We decided to maintain consistency within our research framework and focus on a specific set of features. By doing so, we aimed to analyze the selected features' effectiveness in our model comprehensively. Moreover, we acknowledge the critical influence that feature selection can have on the performance of models, particularly when dealing with imbalanced data Bruce and Bruce (2017); Butcher and Smith (2020). We intended not to disregard the significance of these Deep learning models but rather to streamline our investigation and isolate the impact of the chosen features. Furthermore, we draw attention to the work of Zeng et al. Zeng et al. (2021) and Pornprasit and Tantithamthavorn Pornprasit and Tantithamthavorn (2021) who showed that traditional machine learning models such as a logistic regression classifier outperform deep learning models when working with large datasets. That being said, as part of future work, we intend to expand our research to include deep learning algorithms and semantic feature sets.

The paper addresses the following three new research questions (RQ):

- RQ1: What is the overall performance of OCC algorithms compared to their binary classifier counterparts?
- RQ2: How do OCC algorithms perform compared to binary classifiers when considering the data imbalance ratio?
- RQ3: Which features affect the accuracy of OCC algorithms compared to their corresponding binary classifiers?

Regarding RQ1, our findings suggest that binary classifiers tend to perform better than OCC algorithms in balanced data settings. For RQ2, we consider the data imbalance ratio (IR), which indicates the proportion of normal commits to buggy ones. We found that OCC methods consistently outperformed binary classifiers for projects with a medium to high imbalance ratio, with a medium to large effect size. As for RQ3, our findings indicate that the choice of features has an impact on the accuracy of the algorithm. Projects with medium to high IR require fewer features to train than the other projects.

Researchers and practitioners can benefit from this study by developing JIT-SDP tools that use OCC algorithms instead of binary classifiers for systems with high data imbalance ratios. OCC methods not only eliminate the need for data balancing techniques but do not require the availability of commits from both classes, i.e., normal and buggy commits. These algorithms can also be trained on fewer features, which shortens the training and response time and allows for a better understanding of the behavior of the algorithms.

Organization of the paper: The next section reviews software defect prediction and techniques for learning from imbalanced data. Section 3 describes three one-class classifiers, which will be used in our experiments. Section 4 describes methods and experiments including datasets, features, performance metrics, and experimental protocol used for conducting the experiments. In Section 5, we present the results to provide answers to the research questions. Potential threats to validity and our mitigating actions are presented in Section 6, followed by the conclusions and future work in Section 7.

2. Related Work

Lomio et al. Lomio et al. (2022) investigated the use of anomaly detection algorithms, more particularly, OC-SVM, IOF, and Local Outlier Factor for fine-grained JIT Pascarella, Palomba and Bacchelli (2019) defect prediction, where the predicted class has three labels, namely buggy, partial-buggy, and normal, instead of buggy and normal. The authors found that one-class classification algorithms perform similarly to binary classifiers. There are many key differences between Lomio et al's approach and our study. First the authors focused on predicting files within the commits that may potentially be buggy and not the commits. In addition, they used a cross-project JIT-SDP method, meaning that, using a dataset of n projects, they train a model using n-1 projects and then test it on the remaining project. In this study, we apply JIT-SDP to single projects and not cross-projects. This is because our objective is to determine

whether and when OCC algorithms provide better results than their corresponding binary classifier. Using a cross-project experimental setting makes it difficult to conclude if the obtained results are due to the type of classifier (binary or OCC) or simply because the models are trained on larger datasets (commits from multiple projects). The second difference is that we compare OCC algorithms with their corresponding binary classifiers with and without data balancing techniques. This is because data balancing is used to address the imbalance data problem. Therefore, we must compare OCC to binary classifiers with data balancing to reach strong conclusions. In addition, unlike Lomio et al.'s study, we examine the impact of the ratio of data imbalance on the accuracy to determine a threshold beyond which OCC algorithms should be favored over binary classification algorithms. In their study, the authors did not provide such a threshold. Finally, we also investigate the impact of various feature sets on the accuracy of OCC algorithms to draw a full picture of the value and usefulness of these algorithms in practice.

There exist other studies that investigate the problem of data imbalance in JIT-SDP tasks using mainly binary classifiers. Cabral et al. Cabral et al. (2019) showed that JIT-SDP suffers tremendously from data imbalance issues by significantly reducing the predictive performance of existing JIT-SDP methods. Their study is based on the analysis of commits of 10 projects. Wang et al. Wang and Yao (2013) studied the problem of class imbalance learning methods in the field of software defect prediction. They examined various class imbalance learning methods, including re-sampling techniques, threshold moving, and ensemble algorithms. They found that AdaBoost.NC yields the best overall performance. The authors further improved the performance of AdaBoost.NC by proposing a dynamic version, which adjusts its parameters automatically during training.

Song et al. Song et al. (2019) conducted a comprehensive study to understand the impact of defects in software components by applying various imbalance learning techniques. They conducted experiments with 27 datasets to understand the impact of specific classifiers, data balancing techniques, and features on prediction accuracy. The authors found that a moderate and severe level of imbalanced data can directly influence the SDP model's performance. They also found that the selection of the machine learning algorithm is important when dealing with data imbalance in the context of SDP.

Other recent studies on SDP use a variety of machine learning algorithms and their combinations to improve model accuracy. Tong et al. Tong, Liu and Wang (2018) proposed a two-stage ensemble approach to improve the accuracy of SDP. The proposed model, called SDAEsTSE, builds on a two-phase ensemble learning based on stacked denoising auto-encoders. In the first phase, the auto-encoders were used to represent the traditional software metrics as deep representations. After that, the two layers of the ensemble technique were used to build the prediction model to overcome the class imbalance issue. The proposed approach outperforms existing state-of-the-art SDP models significantly when applied to commits from 12 NASA projects.

TLEL is an approach proposed by Yang et al. Yang, Lo, Xia and Sun (2017) to use two-layer set learning that uses decision trees and ensemble learning to improve the performance of JIT-SDP prediction. On average, TLEL was able to identify more than 70% of defects by only 20% of code lines compared to around 50% for a baseline model. The researchers used random under-sampling to overcome the imbalance issue.

The accuracy across supervised and unsupervised models for investigative JIT-SDP techniques has been examined by Yang, Zhou, Liu, Zhao, Lu, Xu, Xu and Leung (2016). The authors found that unsupervised techniques, which typically require less time to build the model, yield similar results as the supervised models. Fu et al. Fu and Menzies (2017) conducted a replication study of that of Yang et al. Yang et al. (2016). The authors reported that unsupervised models did not perform better than the supervised ones. They contended that unsupervised learners should be combined to achieve comparable performance to supervised algorithms.

Fukushima et al. Fukushima, Kamei, McIntosh, Yamashita and Ubayashi (2014) examined the performance of JIT-SDP models using two case studies: single-projects and cross-projects. They started by examining the effect of using the 14 code-based and process-based features Kamei et al. (2013) by extracting these features from 11 projects. They ended up using only six projects. The authors used the Random Forest algorithm for building the classifier and showed that cross-project techniques provide superior performance compared to single-project methods.

Yan et al. Yan, Xia, Fan, Hassan, Lo and Li (2022) designed a framework to detect the buggy changes for code and then recognize the buggy code location from the newly added lines. This technique comprises two main phases: Identification and Localization. In the Identification phase, the JIT-SDP model is trained and tested using 14 features proposed by Kamei et al. (2013), where the training data is 60% of early commits and the next 40% of commits used to test the model. Yan et al. (2022) did not focus on the performance of the JIT-SDP model, which directly affects the Localization step after identifying the buggy changes. Their approach focuses on finding the location of the buggy code if the JIT-SDP predicts the code changes as buggy.

Wang et al. Wang et al. (2016) applied the Deep Belief Network (DBN) model as a semantic feature generator. The Abstract Syntax Tree (AST) is used to represent the source code and use it to train the DBN model. They used the Naive Bayes (NB) and Logistic Regression (LR) classifiers for building the prediction models, which were trained on 10 open-source Java projects from various domains to ensure model generalization. The proposed method increases the F1-score of cross-projects and within-project approaches by 8.9% and 14.2%, receptively.

In contrast, Pornprasit and Tantithamthavorn Pornprasit and Tantithamthavorn (2021) proposed the JITLine tool using the JIT-SDP model to predict the buggy changes and

find the location of buggy code for buggy predictions. The authors evaluated the performance of the JITLine with 3 models (EARL Kamei et al. (2013), DeepJIT Hoang et al. (2019), and CC2Vec Hoang et al. (2020)). They used the AUC, F1, Precision, and Recall evaluation metrics. The JITLine achieved AUC = 82%, while the best AUC for EARL, DeepJIT, and CC2Vec reaches 64%, 76%, and 81%, respectively. The JITLine approach also provides faster and simpler machine learning models to build JIT-SDP models rather than deep learning approaches (e.g., DeepJIT Hoang et al. (2019) and CC2Vec Hoang et al. (2020)).

Nayrolles and Hamou-Lhadj Nayrolles and Hamou-Lhadj (2018) introduced CLEVER, a JIT-SDP technique that creates a training model by merging contributions from multiple video game systems that use the same game engines. Instead of working on each project independently, the authors argued that developing a training model that incorporates commits from several interconnected systems made more sense in this situation. CLEVER can detect buggy commits with 79% precision and 65% recall, and the F1-score is 79.10%.

Shehab et al. Shehab, Hamou-Lhadj and Alawneh (2022) proposed merging commit data from a collection of projects as part of a bigger cluster to train the JIT-SDP model. ClusterCommit depends solely on code and process metrics, making no assumptions about how projects are built. Furthermore, unlike Nayrolles and Hamou-Lhadj's work Nayrolles and Hamou-Lhadj (2018), Shehab et al. Shehab et al. (2022) adopt a time-sensitive validation technique to test the prediction accuracy of ClusterCommit, which considers the temporal sequence of commits. ClusterCommit gets an F1-score of 73% and 0.44 MCC.

Except for the work of Lomio et al. Lomio et al. (2022), all other existing studies resort to machine learning techniques for JIT-SDP using binariy classification models and use data balancing methods to avoid bias. In this paper, we propose the use of one-class classification to train JIT-SDP models using the majority class, i.e., the normal commits. The OCC approach eliminates the need for data balancing. Also, as we show in this paper, the OCC models perform better than binary algorithms when the data imbalance ratio is high. We also determine using 27 datasets a threshold for the data imbalance ratio beyond which it is preferable to use OCC instead of binary classification.

3. One-class classification

OCC techniques rely on data from the majority (negative) class to train the machine learning model, as opposed to binary classifiers, which need labeled data from both positive and negative classesBellinger, Sharma and Japkowicz (2012). Once trained, the OCC model is used to classify new examples as either belonging to the majority class or not (which can then be considered outliers or anomalies). One-class algorithms are well suited for tasks where the minority (positive) class does not exhibit a consistent pattern or structure in the feature space, which makes it harder for binary classification models to learn the class boundary. OCC algorithms attempt to group the majority class instances into a high-density region in the feature space as normal behavior (see Figure 1) and then detect deviations from this expected behavior as anomalies or outliers Hart, Stork and Duda (2000).

In this paper, we examine three commonly used OCC techniques which are based on three fundamental machine learning approaches, and compare them to their binary classifier counterparts. These OCC algorithms are One-Class Support Vector Machine (OC-SVM), which relies on a margin-based algorithm; One-Class k-Nearest Neighbors (OC-k-NN), which relies on a distance-based algorithm; and Isolation Forest (IOF), which relies on a tree-based algorithm. We explain each algorithm in more detail in the following subsections.

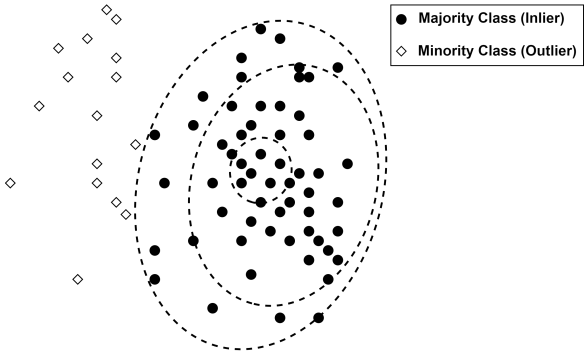

Figure 1: An illustration of OCC approach learning from the majority class and detecting deviations as anomalies or outliers.

3.1. OC-SVM

Support Vector Machine (SVM) is a binary supervised machine learning approach that separates classes based on the maximum margin hyperplane Bishop (2006); He and Garcia (2009). In addition to linear hyperplanes, SVM can rely on other kernels such as polynomial, radial basis function (RBF), and sigmoid to detect nonlinear boundaries between classes Bishop (2006). OC-SVM is a version of SVM adapted to the OCC approach that only learns from the majority class Schölkopf, Platt, Shawe-Taylor, Smola and Williamson (2001). OC-SVM creates discrimination boundaries based on the high-density region in the feature space of the training data.

Given a training data Xi of size n and K kernel function, the OC-SVM training is based on the following dual problem (Eq. 1):

$$\min_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j K(x_i, x_j) \tag{1}$$

$$\text{subject to } 0 \leq \alpha_i \leq \frac{1}{\nu n}, \sum_i \alpha_i = 1 \tag{2}$$

where α_i are the support vectors, K is the kernel, and $\nu \in (0, 1)$ controls the upper bound on the fraction of outliers and the lower bound on the fraction of support vectors. After obtaining the coefficients of the support vectors ($\alpha_i > 0$), the decision function is computed based on the sign (positive or negative) of the fowling function (Eq. 3):

$$f(x) = sign(\sum_{ij} \alpha_i \alpha_j K(x_i, x_j) - \rho) \quad (3)$$

where ρ denotes the offset of the separating hyperplane.

3.2. OC-k-NN

k-NN is a supervised machine learning approach that uses lazy processing to classify the data Wang and Zucker (2000). The lazy approach uses the training data on the prediction time as a memory instead of training the model to detect the patterns on the training time Jiang, Cai, Wang and Jiang (2007). The k-NN algorithm calculates the distance between a new data point and the k closest points. Then, it uses the voting method to determine the best label for that data point Jiang et al. (2007).

The distance between the data point and training points is measured using Minkowski distance as shown in Equation (4). The Minkowski distance d is the generalized formula of both (Manhattan for $p = 1$ and Euclidean for $p = 2$) distances Jiang et al. (2007). After selecting the distance measure, we only need to tune the k value, the number of the closest neighbors to the new incoming point.

$$d = \left(\sum_{i=1}^{n} |p_i - q_i|^p \right)^{\frac{1}{p}} \quad (4)$$

The OC-k-NN algorithm is a modified version of k-NN, which also relies on training a dataset (comprising only the majority class) to determine whether a new instance belongs to the majority class or not. For a given test example x, the distance d to the nearest k neighbors of x is first calculated. Then, the average (using the mean or median) of these distances is computed and compared to a tunable threshold δ to determine whether x belongs to the majority class or not. Therefore, OC-k-NN requires two tunable parameters, the value of k and the threshold δ Zhao, Nasrullah and Li (2019).

3.3. IOF

The Isolation Forests (IOF) is a tree-based ensemble algorithm, the OCC counterpart of the Random Forests (RF) binary classifier Liu, Ting and Zhou (2008). The main idea is to build isolation trees by creating partitions such that each data point is isolated, i.e., a particular partition contains only one data point. The intuition behind isolation trees is that a regular point is much harder to isolate than an anomalous point. Therefore, an anomalous point requires fewer partitions than a regular point. The algorithm creates multiple isolation trees by selecting random features and random partitions from different subsets of the training data. This process of partitioning or branching is performed recursively until reaching a single point or the maximum allowable tree depth (a tunable parameter) Hariri et al. (2021).

Given a new observation x, the IOF algorithm parses the x value into the isolation trees. If x ends up in a leaf node or reaches the maximum allowable tree depth it is considered a normal point (belonging to the majority class). Otherwise, if the x couldn't reach a leaf node or the maximum allowable depth then is classified as abnormal (belonging to the minority class) Hariri et al. (2021). Finally, the anomalous score of a particular point x is calculated as shown in Equation (5):

$$s(x, n) = 2^{-E(\frac{h(x)}{c(n)})} \quad (5)$$

Where $h(x)$ is the mean value of depth of the point x in all the isolation trees, $c(n)$ is the average of $h(x)$ or the average depth of all points, and n is the number of points used to build the trees.

4. Experimental Protocol

4.1. Dataset

In this paper, we use datasets of commits from 34 open-source projects from the Apache organization. The total number of commits to all these projects is 259,925. Table 1 shows the characteristics of the datasets. The first column refers to the project name, followed by the number of normal commits, the number of buggy commits, and the data imbalance ratio (IR) measured as the ratio of the number of normal commits to the number of buggy commits. For example, an IR of 4 means that there are 4 normal commits for each 1 buggy commit. The last column shows the total number of commits to the project. The category column has 3 types (low, medium, and large) labeled using a k-mean clustering algorithm based on the IR column. We make the datasets, the scripts, and the results of this study available online[1].

4.2. Feature Extraction and Data Labeling

In this study, we use 14 features proposed by Kamei et al. Kamei et al. (2013) to build JIT-SDP models (also known as Process metrics). These features are organized into five dimensions: diffusion, size, purpose of commit, history, and experience as detailed in Table 2. The selection of these features is motivated by their widespread usage in JIT-SDP research as shown in a survey study conducted by Yunhua et al. Zhao, Damevski and Chen (2023), and their effectiveness in characterizing normal and buggy commits as shown by Rahman et al. Rahman and Devanbu (2013). They are also simple to compute and easy to interpret, allowing us to understand the model behaviour. The simpler the features used to build models, the clearer the interpretation of model predictions Zheng and Casari (2018).

To label the data into normal and buggy commits, we use the Refactoring Aware SZZ Implementation (RA-SZZ) algorithm, proposed by Neto et al. Neto, da Costa and Kulesza (2018). RA-SZZ labels the commits as normal or buggy by

[1] https://github.com/wahabhamoulhadj/jit-occ

Table 1
Description of the Datasets for JIT-SDP

Project Name	Normal	Buggy	IR	Category	Total
Drill	2,288	1,643	1.39	Low	3,931
Flume	1,151	661	1.74	Low	1,812
Openjpa	3,404	1,706	2.00	Low	5,110
Camel	9,032	3,990	2.26	Low	13,022
Zookeeper	1,453	577	2.52	Low	2,030
Flink	20,369	4,613	4.42	Low	24,982
Carbondata	4,249	552	7.70	Low	4,801
Zeppelin	4,259	543	7.84	Low	4,802
Ignite	13,969	1,609	8.68	Low	15,578
Avro	2,151	235	9.15	Low	2,386
Tez	2,426	232	10.46	Low	2,658
Airavata	6,729	497	13.54	Low	7,226
Hadoop	9,881	627	15.76	Low	10,508
Hbase	16,721	1,058	15.80	Low	17,779
Falcon	2,096	130	16.12	Low	2,226
Derby	7,795	473	16.48	Low	8,268
Accumulo	9,541	552	17.28	Low	10,093
Parquet-mr	2,126	114	18.65	Low	2,240
Phoenix	3,284	168	19.55	Low	3,452
Oozie	2,244	114	19.68	Low	2,358
Cayenne	6,365	285	22.33	Medium	6,650
Hive	11,759	518	22.70	Medium	12,277
Jackrabbit	8,488	370	22.94	Medium	8,858
Oodt	2,006	85	23.60	Medium	2,091
Gora	1,314	52	25.27	Medium	1,366
Bookkeeper	2,289	84	27.25	Medium	2,373
Storm	10,178	239	42.59	Large	10,417
Spark	19,591	376	52.10	Large	19,967
Reef	3,813	60	63.55	Large	3,873
Helix	3,672	56	65.57	Large	3,728
Bigtop	2,567	31	82.81	Large	2,598
Curator	2,690	28	96.07	Large	2,718
Cocoon	13,094	66	198.39	Large	13,160
Ambari	24,477	110	222.52	Large	24,587
Total	***237,471***	***22,454***	-	-	***259,925***

Table 2
The features used to build the prediction model

Dimension	Name	Description
Diffusion	NS	Number of modified sub-systems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across files
Size	LA	Added lines
	LD	Deleted lines
	LT	Line of code before edit
Purpose of Change	Fix	Whether or not the change is a defect or fix
History	NDEV	Number of developers that changed the file
	AGE	The average time between file changes
	NUC	The number of unique changes
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on sub-systems

analyzing bug reports from the bug tracking system (in our case, Jira). The RA-SZZ algorithm retrieves all resolved bug reports and links them to the commits using the bug report's unique identifier contained within the commit message (if available). Finally, the algorithm examines the commit history to determine the original commits that introduced the bugs and label them as buggy. Furthermore, we have taken steps to ensure that the most recent commits in our dataset date back to at least one year from the labeling process initiation, thereby significantly reducing the likelihood of mislabeling the commits. This practice was proposed by Herbold et al. Herbold, Trautsch, Trautsch and Ledel (2022). The RA-SZZ is an extension of SZZ Śliwerski, Zimmermann and Zeller (2005) that takes into account code refactoring, which are changes to the code that do not change its external behavior. Refactoring activities tend to complicate the bug localization process because they can move code around and change its structure, making it difficult to determine which lines of code are responsible for a bug Neto et al. (2018); Fan, Xia, da Costa, Lo, Hassan and Li (2021). The RA-SZZ addresses this issue by identifying and accounting for refactorings when analyzing code changes between two versions. This allows it to provide more accurate and reliable results than SZZ when dealing with systems that undergo frequent refactoring Fan et al. (2021). In addition, Fan et al. Fan et al. (2021) showed that RA-SZZ reduces the noise when compared to SZZ. For all these reasons, we opted to use for RA-SZZ in this research instead of the traditional SZZ.

4.3. Evaluation Metrics

Several metrics have been used to evaluate the performance of binary classification problems in general and JIT-SDP models in particular Kamei et al. (2013); McIntosh and Kamei (2018); Catolino et al. (2019); Huang, Xia and Lo (2019). These include threshold-based metrics such as Precision, Recall, F1-score, and the Matthews Correlation Coefficient (MCC), and threshold-independent metrics such as the Receiver Operating Characteristic (ROC) curve and the Area Under the ROC (AUC-ROC).

Threshold-based metrics rely on setting a cut-off point on the classifier's score to compute the confusion matrix based on the following quantities:

- True Positive (TP): The number of buggy commits that are correctly classified as buggy
- False Positive (FP): The number of normal commits, classified as buggy (a.k.a false alarms)
- False Negative (FN): The number of buggy commits that are classified as normal
- True Negative (TN): The number of normal commits that are correctly classified as normal

The F1-score or F1-measure is a popular accuracy measurement used to evaluate machine models Baeza-Yates and Ribeiro-Neto (1999) based on a specific threshold. It is the harmonic mean of the precision (TP/(TP + FP)) and recall (TP/(TP + FN)). However, the F1-score is not suitable for measuring the performance of classifiers when dealing with class imbalance since it gives equal importance to precision and recall, does not account for the TN instances, and varies

with swapping the target class Song et al. (2019); Chicco and Jurman (2020).

Threshold-independent metrics such as the Receiver Operating Characteristic (ROC) and the Area Under the ROC Curve (AUC) do not commit to a threshold. The ROC is a graphical plot (see Figure 2 for an example), which illustrates the performance of a classifier as its discrimination threshold is varied."' The ROC plots the false positive rate fpr=FP/(FP+TN) against the true positive rate tpr=TP/(TP+FP) for every decision threshold Fawcett (2006). A ROC curve allows the visualization of the performance of detectors and the selection of optimal operational points, without committing to a single decision threshold. It presents the classifier's performance across the entire range of class distribution and error costs. The default decision threshold (minimizing overall errors and costs) corresponds to the vertex that is closest to the upper-left corner of the ROC plane (see the red lines on Figure 2. This threshold assumes balanced classes and an equal cost of errors. When the number of positives is larger than the negatives, this threshold can be adjusted to account for the data imbalance ratio by rotating the iso-performance line (blue line on Figure 2) proportionally to the imbalance ratio Fawcett (2006). The AUC has been proposed as a robust (global) measure for the evaluation and selection of classifiers Huang and Ling (2005). The AUC is the average of the tpr values over all fpr values (independently of decision thresholds and prior class distributions). The AUC evaluates how well a classifier is able to sort its predictions according to the confidence it assigns to these predictions. An AUC = 1 means all positives are ranked higher than the negatives, which indicates perfect discrimination between the positive and negative classes. An AUC = 0.5 means that both classes are ranked at random and the classifier is no better than random guessing.

Because the objective of this paper is to understand the overall performance of OCC algorithms compared to binary classifiers, we use the AUC as a threshold-independent metric and the F1-score as a threshold-dependent metric to assess the performance of the algorithms.

Additionally, we use Cliff's δ effect size to assess the magnitude of the difference between the results of OCC algorithms and binary classifiers. The Cliff's test is a non-parametric effect size measure that quantifies the magnitude of dominance as the difference between two groups X and Y Cliff (1993); Macbeth, Razumiejczyk and Ledesma (2010); Romano, Kromrey, Coraggio, Skowronek and Devine (2006). Cliff's δ ranges from –1 to +1. A Cliff's δ that is equal to -1 means that all observations in Y are larger than all observations in X. It is equal to +1 if all observations in X are larger than the observations in Y. A Cliff's δ value that converges to 0 indicates that the distribution of the two observations is identical. The Cliff's δ effect size can also be grouped into ranges Cohen (1992). The effect is considered small for $0.147 \leq |\delta| < 0.330$, moderate for $0.330 \leq |\delta| < 0.474$, or large for $|\delta| \geq 0.474$ Romano et al. (2006); Cohen (1992). Cliff's δ is defined using Equation (6), with x and y

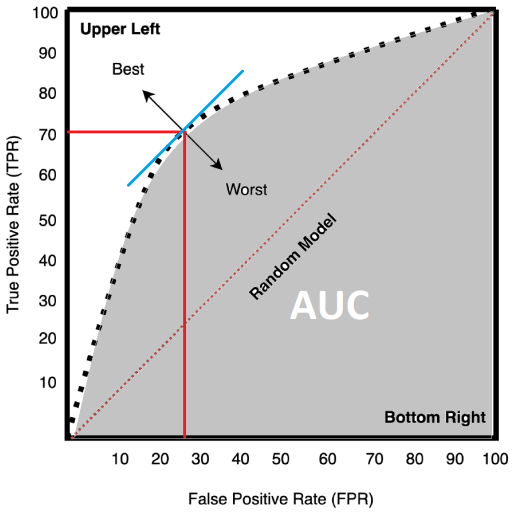

Figure 2: An illustration of a ROC curve and the area under the curve (AUC) along with the default decision threshold.

representing two data vectors and n_x and n_y, the size of these vectors.

$$Cliff's\ \delta = \frac{\sum_i \sum_j sign(y_i - x_j)}{n_y . n_x} \tag{6}$$

4.4. Training and Testing the Algorithms

We experimented with six classification algorithms including three OCC algorithms, OC-SVM, IOF, and OC-k-NN, and three binary classifiers, SVM, RF, and k-NN. For each project in the datasets, we train each of the six algorithms using the 14 features shown in Table 2. In addition, each binary classifier is trained without balancing the data, and with balancing the data using over-sampling, SMOTE, and under-sampling techniques. The choice of these techniques is discussed in Section 5.1. In total, we trained 408 models ((3 **binary models** $*$ 3 **balancing methods** + 3 **OCC models**) $*$ 34 **projects**) repeated 30 times.

We use the PyOD library[2] to build the one-class JIT-SDP models. PyOD is a comprehensive and scalable Python library that is developed on the top of Scikit-learn Pedregosa, Varoquaux, Gramfort, Michel, Thirion, Grisel, Blondel, Prettenhofer, Weiss, Dubourg, Vanderplas, Passos, Cournapeau, Brucher, Perrot and Duchesnay (2011). It supports over 40 anomaly detection algorithms, and has been used in a variety of academic research and commercial products Zhao et al. (2019). For binary classifiers, we used the well-known Scikit-learn Pedregosa et al. (2011) library, which is widely used in this field.

A classification model is built in three steps: training, validation, and testing. The initial model is built during the training step. The validation step is used to fine-tune and optimize the model parameters. The testing phase is used to assess the model's performance. Because one of our goals

[2]https://pyod.readthedocs.io/en/latest/

is to compare OCC algorithms with binary classifiers, the models must be tested on the same testing sets.

4.4.1. Cross-validation approach

In this paper, we use a cross-validation (CV) approach to build and assess the JIT-SDP models. We split each dataset into 70% training and 30% testing sets using a stratified sampling technique to ensure that the ratio of normal to buggy remains the same for both splits. We use k-fold (CV) to train, validate, and select the best model parameters as shown in Figure 3.

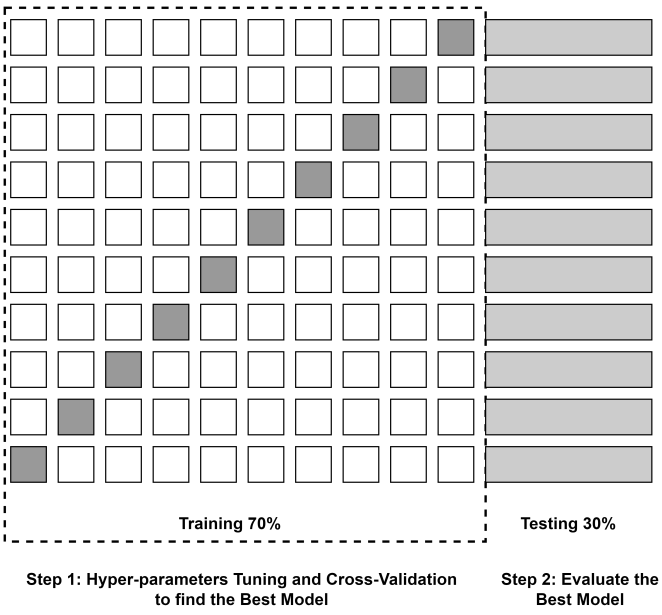

Figure 3: Cross-Validation for JIT-SDP Model Training and Evaluation. In Step 1, models are repeatedly trained on k - 1 folds (white boxes), and their parameters are evaluated on the remaining kth fold (gray box). The overall best models are selected on the test set (Step 2).

Algorithm 1 shows the steps for building the binary models. In Line 1, we split the dataset into training and testing sets. Note that we ensure the testing set is identical for binary and OCC models to enable fair comparison. In Line 2, the training data is used to generate training and validation sets using k folds, where the training data is k-1 folds and the validation data is the remaining fold. From Lines 3 to 6, the hyper-parameters tuning process is performed to find the best model as shown in Figure 3 (Step 1). As for training, we used 70% of normal commits and 70% of buggy commits. We validated the trained model through k-fold cross-validation. Traditionally, k is set to 10. However, in our case, for projects with a number of buggy commits in the validation set that has less than 10 buggy commits (e.g., camel-1.0 and jedit-4.3), we set k to 5, otherwise k = 10. In Line 7, the best binary models without balancing the data are evaluated using the testing set (30% of normal commits and 30% of buggy commits) as shown in Figure 3 (Step 2). The same steps are applied to build the binary models from Lines 8 to 15 but with data balancing methods.

For each one-class classification algorithm, we build the training, validation, and testing sets using the following protocol: In Line 1 of Algorithm 2, the dataset is split for training (70%) and testing (30%), similar to binary classifiers. In Line 2, k folds are used for training and validating the OCC models as shown in Figure 3 (Step 1). From Line 3 through 8, we used 60% of normal commits to train the initial model. The remaining 10% of normal commits are merged with 70% of the buggy commits as the validation data to optimize and hyper-tune the model parameters. Finally, in Line 9, the best OCC model is evaluated using the testing set (30% of normal commits and 30% of buggy commits), as shown in Figure 3 (Step 2).

Algorithm 1: Process of training, validation, and testing of binary algorithms using the Cross-Validation approach.

Data: $Data$
Result: $Results_{imbalance}, Results_{Balanced}$
`/* The size of data tuning is 70% and testing is 30% */`
1 $Data_{Tuning}, Data_{Testing} \leftarrow split\_data(Data)$
2 $folds \leftarrow Generate\_Folds(Data_{Tuning}, 10)$
`/* Evaluate the binary model without balancing the data */`
3 **for** $index = 1; index < Size(folds); index+ = 1$ **do**
4 $\quad Data_{Training} \leftarrow All\ Folds\ Except\ folds[index]$
5 $\quad Data_{Validating} \leftarrow folds[index]$
6 $\quad Model_{Imbalance} \leftarrow Get\_Best\_Model(Data_{Training}, Data_{Validating})$
7 $Results_{Imbalance} \leftarrow Model_{Imbalance}.Evaluate(Data_{Testing})$
`/* Evaluate the binary model after balancing the data */`
8 $ImbalanceMethods \leftarrow \{OverSampling, DownSampling\}$
9 **foreach** $Method \in Imbalance\_Methods$ **do**
10 $\quad Data_{Tuning} \leftarrow Balancing(Data_{Tuning}, Method)$
11 $\quad$ **for** $index = 1; index < k; index+ = 1$ **do**
12 $\quad\quad Data_{Training} \leftarrow All\ Folds\ Except\ folds[index]$
13 $\quad\quad Data_{Validating} \leftarrow folds[index]$
14 $\quad\quad Model_{Balanced} \leftarrow Get\_Best\_Model(Data_{Training}, Data_{Validating})$
15 $\quad Results_{Balanced} \leftarrow Model_{Balanced}.Evaluate(Data_{Testing})$

In our case, for OC-SVM, the cross-validation set is used to determine the best kernel and ν parameters. Note that many studies do not use a validation set and simply rely on the default parameters provided by the ML library. Based on best practices in ML, the use of cross-validation is

Algorithm 2: Process of training, validation, and testing of OCC algorithms using Cross-Validation approach.

Data: $Data_{Tuning}$
Result: $Results_{One-class}$
/* The size of data tuning is 70% and testing is 30% */
1 $Data_{Tuning}, Data_{Testing} \leftarrow split\_data(Data)$
2 $folds \leftarrow Generate\_Folds(Data_{Tuning}, 10)$
3 **for** $index = 1; index < k; index+ = 1$ **do**
4 $Data_{Training} \leftarrow$ $All\ Folds\ Except\ folds[index]$
5 $Data_{Validating} \leftarrow folds[index]$
6 $Data_{normal}, Data_{buggy} \leftarrow$ $DataFilter(Data_{Training})$
/* Merge the buggy data with Validation set */
7 $Data_{Validating} \leftarrow$ $Merge(Data_{Validating}, Data_{buggy})$
/* Train the OCC model only with normal data */
8 $Model_{Best} \leftarrow$ $Get\_Best\_Model(Data_{normal}, Data_{Validating})$
9 $Results_{One-class} \leftarrow$ $Model_{Best}.Evaluate(Data_{Testing})$

highly recommended in order to build more reliable models, avoid overfitting, and improve generalization to unseen data Yang and Shami (2020); Feurer and Hutter (2019). The k-fold cross-validation generally results in less biased models (compared to hold-out validation) since each instance in the training dataset is used for training and testing without overlapping Yang and Shami (2020); Feurer and Hutter (2019). Note that in this case we do not consider the time of commits and it is selected randomly between training and testing sets.

The entire process of training each algorithm (OCC and binary) is replicated 30 times and the average AUC for each classifier is reported along with the summary statistics shown in a boxplot (see Figure 4).

4.4.2. Time-sensitive validation approach

Tan el al. Tan, Tan, Dara and Mayeux (2015) proposed a time-sensitive validation (TV) approach to train JIT-SPD models. This method sorts commits chronologically and divides data into three-time windows: train, gap, and test. The goal is to prevent situations where future commits are predicted based on a training set that contains older commits.

Algorithm 3 describes the protocol of training models using the Time-sensitive validation approach. Line 1 organizes the data chronologically, arranging commits from the oldest to the newest, as shown in Figure 5 (Step 1). In Lines 3 and 4, the dataset is divided into three distinct parts: training (50%), validation (20%), and testing (30%)

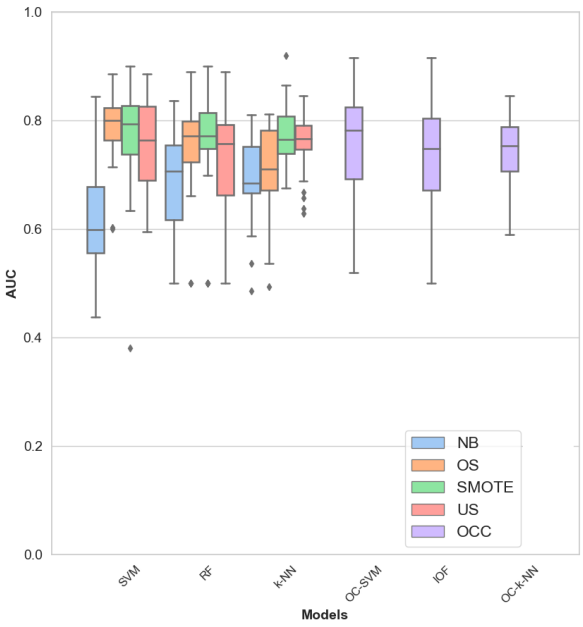

Figure 4: Overall performance of binary and OCC models using average AUC for all projects (Cross-Validation).

sets (Figure 5, Step 2). The data's specific characteristics influenced the decision to allocate 30% of the data for testing. Upon chronological sorting, we observed that most projects exhibited buggy data within the last 30% of commits. In Line 5, the training and validation sets were employed for hyper-parameter tuning to determine the optimal model, similar to the cross-validation approach. During the hyper-parameter tuning process, we implemented a bootstrapping approach on the training data to generate new sets for each test case. Assuming that the training data is represented as TR, with a population size of N (i.e., $TR = Tr_1, Tr_2, Tr_3, ..., Tr_N$), bootstrapping entails randomly selecting data points with replacement from TR to create a new training data of the same size as N, ensuring that the size of the sample remains the same as the original training data (50% as shown in Figure 5). This data is then used for model training and parameter validation, using the validation set as shown in Figure 5 (Step 3). Finally, in Line 6, the best model is evaluated by testing data. From Lines 7 to 11, the same procedure is applied with data balancing methods (i.e., OS, US, and SMOTE).

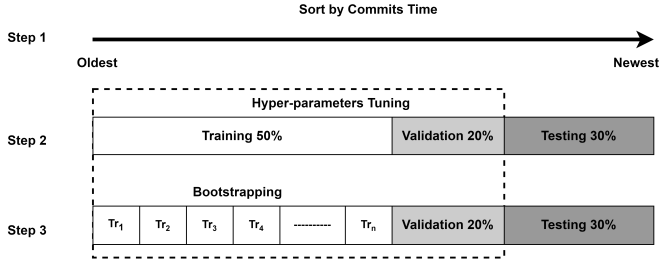

Figure 5: Splitting data using the time-sensitive validation Approach.

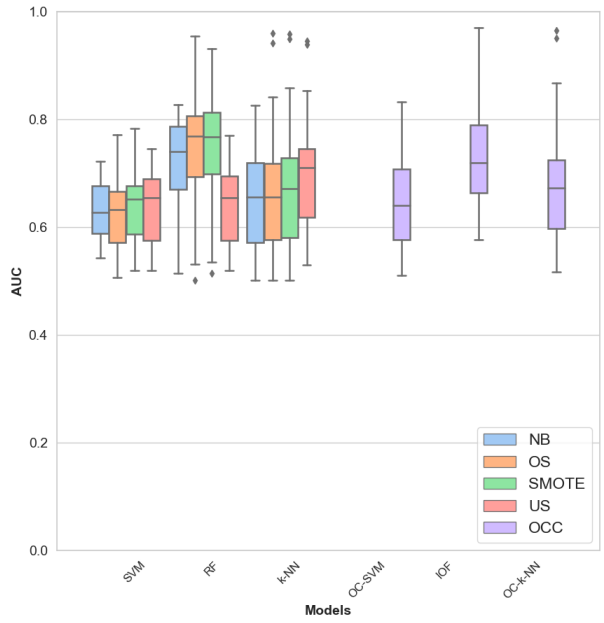

Figure 6: Overall performance of binary and OCC models using average AUC for all projects (time-sensitive validation).

Algorithm 3: Process of training, validation, and testing of binary algorithms using the Time-sensitive Validation approach.

1 **Data:** $Data$
Result: $Results_{imbalance}, Results_{Balanced}$
2 $Data \leftarrow Sort\_byTime(Data)$
/* The size of data tuning is 70% and testing is 30% */
3 $Data_{Tuning}, Data_{Testing} \leftarrow split\_data(Data)$
/* The size of data training is 50% and validation is 20% */
4 $Data_{Training}, Data_{Validating} \leftarrow split\_data(Data_{Tuning})$
/* Evaluate the binary model without balancing the data */
5 $Model_{Imbalance} \leftarrow Get\_Best\_Model(Data_{Training}, Data_{Validating})$
6 $Results_{Imbalance} \leftarrow Model_{Imbalance}.Evaluate(Data_{Testing})$
/* Evaluate the binary model after balancing the data */
7 $ImbalanceMethods \leftarrow \{OS, US, SMOTE\}$
8 **foreach** $Method \in Imbalance\_Methods$ **do**
9 $Data_{Tuning} \leftarrow Balancing(Data_{Training}, Method)$
10 $Model_{Balanced} \leftarrow Get\_Best\_Model(Data_{Training}, Data_{Validating})$
11 $Results_{Balanced} \leftarrow Model_{Balanced}.Evaluate(Data_{Testing})$

Algorithm 4 illustrates the protocol for OCC algorithms. In Lines 1 to 4, the same steps are applied to organize and split the data into training (50%), validation (20%), and testing (30%) sets (Figure 5, Step 2). In Line 5, the training data is filtered as normal and buggy. To keep the principles of the time-sensitive Validation approach, we omit the buggy commits from the training set to prevent any violation. In Line 6, we exclusively utilize the normal commits from the training data for the hyper-parameter tuning process with OCC algorithms. Subsequently, model validation takes place using normal and buggy commits from the validation data (20%).

Conversely, in the case of binary models, the inclusion of buggy data is essential during the training phase. Lastly, the best OCC model is evaluated by testing data (30%) in Line 7. We repeated this process 30 times, calculating the average AUC for each classifier as shown in Figure 6. This approach aims to assess the model's performance and stability by generating multiple samples that mimic the characteristics of the original training data.

Algorithm 4: Process of training, validation, and testing of OCC algorithms using the time-sensitive validation approach

1 **Data:** $Data_{Tuning}$
Result: $Results_{One-class}$
2 $Data \leftarrow Sort\_byTime(Data)$
/* The size of data tuning is 70% and testing is 30% */
3 $Data_{Tuning}, Data_{Testing} \leftarrow split\_data(Data)$
/* The size of data training is 50% and validation is 20% */
4 $Data_{Training}, Data_{Validating} \leftarrow split\_data(Data_{Tuning})$
/* Train the OCC model only with normal data */
5 $Data_{normal}, Data_{buggy} \leftarrow DataFilter(Data_{Training})$
6 $Model_{Best} \leftarrow Get\_Best\_Model(Data_{normal}, Data_{Validating})$
7 $Results_{One-class} \leftarrow Model_{Best}.Evaluate(Data_{Testing})$

5. Results and Discussions

In this section, we present and discuss the results of the experiment by providing answers to our research questions in the subsection sections.

5.1. RQ1: What is the overall performance of OCC algorithms compared to their binary classifier counterparts?

5.1.1. RQ1.1: Results using cross-validation

In this question, we look at the average AUC and F1-score achieved by the six models for JIT-SDP using three

Table 3
The average results of the JIT-SDP trained models with no balancing with cross-validation.

Classifiers	AUC	Improvement	F1 -Score	Improvement
OC-SVM	0.759	0.0%	0.769	0.0%
SVM	0.619	-18.4%	0.578	-24.9%
IOF	0.748	0.0%	0.779	0.0%
RF	0.678	-9.5%	0.678	-12.9%
OC-k-NN	0.755	0.0%	0.776	0.0%
k-NN	0.696	-7.8%	0.706	-9.0%

Table 4
Results of comparison between OCC and binary classifiers with balancing techniques OS, US, SMOTE using cross-validation.

Classifiers	AUC	Improvement	F1 -Score	Improvement
OC-SVM	0.759	0.0%	0.769	0.0%
SVM OS	0.785	3.5%	0.789	2.6%
SVM SMOTE	0.765	0.8%	0.769	0.1%
SVM US	0.748	-1.5%	0.753	-2.0%
IOF	0.748	0.0%	0.779	0.0%
RF OS	0.750	0.1%	0.766	-1.6%
RF SMOTE	0.752	0.5%	0.767	-1.5%
RF US	0.722	-3.5%	0.737	-5.4%
OC-k-NN	0.755	0.0%	0.776	0.0%
k-NN OS	0.708	-6.2%	0.720	-7.2%
k-NN SMOTE	0.769	1.9%	0.782	0.8%
k-NN US	0.757	0.3%	0.774	-0.3%

binary classifiers SVM, RF, and k-NN, and their corresponding OCC algorithms, i.e., OC-SVM, IOF, and OC-k-NN. These models are trained on 34 projects for JIT-SDP with and without balancing techniques. In RQ2, we dig deeper by examining the performance of the algorithms on individual projects.

Table 3 shows the results of the binary classifiers without balancing techniques. On average, OC-SVM performs the best among all classifiers with AUC = 0.759. It outperforms SVM, AUC = 0.619. Also, IOF performs better than RF on average (AUC = 0.748 compared to AUC = 0.678). The OC-k-NN achieved AUC = 0.755 compared to k-NN (AUC = 0.696). We also compute the improvement achieved by each binary method over its one-class counterpart. Improvement of A over B is calculated as (A-B)/B. We see that OC-SVM, IOF, and OC-KNN improve over SVM, RF, and K-NN by 18.4%, 9.5%, and 7.8%.

Table 3 also shows improvements with F1-score. Where the highest F1-score is recorded by IOF 0.779. In this case, we chose the optimal point on the ROC curve to measure the F1-score. The results of F1-score are similar to the AUC ones. We see that OC-SVM, IOF, and OC-KNN improve over SVM, RF, and K-NN by 24.9%, 12.9%, and 9.0% with F1-score. This point is observed with the CV approach due to the same IR between training and testing data compared to the TV approach where the IR is different between training and testing. More explanations are reported in the next section.

Table 4 shows that the average AUC of the JIT-SDP of all binary classifiers achieves a better average AUC when using over-sampling or SMOTE compared to one-class classifiers. The best improvement was achieved when using SVM OS (4.5%). Under-sampling did not improve the results of binary classifiers over OCC except for k-KNN, which improves by 1.5% the result obtained with OC-KNN. These results show that binary classifiers trained with balancing data approaches do not result in major improvements over OCC.

Furthermore, Table 4 shows results of F1-score where SVM outperform OC-SVM with OS and SMOTE balancing techniques with 2.6% and 0.1%, respectively. While the US degraded the performance of SVM compared to OC-SVM average F1-score. The IOF still outperforms RF with all balancing approaches in terms of F1-score. Finally, the k-NN outperforms OC-k-NN only with the SMOTE balancing approach with a small improvement of 0.8% in the F1-score.

5.1.2. RQ1.2: Results using time-sensitive validation

In this section, we discuss the results of time-sensitive validation. Five projects are excluded from these results (Derby, Oozie, Gora, Bookkeeper, and Helix). We decided to exclude these projects because the number of buggy commits in the testing set is less than 10, which resulted in outcomes that aren't robust, with significant variations upon replication, often yielding irrelevant results.

Overall, the performance of all models remained quite consistent with our previous experiments, both in binary and one-class classification scenarios. In Table 5, we present a comparison of the average AUC values and F1-scores for OCC and binary models without the use of data balancing techniques during training.

When examining the performance metrics, the OC-SVM model outperforms its binary counterpart, achieving higher AUC, and F1-score values. It achieves an average AUC of 0.646, while the SVM model reaches 0.628. Similarly, the F1-scores are 0.641 for OC-SVM and 0.618 for SVM, respectively.

In addition, the IOF and OC-k-NN models exhibit superior performance compared to their binary versions, RF and k-NN. The IOF model attains an AUC of 0.737, outperforming RF's 0.721. Likewise, the OC-k-NN achieves an AUC of 0.679, surpassing k-NN's 0.649.

Furthermore, in terms of F1-scores, the IOF model yields a result of 0.704, while RF scores 0.666. Similarly, the OC-k-NN model achieves an F1-score of 0.679, outpacing k-NN's 0.649.

Table 6 exhibits the outcomes of both OCC and binary models, incorporating balancing techniques (OS, SMOTE, and US), following the implementation of a time-sensitive

Table 5
The average results of the JIT-SDP trained models with no balancing with time-sensitive validation

Classifiers	AUC	Improvement	F1 -Score	Improvement
OC-SVM	0.646	0.0%	0.641	0.0%
SVM	0.628	-2.7%	0.618	-3.6%
IOF	0.737	0.0%	0.704	0.0%
RF	0.721	-2.1%	0.666	-5.3%
OC-k-NN	0.679	0.0%	0.677	0.0%
k-NN	0.649	-4.5%	0.559	-17.4%

Table 6
Results of comparison between OCC and binary classifiers with balancing techniques OS, US, SMOTE using time-sensitive validation.

Classifiers	AUC	Improvement	F1 -Score	Improvement
OC-SVM	0.646	0.0%	0.641	0.0%
SVM OS	0.626	-3.1%	0.592	-7.7%
SVM SMOTE	0.639	-1.0%	0.609	-5.0%
SVM US	0.671	4.0%	0.620	-3.3%
IOF	0.737	0.0%	0.704	0.0%
RF OS	0.748	1.6%	0.686	-2.6%
RF SMOTE	0.745	1.1%	0.688	-2.2%
RF US	0.646	-12.3%	0.623	-11.5%
OC-k-NN	0.679	0.0%	0.677	0.0%
k-NN OS	0.670	-1.4%	0.585	-13.7%
k-NN SMOTE	0.680	0.2%	0.607	-10.4%
k-NN US	0.707	4.1%	0.685	1.2%

approach. After applying data balancing strategies, the binary models consistently outperformed the OCC counterparts regarding AUC, and F1-score in some cases. For example, when employing the US balancing technique, SVM surpasses OC-SVM, achieving an AUC of 0.671 compared to OC-SVM's 0.646. However, OC-SVM remains superior to SVM when utilizing oversampling techniques (OS and SMOTE). Moreover, OC-SVM outperforms SVM across all balancing techniques when considering the F1-score.

Shifting our attention to RF, it attains average AUC values of 0.748 and 0.745 with the OS and SMOTE balancing techniques, respectively, compared to IOF's 0.737. Nevertheless, IOF still outperforms RF when employing the US balancing technique, where RF achieves an AUC of 0.646. Additionally, IOF surpasses RF across all balancing techniques when evaluating the F1-score. The MCC, however, demonstrates that IOF outperforms RF only when using the SMOTE and US balancing techniques.

Finally, the average AUC for OC-k-NN remains steady at 0.679. K-NN, on the other hand, outperforms OC-k-NN when applying the SMOTE and US balancing techniques, yielding AUC values of 0.680 and 0.707, respectively. K-NN's average AUC is 0.670, which is only slightly different from OC-k-NN's result. Notably, OC-k-NN outperforms K-NN across all balancing techniques except when employing the US technique, particularly in terms of F1-score.

The F1-score, while informative, does not provide a complete understanding of model performance because it is a threshold-dependent measure and is sensitive to imbalanced data. In contrast, as discussed in Section 4.3, the ROC curve offers a threshold-independent evaluation, illustrating the model's performance across all possible thresholds. At each point on the ROC curve, a different F1-score can be calculated, however, the AUC (area under the ROC curve) provides a more holistic metric for overall model accuracy, which is insensitive to the data imbalance. Most machine learning libraries internally select the "optimal" threshold by balancing the tpr and fpr (see Section 4.3). This threshold aligns, by default, with the point closest to the upper left corner of the ROC curve, as depicted in Figure 2. All threshold-dependent metrics, including the F1-score, are derived based on this threshold. However, relying solely on these threshold-based metrics can lead to deceptive results, particularly when the chosen threshold lies outside the domain application's region of interest, as demonstrated in Figure 7. For instance, the ROC curve in Figure 7 depicts k-NN's performance without data balancing for the Hive project. While the F1-score appears promisingly high at 0.61, this may not truthfully represent the model's genuine performance, as suggested by the AUC value of 0.53 – barely better than a random guess. More importantly, in the desired region where the fpr is low (specifically, below 20%), this model demonstrates little to no detection capability as the tpr approaches zero. Furthermore, if one were to choose this model based on its F1-score, corresponding to an operating point with an fpr of 60% and a tpr of 95%, it would prove impractical. Such a selection implies that, out of 100 commits, 60 healthy commits would be erroneously flagged as buggy. Given these considerations, the F1-score's capacity to accurately represent model performance becomes questionable, especially when there is an imbalance ratio discrepancy between the training and testing sets. Consequently, we have chosen to prioritize the AUC for subsequent research questions (RQ2 and RQ3).

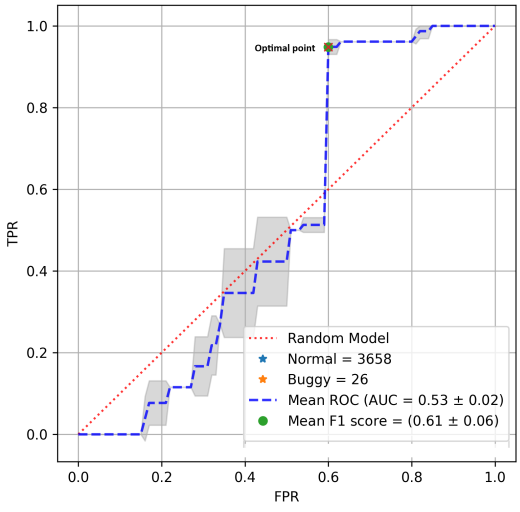

Figure 7: An example of testing for a project to display F1-score and AUC based on the ROC curve.

Finding RQ1: All one-class classifiers outperform their binary counterparts when no balancing technique is applied, resulting in a notable average improvement in AUC of up to 18.4% and 4.5% using CV and TV data splitting approaches, respectively. When data balancing is used, binary classifiers achieve slightly better than OCC methods. The improvement is between 1.9% to 3.5%. when using CV and between 1.1% to 4.1% when using the TV data splitting approach.

5.2. RQ2: How do OCC algorithms perform compared to binary classifiers when considering the data imbalance ratio?

In this question, we want to know how OCC methods perform when the data Imbalance Ratio (IR) of normal versus buggy commits is considered. This will help in determining when it is preferable to use OCC algorithms over binary algorithms.

5.2.1. RQ2.1: Results using cross-Validation

Table 21 (see Appendix) shows detailed results of the algorithms' performance using AUC. Individual project outcomes differ depending on the algorithm used. A closer examination of the results reveals that for projects cayenne, hive, jackrabbit, oodt, gora, bookkeeper, storm, spark, reef, helix, bigtop, curator, cocoon, and ambari, which have a medium to high data imbalance ratio (IR >= 22), all OCC algorithms (i.e., OC-SVM, IOF, and OC-kNN) consistently outperform binary classifiers with and without data balancing. This is also shown in Figure 8 using a boxplot of the average AUC for projects with medium and high IR (IR >= 22). The figure also shows that the OCC algorithms have less variability in their AUC results, which suggests that they are more stable and robust to noise in the data than their binary counterparts for projects with medium and high data imbalance ratios.

This finding suggests that software projects with a medium to high data imbalance ratio would benefit more from using one-class classifiers than a binary classification method to build JIT-SDP models. We show that when the number of normal commits to the number of buggy commits exceeds a certain threshold (in our case an IR ratio of 22 normal commits to 1 buggy commit), OCC algorithms should be considered. This is a significant finding because large software systems are expected to exhibit such imbalance. Considering the fact that OCC algorithms do not require the balancing of data during training and only need to be trained on normal commits, we believe that they are a better alternative than binary methods for large software systems.

The challenge of using OCC in practice is to determine automatically the threshold beyond which OCC algorithms should be used. Software developers can use different criteria including the maturity of the project, the criticality of the project, IR ratios based on past releases, the quality of the project, the overall development and quality assurance processes in place, etc. For example, for mature and stable projects that are developed by experienced developers, one may expect to see fewer defects being introduced, resulting in higher IR. Future work should concentrate on determining the criteria that affect the data imbalance ratio and how these criteria should be used to determine the threshold beyond which OCC should be used.

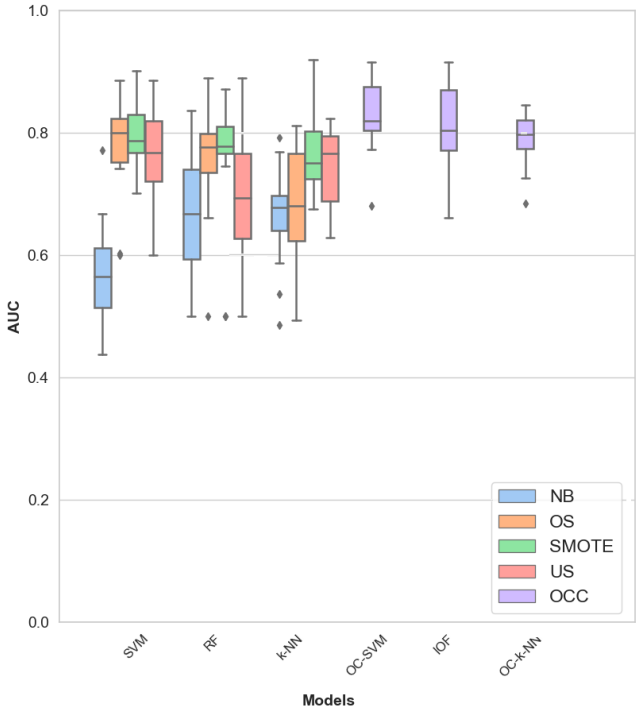

Figure 8: Average AUC of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (Cross-Validation).

For projects with low IR (a total of 20 projects out of 34), the results show that binary classifiers perform usually better than OCC methods (see Figure 9). Although the results vary from one algorithm to another, we can clearly see that OC-SVM performs worse than SVM for 16 projects out 34 (e.g., drill, flume, openjpa, camel, zookeeper, flink, carbondata, zeppelin, tez, phoenix, and oozie). IOF does well on only two projects (ignite and hadoop) out of 34 (i.e., 5% of the projects). OC-k-NN performs well on 1 project out of 34 (i.e., 2%). When comparing the accuracy of all the algorithms independently from the type of the algorithm (see the results highlighted in bold and underlined), we can see that, for projects with IR < 22, OCC algorithms provide the best results for only 6 projects (ignite, avro, hadoop, falcon, derby, and accumulo) out of 34 (i.e., a ratio of 17%). These results are obtained when using OC-SVM. In all other cases, binary classifiers (sometime even without data balancing - see for example flume and openjpa when using SVM with no balancing of data) perform better than OCC. These results clearly demonstrate that for projects with low IR (in our case IR < 22), it is preferable to use a binary classifier. The use of

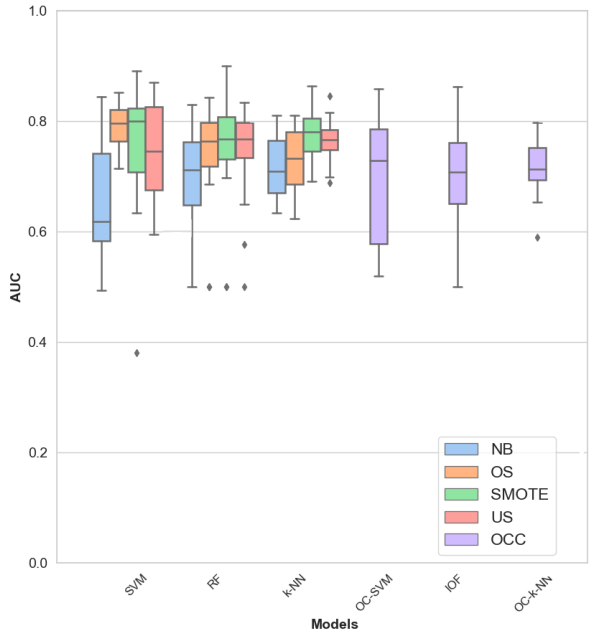

Figure 9: Average AUC of binary and OCC models for projects with low IR (IR<22) (Cross-Validation).

a balancing technique is also recommended as it was already stated in related work (see Song et al. Song et al. (2019)).

Table 7 shows the value of Cliff's δ for all six classifiers for projects with IR >= 22. The rows of the table represent the OCC models and the columns show the binary classifiers. We assess the magnitude of the difference between the AUC of a one-class algorithm and its binary version with no balancing, data balancing with over-sampling, SMOTE, and under-sampling. The results from Cliff's test indicate the extent of the differences between OC-SVM, OC-k-NN, and IOF, along with their corresponding binary classifiers.

We found that in 50% of the cases (6 out of 12), the accuracy of one-class algorithms exhibited a large effect size when compared to that of their binary versions with and without data balancing. For example, the accuracy of both OC-SVM and IOF show a large effect size ($\delta \geq 0.474$) when compared to their binary versions without balancing and with under-sampling. For the remaining cases, 4 out of 12 cases (33%) show a moderate effect size. There are only two cases where the effect size is small and this is between IOF and RF-OS (δ =0.327) and IOF and RF-SMOTE (δ =0.270). A moderate to large effect size means that this research finding has a practical significance Romano et al. (2006); Cohen (1992).

Table 8 shows the Cliff's δ values for all models for projects with low IR (IR < 22). The results indicate a moderate effect size for 5 out of 12 cases (41.66%) (see for example, the effect size between IOF and RF-US, which is -0.403), a large size effect in 2 cases out of 12 (16.16%), and a small size effect in 5 cases out of 12 (41.66%). We also observe that when no data balancing is used, the effect size in all cases is small.

Table 7
The Cliff's δ of AUC between OCC and binary models for project with medium and high IR (Cross-Validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	0.990	0.372	0.490	0.342
	NB-RF	**OS-RF**	**US-RF**	**SMOTE-RF**
IOF	0.740	0.327	0.602	0.270
	NB-k-NN	**OS-k-NN**	**US-k-NN**	**SMOTE-k-NN**
OC-k-NN	0.857	0.694	0.418	0.332

Table 8
The Cliff's δ of AUC between OCC and binary models with low IR (Cross-Validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	0.180	-0.432	-0.212	-0.355
	NB-RF	**OS-RF**	**US-RF**	**SMOTE-RF**
IOF	-0.005	-0.360	-0.403	-0.432
	NB-k-NN	**OS-k-NN**	**US-k-NN**	**SMOTE-k-NN**
OC-k-NN	0.015	-0.180	-0.575	-0.657

Also Figure 10 shows the results of F1-score using CV approach with medium and high IR (the detailed F1-score results for CV can be found in Table 22 within the Appendix section). It can be clearly seen that the OCC models' performance is higher than binary ones even with balanced data. On the other side, the binary models start to outperform OCC models, especially with data balancing methods as shown in Figure 11 when IR is low.

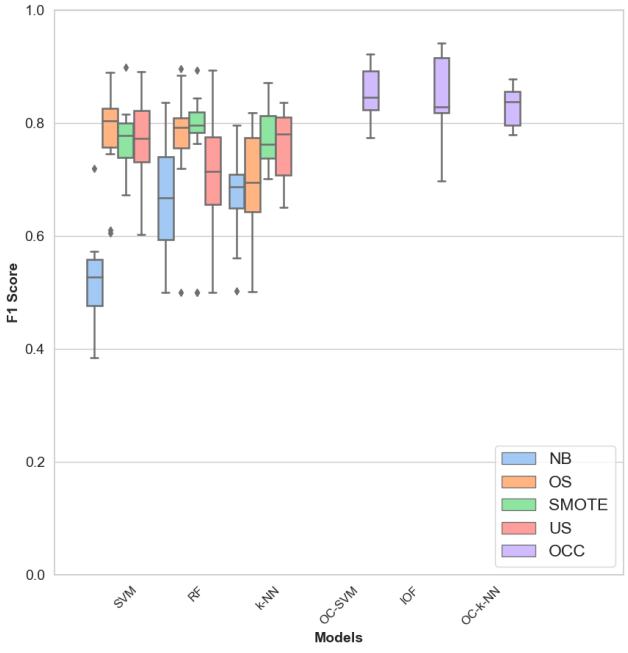

Figure 10: Average F1-score of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (Cross-Validation).

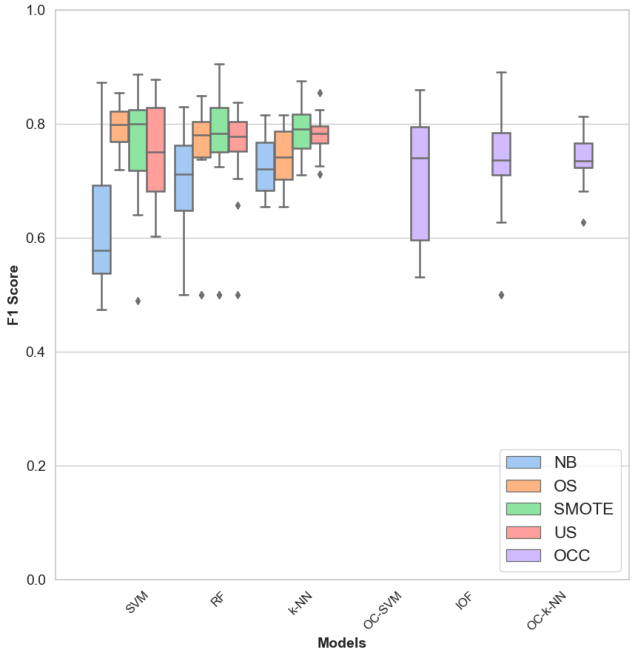

Figure 11: Average F1-score of binary and OCC models for projects with low IR (IR<22) (Cross-Validation).

Table 9
The Cliff's δ of F1-score between OCC and binary models for projects with medium and high IR (Cross-Validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	1.000	0.628	0.673	0.806
	NB-RF	**OS-RF**	**US-RF**	**SMOTE-RF**
IOF	0.867	0.607	0.750	0.582
	NB-k-NN	**OS-k-NN**	**US-k-NN**	**SMOTE-k-NN**
OC-k-NN	0.964	0.827	0.612	0.628

Table 9 presents the Cliff's δ values for F1-scores in projects characterized by medium and high IR. In all cases, the Cliff's δ values clearly demonstrate that OCC models statistically outperform binary models, and large effect sizes characterize these differences.

However, when we examine the Cliff's δ values in Table 10, focusing on projects with low IR, we observe that OCC models outperform binary models only when data balancing is not applied. To illustrate, without data balancing, both OC-SVM and IOF exhibit superior performance compared to NB-SVM and NB-RF, with moderate effect sizes ($0.147 \leq \delta \leq 0.474$). OC-k-NN outperforms NB-k-NN with a small effect size ($\delta = 0.147$). Interestingly, binary models take the lead and exceed OCC models once data balancing is introduced, displaying moderate to large δ values.

Table 10
The Cliff's δ of F1-score between OCC and binary models with low IR (Cross-Validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	0.465	-0.415	-0.205	-0.333
	NB-RF	**OS-RF**	**US-RF**	**SMOTE-RF**
OC-RF	0.220	-0.340	-0.325	-0.388
	NB-k-NN	**OS-k-NN**	**US-k-NN**	**SMOTE-k-NN**
OC-k-NN	0.147	-0.280	-0.575	-0.593

Finding RQ2.1: We found that OCC algorithms outperform binary classifiers with and without data balancing techniques for all projects with medium to high data imbalance ratio (IR >= 22 in our case). For projects with a low IR (IR < 22 for our datasets), binary classifiers perform better than OCC ones in the majority of the cases. This finding suggests that OCC JIT-SDP models should be used in situations with IR is high enough. The challenge, however, is to determine the right IR threshold beyond which the use of OCC is warranted.

5.2.2. RQ2.2: Results using time-sensitive validation

Table 23 (see Appendix) provides a detailed breakdown of the results for all six models when using the time-sensitive validation approach. Five projects (Derby, Oozie, Gora, Bookkeeper, and Helix) were excluded due to their limited presence of buggy commits in the testing set (fewer than 10). Across the board, the OCC algorithms consistently outperform the binary ones, particularly when dealing with projects with medium and high IR as measured by AUC. For instance, the OCC models achieved the best results on 9 out of 29 projects (representing 31%) except for two projects, Jackrabbit and Bigtop. This discrepancy arises from variations in IR values between the training, validation, and testing sets, resulting from the data distribution when sorted chronologically. Section (5.2.3) elaborates further on these cases. This observation persists even when applying balancing techniques such as OS, US, and SMOTE.

Figure 12 visualize the overall AUC results of 11 projects when the IR is medium or high (more than 21 in our case). As mentioned previously, the OCC algorithms get an advantage when IR is medium or high compared to their binary counterparts. On the other hand, Figure 13 displays the AUC of all 29 projects with low IR also using time-sensitive validation. As we can see, the binary algorithms perform better than OCC ones. For example, the OC-SVM achieved lower performance on 14 projects from a total of 29 projects. IOF performed worst on 16 projects from a total

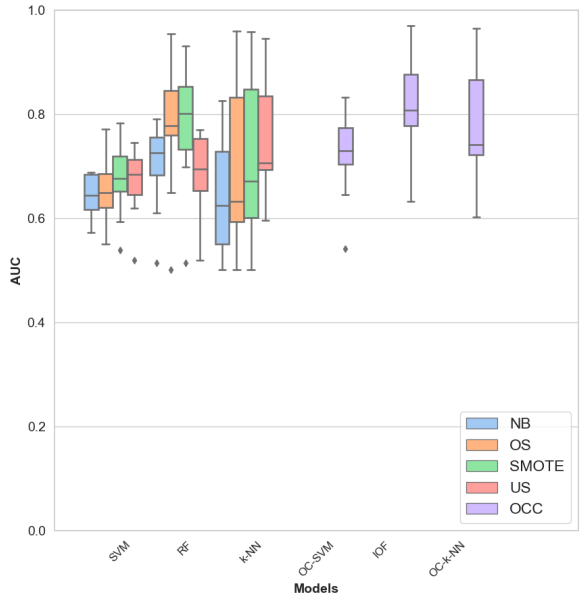

Figure 12: Average AUC of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (time-sensitive validation).

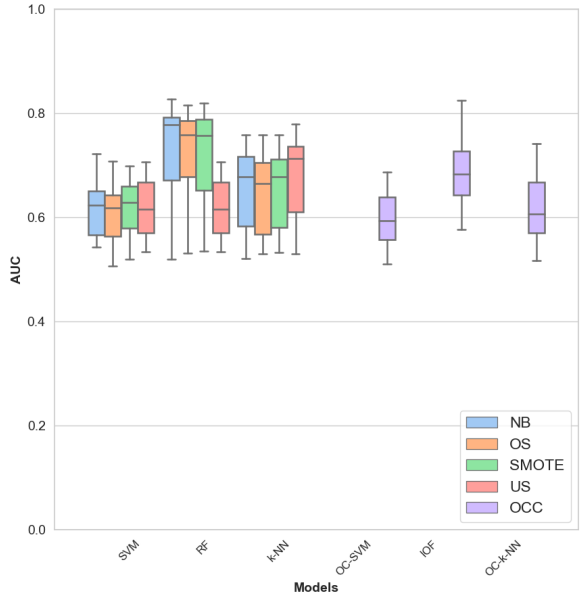

Figure 13: Average AUC of binary and OCC models for projects with low IR (IR<22) (time-sensitive Validation).

of 29 ones and OC-k-NN had 15 worst results out of a total of 29 projects.

Table 11 represents the Cliff's δ for OCC models and their binary versions with medium and high IR using time-sensitive validation. The OC-SVM shows a larger impact than SVM, even with balancing approaches (all $\delta \geq 0.474$).

Table 11
The Cliff's δ of AUC between OCC and binary models for projects with medium and high IR (time-sensitive validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	0.736	0.562	0.512	0.478
	NB-RF	**OS-RF**	**US-RF**	**SMOTE-RF**
IOF	0.661	0.207	0.785	0.190
	NB-k-NN	**OS-k-NN**	**US-k-NN**	**SMOTE-k-NN**
OC-k-NN	0.595	0.306	0.281	0.289

Table 12
The Cliff's δ of AUC between OCC and binary models with low IR (time-sensitive validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	-0.201	-0.086	-0.210	-0.225
	NB-RF	**OS-RF**	**US-RF**	**SMOTE-RF**
IOF	-0.398	-0.370	-0.562	-0.296
	NB-k-NN	**OS-k-NN**	**US-k-NN**	**SMOTE-k-NN**
OC-k-NN	-0.302	-0.281	-0.463	-0.343

The Cliff's δ of IOF is large compared to RF with imbalanced and US approaches. While it shows a small size impact using OS and SMOTE balancing techniques with RF. Finally, the OC-k-NN Cliff's δ shows a large size impact compared to k-NN without balancing data with a small size impact with OS, US, and SMOTE. These results collectively indicate that OCC models consistently outperform binary models in a statistically significant manner when dealing with projects with medium or high IR scenarios.

Table 12 presents the Cliff's δ values for projects characterized by low IR. All the results in the table are negative, indicating that in these cases, OCC models underperform compared to their binary counterparts. For instance, SVM indicates a statistically significant but small size impact compared to OC-SVM. Additionally, RF outperforms IOF with a moderate effect size when employing NB and OS techniques, while RF achieves superior results over IOF using the US with a large effect size. Furthermore, RF records a small effect size compared to IOF when using SMOTE. Lastly, k-NN demonstrates a moderate effect size compared to OC-k-NN when employing NB and OS, and Cliff's δ indicates a moderate effect size when using US and SMOTE.

In terms of F1-score, Figure 14 illustrates the average results using the TV approach through boxplots for projects characterized by medium and high IR. The detailed F1-score results for TV can be found in Table 24 within the Appendix section. Among these results, OC-SVM shows only slight variations compared to SVM, while IOF consistently yields higher results than RF, even when data balancing techniques are applied, except in the case of RF-SMOTE, which exhibits similar results to IOF. Additionally, OC-k-NN consistently outperforms k-NN when used in conjunction with NB, OS,

Table 13
The Cliff's δ of F1-score between OCC and binary models with medium and high IR (time-sensitvie validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	0.397	0.355	0.116	0.331
	NB-RF	**OS-RF**	**US-RF**	**SMOTE-RF**
IOF	0.537	0.157	0.570	0.083
	NB-k-NN	**OS-k-NN**	**US-k-NN**	**SMOTE-k-NN**
OC-k-NN	0.744	0.405	0.207	0.339

and SMOTE balancing methods, while k-NN-US demonstrates similar performance to the OC-k-NN model.

Table 13 provides insight into the Cliff's δ values for the F1-score in projects characterized by medium and high IR. OC-SVM exhibits a moderate effect size (with δ values ranging from 0.147 to 0.474) when compared to SVM-NB, SVM-OS, and SVM-SMOTE. Conversely, it shows a small effect size when compared to SVM-US (δ values less than or equal to 0.147). In contrast, IOF demonstrates a large effect size when compared to RF-NB and RF-US, with Cliff's δ values greater than or equal to 0.474. However, when compared to RF-OS and RF-SMOTE, the effect size is small, indicated by Cliff's δ values less than or equal to 0.147. Finally, OC-k-NN exhibits a moderate to large effect size when compared to k-NN, both with and without balancing methods (OS and SMOTE), while k-NN-US shows a moderate effect size.

Transitioning to projects with low IR, it becomes evident that binary models consistently maintain their advantage over one-class classifiers in terms of the F1-score. This pattern is clearly illustrated in Figure 15, regardless of whether data balancing techniques are applied or not. Further confirmation of this trend is found in the Cliff's δ values presented in Table 14, where all δ values are negative. These negative values indicate that across the board, binary models consistently outperform OCC models, and the magnitude of this advantage is moderate to large in terms of effect size.

Finding RQ2.2: Our investigation reveals that OCC algorithms consistently outperform binary classifiers across all projects characterized by medium to high IR levels (IR >= 22 in our study), even when using the TV data splitting approach. It is worth noting that two projects, Bigtop and Jackrabbit, exhibit unique behavior due to the distribution of buggy data after commits are sorted by time. These projects represent special cases and require separate consideration. In contrast, for projects with a low IR, binary classifiers consistently exhibit superior performance compared to OCC models in most cases, even without data balancing techniques.

5.2.3. Fine-grained discussion

We also carefully analyzed projects with IR exceeding 21, revealing that data distribution significantly affects OCC algorithms in contrast to binary ones. This observation

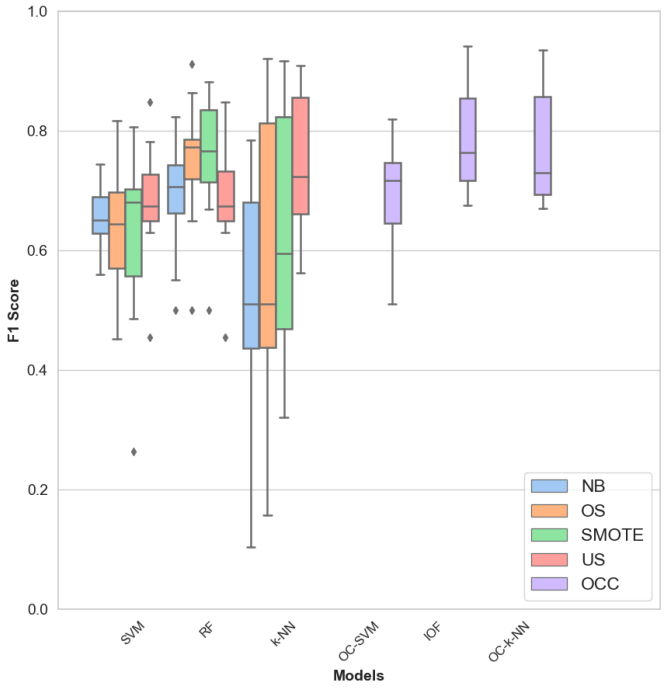

Figure 14: Average F1-score of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (time-sensitive Validation).

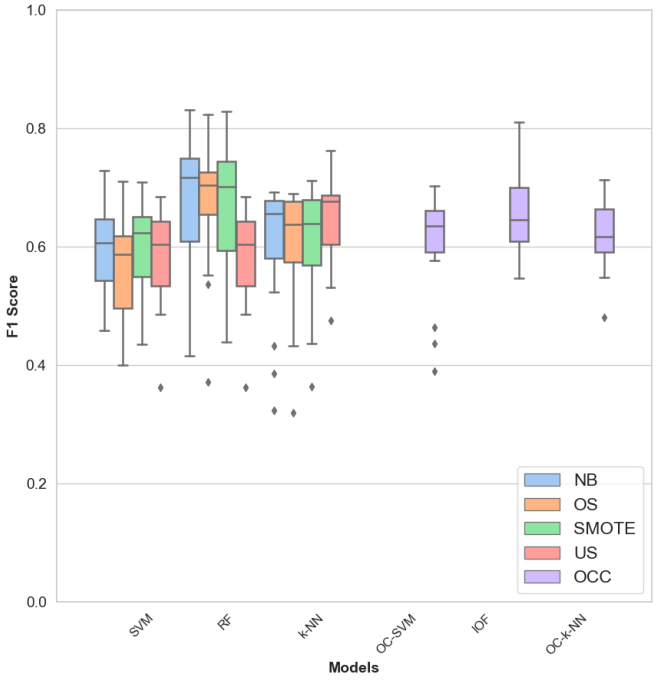

Figure 15: Average F1-score of binary and OCC models for projects with low IR (IR<22) (time-sensitive validation).

becomes obvious when the data is chronologically sorted, leading to variations in the distribution of buggy commits. For instance, when we sort the data by time and divide it

Table 14
The Cliff's δ of F1-score between OCC and binary models with low IR (time-senstivie validation)

	NB-SVM	OS-SVM	US-SVM	SMOTE-SVM
OC-SVM	-0.501	-0.404	-0.210	-0.148
	NB-RF	OS-RF	US-RF	SMOTE-RF
IOF	-0.321	-0.340	-0.451	-0.238
	NB-k-NN	OS-k-NN	US-k-NN	SMOTE-k-NN
OC-k-NN	-0.196	-0.386	-0.330	-0.133

into three parts: training, validation, and testing (see Figure 5), we encounter varying counts of buggy commits in each segment, reflecting real-world scenarios. However, this distribution discrepancy poses challenges for OCC models, particularly when the number of buggy commits in the validation set is lower than in the training set. The OCC models require additional information during the hyper-tuning process since they assess their parameters based on buggy commits in the validation set. In contrast, binary models outperform OCC models due to their inherent nature, which employs two classes during training, as opposed to OCC models that rely solely on one class.

For example, the OCC models do not perform well with projects (Jackrabbit and Bigtop), even with higher IR. When we dug deeper into data distribution we found that the count of buggy data represented after sorting data by commit time is as follows for project Jackrabbit: training (228), validation (50), and testing (92). As we can see most of the buggy commits are placed in the training set. In the case of OCC, this data is dropped and not used at all, while binary classifiers use them to discover the class boundaries. Therefore, the binary models get an advantage over OCC.

In contrast, OCC models exhibit superior performance compared to binary models in situations where the validation set contains more buggy commits. Take, for example, the distribution of buggy commits in the project Parquet-mr after sorting the data: training (7), validation (30), and testing (77). In such cases, binary classifiers encounter challenges due to the limited information available in the training data. Additionally, balancing methods yield little improvements, primarily due to the noise in the data. On the other hand, OCC models perform better for two key reasons. Firstly, the OCC models are trained exclusively on normal data and evaluated on the validation set. This approach helps them adapt effectively to scenarios where buggy commits are more prevalent in the validation data. Secondly, the OCC models are less sensitive to the noise introduced by data balancing processes, as they do not require such balancing. They contribute to their improved performance in situations with imbalanced data.

In such instances, OCC identifies buggy commits with a higher IR (exceeding 21). Interestingly, when data is balanced using various techniques like OS, SMOTE, and US, binary models tend to identify the same buggy commits as OCC. However, these balancing methods directly impact binary models, as highlighted by Song et al. Song et al. (2019). Notably, we discovered that binary models tend to generate more false positives due to these balancing methods, leading to a degradation in their overall performance. In other words, while OCC and (binary models + balanced data) identify the same buggy commits, binary models exhibit reduced performance due to increased false positive results.

> **Finding RQ2.3:** We found that the distribution of buggy data during time validation has an impact on the performance of binary and OCC models. To illustrate this, OCC models exhibit better performance when most of the buggy data are included in the validation set during the model-building process. Conversely, having buggy data in the training set does not affect OCC models since these models are trained on normal data only. In contrast, binary models need to incorporate both normal and buggy data during training.

5.3. RQ3: Which features affect the accuracy of OCC algorithms compared to their binary counterparts?

In the previous question, we found that the accuracy of OCC algorithms depends on the data imbalance ratio of the project. In this question, we aim to understand which feature set (i.e., diffusion, size, purpose, history, and experience) has the most impact on the results. The feature sets used to train the algorithms are shown in Table 2. We rank the features based on their importance and investigate the effect of using the top 9 features on the performance of the OCC and binary algorithms.

To achieve this, we extract the most important features from the Random Forest decision trees. It calculates feature importance by assessing how much each feature contributes to reducing impurity when splitting decision tree nodes, making it a straightforward and interpretable choice Zheng and Casari (2018). Table 15 lists the 14 features in the descending order of importance. Table 16 shows the average AUC of different algorithms trained on datasets with low IR (IR $<$ 22). It includes the results when all features are used and when only using the top 9 features. We only kept the top 9 features shown in Table 15 because the other features (ND, Fix, REXP, SEXP, and EXP) did not result in any improvements to the models. The algorithm with the highest accuracy is highlighted with a gray background. We can observe that the accuracy of all algorithms has improved when using the top 9 features. We also found that the accuracy of binary classifiers with data balancing techniques remains superior to that of OCC algorithms for projects with low IR.

Table 17 presents the average AUC of various algorithms trained on datasets with medium and high IR (IR $>=$ 22). The table includes results for all features and the top 9 features. We see that the best results are obtained when using

Table 15
Ranking of feature importance for JIT-SDP classifiers using Cross-Validation.

Features	Importance Ranking
NF	26.223
NS	25.577
LT	25.434
LA	24.883
Entropy	23.042
AGE	20.012
LD	18.052
NDEV	15.741
NUC	15.043
ND	5.412
Fix	3.652
REXP	3.256
SEXP	2.124
EXP	2.004

Table 16
Impact of feature sets on average AUC for JIT-SDP projects with low IR (IR<22)

Classifiers	Metrics	Average AUC of low IR			
		NB	US	OS	SMOTE
RF	All	0.691	0.748	0.742	0.736
	Top 9 only	0.698	0.749	0.730	**0.758**
IOF	All	0.691	-	-	-
	Top 9 only	0.708	-	-	-
k-NN	All	0.718	0.765	0.730	0.760
	Top 9 only	0.744	0.770	0.735	**0.779**
OC-k-NN	All	0.715	-	-	-
	Top 9 only	0.700	-	-	-
SVM	All	0.655	0.740	0.791	0.784
	Top 9 only	0.706	0.796	**0.806**	0.783
OC-SVM	All	0.696	-	-	-
	Top 9 only	0.736	-	-	-

Table 17
Impact of feature sets on average AUC for JIT-SDP projects with medium & high IR (Cross-Validation)

Classifiers	Metrics	Average AUC of medium & high IR			
		NB	US	OS	SMOTE
RF	All	0.659	0.685	0.761	0.776
	Top 9 only	0.670	0.630	0.707	0.760
IOF	All	0.804	-	-	-
	Top 9 only	**0.830**	-	-	-
k-NN	All	0.665	0.747	0.677	0.782
	Top 9 only	0.685	0.719	0.671	0.773
OC-k-NN	All	0.790	-	-	-
	Top 9 only	**0.805**	-	-	-
SVM	All	0.568	0.758	0.776	0.737
	Top 9 only	0.613	0.807	0.795	0.801
OC-SVM	All	0.826	-	-	-
	Top 9 only	**0.863**	-	-	-

Table 18
Ranking of feature importance for JIT-SDP classifiers using time-sensitive validation.

Features	Importance Ranking
NS	12.507
NF	11.552
LT	10.415
Entropy	10.167
LA	10.071
AGE	8.268
LD	7.217
NUC	6.737
NDEV	6.178
ND	2.352
Fix	1.352
REXP	1.304
SEXP	1.054
EXP	1.007

OCC with the top 9 features. IOF, OC-K-NN, and OC-SVM achieve an average AUC with the Top 9 features of 0.830, 0.805, and 0.863.

The conclusion from answering RQ3 is that the choice of the feature sets has an impact on the accuracy. The results suggest that projects with medium to high IR (in our case, IR >= 22) require fewer feature sets than projects with low IR (IR<22). Feature selection reduces the redundancy and multicollinearity among features, which can improve the accuracy of machine learning algorithms. Furthermore, feature selection alleviates the curse of the dimensionality problem, which indicates that the number of instances in the training data set that need to be accessed grows exponentially with the underlying dimensionality (number of features). This becomes a bigger issue when training binary classifiers on imbalanced datasets due to the difficulty in obtaining more positive examples, while a large number of negative examples are typically available (or easy to acquire) for training OCC algorithms. More importantly, for practical applications, selecting fewer features reduces the training and response time and allows for a better understanding of the data and the behavior of the algorithms Chandrashekar and Sahin (2014).

We also do the same process with the time-sensitive validation protocol. Table 18 displays the 14 features ranked based on their importance using the RF algorithm. The rank of features is different after applying the time-sensitive validation approach compared to Cross-Validation. However, the RF model shows the same top 9 features with lower values in importance.

Table 19 displays the average AUC of projects with low IR using the time-sensitive Validation approach. The best results are recorded when we use only the top 9 features. The binary models still perform better than OCC ones, but balancing approaches show changes. For instance, we can

Table 19
Impact of feature sets on average AUC for JIT-SDP projects with low IR (time-sensitive validation)

Classifiers	Metrics	Average AUC of low IR			
		NB	US	OS	SMOTE
RF	All	0.732	0.619	0.728	0.722
	Top 9 only	0.779	0.678	0.791	**0.798**
IOF	All	0.687	-	-	-
	Top 9 only	0.760	-	-	-
k-NN	All	0.654	0.678	0.718	0.731
	Top 9 only	0.693	**0.742**	0.718	0.731
OC-k-NN	All	0.619	-	-	-
	Top 9 only	0.703	-	-	-
SVM	All	0.618	0.674	0.607	0.618
	Top 9 only	0.677	0.674	0.667	**0.680**
OC-SVM	All	0.597	-	-	-
	Top 9 only	0.670	-	-	-

Table 20
Impact of feature sets on average AUC for JIT-SDP projects with medium & high IR (time-sensitive Validation)

Classifiers	Metrics	Average AUC of medium & high IR			
		NB	US	OS	SMOTE
RF	All	0.704	0.692	0.782	0.783
	Top 9 only	0.721	0.712	0.770	0.795
IOF	All	0.819	-	-	-
	Top 9 only	**0.836**	-	-	-
k-NN	All	0.640	0.754	0.708	0.724
	Top 9 only	0.658	0.733	0.686	0.706
OC-k-NN	All	0.785	-	-	-
	Top 9 only	**0.788**	-	-	-
SVM	All	0.646	0.671	0.656	0.675
	Top 9 only	0.660	0.689	0.693	0.721
OC-SVM	All	0.726	-	-	-
	Top 9 only	**0.740**	-	-	-

see RF still recorded the best average AUC using SMOTE. On the other side, the k-NN algorithm gets the best average AUC with US, while it gets the best average AUC using SMOTE with the Cross-Validation approach. Also SVM algorithm gets a different best average AUC, where the best result with time-sensitive Validation is SVM-SMOTE. While the SVM-OS is the best result with Cross-Validation.

Table 20 presents the average AUC values for projects with moderate to high IR using the time-sensitive validation approach. The OCC algorithms achieve the most favorable average AUC scores when the top 9 features are considered. Interestingly, the IOF method demonstrates a similar average AUC performance when employing both Cross-Validation and time-sensitive Validation approaches. However, when comparing the OC-SVM and OC-k-NN algorithms, we observe that they yield higher average AUC values with the Cross-Validation approach than the time-sensitive Validation approach. Nevertheless, it is worth noting that OCC algorithms consistently outperform counterpart models when dealing with moderate to high IR.

Finding RQ3 : We found that the accuracy of the algorithms improved when using the top 9 ranked features based on their importance. For projects with low IR, binary classifiers outperform OCC algorithms when using the top 9 features. For projects with medium to high IR, all OCC algorithms outperform their binary versions with and without data balancing using both data splitting approaches CV and TV.

6. Threats to Validity

We now discuss the threats to the validity of our results and recommendations.

Construct Validity: Construct validity threats concern the accuracy of the observations with respect to the theory. We used six machine learning algorithms that are well-studied in the literature. We followed the conventional way of training, validation, and testing. We also used the AUC, a threshold-independent evaluation metric, to assess the performance of the classification algorithms. We argued that the AUC is a more representative metric than the F1-score, which is tied to a specific threshold. Thus, we believe that there is no threat to the construct validity of our results and recommendations besides the threat to any experimental studies in software engineering where the use of other datasets, especially those from industry, may impact the results.

Internal Validity: Internal validity threats are factors that may have an impact on our results. The selection of the algorithms is one possible threat. We mitigated this threat by using powerful algorithms that are known to perform well in various classification tasks and are used in many research fields. Another threat is concerned with the datasets that we selected. Although, we experimented with 34 different Java Apache projects, using additional datasets including those written in different programming languages should provide better generalizability of the results. Another threat to internal validity is the implementation of the scripts we use to run the experiments. To mitigate this threat, all authors have tested the scripts rigorously to ensure that they work properly. We also make all the data and scripts available online[3] to other researchers.

Conclusion Validity: Conclusion validity threats correspond to the correctness of the obtained results. We selected six machine learning algorithms based on their excellent performance in various research fields. We made every effort to follow proper machine learning procedures to conduct the experiments. We also make the data and scripts available online to allow the assessment and reproducibility of our results.

[3]https://github.com/wahabhamoulhadj/jit-occ

External Validity: External validity is related to the generalizability of the results. We experimented with 34 datasets from different software projects. We do not claim that our results can be generalized to all projects, in particular industrial, proprietary systems to which we did not have access.

In addition, we used the implementation of RA-SZZ that is provided by the authors [4] to label the dataset of into normal and buggy commits. Although RA-SZZ is a powerful labelling technique, errors in the implementation may occur, which can impact our results.

7. Conclusion

In this paper, we investigate the use of OCC algorithms for JIT-SDP. To achieve this, we experimented with three OCC algorithms, OC-SVM, IOF, and OC-k-NN using 34 datasets. We compared their performance to their corresponding binary classifiers, SVM, RF, and k-NN using two data splitting and evaluation approaches (Cross-Validation and Time-aware Validation). We found that for projects with medium to high IR (in our case IR >= 22), OCC algorithms outperform binary classifiers for all projects. We also found that for these projects, OCC requires fewer features for training than the other projects. These findings are significant because they show that for projects with a medium to high IR, OCC should be favored over binary classification. The challenge, however, is to determine the threshold beyond which OCC methods should be favored. We expect that this threshold would vary from one project to another.

Future directions should focus on the following aspects. First, we need to work towards determining the criteria that software engineers should use to determine the IR threshold beyond which OCC should be used. Examples include the maturity of the subject system, IR ratios from past releases, etc. Because a software system evolves over time, there is a need to constantly check that the criteria hold for major subsequent changes in the systems to determine whether OCC is still a viable option. Second, we need to experiment with more systems from different domains that are written in various programming languages to generalize our findings. Furthermore, we should explore other OCC algorithms and the combination of the algorithms such as those used in anomaly detection research for the detection of outliers (e.g., Islam et al. (2018); Khreich, Murtaza, Hamou-Lhadj and Talhi (2018)). We also need to compare with more binary classifiers using different balancing techniques. Moreover, we should also apply OCC to cross-projects and determine the best IR for cross-project JIT-SDP tasks where data from many projects are used for training, which may result in a higher data imbalance ratio, further favoring the use of OCC. Finally, we need to experiment with deep learning algorithms and other feature sets such as semantic features extracted from commit messages and code change.

[4] RA-SZZ Github repository: https://github.com/danielcalencar/ra-szz

8. Replication Package

All the data, scripts, and results discussed in this paper are available on Github:
https://github.com/wahabhamoulhadj/jit-occ

9. Acknowledgment

We would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for partly supporting this research.

References

Baeza-Yates, R.A., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., USA.

Bellinger, C., Sharma, S., Japkowicz, N., 2012. One-class versus binary classification: Which and when?, in: 2012 11th International Conference on Machine Learning and Applications, pp. 102–106. doi:10.1109/ICMLA.2012.212.

Bishop, C.M., 2006. Pattern recognition and machine learning. Information science and statistics, Springer, New York, NY.

Bruce, P., Bruce, A., 2017. Practical statistics for data scientists: 50 essential concepts. O'Reilly Media, Inc.

Butcher, B., Smith, B.J., 2020. Feature engineering and selection: A practical approach for predictive models. The American Statistician 74, 308–309. doi:10.1080/00031305.2020.1790217.

Cabral, G.G., Minku, L.L., Shihab, E., Mujahid, S., 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 666–676. doi:10.1109/ICSE.2019.00076.

Catolino, G., Di Nucci, D., Ferrucci, F., 2019. Cross-project just-in-time bug prediction for mobile apps: An empirical assessment, in: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 99–110.

Chandrashekar, G., Sahin, F., 2014. A survey on feature selection methods. Computers & Electrical Engineering 40, 16–28. doi:https://doi.org/10.1016/j.compeleceng.2013.11.024. 40th-year commemorative issue.

Chicco, D., Jurman, G., 2020. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. BMC genomics 21, 1–13.

Cliff, N., 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological Bulletin 114, 494–509.

Cohen, J., 1992. A power primer. Psychological Bulletin 112, 155–159.

Fan, Y., Xia, X., da Costa, D.A., Lo, D., Hassan, A.E., Li, S., 2021. The impact of mislabeled changes by szz on just-in-time defect prediction. IEEE Transactions on Software Engineering 47, 1559–1586. doi:10.1109/TSE.2019.2929761.

Fawcett, T., 2006. An introduction to roc analysis. Pattern Recognition Letters 27, 861–874. URL: https://www.sciencedirect.com/science/article/pii/S016786550500303X, doi:https://doi.org/10.1016/j.patrec.2005.10.010. rOC Analysis in Pattern Recognition.

Feurer, M., Hutter, F., 2019. Hyperparameter Optimization. Springer International Publishing, Cham. pp. 3–33. doi:10.1007/978-3-030-05318-5_1.

Fu, W., Menzies, T., 2017. Revisiting unsupervised learning for defect prediction, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 72–83. URL: https://doi.org/10.1145/3106237.3106257, doi:10.1145/3106237.3106257.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N., 2014. An empirical study of just-in-time defect prediction using cross-project models, in: Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14), p. 172–181. doi:10.1145/2597073.2597075.

Hariri, S., Kind, M.C., Brunner, R.J., 2021. Extended isolation forest. IEEE Transactions on Knowledge and Data Engineering 33, 1479–1489. doi:10.1109/TKDE.2019.2947676.

Hart, P.E., Stork, D.G., Duda, R.O., 2000. Pattern classification. Wiley Hoboken.

He, H., Garcia, E.A., 2009. Learning from imbalanced data. IEEE Transactions on Knowledge and Data Engineering 21, 1263–1284. doi:10.1109/TKDE.2008.239.

Herbold, S., Trautsch, A., Trautsch, F., Ledel, B., 2022. Problems with szz and features: An empirical study of the state of practice of defect prediction data collection. Empirical Softw. Engg. 27. URL: https://doi.org/10.1007/s10664-021-10092-4, doi:10.1007/s10664-021-10092-4.

Hoang, T., Kang, H.J., Lo, D., Lawall, J., 2020. Cc2vec: Distributed representations of code changes, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 518–529.

Hoang, T., Khanh Dam, H., Kamei, Y., Lo, D., Ubayashi, N., 2019. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 34–45. doi:10.1109/MSR.2019.00016.

Huang, J., Ling, C., 2005. Using auc and accuracy in evaluating learning algorithms. IEEE Transactions on Knowledge and Data Engineering 17, 299–310. doi:10.1109/TKDE.2005.50.

Huang, Q., Xia, X., Lo, D., 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empirical Software Engineering 24, 2823–2862.

Islam, M.S., Khreich, W., Hamou-Lhadj, A., 2018. Anomaly detection techniques based on kappa-pruned ensembles. IEEE Transactions on Reliability 67, 212–229.

Jiang, L., Cai, Z., Wang, D., Jiang, S., 2007. Survey of improving k-nearest-neighbor for classification, in: Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007), pp. 679–683. doi:10.1109/FSKD.2007.552.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39, 757–773. doi:10.1109/TSE.2012.70.

Khreich, W., Khosravifar, B., Hamou-Lhadj, A., Talhi, C., 2017. An anomaly detection system based on variable n-gram features and one-class svm. Information and Software Technology 91, 186–197.

Khreich, W., Murtaza, S.S., Hamou-Lhadj, A., Talhi, C., 2018. Combining heterogeneous anomaly detectors for improved software security. Journal of Systems and Software (JSS) 137, 415–429.

Kiehn, M., Pan, X., Camci, F., 2019. Empirical study in using version histories for change risk classification, in: Proceedings of the 16th International Conference on Mining Software Repositories, IEEE Press. p. 58–62. URL: https://doi.org/10.1109/MSR.2019.00018, doi:10.1109/MSR.2019.00018.

Liu, F.T., Ting, K.M., Zhou, Z.H., 2008. Isolation forest, in: 2008 Eighth IEEE International Conference on Data Mining, pp. 413–422. doi:10.1109/ICDM.2008.17.

Lomio, F., Pascarella, L., Palomba, F., Lenarduzzi, V., 2022. Regularity or anomaly? on the use of anomaly detection for fine-grained just-in-time defect prediction, in: 30th IEEE/ACM International Conference on Program Comprehension (ICPC 2022), pp. 1–10.

Macbeth, G., Razumiejczyk, E., Ledesma, R.D., 2010. Cliff´s delta calculator: A non-parametric effect size program for two groups of observations. Universitas Psychologica 10, 545–555. URL: https://app.dimensions.ai/details/publication/pub.1113188555, doi:10.11144/javeriana.upsy10-2.cdcp.

McIntosh, S., Kamei, Y., 2018. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering 44, 412–428. doi:10.1109/TSE.2017.2693980.

Nayrolles, M., Hamou-Lhadj, A., 2018. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects, in: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 153–164.

Neto, E.C., da Costa, D.A., Kulesza, U., 2018. The impact of refactoring changes on the szz algorithm: An empirical study, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 380–390. doi:10.1109/SANER.2018.8330225.

Pascarella, L., Palomba, F., Bacchelli, A., 2019. Fine-grained just-in-time defect prediction. Journal of Systems and Software 150, 22 – 36. doi:https://doi.org/10.1016/j.jss.2018.12.001.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830.

Pornprasit, C., Tantithamthavorn, C.K., 2021. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 369–379. doi:10.1109/MSR52588.2021.00049.

Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better, in: 2013 35th International Conference on Software Engineering (ICSE), pp. 432–441. doi:10.1109/ICSE.2013.6606589.

Romano, J., Kromrey, J., Coraggio, J., Skowronek, J., Devine, L., 2006. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohensd indices the most appropriate choices, in: Annual meeting of the Southern Association for Institutional Research.

Schölkopf, B., Platt, J.C., Shawe-Taylor, J., Smola, A.J., Williamson, R.C., 2001. Estimating the Support of a High-Dimensional Distribution. Neural Computation 13, 1443–1471. URL: https://doi.org/10.1162/089976601750264965, doi:10.1162/089976601750264965.

Shehab, A.M., Hamou-Lhadj, A., Alawneh, L., 2022. Clustercommit: A just-in-time defect prediction approach using clusters of projects, in: 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'22), pp. 1–5.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes?, in: Proceedings of the 2005 International Workshop on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA. p. 1–5. URL: https://doi.org/10.1145/1083142.1083147, doi:10.1145/1083142.1083147.

Song, Q., Guo, Y., Shepperd, M., 2019. A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Transactions on Software Engineering 45, 1253–1269. doi:10.1109/TSE.2018.2836442.

Song, Q., Jia, Z., Shepperd, M., Ying, S., Liu, J., 2011. A general software defect-proneness prediction framework. IEEE Transactions on Software Engineering 37, 356–370. doi:10.1109/TSE.2010.90.

Tan, M., Tan, L., Dara, S., Mayeux, C., 2015. Online defect prediction for imbalanced data, in: Proceedings of the 37th International Conference on Software Engineering Volume2, IEEE Press. p. 99–108.

Tong, H., Liu, B., Wang, S., 2018. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. Information and Software Technology 96, 94–111. URL: https://www.sciencedirect.com/science/article/pii/S0950584917300113, doi:https://doi.org/10.1016/j.infsof.2017.11.008.

Wang, J., Zucker, J.D., 2000. Solving multiple-instance problem: A lazy learning approach. URL: http://cogprints.org/2124/.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 297–308. URL: https://doi.org/10.1145/2884781.2884804, doi:10.1145/2884781.2884804.

Wang, S., Minku, L.L., Yao, X., 2018. A systematic study of online class imbalance learning with concept drift. IEEE Transactions on Neural Networks and Learning Systems 29, 4802–4821. doi:10.1109/TNNLS.2017.2771290.

Wang, S., Yao, X., 2009. Diversity analysis on imbalanced data sets by using ensemble models, in: 2009 IEEE symposium on computational intelligence and data mining, IEEE. pp. 324–331.

Wang, S., Yao, X., 2013. Using class imbalance learning for software defect prediction. IEEE Transactions on Reliability 62, 434–443.

Yan, M., Xia, X., Fan, Y., Hassan, A.E., Lo, D., Li, S., 2022. Just-in-time defect identification and localization: A two-phase framework. IEEE Transactions on Software Engineering 48, 82–101. doi:10.1109/TSE.2020.2978819.

Yang, L., Shami, A., 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing 415, 295–316.

Yang, X., Lo, D., Xia, X., Sun, J., 2017. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology 87, 206 – 220. doi:https://doi.org/10.1016/j.infsof.2017.03.007.

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H., 2016. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 157–168. URL: https://doi.org/10.1145/2950290.2950353, doi:10.1145/2950290.2950353.

Yousef, M., Jung, S., Showe, L.C., Showe, M.K., 2008. Learning from positive examples when the negative class is undetermined-microrna gene identification. Algorithms for molecular biology 3, 1–9.

Zeng, Z., Zhang, Y., Zhang, H., Zhang, L., 2021. Deep just-in-time defect prediction: How far are we?, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. p. 427–438. URL: https://doi.org/10.1145/3460319.3464819, doi:10.1145/3460319.3464819.

Zhao, Y., Damevski, K., Chen, H., 2023. A systematic survey of just-in-time software defect prediction. ACM Comput. Surv. 55. URL: https://doi.org/10.1145/3567550, doi:10.1145/3567550.

Zhao, Y., Nasrullah, Z., Li, Z., 2019. Pyod: A python toolbox for scalable outlier detection. Journal of Machine Learning Research 20, 1–7. URL: http://jmlr.org/papers/v20/19-011.html.

Zheng, A., Casari, A., 2018. Feature engineering for machine learning: principles and techniques for data scientists. " O'Reilly Media, Inc.".

A. Appendix

Table 21
The relationship between JIT-SDP model accuracy using AUC and the data imbalance ratio (IR) with cross-validation. NB stands for No balancing, OS stands for Over-sampling, US stands for Under-sampling, and SMOTE stands for Synthetic Minority Oversampling Technique.

		SVM					RF					k-NN				
Project name	**IR**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**
Drill	1.4	0.804	0.801	0.709	**0.808**	0.582	0.755	0.763	0.500	0.763	0.707	0.787	0.780	0.784	0.774	0.717
Flume	1.7	**0.844**	0.833	0.704	0.829	0.530	0.711	0.690	0.716	0.752	0.500	0.804	0.782	0.803	0.787	0.705
Openjpa	2.0	0.784	0.767	**0.822**	0.779	0.519	0.706	0.763	0.735	0.786	0.500	0.754	0.747	0.763	0.722	0.689
Camel	2.3	0.801	0.809	**0.852**	0.812	0.567	0.789	0.790	0.790	0.806	0.731	0.795	0.789	0.802	0.768	0.740
Zookeeper	2.5	0.828	0.843	0.689	0.846	0.567	0.802	0.755	**0.876**	0.794	0.586	0.790	0.790	0.823	0.765	0.590
Flink	4.4	0.703	0.764	**0.876**	0.765	0.588	0.728	0.765	0.753	0.771	0.674	0.757	0.711	0.740	0.746	0.653
Carbondata	7.7	0.681	**0.838**	0.808	0.833	0.608	0.830	0.829	0.774	0.834	0.806	0.810	0.811	0.810	0.816	0.761
Zeppelin	7.8	0.727	0.780	0.802	**0.825**	0.555	0.783	0.799	0.823	0.816	0.776	0.747	0.671	0.759	0.764	0.671
Ignite	8.7	0.611	0.763	**0.801**	0.635	0.779	0.651	0.705	0.764	0.649	0.709	0.678	0.732	0.708	0.748	0.750
Avro	9.2	0.605	0.819	0.891	0.845	0.858	0.500	0.800	**0.900**	0.760	0.630	0.648	0.790	0.823	0.800	0.797
Tez	10.5	0.593	0.792	0.380	0.720	0.790	0.753	0.786	**0.817**	0.787	0.766	0.669	0.732	0.794	0.773	0.761
Airavata	13.5	0.582	0.727	**0.818**	0.618	0.728	0.656	0.722	0.698	0.736	0.663	0.634	0.623	0.691	0.710	0.668
Hadoop	15.8	0.657	0.810	0.800	0.693	**0.814**	0.641	0.686	0.769	0.755	0.759	0.657	0.710	0.747	0.784	0.709
Hbase	15.8	0.554	0.779	**0.852**	0.601	0.785	0.738	0.786	0.767	0.726	0.738	0.673	0.672	0.776	0.749	0.720
Falcon	16.1	0.583	0.824	0.736	0.712	**0.852**	0.602	0.797	0.759	0.817	0.734	0.722	0.680	0.817	0.783	0.734
Derby	16.5	0.509	0.818	0.829	0.725	**0.843**	0.500	0.500	0.500	0.727	0.695	0.671	0.774	0.797	0.813	0.702
Accumulo	17.3	0.601	0.722	0.634	0.603	0.729	0.500	0.500	**0.805**	0.577	0.544	0.694	0.627	0.722	0.699	0.695
Parquet-mr	18.7	0.625	0.714	**0.790**	0.689	0.727	0.658	0.732	0.721	0.775	0.657	0.648	0.721	0.729	0.688	0.710
Phoenix	19.6	0.517	0.763	0.760	0.595	0.736	0.713	0.818	0.771	**0.834**	0.789	0.695	0.687	0.762	0.757	0.756
Oozie	19.7	0.494	0.852	0.636	**0.870**	0.764	0.795	0.843	0.823	0.500	0.862	0.735	0.773	0.864	0.845	0.784
Cayenne	22.3	0.525	0.814	0.789	0.714	**0.826**	0.734	0.792	0.500	0.745	0.812	0.679	0.685	0.774	0.771	0.781
Hive	22.7	0.597	0.799	0.743	0.799	**0.806**	0.500	0.500	0.500	0.500	0.661	0.674	0.677	0.738	0.668	0.768
Jackrabbit	22.9	0.616	0.816	0.774	0.659	0.823	0.743	0.799	0.818	0.500	**0.892**	0.769	0.709	0.813	0.795	0.835
Oodt	23.6	0.511	0.773	0.785	0.773	0.803	0.659	0.727	0.766	0.679	0.793	0.701	0.788	0.766	0.762	**0.805**
Gora	25.3	0.444	0.600	0.680	0.600	**0.889**	0.609	0.779	0.785	0.708	0.811	0.631	0.635	0.754	0.638	0.761
Bookkeeper	27.3	0.620	0.826	0.782	0.826	**0.883**	0.667	0.758	0.767	0.763	0.769	0.680	0.788	0.828	0.792	0.838
Storm	42.6	0.491	0.762	0.728	0.762	**0.810**	0.667	0.705	0.746	0.500	0.794	0.666	0.619	0.747	0.751	0.790
Spark	52.1	0.559	0.837	0.701	0.837	0.887	0.766	0.827	0.827	0.797	**0.890**	0.738	0.786	0.862	0.753	0.866
Reef	63.6	0.771	0.886	0.901	0.886	0.910	0.836	0.878	0.872	0.889	**0.914**	0.792	0.812	0.819	0.820	0.838
Helix	65.6	0.576	0.749	0.718	0.749	**0.881**	0.708	0.769	0.820	0.768	0.880	0.678	0.666	0.676	0.773	0.773
Bigtop	82.8	0.668	0.741	0.765	0.741	0.773	0.500	0.799	0.787	0.631	**0.905**	0.587	0.588	0.720	0.823	0.835
Curator	96.1	0.561	0.603	0.803	0.603	0.816	0.588	0.774	0.775	0.626	0.780	0.486	0.494	0.713	0.657	**0.844**
Cocoon	198.4	0.438	0.866	0.797	0.866	0.915	0.755	0.889	0.780	0.829	**0.916**	0.689	0.687	0.747	0.821	0.846
Ambari	222.5	0.568	0.800	0.839	0.800	**0.852**	0.500	0.661	0.773	0.657	0.806	0.536	0.537	0.685	0.629	0.775
Average		***0.619***	***0.785***	***0.765***	***0.748***	***0.759***	***0.678***	***0.750***	***0.752***	***0.722***	***0.748***	***0.696***	***0.708***	***0.769***	***0.757***	***0.755***

Table 22
The relationship between JIT-SDP model accuracy using F1-score and the data imbalance ratio (IR) with cross-validation. NB stands for No balancing, OS stands for Over-sampling, US stands for Under-sampling, and SMOTE stands for Synthetic Minority Oversampling Technique.

		SVM					RF					k-NN				
Project name	**IR**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**
Drill	1.4	0.756	0.803	0.722	**0.810**	0.597	0.755	0.779	0.500	0.779	0.743	0.788	0.782	0.797	0.788	0.745
Flume	1.7	**0.873**	0.834	0.706	0.832	0.549	0.711	0.738	0.748	0.773	0.500	0.809	0.789	0.815	0.799	0.724
Openjpa	2.0	0.738	0.770	**0.827**	0.785	0.531	0.706	0.782	0.752	0.788	0.500	0.760	0.758	0.776	0.741	0.712
Camel	2.3	0.754	0.811	**0.850**	0.816	0.595	0.789	0.804	0.800	0.809	0.758	0.799	0.794	0.813	0.785	0.757
Zookeeper	2.5	0.765	0.844	0.693	0.850	0.576	0.802	0.780	**0.893**	0.802	0.667	0.799	0.798	0.831	0.782	0.627
Flink	4.4	0.677	0.771	**0.883**	0.771	0.601	0.728	0.779	0.771	0.782	0.720	0.761	0.720	0.754	0.768	0.682
Carbondata	7.7	0.660	**0.840**	0.802	0.834	0.622	0.830	0.835	0.782	0.836	0.819	0.815	0.816	0.820	0.823	0.780
Zeppelin	7.8	0.659	0.781	0.801	0.828	0.564	0.783	0.804	**0.834**	0.821	0.795	0.757	0.676	0.768	0.780	0.721
Ignite	8.7	0.560	0.767	**0.807**	0.638	0.794	0.651	0.741	0.782	0.704	0.731	0.685	0.747	0.728	0.773	0.766
Avro	9.2	0.542	0.820	0.887	0.855	0.860	0.500	0.813	**0.905**	0.776	0.694	0.656	0.803	0.833	0.813	0.813
Tez	10.5	0.546	0.794	0.490	0.730	0.796	0.753	0.795	**0.827**	0.788	0.787	0.683	0.736	0.803	0.790	0.780
Airavata	13.5	0.535	0.742	**0.802**	0.624	0.738	0.667	0.741	0.725	0.756	0.716	0.654	0.655	0.710	0.736	0.708
Hadoop	15.8	0.615	0.812	0.800	0.696	**0.821**	0.641	0.738	0.791	0.768	0.783	0.665	0.718	0.758	0.794	0.727
Hbase	15.8	0.517	0.783	**0.855**	0.603	0.792	0.738	0.795	0.785	0.739	0.758	0.682	0.682	0.784	0.762	0.740
Falcon	16.1	0.538	0.826	0.747	0.721	**0.856**	0.602	0.804	0.775	0.821	0.761	0.734	0.696	0.825	0.796	0.759
Derby	16.5	0.474	0.819	0.824	0.731	**0.854**	0.500	0.500	0.500	0.739	0.728	0.686	0.786	0.810	0.825	0.730
Accumulo	17.3	0.541	0.739	0.640	0.602	0.739	0.500	0.500	**0.831**	0.657	0.627	0.692	0.664	0.739	0.726	0.726
Parquet-mr	18.6	0.595	0.720	**0.794**	0.702	0.741	0.658	0.758	0.739	0.778	0.718	0.663	0.729	0.742	0.712	0.728
Phoenix	19.5	0.484	0.767	0.760	0.623	0.750	0.713	0.826	0.787	**0.838**	0.821	0.708	0.704	0.776	0.775	0.768
Oozie	19.7	0.480	0.855	0.643	0.878	0.778	0.795	0.850	0.837	0.500	**0.891**	0.743	0.778	0.875	0.855	0.802
Cayenne	22.3	0.471	0.815	0.782	0.726	**0.835**	0.734	0.804	0.500	0.758	0.818	0.690	0.700	0.787	0.793	0.796
Hive	22.7	0.553	0.802	0.753	0.803	**0.818**	0.500	0.500	0.500	0.500	0.697	0.681	0.689	0.760	0.688	0.782
Jackrabbit	22.9	0.562	0.819	0.778	0.667	0.836	0.743	0.804	0.830	0.500	**0.900**	0.775	0.716	0.822	0.806	0.854
Oodt	23.6	0.494	0.777	0.791	0.776	0.816	0.659	0.751	0.783	0.700	0.805	0.712	0.797	0.780	0.776	**0.824**
Gora	25.3	0.437	0.605	0.673	0.604	**0.898**	0.609	0.795	0.798	0.729	0.827	0.638	0.643	0.764	0.651	0.779
Bookkeeper	27.3	0.560	0.829	0.778	0.828	**0.887**	0.667	0.771	0.785	0.775	0.830	0.694	0.797	0.839	0.811	0.856
Storm	42.6	0.466	0.769	0.735	0.769	0.819	0.667	0.741	0.763	0.500	**0.823**	0.682	0.645	0.758	0.769	0.801
Spark	52.1	0.517	0.839	0.710	0.839	0.894	0.766	0.845	0.844	0.808	**0.918**	0.743	0.794	0.871	0.768	0.878
Reef	63.6	0.720	0.889	0.899	0.891	0.917	0.836	0.884	0.894	0.894	**0.937**	0.796	0.818	0.828	0.831	0.856
Helix	65.6	0.536	0.753	0.735	0.756	0.883	0.708	0.782	0.825	0.775	**0.908**	0.685	0.678	0.704	0.786	0.787
Bigtop	82.8	0.573	0.746	0.768	0.748	0.774	0.500	0.810	0.801	0.660	**0.933**	0.605	0.606	0.731	0.834	0.852
Curator	96.1	0.523	0.611	0.808	0.602	0.835	0.588	0.789	0.795	0.654	0.820	0.502	0.501	0.726	0.684	**0.862**
Cocoon	198.4	0.385	0.867	0.803	0.869	0.922	0.755	0.896	0.800	0.837	**0.942**	0.699	0.700	0.761	0.836	0.863
Ambari	222.5	0.531	0.805	0.815	0.805	**0.856**	0.500	0.720	0.791	0.697	0.795	0.561	0.568	0.701	0.661	0.799
Average		***0.578***	***0.789***	***0.769***	***0.753***	***0.769***	***0.678***	***0.766***	***0.767***	***0.737***	***0.779***	***0.706***	***0.720***	***0.782***	***0.774***	***0.776***

Table 23
The relationship between JIT-SDP model accuracy using AUC and the data imbalance ratio (IR) with time-sensitive validation. NB stands for No balancing, OS stands for Over-sampling, US stands for Under-sampling, and SMOTE stands for Synthetic Minority Oversampling Technique.

		SVM					RF					k-NN				
Project name	**IR**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**
Drill	1.4	0.651	0.622	0.599	0.654	0.604	0.789	0.788	**0.789**	0.654	0.683	0.706	0.710	0.704	0.711	0.616
Flume	1.7	0.557	0.572	0.699	0.566	0.558	0.724	0.763	**0.767**	0.566	0.730	0.699	0.700	0.703	0.716	0.609
Openjpa	2.0	0.648	0.642	0.625	0.586	0.582	0.780	0.777	**0.780**	0.586	0.676	0.671	0.656	0.667	0.660	0.597
Camel	2.3	0.642	0.644	0.631	0.594	0.605	**0.822**	0.816	0.820	0.594	0.703	0.759	0.758	0.759	0.746	0.605
Zookeeper	2.5	0.634	0.666	0.667	0.689	0.516	**0.818**	0.806	0.814	0.689	0.615	0.727	0.718	0.730	0.728	0.568
Flink	4.4	0.625	0.648	0.652	0.673	0.675	**0.759**	0.755	0.730	0.673	0.690	0.656	0.652	0.646	0.670	0.687
Carbondata	7.7	0.679	0.639	0.587	0.706	0.638	0.781	0.769	**0.784**	0.706	0.770	0.720	0.706	0.715	0.764	0.650
Zeppelin	7.8	0.622	0.599	0.663	0.575	0.655	**0.785**	0.766	0.770	0.575	0.759	0.724	0.723	0.728	0.734	0.688
Ignite	8.7	0.588	0.574	0.578	0.534	0.529	0.537	0.555	0.560	0.534	**0.636**	0.532	0.534	0.532	0.553	0.615
Avro	9.2	0.563	0.561	0.578	0.568	0.657	0.794	**0.806**	0.800	0.568	0.594	0.645	0.629	0.652	0.610	0.517
Tez	10.5	0.674	0.708	0.635	0.705	0.556	**0.827**	0.813	0.813	0.705	0.684	0.707	0.688	0.704	0.779	0.586
Airavata	13.5	0.560	0.506	0.546	0.569	0.576	0.639	**0.672**	0.666	0.569	0.576	0.574	0.558	0.568	0.610	0.540
Hadoop	15.8	0.722	0.624	0.692	0.644	0.625	**0.799**	0.694	0.648	0.644	0.641	0.521	0.530	0.532	0.588	0.673
Hbase	15.8	0.677	0.684	0.656	0.636	0.582	0.740	**0.751**	0.747	0.636	0.749	0.608	0.596	0.620	0.738	0.700
Falcon	16.1	0.610	0.614	0.661	0.671	0.545	0.654	**0.741**	0.731	0.671	0.663	0.686	0.674	0.689	0.714	0.578
Derby	16.5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Accumulo	17.3	0.543	0.547	0.549	0.534	0.510	0.519	0.531	0.535	0.534	**0.649**	0.556	0.552	0.542	0.530	0.558
Parquet-mr	18.6	0.549	0.544	0.519	0.575	0.687	0.631	0.645	0.602	0.575	**0.719**	0.552	0.549	0.551	0.575	0.609
Phoenix	19.5	0.575	0.539	0.582	0.658	0.640	0.777	0.648	0.635	0.658	**0.825**	0.727	0.703	0.719	0.777	0.742
Oozie	19.7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cayenne	22.3	0.680	0.669	0.692	0.730	0.773	0.774	0.851	0.842	0.770	0.872	0.765	0.960	0.959	0.946	**0.965**
Hive	22.7	0.644	0.650	0.727	0.734	0.806	0.696	0.955	0.914	0.734	**0.970**	0.529	0.842	0.858	0.853	0.867
Jackrabbit	22.9	0.604	0.633	0.676	0.692	0.646	0.787	0.818	0.801	0.692	0.714	0.826	0.823	**0.838**	0.817	0.719
Oodt	23.6	0.686	0.646	0.593	0.619	0.730	0.708	0.841	0.865	0.619	0.880	0.714	0.942	0.950	0.939	**0.951**
Gora	25.3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Bookkeeper	27.3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Storm	42.6	0.614	0.680	0.668	0.685	0.720	0.738	0.778	0.720	0.685	**0.790**	0.571	0.576	0.581	0.691	0.724
Spark	52.1	0.688	0.550	0.539	0.695	0.701	0.726	0.769	0.699	0.695	**0.774**	0.633	0.632	0.658	0.696	0.725
Reef	63.6	0.684	0.691	0.648	0.671	**0.833**	0.730	0.751	0.753	0.761	0.783	0.609	0.611	0.622	0.705	0.742
Helix	65.6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Bigtop	82.8	0.684	0.772	0.783	0.745	0.542	0.610	0.650	**0.834**	0.745	0.633	0.625	0.617	0.671	0.707	0.602
Curator	96.1	0.627	0.567	0.657	0.519	0.708	0.670	0.772	0.746	0.519	**0.846**	0.521	0.518	0.544	0.596	0.705
Cocoon	198.4	0.620	0.609	0.725	0.621	0.754	0.514	0.501	0.514	0.621	**0.808**	0.501	0.501	0.501	0.618	0.765
Ambari	222.5	0.573	0.747	0.714	0.676	0.774	0.791	0.918	0.931	0.770	**0.938**	0.744	0.759	0.784	0.731	0.866
Average		***0.628***	***0.626***	***0.639***	***0.639***	***0.646***	***0.721***	***0.748***	***0.745***	***0.646***	***0.737***	***0.649***	***0.670***	***0.680***	***0.707***	***0.682***

Table 24
The relationship between JIT-SDP model accuracy using F1-score and the data imbalance ratio (IR) with time-sensitive validation. NB stands for No balancing, OS stands for Over-sampling, US stands for Under-sampling, and SMOTE stands for Synthetic Minority Oversampling Technique.

		SVM					RF					k-NN				
Project name	**IR**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**
Drill	1.4	0.648	0.637	0.629	0.602	0.624	0.720	**0.727**	0.724	0.602	0.654	0.680	0.685	0.680	0.681	0.630
Flume	1.7	0.545	0.400	0.671	0.362	0.622	0.642	0.720	**0.753**	0.362	0.701	0.655	0.662	0.656	0.683	0.600
Openjpa	2.0	0.624	0.618	0.601	0.586	0.577	0.715	0.711	**0.734**	0.586	0.636	0.630	0.620	0.628	0.627	0.590
Camel	2.3	0.629	0.622	0.619	0.553	0.649	**0.752**	0.747	0.747	0.553	0.659	0.687	0.690	0.695	0.689	0.619
Zookeeper	2.5	0.657	0.600	0.636	0.644	0.670	**0.770**	0.749	0.755	0.644	0.589	0.674	0.680	0.702	0.675	0.556
Flink	4.4	0.585	0.608	0.604	0.640	0.665	**0.708**	0.685	0.687	0.640	0.659	0.607	0.615	0.606	0.638	0.671
Carbondata	7.7	0.645	0.618	0.600	0.684	0.638	**0.729**	0.722	0.706	0.684	0.717	0.686	0.666	0.678	0.719	0.643
Zeppelin	7.8	0.623	0.574	0.663	0.556	0.665	**0.749**	0.725	0.719	0.556	0.715	0.671	0.687	0.681	0.698	0.673
Ignite	8.7	0.459	0.433	0.435	0.491	**0.664**	0.580	0.552	0.570	0.491	0.593	0.524	0.562	0.552	0.571	0.616
Avro	9.2	0.542	0.561	0.654	0.607	0.464	**0.831**	0.824	0.829	0.607	0.547	0.685	0.663	0.673	0.669	0.481
Tez	10.5	0.729	0.711	0.631	0.675	0.580	**0.822**	0.771	0.770	0.675	0.633	0.692	0.644	0.631	0.762	0.605
Airavata	13.5	0.583	0.434	0.509	0.528	0.437	0.576	0.651	**0.655**	0.528	0.579	0.571	0.559	0.556	0.596	0.558
Hadoop	15.8	0.722	0.473	0.709	0.632	0.631	**0.749**	0.657	0.565	0.632	0.603	0.386	0.432	0.436	0.587	0.669
Hbase	15.8	0.683	0.704	0.644	0.634	0.654	0.698	0.696	0.660	0.634	**0.708**	0.635	0.608	0.630	0.682	0.699
Falcon	16.1	0.589	0.613	0.653	0.658	0.631	0.598	0.671	**0.697**	0.658	0.627	0.656	0.632	0.647	0.677	0.594
Derby	16.5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Accumulo	17.3	0.511	0.529	0.474	0.511	**0.654**	0.476	0.536	0.542	0.511	0.629	0.432	0.467	0.495	0.475	0.648
Parquet-mr	18.6	0.479	0.499	0.529	0.486	0.389	0.416	0.371	0.439	0.486	**0.697**	0.323	0.320	0.363	0.531	0.548
Phoenix	19.5	0.484	0.495	0.533	0.643	0.703	0.742	0.653	0.573	0.643	**0.810**	0.667	0.681	0.712	0.753	0.713
Oozie	19.7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Cayenne	22.3	0.705	0.692	0.693	0.732	0.743	0.745	0.794	0.774	0.732	0.798	0.750	0.906	0.905	0.872	**0.910**
Hive	22.7	0.630	0.703	0.807	0.848	0.511	0.742	0.912	0.882	0.848	**0.941**	0.605	0.836	0.864	0.861	0.855
Jackrabbit	22.9	0.560	0.600	0.645	0.674	0.630	0.743	0.778	0.766	0.674	0.675	0.785	**0.791**	0.784	0.759	0.675
Oodt	23.6	0.744	0.547	0.502	0.630	0.750	0.710	0.776	0.823	0.630	0.865	0.701	0.921	0.917	0.909	**0.935**
Gora	25.3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Bookkeeper	27.3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Storm	42.6	0.627	0.644	0.680	0.659	0.702	0.700	0.717	0.689	0.659	**0.750**	0.373	0.401	0.426	0.689	0.687
Spark	52.1	0.661	0.537	0.486	0.653	0.654	0.686	**0.722**	0.669	0.653	0.693	0.510	0.510	0.586	0.661	0.670
Reef	63.6	0.649	0.659	0.612	0.678	**0.792**	0.706	0.728	0.739	0.732	0.731	0.393	0.394	0.437	0.660	0.730
Helix	65.6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Bigtop	82.8	0.716	0.791	0.781	0.722	0.636	0.551	0.650	**0.846**	0.722	0.704	0.481	0.474	0.595	0.723	0.717
Curator	96.1	0.675	0.594	0.264	0.646	0.717	0.639	0.773	0.763	0.646	**0.843**	0.104	0.157	0.321	0.562	0.701
Cocoon	198.4	0.651	0.452	0.702	0.455	0.719	0.000	0.000	0.000	0.455	0.764	0.000	0.000	0.000	0.618	**0.786**
Ambari	222.5	0.566	0.817	0.703	0.782	0.820	0.823	0.863	0.881	0.817	**0.882**	0.659	0.692	0.746	0.850	0.859
Average		***0.618***	***0.592***	***0.609***	***0.620***	***0.641***	***0.666***	***0.686***	***0.688***	***0.623***	***0.704***	***0.559***	***0.585***	***0.607***	***0.685***	***0.677***