Techniques to Enhance Just-In-Time Software Defect Prediction Models

Mohammed Shehab

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Electrical and Computer Engineering) at

Concordia University

Montréal, Québec, Canada

January 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Mohammed Shehab**

Entitled: **Techniques to Enhance Just-In-Time Software Defect Prediction Models**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair

_____ External Examiner

_____ External to Program

_____ Examiner

_____ Examiner

_____ Supervisor

Approved by _____
Dr. Yousef Shayan, ECE, Department Chair

_____ 2024

_____

Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Techniques to Enhance Just-In-Time Software Defect Prediction Models

Mohammed Shehab, PhD Candidate.

Concordia University, 2024

Software defects can lead to significant consequences, adversely affecting system performance by resulting in critical failures. The objective of Just-In-Time Software Defect Prediction (JIT-SDP) techniques is to identify potential defects at an early stage of development, thereby enhancing the reliability and maintainability of software. This thesis contributes novel advancements to JIT-SDP, specifically addressing project clusters, data imbalance, and classifier combination challenges. Additionally, all contributions are evaluated using diverse software projects and 34 datasets, encompassing a total of 259k commits.

The first contribution introduces ClusterCommit, a JIT-SDP approach tailored for project clusters sharing libraries and functionalities. Unlike traditional methods, ClusterCommit employs a machine learning model trained on commits from various projects within a cluster. The study incorporates six machine learning and three deep learning models. The results reveal noteworthy improvements, with mean Area Under the Curve (AUC) values ranging from 4% to 12%, particularly prominent in complex models such as Random Forest (RF) and Support Vector Machine (SVM) when dealing with large clusters. In contrast, simpler models like Naive Bayes (NB), Logistic Regression (LR), Decision Tree (DT), and k-Nearest Neighbors (k-NN) do not perform as well when applied to clusters of projects.

This observed trend extends to deep learning models, where all models experience a performance from 3% to 30% with the ClusterCommit approach, irrespective of cluster size.

The second contribution addresses the challenge of data imbalance in JIT-SDP models. We introduce a novel One-Class Classification (OCC) approach, wherein JIT-SDP models are exclusively trained on data from the majority class (normal commits). It eliminates the need to employ balancing techniques. OCC algorithms, including One-class SVM, Isolation Forest, and One-class k-NN, demonstrate superior performance compared to binary classifiers, especially in projects characterized by medium to high data imbalance ratios. The OCC algorithms achieve mean AUCs of 83%, 81%, and 86% for IOF, OC-k-NN, and OC-SVM, respectively, using cross-validation. In time-sensitive validation, IOF, OC-k-NN, and OC-SVM achieve 84%, 79%, and 74%, respectively. Additionally, these algorithms require fewer features, reducing computational overhead.

The third contribution introduces JITBoost, a framework utilizing a Boolean Combination of Classifiers (BCC) to construct robust JIT-SDP models. Three BCC algorithms—Brute-force Boolean Combination (BBC), Iterative Boolean Combination (IBC), and Weighted Pruning Iterative Boolean Combination (WPIBC)—are investigated. The approach involves generating a set of classifier predictions based on the available features in the dataset, followed by combining these predictions in the Receiver Operator Characteristics (ROC) curve space using Boolean operators to form the final new classifier. JITBoost achieves superior performance by combining decisions from six traditional machine learning and deep learning algorithms, with mean AUCs of 89%, 87%, and 88% for JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC, respectively.

Acknowledgments

I want to express my deep gratitude to my faith and the guidance of Allah, who provided me with the strength and opportunity to complete this significant phase of my life. I want to begin by thanking my supervisor, Dr. Abdelwahab Hamou-Lhadj. His knowledge and generosity have been instrumental during this pivotal chapter of my academic journey. I have learned a great deal from him, not only in research but also in personal and leadership skills. Throughout this journey, I had the opportunity to engage in research and oversee projects that enhanced my creativity, imagination, and passion, both in academic and industrial contexts. I also extend my sincere appreciation to all members of the academic community. Your precise feedback has significantly improved the quality of my proposals and seminars. I would like to express my gratitude to my colleague, Fatima Ait-Mahammed, for her time and constructive assistance in scientific research and discussions.

Furthermore, I want to express my heartfelt thanks to my parents, brother, and dear sister. Their unwavering trust and support are truly invaluable. I'd also like to offer a special thanks to my uncle, Walid Al Hammadi, for his emotional and financial support. Thanks a lot, Uncle Walid.

To my wife, Batool Alkaddah, I am at a loss for words to express my gratitude for your unwavering support and assistance during the most challenging times and circumstances. You are the strong fortress who protecting all of these achievements and dreams.

Lastly, I want to thank Audrey Veilleux for her time, guidance, and support. Her professional explanations and responses have consistently made our journey smoother.

Contents

Acronyms

AST Abstract Syntax Tree. 10, 17

AUC-ROC Area Under the ROC Curve. 5–7, 18, 24, 27, 51

BBC Brute-force Boolean Combination. 114, 118–120

BCC Boolean Combination of Classifiers. 6, 114

BR Bug Report. 23, 64

CNN Convolutional Neural Networks. 17, 49

DBN Deep Belief Network. 18, 50

DeepJIT Deep Just-In-Time. 50

DL Deep Learning. 1, 3–6, 11, 43, 44, 47, 50, 58, 59, 61

DT Decision Tree. 48

EALR Effort-Aware Linear Regression. 12, 13, 50

FN False Negative. 24

FP False Positive. 24

FPR False Positive Rate. 27

HAN Hierarchical Attention Network. 51

IBC Iterative Boolean Combination. 114, 118–120

JIT-SDP Just-In-Time Software Defect Prediction. 1–7, 9, 11–13, 17, 30, 31, 42–44, 47, 48, 59, 67, 139

k-NN k-nearest neighbors. 48

LP Label Propagation. 34

LR Logistic Regression. 12, 13, 17, 18, 48

LSTM Long Short-Term Memory. 17

MCC Matthews Correlation Coefficient. 24, 38

ML Machine Learning. 1, 3, 5–7, 11, 43, 44, 47, 58, 59, 61

NB Naive Bayes. 12, 18, 48

OCC One-Class Classification. 5, 15

RF Random Forest. 12, 17, 35, 48, 58

ROC Receiver Operating Characteristic Curve. 6, 24, 27, 116

SDP Software Defects Prediction. 7, 9–11

SGD Stochastic Gradient Descent. 49

SVM Support Vector Machine. 48

TN True Negative. 25

TP True Positive. 24

TPR True Positive Rate. 27

WPIBC Weighted Pruning Iterative Boolean Combination. 114, 120

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Software maintenance involves various activities such as bug fixing, adapting to changing environments, and incorporating new features to meet customer requirements [1]. It has been observed that the cost of software maintenance can account for up to 70% of the entire software development life cycle, highlighting the need for techniques and tools to reduce this burden and improve code quality [2].

In recent years, the use of Machine Learning (ML) and Deep Learning (DL) in supporting software maintenance activities has gained significant traction. One notable area of research focuses on predicting bugs during the commit phase before changes are integrated into the central code repository. By analyzing commits, which mark the completion of specific tasks, it becomes possible to identify and address unwanted modifications that could introduce bugs [3] [4] [5]. This approach, known as Just-In-Time Software Defect Prediction (JIT-SDP), offers the advantage of providing immediate feedback to developers, enabling them to rectify potential issues while the changes are fresh in their minds [5] [4] [6]. In contrast, traditional bug prediction techniques operate on the entire source code, delaying the provision of feedback [7] [8] [9]. Additionally, JIT-SDP techniques seamlessly integrate with developers' workflow, as they can be implemented as part of a code versioning system.

The advantages of JIT-SDP can be summarized in what follows:

(1) **Preliminary Detection**: the JIT-SDP approach enables the identification of potential software bugs at the premature stage of the software development cycle, which is during the commit phase. It allows developers to address issues before they become deeply embedded in the repository [4, 5, 10].

(2) **Immediate Feedback**: JIT-SDP provides developers immediate feedback by predicting buggy changes at commit time. This rapid feedback loop is highly beneficial, as it allows developers to fix their code while it is still fresh in their minds. It fosters an agile development environment by reducing the time and effort required to fix issues [4, 5, 11].

(3) **Integration with Developer Workflow**: JIT-SDP can seamlessly integrate into code versioning systems. It means developers can access bug predictions as a natural part of their workflow, eliminating the need for external tools and making the bug prediction process more developer-friendly [4, 10].

(4) **Reduced Quality Assurance Costs**: By identifying and addressing bugs early in the development process, JIT-SDP bug prediction contributes to reducing the costs associated with software quality assurance [5, 12]. It helps prevent bugs from propagating throughout the software, which can be expensive and time-consuming to fix later in the development cycle.

(5) **Enhanced Software Quality and Reliability**: Ultimately, JIT-SDP aims to enhance the overall quality of software systems and reliability by minimizing the introduction of defects. It improves customer satisfaction and reduces the likelihood of costly post-release bug fixes [1, 2, 6].

JIT-SDP techniques rely heavily on machine learning and deep learning approaches to classify code commits as either ”buggy” (commits that may potentially introduce defects) or

”normal” (non-buggy commits) [13, 14]. These techniques train machine learning models using historical commit data and various features. The choice of features and algorithms may vary among different JIT-SDP methods [15].

In summary, JIT-SDP is valuable for maintaining software quality and preventing defects. It aligns with the principles of continuous integration and continuous delivery (CI/CD) by providing rapid feedback to developers, making it an integral part of modern software development practices.

1.1 Problem Statement

The performance of JIT-SDP models remains a challenging endeavor as noted in previous research studies [3, 16, 17]. This challenge is exacerbated by the data imbalance problem between the number of normal commits to the number of buggy commits [3, 18]. Moreover, model performance can exhibit instability over time due to a multitude of factors such as:

- **Data heterogeneity**: Increased dataset size improves the accuracy of ML and DL models. It improves the generalization by providing more patterns in the data and reducing overfitting issue [19–21]. However, using dataset from other projects can lead to data heterogeneity issue [22]. This issue occurs due to different functional and non-functional requirements between software applications [22].

- **Imbalance data**: Another challenge in JIT-SDP is ensuring that the machine learning models are effective, especially in scenarios where there is an imbalance between the number of buggy and normal commits [15, 23]. Imbalanced data can lead to bias in the model, where it predicts commits as normal, while it is buggy change [15].

Therefore, many JIT-SDP studies provide data balancing techniques such as oversampling or undersampling to address this issue [15]. However, the balancing techniques produce high false positive rate results, whereas the JIT-SDP model predicts the normal as buggy. So, it can be useless for developers in real-life scenario [4, 20].

- **Performance degradation**: the JIT-SDP models performance is degraded overtime [3, 17]. This issue is raises due to feature important fluctuation [17] and the data distributions over timeline [3, 15]. To this end, recent studies (e.g., [10, 13, 14]) use DL approach to reduce the effect of JIT-SDP performance degradation overtime. However, using DL approaches are costly and require optimization of neural networks architecture, which can be challengeable [13, 21].

1.2 Research Contributions and Implemented Tools

In this research, we present a series of innovative contributions that push the boundaries of JIT-SDP and offer exciting possibilities for improving software maintenance and code quality.

Firstly, we introduce a new category of JIT-SDP approaches that operate on clusters of interrelated projects. By harnessing the power of shared functionalities and common dependencies within project ecosystems, our approach leverages the collective knowledge of multiple projects to enhance bug prediction accuracy and reduce the heterogeneity issue. This novel methodology breaks away from the limitations of traditional single-project methods and opens up exciting avenues for more effective bug detection.

We have developed ClusterCommit, an automated approach that predicts buggy commits by leveraging project dependencies. ClusterCommit uses a community graph clustering algorithm and Maven[1] dependency manager to group similar projects into clusters.For

[1] https://maven.apache.org/

each cluster of projects, ClusterCommit builds a training model that combines commits of all the projects of the cluster. The model is then used to predict buggy commits for each project individually. When applying ClusterCommits to 16 projects that revolve around the Hadoop ecosystem and 10 projects of the Hive ecosystem, the results show that ClusterCommit achieves an F1-score of 73% and MCC of 0.44 for both clusters. These preliminary results are very promising and may lead to new JIT-SDP techniques geared towards projects that are part of a large cluster.

Following this, we expand our research to explore a wider range of ML and DL models using the ClusterCommit methodology. We investigate six ML models: Naive Bayes (NB), Decision Tree (DT), Logistic Regression (LR), Support Vector Machine (SVM), Random Forest (RF), and k-Nearest Neighbors (k-NN). Additionally, we experiment with ClusterCommit using three deep learning models: DeepJIT [14], DBN-JIT [24], and CC2Vec [25]. Furthermore, we increased the dataset by including more projects with additional clusters, totaling 34 grouped into five clusters. The clustering approach is enhanced by eliminating Java utilities. By integrating ClusterCommit's capabilities with advanced deep learning methods, our study showcases its potential to improve the accuracy of bug prediction, specifically complex ML models and all DL models, in interconnected projects. ClusterCommit records improvements in Area Under the ROC Curve (AUC-ROC) ranging from 3% to 12% complex ML models (e.g., RF and SVM). Also, the ClusterCommit approach improves all DL models AUC-ROCs with a range between 3% to 30%.

In Contribution 2, we delve into the issue of imbalanced data in JIT-SDP methods [3, 15]. With a comprehensive study of this problem, we present novel techniques that handle imbalanced datasets while maintaining accurate bug prediction. Leveraging the power of One-Class Classification (OCC) methods commonly used in anomaly detection [26, 27]. We propose a method that trains models using normal commits only and detects buggy commits based on their deviation from normalcy [28]. We compare the accuracy

of three OCC algorithms, One-class SVM, Isolation Forest, and One-class k-NN, to their binary counterparts - SVM, RF, and k-NN. The OCC algorithms perform well with medium and high Imbalance Ratio (IR), scoring around mean AUC-ROCs 83%, 81%, and 86% in cross-validation for IOF, OC-k-NN, and OC-SVM. In time-sensitive testing, they get 84%, 79%, and 74%. These algorithms also need fewer features, making them faster and using less computer power.

However, we observed a degradation in the performance of JIT-SDP models over time [3, 17]. The time-sensitive validation approach results in a changing distribution of buggy commits, and in some cases, this data is unavailable [16, 17]. While OCC models perform better in this situation by training exclusively with the majority class (normal commits), they still require buggy commits in the validation step. Consequently, we draw inspiration from the anomaly detection approach, introducing Boolean Combination of Classifiers (BCC) [29–31]. The BCC algorithms are employed to enhance the performance of ML models by combining decisions from multiple classifiers into a single classifier. These algorithms apply Boolean operators on the Receiver Operating Characteristic Curve (ROC) curve space among multiple classifiers. These classifiers can be of various types, including decision trees, Support Vector Machines, and logistic regression [32, 33].

To this end, we propose JITBoost, a framework that combines the strength of multiple ML and DL algorithms through the use of BCC [29–31]. JITBoost offers a powerful prediction mechanism for identifying buggy commits, outperforming individual JIT-SDP algorithms. By harnessing the collective intelligence of various algorithms, JITBoost showcases its potential to revolutionize bug prediction techniques and pave the way for more accurate and reliable software maintenance. The JITBoost models perform better than traditional ML and DL algorithms when used individually. Specifically, JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC achieve mean AUC-ROCs of 0.891, 0.879, and 0.886, respectively, with cross-validation. With a time-aware data-splitting approach,

they achieve mean AUC-ROCs of 0.863, 0.854, and 0.857, respectively.

Overall, integrating ML models within the JITBoost framework can enhance the performance of JIT-SDP models. Furthermore, JITBoost models exhibit superior performance compared to Deep-JIT, with minimal impact on performance degradation over time.

Within this thesis, we have implemented tools for all these contributions, as detailed in Table 1.1, which represent the three contributions of the thesis. These tools, along with the datasets and results, are readily available online for easy access.

Table 1.1: Tools built for each thesis contribution.

Tool Name	Start Date	URL
ClusterCommit	2019	https://github.com/wahabhamoulhadj/OpenCommitBeta
OCC-JIT	2020	https://github.com/wahabhamoulhadj/jit-occ
JITBoost Framework	2022	https://github.com/wahabhamoulhadj/bcml

1.3 Thesis Organization

The thesis organization is as follows: in Chapter 2, we provide the Traditional Software Defects Prediction (SDP) Approach, JIT-SDP Approach, and a Literature Review. In Chapter 3, we present Dataset preparation, including feature extraction, labeling, etc. Chapters 4, 5, 6, and 7 are dedicated to the main contributions of this thesis we mentioned in the previous section. Finally, we conclude the thesis in Chapter 8, following with future directions and closing remarks.

1.4 Related Publications

This section displays the list of publications from the thesis contributions, followed by two additional types of research in our lab.

(1) **Mohammed A. Sheheb**, Abdelwahab Hamou-Lhadj, and Luay Alawneh. "*Cluster-Commit: A Just-in-Time Defect Prediction Approach Using Clusters of Projects*." International Conference on Software Analysis, Evolution and Reengineering (SANER-NIER). IEEE, 2022. pp. 333-337

(2) **Mohammed A. Sheheb**, Abdelwahab Hamou-Lhadj, and Venkata Sai Gunda. "*JIT-Boost: Boosting Just-In-Time Defect Prediction Using Boolean Combination of Classifiers*." International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2023. pp. 95-104

(3) **Mohammed A. Sheheb**, Wael Khreich, Abdelwahab Hamou-Lhadj, and Issam Sedki "*Commit-time defect prediction using one-class classification*." Journal of Systems and Software (2023): pp. 1-20.

(4) **Mohammed A. Sheheb**, Abdelwahab Hamou-Lhadj. "*Extending ClusterCommit Using a Large Set of Machine Learning and Deep Learning Algorithms*." Journal of Systems and Software, 2024 (In review process).

Additional papers published through collaborations with the research group:

(1) Issam Sedki, Abdelwahab Hamou-Lhadj, Otmane Ait-Mohamed, **Mohammed A. Shehab**, *An Effective Approach for Parsing Large Log Files*, 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME): pp. 1-12

(2) Issam Sedki, Abdelwahab Hamou-Lhadj, Otmane Ait-Mohamed, **Mohammed A. Shehab**, *Understanding Log Parsing Errors Using Open Coding and Mining of Log Files*, 2024 (In review process)

Chapter 2

BACKGROUND AND LITERATURE REVIEW

This chapter covers the background needed to understand this thesis. The chapter starts by discussing traditional SDP and JIT-SDP techniques. It continues with a detailed literature review of other JIT-SDP research studies.

2.1 Traditional Software Defect Prediction

SDP models have played a significant role in identifying defect-prone modules or components in software systems [34,35]. These models use historical data and software metrics to make predictions about areas of the codebase that are more likely to contain defects. By analyzing patterns and metrics from past data, traditional SDP models assist in allocating testing efforts, identifying high-risk areas for code review, and prioritizing resources for debugging and maintenance [35, 36].

Traditional SDP models typically come into play further along in the development cycle, often during testing or after the release phase [37,38]. Through the analysis of historical

data, these models identify modules or components that have a higher likelihood of containing defects [9]. By pinpointing areas that require additional attention and resources, SDP models help ensure software quality by focusing efforts on areas with a higher risk of defects [7].

However, traditional SDP techniques primarily focus on identifying defects during later stages, such as testing [34–36], leading to delayed feedback for developers [5, 6]. These SDP techniques have the following limitations:

- **Reviewing cost**: Providing feedback in the later stages of the software life cycle, when the source code has grown larger, results in increased costs for code reviews [5, 12]. Additionally, assigning fixing tasks to the correct developer becomes challenging when multiple developers make changes to the same file [4, 5].

- **Prediction time**: Several studies (e.g., [39] [8] [9]) suggested changing the standard features to the semantic view by using Abstract Syntax Tree (AST). The AST reduces the data dimensions. For instance, the iterative controls become a cycle graph, and the function declarations change to a node. With a large size of code, converting the code to AST takes between hours to a few days [39], which delays the model prediction.

- **Bug localization**: Identifying the bug location in source code becomes challengeable [10, 39]. It becomes difficult to find the buggy line after converting the code to AST [39]. Even if the code is not converted to AST, finding the correct buggy line inside the large source file increases the false positive rates [10, 18].

- **Agile workflow integration**: Traditional SDP models lack the capability to provide immediate feedback to developers, potentially slowing down team productivity [5]. Applying these models to real-life scenarios, such as(CD/CI), may prove impractical.

While traditional SDP models prove valuable in identifying defect-prone areas, the rise of JIT-SDP models presents a more proactive approach to defect prediction by providing immediate predictions at the commit level [5, 18, 40].

2.2 Just-In-Time Software Defect Prediction

JIT-SDP models aim to provide real-time predictions at the level of code commits. [5, 10, 24]. By analyzing code changes as they occur or just before they are merged into the codebase, JIT-SDP models offer developers the opportunity to address potentially risky changes promptly. These models provide alerts about code changes that are more likely to introduce defects, enabling developers to take proactive measures, such as performing additional testing, code reviews, or making necessary adjustments to mitigate the risk of defects [4, 5, 10, 24]. JIT-SDP models operate by analyzing various code-related features [10, 12], contextual information [13, 14], and historical data at the commit level [3]. They integrate with version control systems or code review tools to provide timely feedback to developers [4, 5, 10, 24].

The application of JIT-SDP models has the potential to improve software quality, reduce maintenance costs, and enhance software development processes. By allowing developers to identify and address potential defects early in the development lifecycle, JIT models contribute to a more proactive approach to defect prevention [5, 10, 13, 24].

Nevertheless, JIT-SDP models exhibit limitations, including data heterogeneity, imbalanced data, and performance degradation (refer to section 1.1). These challenges have prompted recent studies (e.g., [7, 13, 14, 24, 25]) to shift the utilization of DL algorithms, known for their complexity and time-consuming construction [10, 41].

Researchers have discovered that various simple approaches often outperform deep learning methods in software engineering tasks [10, 42, 43]. In this thesis, we adopt diverse strategies to enhance JIT-SDP using ML models, with due consideration for DL models.

2.3 Literature Review

2.3.1 Machine learning approaches for JIT-SDP

JIT-SDP models use both supervised and unsupervised machine learning techniques. Yang et al. [44] studied the accuracy of supervised and unsupervised models for within-project and cross-project approaches. They concluded that unsupervised approaches, which also require less time to build the model, provide similar results to the supervised models. On the other hand, Fu and Menzies [45] proposed the OneWay supervised learning model that used the 12 features proposed by Kim et al. [5]. The results showed that the OneWay model outperformed the unsupervised model proposed by Yang et al. [44] in terms of recall and P_{opt} measures for the within-project approach.

The unsupervised technique is categorized into two types (distance-based and connectivity-based). The distance-based technique (e.g., Yang et al. [44]) showed insufficient performance compared to the supervised models. Therefore, Zhang et al. [22] considered using connectivity-based techniques for the cross-project approach. The proposed method is known as spectral clustering. This method is compared with three supervised methods (Random Forest (RF), Naive Bayes (NB), and Logistic Regression (LR)). To evaluate the spectral clustering method, the authors used 26 projects from three datasets (AEEEM, NASA, and PROMISE). The results showed that the spectral clustering method does not outperform supervised models for the within-project approach. On the other side, it performs approximately similar to the supervised models (e.g., RF) in the cross-project approach.

Moreover, Huang et al. [46] pointed that the unsupervised Line of code before edit (LT) model proposed by Yang et al. [44] does not outperform Effort-Aware Linear Regression (EALR) when the harmonic mean of recall and precision (i.e., F1-score) is considered. The authors did the same steps as Yang et al. [44] to build the LT model and exported its

results. They typically applied the technique by Kamei et al. [5] to get the results of the EALR model and compared them with the LT model. Thereafter, Huang et al. proposed an improved supervised JIT model called CBS+ (Classify-Before-Sort)+ based on the LR model. The CBS+ shows similar results as EALR but with about 15% - 26% more defective changes and better results than the LT model.

Catolino et al. [6] investigated single, and ensembling supervised machine learning models to build the JIT-SDP models with mobile apps. The authors analyzed the 14 metrics proposed by [5] to decide the best metrics that improve JIT-SDP model performance. Thus, the information gain technique between the 14 metrics and label measured to select the top features to filter relevant features with risky changes. Their main finding is that Naive Bayes performs better than ensemble models (e.g., RF) and single models after using the information gain as a feature selection technique.

McIntosh et al. [17] examined the performance of LR models for JIT-SDP in the within-project approach. They trained the models using short-term and long-term data. The short-term used one month of data for training, while the long-term used data spanning a range of several months of data. To evaluate the proposed method, the authors used data from the QT and OpenStack projects. They built the model using 14 features from [5] and three additional features extracted from the code review system (Awareness, Comments Review, and Review window). These additional features ensure that the bug report is fixed and closed. This study found that the accuracy increased by 16%–24% and 6%–12% AUC for the long-term and short-term scenarios, respectively.

Kiehn et al. [47] examined the effect of combining the two software metrics (code and process). They extracted 70 features from 50,000 changes, which were later reduced to 57. Then, the selected features were fed to a Neural Network model. Their approach achieved 70% and 72% precision and recall, respectively. Pascarella et al. [48] proposed to include partially defective commits, another prediction class. The authors also tested

seven supervised machine-learning models over 10 projects. The best-trained model was the random forest model with up to an AUC of 82%.

The accuracy across supervised and unsupervised models for investigative JIT-SDP techniques has been examined by Yang et al. [44]. 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. [45] conducted a replication study of that of Yang et al. [44]. 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. [49] 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 [5] 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.

Cabral et al. [23] 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 and Yao [50] 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.

Yan et al. [18] 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. [5], where the training data is 60% of early commits and the next 40% of commits used to test the model. Yan et al. [18] 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 buggy code if the JIT-SDP predicts the code changes as buggy.

Lomio et al. [3] investigated the use of anomaly detection algorithms, more particularly, OC-SVM, IOF, and Local Outlier Factor for fine-grained JIT-SDP [51], 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 second contribution (OCC, see chapter 6). 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 thesis, 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.

Two-Layer Ensemble Learning (TLEL) is an approach proposed by Yang et al. [40] 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. Nayrolles and Hamou-Lhadj [4] 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%.

2.3.2 Deep learning approaches for JIT-SDP

Code changes in software development can introduce bugs and costly mistakes. Researchers have used DL techniques to build predictive models that analyze code changes and identify potential issues. Traditional metrics-based features have limitations in capturing the true meaning of code changes. To address this, Hoang et al. [14] proposed DeepJIT, a Deep Learning Framework for JIT-SDP. DeepJIT combines metrics based on syntactic and semantic features to improve defect prediction accuracy. These features are extracted from Commit Message (CM) and Code Changes (CC). Hoang et al. [14] tested DeepJIT on the QT and OpenStack projects, achieving the best AUC values of 0.788 and 0.814, respectively. They used different data splitting methods (cross-validation, long periods, short periods) but found no significant differences in the results with data splitting approaches.

In addition to DeepJIT, Hoang et al. [25] also proposed a CC2Vec framework for deep JIT-SDP. Like DeepJIT, CC2Vec uses natural language processing techniques to extract syntactic and semantic features from the commit message's and code changes. In addition to these features, two vectors are created from the added and deleted lines in the commit as additional features. These two vectors are then used to enhance the classification process, a step known as Hierarchical Attention Network (HAN). HAN is utilized for training the Neural Tensor Network, a form of deep learning architecture. To achieve the best results for CC2Vec, six parameters must be tuned. Finally, the output of CC2Vec is used as input features for JIT-SDP, such as SVM. The proposed method increases AUC by 4%. Note that CC2Vec is not an end-to-end deep learning framework such as DeepJIT.

Dam et al. [39] proposed to represent the code as an AST and fit it into a Long Short-Term Memory (LSTM) to generate new features that are used to train the models. They used two datasets (Samsung open-source and PROMISE) to test their approach. The dataset is typically split into 90% training and 10% testing using 10 cross-validations. Then, they used the RF and the LR to build the prediction models. Using the F1-score, the RF and the LR models achieved around 90% and 51% respectively. Similarly, Li et al. [9] proposed a defect prediction approach that uses Convolutional Neural Networks (CNN) for feature generation from information extracted from AST. The generated features are combined with 20 traditional features to build the JIT-SDP prediction model. The authors used the PROMISE dataset (consisting of seven Java projects) for training and testing for the within-project approach. The trained model achieves 12% improvements of the F1-score compared to the traditional feature-based method.

On the other hand, Zhou et al. [7] used deep learning to create the prediction model instead of using it for feature generation as in the previous approaches [25, 39]. More specifically, they used the gcForest to build the deep learning model. The gcForest is based

on multiple classifiers, such as decision trees and ensemble learning, that use the layer-by-layer method. The authors validated their approach using four datasets that contain different projects with a different number of features. They used a total of 25 open-source projects to evaluate this approach. Finally, they used the AUC-ROC evaluation metric since it is not affected by data distribution when compared to the precision, recall, accuracy, and F1-score metrics [52]. The results showed that the proposed technique outperforms the LR approach by 10% for the within-project approach.

Wang et al. [8] 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 NB and 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 [10] 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 [5], DeepJIT [14], and CC2Vec [25]). 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 [14] and CC2Vec [25]).

Dinter et al. [53] assessed the performance of ML and DL for JIT-SDP with 12 mobile applications. They used MCC as the evaluation metric to examine MLP, TabNet, and XGBoost. Interestingly, the results indicated that ML models, particularly XGBoost, outperformed DL models, achieving a 32% higher MCC and being 116 times faster than DL algorithms.

Table 2.1: Related work evaluation metrics and JIT-SDP models

Related Work	Year	Model(s)	Evaluation Metric(s)
Wang et al.	2016	NB and LR	F1-score
Yang et al.	2017	NB, SVM, DT, LDA, k-NN	F1-score, Precision, Recall
Tong et al.	2018	RF and NB	F1-score, AUC-ROC, MCC
McIntosh et al.	2018	RF	AUC-ROC
Nayrolles and Hamou-Lhadj	2018	RF	F1-score, Precision, Recall
Dam et al.	2019	RF and LR	F1-score
Zhou et al.	2019	gcForest, DBN, NB, LR, RF, and SVM	AUC-ROC
Huang et al.	2019	EALR	F1-score
Catolino et al.	2019	LR, NB, RF, DT, and SVM	F1-score, AUC-ROC, MCC
Cabral et al.	2019	EALR	G-mean
Hoang et al.	2019	DBN+LR, DBN+NB	AUC-ROC
Hoang et al.	2020	SVM, LR	F1-score, Precision, Recall, AUC-ROC
Hoang et al.	2021	CNN	AUC-ROC
Pornprasit and Tantithamthavorn	2021	RF	AUC-ROC
Lomio et al.	2022	ET, SVM, k-NN, IOF, and LOF	F1-score, AUC-ROC
Dinter et al.	2023	MLP, TabNet, and XGBoost	MCC

Chapter 3

DATA PREPARATION AND EXPERIMENTAL SETUP

In this chapter, we introduce the datasets we used throughout the thesis to evaluate the proposed approaches. We built datasets of commits from 34 open-source projects from the Apache[1] organization. The total number of commits in all these projects is 259,925. Table 3.1 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) grouped by 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[2].

[1] https://www.apache.org/

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

Table 3.1: 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***

3.1 Feature Extraction

We performed feature extraction from each project in Table 3.1 using the GIT version control system. We extracted 14 features proposed by Kamei et al. [5], which are widely used in the JIT-SDP area (e.g., [17] [13] [54] [3]), alongside 2 additional features, Code Change (CC) and Code Message (CM), proposed by Hoang et al. [14]. These features capture semantic information and syntactic structure, both hidden within the source code and were used to construct and evaluate the DeepJIT model. Table 3.2 presents the 16 features we extracted from the projects.

Table 3.2: The features used to build the JIT-SDP models.

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
Commits	CC	Code changes inside commit
	CM	Commit message represented by the developer

3.2 Dataset Labeling

We use the Refactoring Aware SZZ Implementation (RA-SZZ) algorithm, introduced by Neto et al. [55], to classify the data into normal and buggy commits. This algorithm determines the label of each commit by examining Bug Report (BR) from the bug tracking system, specifically Jira in our case. The RA-SZZ algorithm retrieves all resolved BR and establishes connections with the commits by extracting the bug report's unique identifier from the commit message, if available. Subsequently, the algorithm analyzes the commit history to identify the original commits that introduced the bugs, classifying them as buggy. Unlike the traditional SZZ approach [56], RA-SZZ considers code refactoring, which encompasses changes to the code that maintain its external behavior. Including refactoring activities in the bug localization process poses challenges as they can rearrange and modify the code structure, making it difficult to pinpoint the specific lines of code responsible for a bug [54, 55]. To address this challenge, RA-SZZ incorporates refactorings into its analysis of code changes between different versions. This enables it to produce more precise and dependable results than SZZ, particularly in systems that frequently undergo refactoring [54].

In Figure 3.1, we present an illustration demonstrating the impact of different changes on the SZZ version. Notably, SZZ also identifies code refactoring as a risky change. Furthermore, alterations such as flagging comments and adding blank lines may be regarded as risky modifications [57]. To overcome this limitation, Campos Neto et al. [55] leveraged the RefDiff tool proposed by Silva and Valente in [58]. By using this tool, they were able to disregard these specific changes, reducing noise during the data labeling phase.

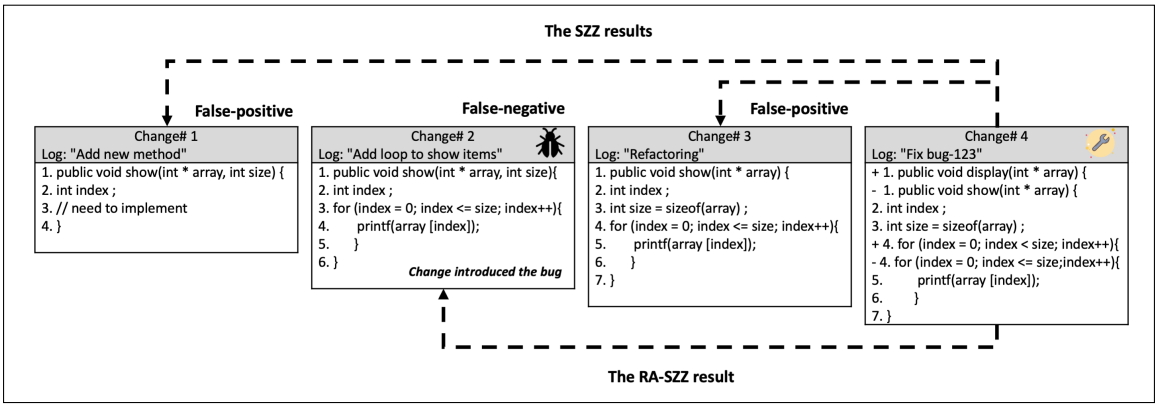


Figure 3.1: An example to compare SZZ vs. RA-SZZ result

3.3 Evaluation Metrics

Several metrics have been used to evaluate the performance of binary classification problems in general and JIT-SDP models in particular [5,6,17,46]. 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).

3.3.1 Threshold-based metrics

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

Precision

Precision is a metric that measures the accuracy of positive predictions made by a model. It is the ratio of true positive predictions to the total number of positive predictions (both true positives and false positives). Equation 1 displays the calculation of Precision.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

Recall

Recall, also known as sensitivity or true positive rate, measures the ability of a model to find all the relevant positive instances. It is the ratio of true positive predictions to the total number of positive instances (true positives and false negatives). Equation 2 shows how to measure the recall from the predictions.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

F1-Score

The F1-score, also known as the F1-measure, is a widely used metric for assessing the accuracy of machine learning models [59]. It is calculated as the harmonic mean of precision (TP/(TP + FP)) and recall (TP/(TP + FN)), as shown in Equation (1) (Ricardo, 1990). However, the F1-score has limitations when evaluating classifiers in the presence of class imbalance and certain data variations [15, 60].

$$F1\text{-score} = 2.\frac{\text{precision . recall}}{\text{precision + recall}} = \frac{2.TP}{2.TP + FP + FN} \tag{3}$$

The F1-score, while widely used, has limitations when dealing with imbalanced datasets, where there is a significant disparity in sample sizes between classes [15]. Relying solely on the F1-score may lead to misleading evaluations of classifier performance as it tends to favor the majority class and overlook the performance of the minority class [61]. Therefore, caution should be exercised when applying the F1-score to imbalanced scenarios to ensure accurate interpretations.

One limitation of the F1-score is its dependence on the selection of a threshold that separates positive and negative predictions based on predicted probabilities or scores [61]. However, determining the optimal threshold is problem-specific and relies on the underlying data distribution [19]. The F1-score does not provide insights into classifier performance at different thresholds, which limits a comprehensive understanding of the model's behavior across various operating points [15, 61]. To gain a more nuanced understanding and make informed decisions, researchers are encouraged to explore additional evaluation metrics that capture performance characteristics at different threshold levels [15, 61].

Recent studies have highlighted the limitations of using metrics such as the F1-score and recommend alternatives like the Matthews correlation coefficient (MCC) [15]. This problem is particularly prominent when dealing with imbalanced data, as biases can arise [15, 62]. Another useful metric is the Area Under the Receiver Operating Characteristic (ROC) curve (AUC), which illustrates the relative trade-offs between the true positive rate (TPR) and false positive rate (FPR) for different classification thresholds [20]. The AUC-ROC is considered in this study as an additional evaluation measure.

Matthews Correlation Coefficient (MCC)

The Matthews correlation coefficient (MCC), known as the phi coefficient in statistics, is another measure of the quality of a classification algorithm. It is similar to the Pearson correlation coefficient for two binary variables. The MCC is calculated using the formula

in Equation (4) and includes the TN along with all other quantities of the confusion matrix. MCC is the only binary classification metric that provides a high score only if the majority of both positive and negative data instances are correctly predicted. The important property of MCC consists of producing a high score only if the prediction obtained good results in all four confusion matrix quantities (TP, FP, TN, FN) proportionally to the size of the positive and negative examples in the dataset. These characteristics make MCC a more reliable metric than the F1-score, especially for imbalanced data [15, 60, 62–64].

$$MCC = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4}$$

Similar to the F1-score, the MCC can also be influenced by imbalanced datasets. If the number of samples in each class is highly imbalanced, the MCC may be biased towards the majority class. This means that the MCC may not accurately reflect the performance of the minority class [65]. To address this limitation, techniques such as resampling, class weighting, or utilizing alternative evaluation metrics specifically designed for imbalanced datasets can be considered.

3.3.2 Threshold-Independent Metrics

Threshold-independent metrics such as the ROC and the AUC-ROC do not commit to a threshold. The ROC is a graphical plot (see Figure 3.2 for an example), which illustrates the performance of a classifier as its discrimination threshold is varied. The ROC plots the false positive rate False Positive Rate (FPR)=FP/(FP+TN) against the true positive rate True Positive Rate (TPR)=TP/(TP+FP) for every decision threshold [61]. 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 3.2. 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 3.2) proportionally to the imbalance ratio [61]. The AUC has been proposed as a robust (global) measure for the evaluation and selection of classifiers [66]. The AUC is the average of the tpr overall values of the fpr (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.

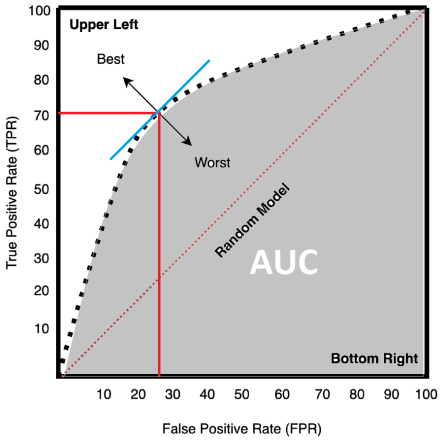


Figure 3.2: An illustration of a ROC curve, the area under the curve (AUC), and the default decision threshold.

3.3.3 Statistical Methods

We also used two statistical methods for data analysis and hypothesis testing, namely, Mann-Whitney U Test [67] and Cliff's δ [68].

1) Mann-Whitney U Test

We employed the non-parametric Mann-Whitney U test [69, 70]. It is used because we cannot assume the distribution follows normal distribution to assess the statistical significance of the model's outcomes. The null hypothesis (h_0) posits that there is no statistical difference in the model's results, whereas the alternative hypothesis (h_1) suggests that the model's results do exhibit statistical dissimilarity. The null hypothesis is deemed invalid when the p-value falls below 0.05, corresponding to a 95% confidence level [67].

Mann-Whitney U test shows if two groups are different or not, but it focuses on determining the significance of the difference and does not provide a specific measure of effect size. Therefore, we also used Cliff's δ to measure that quantifies the magnitude of the difference [20, 71].

2) Cliff's δ Effect size measurement

Additionally, we use Cliff's δ effect size to assess the magnitude of the difference between the results of binary classifiers. 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 [68, 71, 72]. 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 [73]. The effect is considered small for $0.147 \leq |\delta| < 0.330$, moderate for $0.330 \leq |\delta| < 0.474$, or large for $|\delta| \geq 0.474$ [71,73]. Cliff's δ is defined using Equation (5), with x and y representing two data vectors and n_x and n_y, the size of these vectors.

$$Cliff's\ \delta = \frac{\sum_i \sum_j sign(y_i - x_j)}{n_y.n_x} \tag{5}$$

Chapter 4

ClusterCommit: A Just-in-Time Defect Prediction Approach Using Clusters of Projects

4.1 Introduction

Recently, Nayrolles and Hamou-Lhadj [4] conducted a study at Ubisoft, the video game development company, in which they developed CLEVER, a novel JIT-SDP approach. CLEVER is unique in the sense that it relies on training a JIT-SDP model that combines commits from many Ubisoft video game projects that run on the same game engine. The authors argued that, for industrial projects, it is useful to combine commits from highly-coupled projects instead of working on each project separately. This is because these projects reuse libraries and share an important code base, rendering them vulnerable to the same faults.

Inspired by the design of CLEVER, we conducted a study to investigate the use of clusters of projects for JIT-SDP in open-source systems. As a motivating example, we take

the Apache Foundation projects that were introduced in the previous Chapter 3. These projects offer complementary services in diverse fields. Many of them are built by reusing other projects and libraries. For example, the Bigtop, ZooKeeper, and Spark projects are built on Hadoop. Similarly, Ambari, Falcon, and Oozie projects use Hive, another Apache project. By examining the bug reports of many of these projects, we found bug reports of one project that refer to bugs in another project. For example, the description of bug report OOZIE-3563 of the Oozie project refers to the inability to initiate the Hive project. For these interrelated projects, we conjecture that it would be beneficial to treat them as one cluster and build a training model that combines their commits. This led us to the design of ClusterCommit, a JIT-SDP approach based on clusters of projects.

Although ClusterCommit is inspired by CLEVER, the two approaches exhibit important differences in the way they are designed. A key design feature of CLEVER is its reliance on clone detection techniques to predict buggy commits. For each suspected commit, CLEVER extracts the corresponding code block and compares it to a database of known defects. This design choice suggests that CLEVER is too dependent on the way Ubisoft projects are developed, where large code segments may be reused across systems (perhaps because they are written by the same development teams). ClusterCommit, on the other hand, relies solely on code and process metrics, common to any software system. This way, we do not assume anything about the way the projects are developed. This is particularly important for open systems since they are developed by contributors from different organizations. Additionally, ClusterCommit relies on a clustering technique to identify projects that should be grouped together since we cannot rely on domain knowledge to identify groups of interrelated projects, as is the case for industrial systems. Finally, ClusterCommit uses a time-validation approach (see section 4.2.3), which accounts for the temporal order of commits, to evaluate the prediction accuracy. This temporal order is not considered in CLEVER, yielding a situation where past commits may be potentially

compared to future commits.

4.2 The ClusterCommit Approach

Figure 4.1 depicts the overall steps of ClusterCommit. ClusterCommit requires as input a set of projects that share some dependencies. This input can be specified by the user of ClusterCommit. The next step of ClusterCommit is to cluster the projects so as to identify strongly coupled subprojects, which are likely to share a large code base. For each cluster, we build a training model by aggregating past commits (both healthy and buggy) extracted from each project of the cluster. The resulting model is used for testing. At this point, ClusterCommit predicts whether a commit is buggy or not for each project individually.

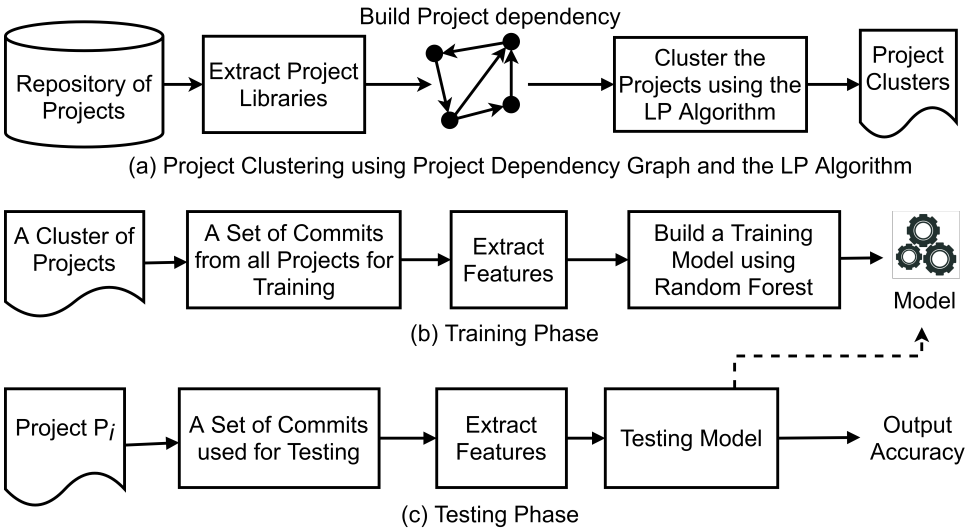


Figure 4.1: Overall approach

4.2.1 Project Clustering

Consider the set $P = \{P_1, P_2, ...P_N\}$ of size N, a set of projects that are given as input by the user. ClusterCommit starts by extracting the libraries that each project uses. This information is found in the dependency management system used by the projects. For

example, Java projects (the focus of this chapter) are managed and built using tools such as the Maven[1] dependency manager to save development time by reusing internal and external (third-party) libraries [74]. We mine Maven information to extract libraries used by each project. Assume $L = \{L_1, L_2, ..., L_M\}$ is a set of distinct libraries used by the projects of P, where M is the number of distinct libraries. ClusterCommit builds a project dependency graph $G = (V, E)$ where V is the set of nodes representing projects of P and libraries of L, and E represents the set of edges between projects and libraries. We define a directed edge from a project P_i to a library L_j if P_i uses the library L_j. This type of graphs is known as a community graph [75]. Instead of having edges between projects, we link projects to their libraries. The idea is to find projects that share a large number of libraries and cluster them together. For this, we use a community-based clustering technique. More particularly, we choose to apply the Label Propagation (LP) algorithm for finding communities [75]. For more details about the LP algorithm, we refer the reader to the work of Raghavan et al. [75].

Consider the set $P = \{P_1, P_2, ...P_N\}$ of size N, a set of projects that are given as input by the user. ClusterCommit starts by extracting the libraries that each project uses. This information is found in the dependency management system used by the projects. For example, Java projects (the focus of this chapter) are managed and built using tools such as the Maven[2] dependency manager, to save development time by reusing internal and external (third-party) libraries [74]. We mine Maven information to extract libraries used by each project. Assume $L = \{L_1, L_2, ..., L_M\}$ is a set of distinct libraries used by the projects of P, where M is the number of distinct libraries. ClusterCommit builds a project dependency graph $G = (V, E)$ where V is the set of nodes representing projects of P and libraries of L, and E represents the set of edges between projects and libraries. We define a directed edge from a project P_i to a library L_j if P_i uses the library L_j. This type of graph is known as a community graph [75]. Instead of having edges between projects, we link projects to their

[1]https://mvnrepository.com/repos/central
[2]https://mvnrepository.com/repos/central

libraries. The idea is to find projects that share a large number of libraries and cluster them together. For this, we use a community-based clustering technique. More particularly, we choose to apply the Label Propagation (LP) algorithm for finding communities [75].

The LP algorithm uses an adjacency matrix of the nodes in the graph. It initializes the matrix with empty cells except for the diagonal cells, where each node matches its label. The algorithm updates the matrix iteratively until it reaches a stable state (i.e., stops updating). Each iteration consists of a forward step and a backward step. After the completion of each iteration, the matrix cells (which represent the edges) will be updated based on the directions of the edges between the nodes in the graph, and the column labels will be updated based on the nodes with the highest frequencies [75].

We use the directed graph shown in Figure 4.2a to illustrate the LP algorithm. The algorithm starts by the forward step and updates the matrix by labeling the neighboring nodes as shown in Figure 4.3a. For example, since node A has an edge to C, the cell at row A and column C will be labeled as C. Moreover, node B has a bidirectional edge with node C. Therefore, the cell at row B and column C will be labeled as C, and the cell at row C and column B will be labeled as B. The same steps will be repeated for all the nodes based on the direction of the edges. Figure 4.3 shows that nodes C and F have the highest frequencies in the matrix when the forward step is completed. Thus, the backward step will update the column labels in the matrix with the nodes C and F as shown in Figure 4.3b. The first two columns were changed to C (since A and B are neighbors to C), the 4^{th} column was changed to F (D is neighbor to F), the 5^{th} column was changed to C (since E has a bidirectional edge with C), and the 7^{th} column was changed to F (G is a direct neighbor of F).

In the second iteration, the forward step will check the rows and update the cells based on the labels of the neighbor nodes. Figure 4.3b shows how the matrix is updated after the completion of the forward step, along with the frequencies for each column. The backward

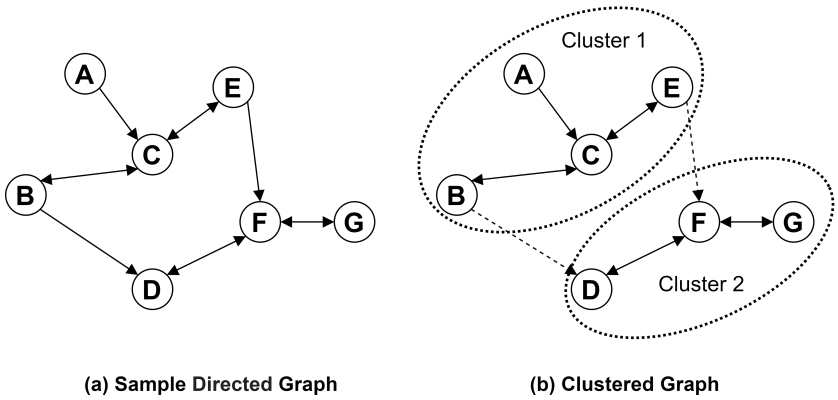


Figure 4.2: Graph Clustering Example

step will check the columns with the highest frequencies and update the column labels (similar to the first iteration). In this example, the column labels will not be updated in the second iteration, which means that the matrix has reached a stable state. Figure 4.3b shows that we have C and F as the super-nodes. Therefore, the algorithm detects two communities (clusters) as shown in Figure 4.2b. It should be noted that Node E has connections with the two super-nodes. However, the count of the labels (C = 2 and F = 1) means that it is more related to cluster C [75] [76].

4.2.2 Classifier

In this chapter, we use RF [77] as the classification algorithm. We chose RF because Pascarella et al. [48] tested seven supervised machine learning models over 10 projects and found that the best performance was obtained with RF. We intend to conduct future studies to compare the impact of various algorithms in Chapter 5.

4.2.3 Evaluating the classifier

One way to validate the performance of a classifier would be to use the traditional 10-fold cross-validation approach. The problem with this approach is that it does not consider

	A	B	C	D	E	F	G
A	**A**		**C**				
B		**B**	**C**	**D**			
C		**B**	**C**		**E**		
D				**D**		**F**	
E			**C**		**E**	**F**	
F				**D**		**F**	**G**
G						**F**	**G**
Count	***1***	***2***	***4***	***3***	***2***	***4***	***2***

(a) Iteration 1

	C	C	C	F	C	F	F
A	**C**		**C**				
B		**C**	**C**	**F**			
C		**C**	**C**		**C**		
D				**F**		**F**	
E			**C**		**C**	**F**	
F				**F**		**F**	**F**
G						**F**	**F**
Count	***1***	***2***	***4***	***3***	***2***	***4***	***2***

(b) Iteration 2

Figure 4.3: Label Propagation Steps

the temporal order of the commits, leading to a situation where commits from the past may be tested against a model that is trained on commits from the future. To address this issue, Tan et al. [16] proposed a time-based validation approach. This approach sorts the commits based on their timestamps and groups them into slots depending on the month of submission. Then, it chooses a time interval for training and another one for testing. In addition, Tan et al. [16] argued that there should be a gap between the commits used for training and those used for testing to account for the time between submitting a buggy commit and the manifestation and reporting of the bug. Therefore, the time-validation approach requires the definition of three-time intervals: train, gap, and test.

Figure 4.4 illustrates how we used time validation with clusters of projects using as example three fictive projects and two runs of validation. To explain how our approach works, we first need to determine the length of the training, gap, and testing time intervals. Based on the literature, we consider 6 months as the length of the training time interval as

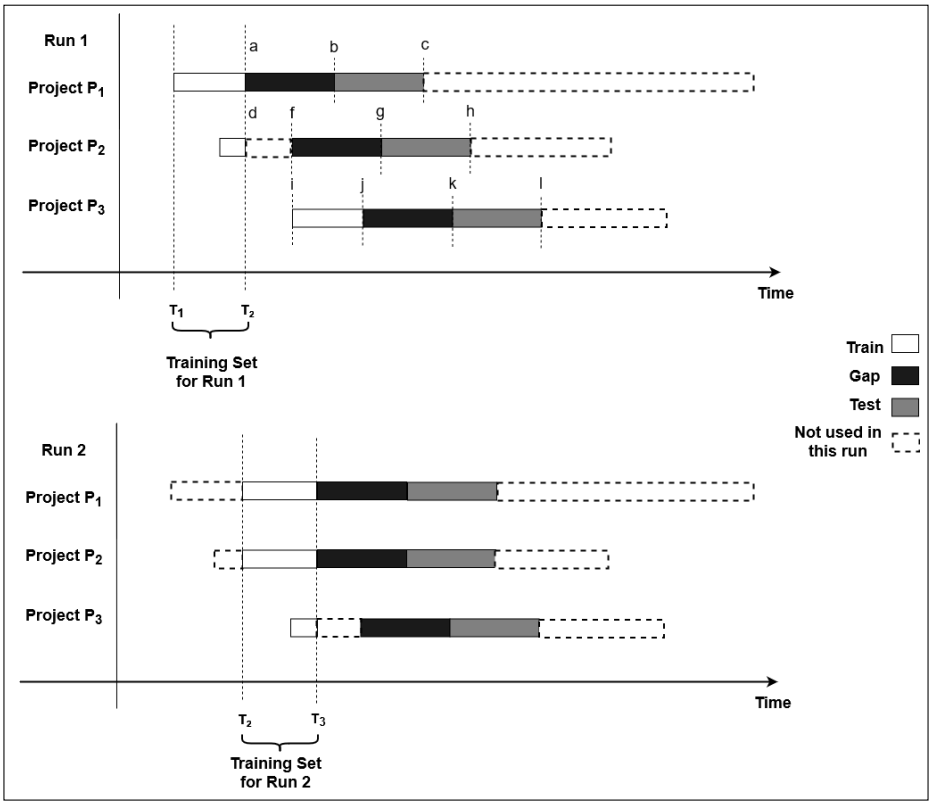


Figure 4.4: An example of time-validation using ClusterCommit with three projects and two runs

suggested by McIntosh et al. [17] since the projects we use in the evaluation have several years of historical commits (see section 4.2.5). For the testing and gap time intervals, we take the minimum of the average fixing times of all projects of the cluster. More formally, consider aft_i as the average time it takes to fix bugs of Project P_i. The length of the gap and testing time intervals is computed as follows: $length = min(aft_1, aft_2, ..., aft_N)$, with N being the total number of projects in a cluster. The rationale behind taking the minimum of the averages instead of, for example, the average of averages, is to ensure that for each project we can perform at least one run of validation. The length of the training, gap, and testing time intervals is fixed for all projects.

Now that we have determined the length of the three time intervals needed for time-validation, we start by iterating through the data to determine the commits used for training

and testing in each iteration. In each run, we build a training model that combines commits that appear in the training time interval. We test each project individually against the model using commits that appear in the testing time interval. For example, for the projects of Figure 4.4, in Run 1, we use the commits of Project P1 and some commits of Project P2 that fall within the training time interval as a training set. To test commits of P1, we use commits that appear in time interval b to c. For P2, we use commits in the time interval g to h, and finally we test P3 with commits in time interval k to l. We are aware that P3 is tested against commits of P1 and P2 and that none of P3 commits are used for training in this iteration. This is perfectly aligned with the core idea of ClusterCommit where strongly interrelated projects can have commits of one project or more used to predict buggy commits in other projects. In Run 2, the process continues by shifting the time window with exactly the length of the training time interval and the process is repeated again. With this approach, we have as many runs as necessary until all projects are covered. For each project, we measure the prediction accuracy (see next section for the evaluation metrics) in each run, and take the average as the final performance of the classifier for each project. Note that the number of runs through the projects is not the same. For example, in the last run of our example, both Projects P2 and P3 in Figure 4.4 will end up contributing commits to a training set that are used to test commits of Project P1 only. This is because the ending time of these projects is earlier than that of project P1.

4.2.4 Evaluation Metrics

We use precision, recall, F1-Score, and MCC [78] to evaluate the effectiveness of ClusterCommit. These metrics are widely used in related studies (e.g., [6] [4]). More explains are represented in Chapter 3, section 3.3.1.

4.2.5 Subject Systems

To evaluate the effectiveness of ClusterCommit, we need a group of related projects. For this, we turn to projects of the Apache Foundation. The Apache Foundation maintains a list of projects categorized depending on their domain such as Big Data, Cloud, Build Management, Logging, etc.]In this chapter, we choose to focus on projects of the Big Data and Database domain. Another category can also be used.

The Big Data and database category contains 72 open source projects (at the time of conducting this study) for managing and processing large data. We further restrict the number of projects to those written in Java and built using the Maven dependency manager. This is because we use Maven to extract shared libraries to build the project dependency graph used for clustering. In addition, we only select projects that have at least 1,000 commits to ensure that we have mature projects with sufficient history. By applying these criteria, we ended up with 34 projects (see Table 3.1. These projects are available on Github and use Jira[2] for bug tracking.

4.2.6 Result of Clustering

In this step, we apply the LP clustering algorithm to the 34 projects. We wrote a script to extract the Maven dependencies and built a project dependency graph that shows the relationships between projects based on the number of libraries they share. We found that these projects have 1,520 distinct libraries. The LP algorithm returned seven distinct clusters, among which two clusters are the most predominant. The first cluster has Hadoop as its super-node and contains 16 projects and 868 (57.10%) shared libraries. The second cluster consists of projects that revolve around Hive as a super-node. It contains 10 projects that share 652 (42.9%) libraries. We refer to these clusters as Hadoop and Hive clusters and use them in this Chapter to illustrate the performance of ClusterCommit.

[2]https://www.atlassian.com/software/jira/features/bug-tracking

Table 4.1 shows the projects that belong to these two clusters. As we can see, the Hadoop cluster contains projects that are built around the Hadoop ecosystem, such as Camel, a Hadoop-based project that integrates various systems that consume and produce data. Camel accepts data stored in the Hadoop Distributed File Management System (HDFS). The same goes for other projects such as Drill, Helix, Spark, and Parquet, which revolve around Hadoop technology. The Hive cluster groups projects that support the management of data warehouses. Table 4.1 shows information about the clusters and their projects, including the project name, version, total number of commits, the ratio of buggy commits (shown as linked commits using RA-SZZ), and the project time period.

4.2.7 ClusterCommit Results and Discussion

We set the length of the training time interval to 6 months as discussed in section 4.2.3. To compute the length of the testing time interval, which is also the length of the gap time interval, we measure the average bug fixing time of each project in a cluster and take the minimum of averages as explained in section 4.2.3. We found that the minimum for both clusters is 8 months.

Table 4.2 shows the results of ClusterCommit. For both clusters. Our approach achieves an F1-Score of around 73% and an MCC of 0.44. Note that MCC varies from -1 to 1 with the latter being the perfect model. We also observe that ClusterCommit has a higher recall than precision. In other words, while it detects more buggy commits (high recall), in both clusters it has around 29% false positives (precision of about 71%). This contradicts the results obtained by Nayrolles and Hamou-Lhadj for their approach CLEVER (79% precision and 65% recall). Further studies with more clusters are needed to generalize the results of this study.

Table 4.1: Subject projects considered in this study

	Name	Version	Commits	Defects (%)	Project Period
Hadoop	Bigtop	1.5.0	2,599	1.2%	Aug/2011 - Dec/2020
	Bookkeeper	4.12.1	2,374	3.5%	Mar/2011 - Dec/2020
	Camel	2.10.7	13,023	30.6%	Mar/2007 - Sep/2013
	Curator	2.0.0	2,718	1.0%	Jul/2011 - Jan/2021
	Drill	1.10.0	2,597	63.3%	Oct/2012 - Feb/2021
	Flink	1.12.1	24,983	18.5%	Dec/2010 - Jan/2021
	Gora	0.1-incub.	1,367	3.8%	Oct/2010 - Nov/2020
	Hadoop	2.6.0	10,508	6.0%	Sep/2009 - Aug/2014
	Helix	1.0.1	3,756	1.5%	Jun/2011 - Jun/2020
	Ignite	1.0.1	10,836	14.8%	Feb/2014 - Sep/2017
	Oodt	1.9	2,084	4.1%	May/2010 - Jan/2021
	Parquet	1.8.0	1,679	6.8%	Aug/2012 - Feb/2021
	Reef	0.16.0	3,749	1.6%	Aug/2012 - Nov/2020
	Spark	2.2.1	19,967	1.8%	Apr/2010 - Nov/2017
	Tez	0.9.2	2,658	8.7%	Mar/2013 - Mar/2019
	Zookeeper	3.6.0	2,030	28.4%	Nov/2007 - Nov/2019
Hive	Accumulo	2.0.1	10,094	5.5%	Oct/2011 - Dec/2021
	Airavata	0.17	7,227	6.9%	Jul/2011 - Mar/2019
	Ambari	2.7.5	24,578	0.4%	Sep/2011 - Jun/2020
	Carbondata	2.1.0	4,746	11.6%	Mar/2016 - Feb/2021
	Falcon	0.11	2,209	5.9%	Nov/2011 - Aug/2018
	Flume	1.9.0	1,812	42.4%	Aug/2011 - May/2020
	Hive	3.1.3	12,277	4.2%	Sep/2008 - Jan/2020
	Oozie	5.2.0	2,332	4.9%	Sep/2011 - Jan/2021
	Storm	2.2.0	10,326	2.3%	Sep/2011 - Feb/2021
	Zeppelin	0.8.2	3,896	13.9%	Jun/2013 - Feb/2021

4.2.8 Threats to Validity

We experimented with 26 projects of the Apache Foundation. We need to conduct more experiments to generalize our results. The size of the time intervals can affect the performance of the prediction models. We need to experiment with different time intervals to see their impact on the result. Furthermore, we relied on the RA-SZZ algorithm and the implementation provided by the authors to label the data. Errors in this implementation may impact our results.

Table 4.2: ClusterCommit Results

	Project name	Precision (%)	Recall (%)	F1 score (%)	MCC (%)
Hadoop	Bigtop	72.63	51.40	60.19	0.34
	Bookkeeper	66.22	85.76	74.74	0.42
	Camel	75.03	76.69	75.85	0.51
	Curator	71.87	90.21	80.00	0.53
	Drill	72.47	79.54	75.84	0.49
	Flink	70.78	73.18	71.96	0.42
	Gora	64.67	66.06	65.36	0.30
	Hadoop	64.88	81.54	72.26	0.38
	Helix	70.50	80.45	75.15	0.45
	Ignite	69.98	59.53	64.33	0.34
	Oodt	60.63	62.90	61.74	0.19
	Parquet	76.37	75.11	75.74	0.51
	Reef	79.76	93.70	86.17	0.70
	Spark	74.00	75.16	74.58	0.49
	Tez	67.95	82.69	74.60	0.44
	Zookeeper	75.64	67.41	71.28	0.46
	Average	***70.84***	***75.08***	***72.89***	***0.44***
Hive	Accumulo	66.64	61.05	63.72	0.31
	Airavata	66.64	72.96	69.65	0.37
	Ambari	71.43	81.49	76.13	0.49
	Carbondata	71.07	81.82	76.07	0.49
	Falcon	65.17	81.90	72.58	0.40
	Flume	78.99	79.57	79.28	0.56
	Hive	69.82	67.12	68.44	0.38
	Oozie	71.56	84.04	77.30	0.53
	Storm	63.97	74.17	68.69	0.33
	Zeppelin	78.58	73.69	76.05	0.54
	Average	***70.39***	***75.78***	***72.98***	***0.44***

4.2.9 Conclusion

In this chapter, we proposed ClusterCommit a new JIT-SDP approach that builds a training model that combines commits from projects of the same cluster and tests commits from each project individually. When applied to two clusters of projects of the Apache Foundation with a total of 26 projects, we showed that ClusterCommit yields promising results. The next chapter shows: (a) experiment with more different size clusters, (b) apply ClusterCommit with more ML and DL algorithms, and (c) investigate factors that can effect the performance of JIT models using ClusterCommit approach.

Chapter 5

Extending ClusterCommit Using a Large Set of Machine Learning and Deep Learning Algorithms

5.1 Introduction

This chapter is an extension of the ClusterCommit approach by investigating six ML models: Naive Bayes (NB), Decision Tree (DT), Logistic Regression (LR), Support Vector Machine (SVM), Random Forest (RF), and k-Nearest Neighbors (k-NN). Additionally, we experiment with ClusterCommit using three DL models: DeepJIT [14], DBN-JIT [24], and CC2Vec [25]. Furthermore, we augment the dataset by including more projects with additional clusters, totaling 34 grouped into five clusters. The clustering approach is enhanced by eliminating Java utilities, with further details explained in Section 5.2.1.

In this chapter, we address the following three new research questions (RQ):

- **RQ1**: Does the utilization of the ClusterCommit method lead to enhanced performance of JIT-SDP models?

- **RQ2**: What is the impact of varying cluster sizes on the performance of JIT-SDP models?

- **RQ3**: What are the main factors that need to be considered using the ClusterCommit approach?

Regarding RQ1, our findings indicate that complex ML classifiers (e.g., SVM and RF) tend to outperform simpler ones (e.g., NB, LR, DT, and k-NN) when employing the ClusterCommit approach. This trend is also observed in DL models, where all models exhibit improvement with the ClusterCommit approach. For RQ2, we examine the impact of cluster data size, revealing that data size directly affects ML models while DL models remain unaffected.

Concerning RQ3, our results suggest that ClusterCommit introduces a heterogeneous data issue by combining datasets from different projects. Consequently, complex ML and DL models share greater improvements using the ClusterCommit approach. Additionally, overlapping bug reports within clusters emerge as a factor influencing the performance of ClusterCommits; higher overlap mitigates the impact of heterogeneous data issues.

5.2 Study Setup

Figure 5.1 shows the steps of our study. The first step is to cluster projects based on their dependencies (see Figure 5.1(a)). This step is similar to the one proposed in Section 4.2.1, with some exceptions regarding the types of dependencies that are used (see Section 5.2.1). The training and testing phases of (shown in Figure 5.1(b) and (c)) use machine learning and deep learning models, that are fundamentally different from ClusterCommit in terms of the features and the classification algorithms that are used.

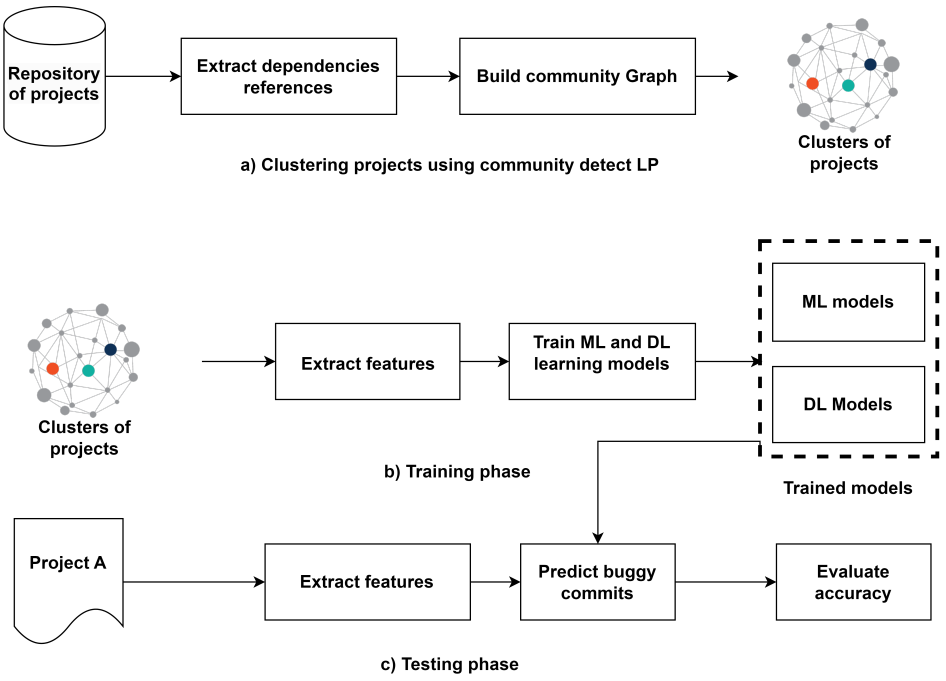


Figure 5.1: Overall approach of a cluster-based deep learning approach.

5.2.1 Community Detection Algorithm

We improved upon the clustering approach initially proposed previous Chapter 4 by excluding utility dependencies, such as references to Java libraries. This modification allows for a more focused analysis of specialized functionalities that exhibit distinctive relationships among projects. Following this refinement, the number of clusters is reduced from seven to five, as illustrated in Figure 5.2, in contrast to the findings of the previous study [79]. The elimination of utility dependencies results in the removal of links between nodes, shifting the emphasis towards more cohesive networks and consequently leading to a reduction in the number of distinct clusters [76, 80]. Through manual verification, we confirmed that the two additional clusters identified are associated with Java utility dependencies. Table 5.1 presents the five clusters and their respective projects.

Table 5.1: Subject projects considered in this study

Cluster ID	Project Name	Normal	Buggy	Total
Hadoop	Bigtop	2,567	31	2,598
	Bookkeeper	2,289	84	2,373
	Camel	9,032	3,990	13,022
	Curator	2,690	28	2,718
	Drill	2,288	1,643	3,931
	Flink	20,369	4,613	24,982
	Gora	1,314	52	1,366
	Hadoop	9,881	627	10,508
	Helix	3,672	56	3,728
	Ignite	13,969	1,609	15,578
	Oodt	2,006	85	2,091
	Parquet-mr	2,126	114	2,240
	Reef	3,813	60	3,873
	Spark	19,591	376	19,967
	Tez	2,426	232	2,658
	Zookeeper	1,453	577	2,030
Hive	Accumulo	9,541	552	10,093
	Airavata	6,729	497	7,226
	Ambari	24,477	110	24,587
	Carbondata	4,249	552	4,801
	Falcon	2,096	130	2,226
	Flume	1,151	661	1,812
	Hive	11,759	518	12,277
	Oozie	2,244	114	2,358
	Storm	10,178	239	10,417
	Zeppelin	4,259	543	4,802
Avro	Avro	2,151	235	2,386
	Jackrabbit	8,488	370	8,858
Cocoon	Phoenix	3,284	168	3,452
	Cocoon	13,094	66	13,160
Hbase	Openjpa	3,404	1706	5,110
	Hbase	16,721	1058	17,779
	Derby	7,795	473	8,268
	Cayenne	6,365	285	6,650

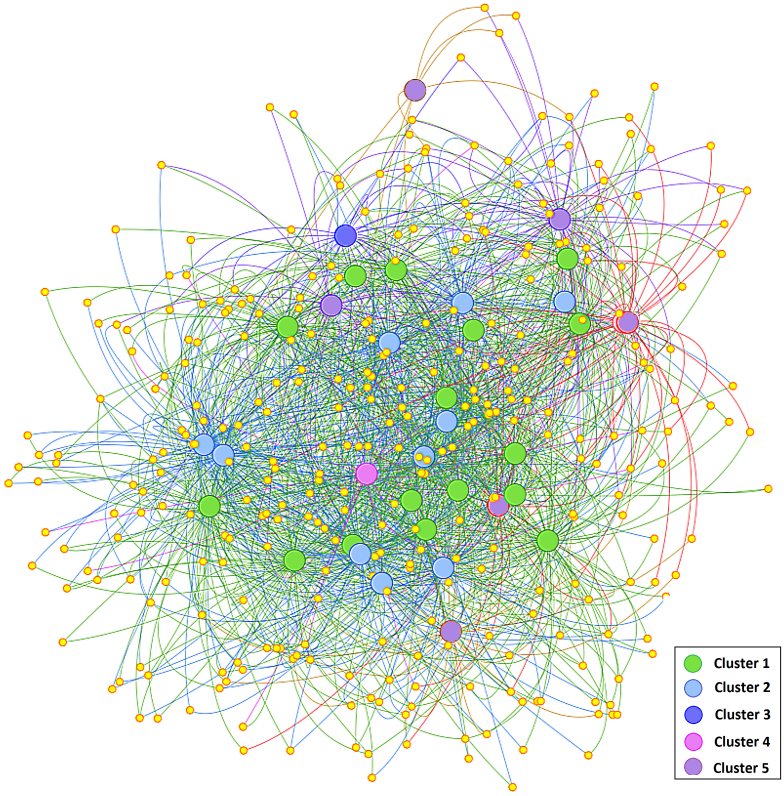


Figure 5.2: Clustering results of 34 Apache Projects using the LP Algorithm

5.2.2 Feature Extraction

We used the same data labeling and preparation in Chapter 3. These features are represented in five categories (Diffusion, Size, Purpose of Change, History, and Experience), which are widely used in the JIT-SDP area (e.g., [17] [13] [54] [3]).

However, using process features alone to capture the true essence of code changes comes with certain limitations. We also augment our approach by incorporating syntactic and semantic code representations, as detailed in references [14, 25], which are employed in training our DL models. Concurrently, we continue to utilize process features in conjunction with traditional ML models.

The DL models leverage features extracted from Commit Message (CM) and Code Changes (CC), as outlined in [13]. These syntactic and semantic features are extracted from both the CM and CC of commits using the GIT blame tool. The data is structured as

follows: commit message features, code changes at the commit level (noted as additions or deletions), commit time for the chronological organization during data preparation (as referenced in Chapter 3), and commit hex for unique commit identification. While the CM and CC features are used for training and testing the models, the other columns play roles in data segmentation and commit identification. Table 3.2 shows the features in six categories with a total of 16 features.

5.2.3 Machine Learning Algorithms

We chose six different ML models, NB, Decision Tree (DT), LR, Support Vector Machine (SVM), RF, and k-nearest neighbors (k-NN) that have been utilized in various JIT-SDP studies (e.g, [4,5,10,23,81]). Our objective is to delve deeper into the ClusterCommit approach and assess its performance when combined with diverse ML models. Building upon the processes previously outlined, including project clustering, dataset preparation, and evaluation metrics. In this part, we used Hyperparameter tuning to find the optimal configuration for each model [19, 20]. We aimed to explore ClusterCommit's potential beyond deep learning techniques (as detailed in the next Section 5.2.4).

5.2.4 Deep Learning Algorithms

These models have gained wide recognition in the field of JIT-SDP [13]. In this section, we employed a fine-tuning approach for the pre-trained models used by Zeng et al. [13]. Hyperparameter tuning focuses on finding the optimal configuration for a model's hyperparameters, while fine-tuning involves adapting a pre-trained model to a new task or dataset by updating its parameters. Both processes aim to enhance the performance of deep learning models, with the fine-tuning approach specifically leading to faster and more efficient training [82]. The following sections explain them in details.

DeepJIT Model

DeepJIT represents a comprehensive deep learning framework that employs a CNN to extract both the syntactic and semantic characteristics from Commit Messages (CM) and Code Changes (CC). The CNN is then trained to make predictions about commits that might contain bugs [14].

In Figure 5.3 the CM and CC are serving as input vectors in the feature learning stage. This stage comprises two sub-processes involving Convolution and the application of the ReLU (Rectified Linear Unit) as a non-linear activation function. The results of these processes are then flattened and directly transmitted to a fully connected layer, which is responsible for classifying the commit as either buggy or normal [14]. Typically, the optimization of the objective function is achieved using Stochastic Gradient Descent (SGD) [14, 83].

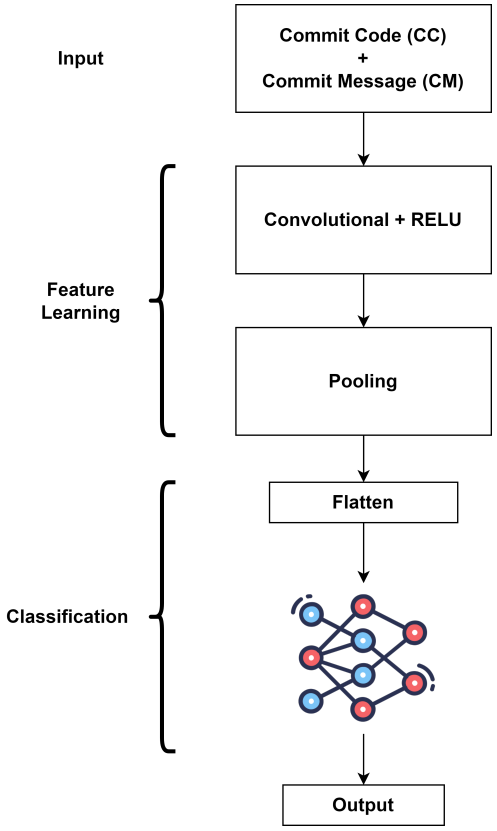


Figure 5.3: Structure of DeepJIT approach using Convolutional Neural Network (CNN).

DBN-JIT Model

In contrast, the DBN-JIT method employs the DBN to derive syntactic and semantic features from the 14 process features introduced by Yang et al. (2015) [24]. The input layer has a dimension of 14, matching the size of the process features suggested by Kamei et al. [5]. The hidden layers are structured as 14-20-12-12, with the last layer as input to ML models like LR, SVM, etc., as outlined in [24]. Consequently, the overall structure is 14 (input features) - 20-12-12 (hidden layers) - 2 (output layer). This study adopts the EALR model proposed by Kamei et al. [5], following a similar implementation by Zeng et al. [13] (see Figure 5.4). DBN-JIT and CC2Vec utilize deep learning algorithms for feature extraction only. In contrast, Deep Just-In-Time (DeepJIT) is an end-to-end DL framework [25].

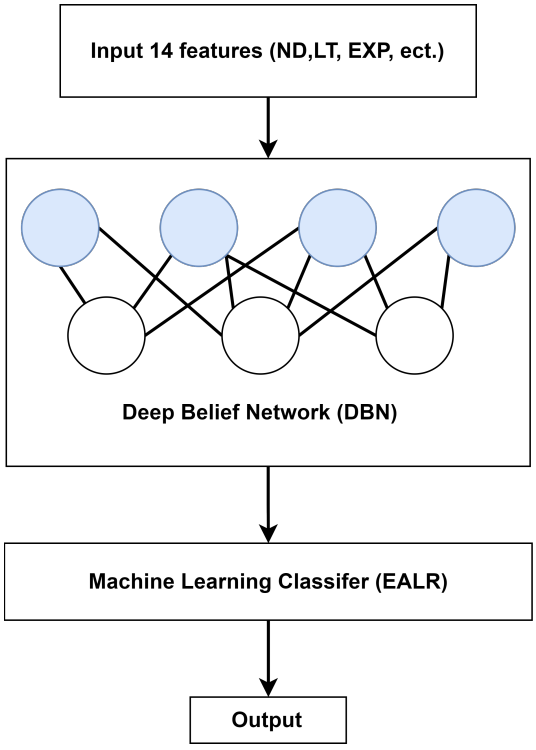


Figure 5.4: Structure of DBN-JIT approach using Restricted Boltzmann's Machines (RBM).

CC2Vec Model

In the case of the CC2Vec approach, the model incorporates these extracted features along with two additional vectors derived from the added and deleted lines in the commit. These extra vectors contribute as supplementary features, enhancing the classification process through the use of a Hierarchical Attention Network (HAN). HAN is applied to train the Neural Tensor Network, a specific type of deep learning architecture [25].

The CC2Vec framework undergoes several stages to handle code changes and extract meaningful features. Illustrated in Figure 5.5, the initial "Preprocessing" phase takes in Code Change (CC) data from a patch, generating a file list containing both removed and added lines of code [25]. The encoding layer then serves as input for extracting features from the HAN. The resulting embedding vectors are merged to form a comprehensive vector representation of code changes in the patch [25].

Next, the CC2Vec framework integrates features and semantic labels from the Commit Message (CM). The goal is to map the vector representation of the code change to a word vector originating from the initial CM line. This word vector captures the likelihood of various words characterizing the patch, chosen based on the CM content for semantic relevance [25].

Following these processes, CC2Vec utilizes each patch with a code change vector obtained from the intermediate output between feature extraction and feature fusion layers. Finally, ML models leverage these features to predict whether the changes are buggy or normal. Our study employed the EALR model suggested by Zeng et al. [13].

5.2.5 Evaluation Metric

Since we aim to examine the overall performance of the models independently from a given threshold, we choose to focus mainly on AUC-ROC, which considers the performance of a classifier across all possible decision thresholds [61], i.e., it evaluates a model's

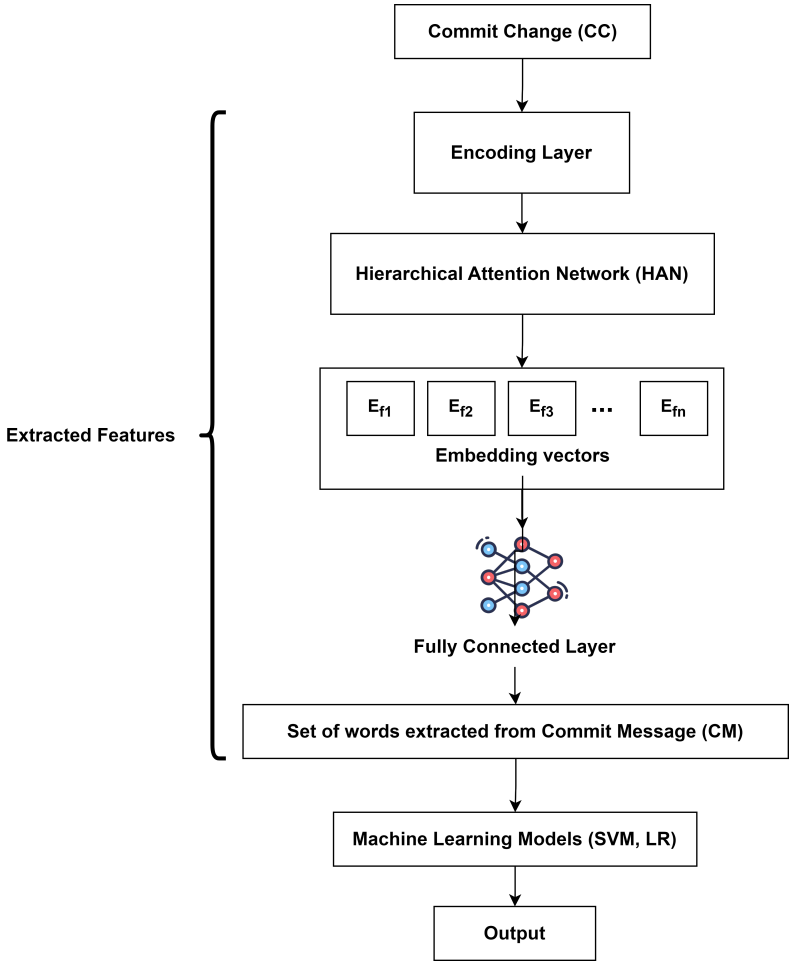


Figure 5.5: Structure of CC2Vec approach using Hierarchical Attention Network (HAN).

ability to discriminate between classes, regardless of the different thresholds (see Section 3.3.2).

5.3 Results and Discussions

5.3.1 RQ1: Does the utilization of the ClusterCommit method lead to enhanced performance in JIT-SPD models?

This section examines the outcomes of nine distinct JIT-SPD models using both Cluster-Commit and Single-Project approaches. To assess the statistical significance of the model

results, the Mann-Whitney U test [69] was employed. The null hypothesis (h_0) posits no statistical difference between the results, and the alternative hypothesis (h_1) suggests a difference. The null hypothesis is rejected when the p-value is less than 0.05 (95%confidence interval) [67]. The effect size, measured using Cliff's δ, quantifies the magnitude of the difference between the two groups [71].

Table 5.2 provides a comprehensive overview of the statistical analysis performed on the six ML results. Shaded cells in the "Improvement" and "Cliff's δ" columns highlight cases where the Single-Projects approach outperforms the ClusterCommit approach. Additionally, gray shading in the "p-value" column indicates statistically significant differences in the results.

In the Hadoop cluster, the ClusterCommit approach leads to a 4% and 5% improvement with the RF and SVM models, respectively, showing a moderate effect size ($0.147 < |\delta| \leq 0.330$). However, other models like NB and DT experience slight performance decreases (1% and 3%, respectively) with small δ values. LR and k-NN also exhibit notable performance degradation using the ClusterCommit approach, with drops of 10% and 6%, respectively, accompanied by large δ values.

For the Hive cluster, RF and SVM models benefit from improvements of 9% and 12%, respectively, when employing the ClusterCommit approach. These improvements have a moderate effect size ($0.147 < |\delta| \leq 0.330$). Conversely, models like NB, DT, LR, and k-NN experience slight performance declines (1%, 3%, 2%, and 2%, respectively) using the ClusterCommit approach, also with moderate effect sizes.

All models demonstrate performance degradation in the Avro and Cocoon clusters when utilizing the ClusterCommit approach. Even in the case of the SVM model in the Cocoon cluster, the δ value is zero, indicating no effect size between the ClusterCommit and Single-Project approaches.

Lastly, in the Hbase cluster, both RF and SVM models exhibit performance improvements of 6% and 12%, respectively. These improvements are associated with a moderate effect size for RF and a large effect size for SVM. These results are interpreted based on the number of commits in the Hbase cluster, which is twice as much as the number of commits in the Avro and Cocoon clusters.

Table 5.2: Statistical Analysis of Machine Learning Models using AUC measurement.

Cluster ID	Classifier	Improvments%	Cliff's δ	p-value
Hadoop	**NB**	-1%	-0.020	0.970
	RF	4%	0.260	0.345
	DT	-3%	-0.190	0.496
	LR	-10%	-0.460	0.089
	k-NN	-6%	-0.330	0.226
	SVM	5%	0.310	0.257
Hive	**NB**	-1%	-0.168	0.429
	RF	9%	0.312	0.137
	DT	-3%	0.277	0.187
	LR	-2%	-0.164	0.440
	k-NN	-2%	-0.117	0.585
	SVM	12%	0.148	0.486
Avro	**NB**	-25%	-1.000	0.333
	RF	-8%	0.000	1.000
	DT	-10%	-1.000	0.333
	LR	-24%	-0.500	0.667
	k-NN	-28%	-1.000	0.333
	SVM	-18%	-0.500	0.667
Cocoon	**NB**	-2%	-0.500	0.667
	RF	-8%	-1.000	0.333
	DT	-5%	0.000	1.000
	LR	-14%	-1.000	0.333
	k-NN	0%	0.000	1.000
	SVM	2%	0.000	1.000
Hbase	**NB**	-11%	-0.500	0.343
	RF	6%	0.250	0.686
	DT	-24%	-1.000	0.290
	LR	-12%	-0.875	0.057
	k-NN	-7%	-0.375	0.486
	SVM	12%	0.500	0.343

Table 5.3 analyzes the three deep learning models based on classifiers, comparing their

performance using different training approaches. The p-values for all models are greater than 0.05, indicating statistically significant differences in the results. The observed improvements in performance are not due to random chance but rather represent actual enhancements achieved by the ClusterCommit approach.

In the Hadoop cluster, the ClusterCommit approach demonstrates 3%, 9%, and 14% improvements for the DeepJIT, DBN-JIT, and CC2Vec models, respectively. These improvements signify the effectiveness of integrating the ClusterCommit approach in enhancing the predictive capabilities of the deep learning models in this cluster.

Moving on to the Hive cluster, the improvement ratios for the DeepJIT, DBN-JIT, and CC2Vec models with the ClusterCommit approach are 5%, 8%, and 13%, respectively. These results further emphasize the benefits of incorporating the ClusterCommit approach in improving the accuracy and performance of deep learning models in the Hive cluster.

In the Avro cluster, the ClusterCommit approach yields substantial improvements of 10%, 14%, and 30% for the DeepJIT, DBN-JIT, and CC2Vec models. These significant enhancements highlight the advantages of leveraging the ClusterCommit approach in boosting the predictive capabilities of deep learning models in the Avro cluster.

Similarly, the Cocoon cluster shows notable improvements with the ClusterCommit approach, with improvement ratios of 9%, 16%, and 25% for the DeepJIT, DBN-JIT, and CC2Vec models, respectively. These results further solidify the effectiveness of the ClusterCommit approach in enhancing the performance of deep learning models in the Cocoon cluster.

Finally, in the Hbase cluster, the ClusterCommit approach leads to 15%, 16%, and 9% improvements for the DeepJIT, DBN-JIT, and CC2Vec models, respectively. These improvements indicate the value of incorporating the ClusterCommit approach in achieving better predictive capabilities in the Hbase cluster.

Overall, the observed improvements across different clusters highlight the significant

benefits of utilizing the ClusterCommit approach, as it consistently enhances the performance of deep learning models and contributes to more accurate predictions in various cluster environments.

The effect size analysis indicates that, in general, the improvements achieved in performance by the ClusterCommit approach are characterized by small effect sizes for most models and clusters. However, there are a few notable exceptions.

Specifically, the CC2Vec model in the Hadoop cluster shows a large effect size, suggesting that the ClusterCommit approach significantly impacts the performance of this particular model in that cluster. Furthermore, the results for the Avro, Cocoon, and Hbase clusters demonstrate large effect sizes for all models except for the CC2Vec-JIT model in the Hbase cluster. This indicates that the ClusterCommit approach significantly improves the performance of the DeepJIT and DBN-JIT models in these clusters.

Overall, while the effect sizes are generally small for most models and clusters, the ClusterCommit approach still leads to marginal improvements in performance. Notably, the CC2Vec model consistently shows a more substantial effect, implying that it benefits the most from integrating the ClusterCommit approach.

Table 5.3: Statistical Analysis of Deep Learning Models using AUC measurement.

	Classifier	Improvments%	Cliff's δ	p-value
Hadoop	**DeepJIT**	3%	0.200	0.473
	DBN-JIT	9%	0.240	0.385
	CC2Vec-JIT	14%	0.510	0.059
Hive	**DeepJIT**	5%	0.188	0.376
	DBN-JIT	8%	0.086	0.692
	CC2Vec-JIT	13%	0.223	0.290
Avro	**DeepJIT**	10%	1.000	0.333
	DBN-JIT	14%	0.500	0.667
	CC2Vec-JIT	30%	1.000	0.333
Cocoon	**DeepJIT**	9%	0.500	0.667
	DBN-JIT	16%	1.000	0.333
	CC2Vec-JIT	25%	1.000	0.333
Hbase	**DeepJIT**	15%	0.500	0.343
	DBN-JIT	16%	1.000	0.290
	CC2Vec-JIT	9%	0.250	0.686

Finding RQ1: Our findings indicate that complex models benefit from the ClusterCommit approach compared to their simpler counterparts. Specifically, models such as RF and SVM exhibit improvements ranging from 3% to 12%. In contrast, simpler models like NB, LR, DT, and k-NN encounter challenges with data combinations. This phenomenon arises due to the increasing complexity of the distribution of buggy commits over time and the compounding complexity resulting from merging datasets within clusters. Notably, this trend is evident in DL models, where all DL models experience enhancements with the ClusterCommit approach compared to the single-project approach. Consequently, using ClusterCommit introduces a heightened level of complexity in detecting buggy commits, thereby enhancing the performance of complex models up to 30%.

5.3.2 RQ2: What is the impact of varying cluster sizes on the performance of JIT-SDP models?

It is clear that the RF and SVM models stand out for their excellent performance in the previous section, especially in dealing with larger clusters like Hadoop, Hive, and Hbase. The interesting point to highlight is that both RF and SVM models also show higher AUC-ROC scores in the Hbase cluster when the ClusterCommit approach is applied. We can see the details in Table 5.4, where the five clusters are listed with their names, the number of projects, and the total number of commits. The Hadoop cluster is the largest, encompassing 16 projects with a total of 113,663 commits, followed by the Hive cluster is the second largest with 10 projects and 80,599 commits. Despite having fewer projects, the Hbase cluster has a substantial commit volume of 37,807. In contrast, the Avro and Cocoon clusters have significantly fewer commits, totaling 11,244 and 16,612, respectively. So, even though the Hbase cluster has fewer projects compared to the larger clusters, its considerably larger commit volume plays a key role in enhancing the performance of the RF and SVM models in this specific scenario.

Table 5.4: Overall cluster sizes based on the number of projects and total commits.

Cluster Name	# Projects	Total Commits
Hadoop	16	113,663
Hive	10	80,599
Avro	2	11,244
Cocoon	2	16,612
Hbase	4	37,807

In Figure 5.6, we observe the distribution of outcomes for the Hadoop cluster. RF model achieved the highest AUC-ROC results among all ML models. Both DeepJIT and CC2Vec-EALR models display exceptional performance in over DL models. Despite the largest Hadoop cluster, including a total of 113,663 commits, ML models show only modest

improvements compared to the Hive and Hbase clusters. A more detailed exploration of these findings can be found in Section 5.3.3, where a thorough investigation uncovers the impact of factors such as bug reports on the outcomes of the ClusterCommit approach.

Moving to Figure 5.7, we visually represent the outcomes of the nine JIT-SDP models within the Hive cluster. Here, the RF and SVM models outperform other ML models regarding AUC-ROC results. Conversely, performance enhancements are evident across all DL models when the ClusterCommit approach is employed. This trend continues in Figure 5.8 for the Hbase cluster, where both RF and SVM models exhibit superior results among ML models, and all DL models demonstrate performance improvements with the ClusterCommit approach. It is worth noting that the Hive cluster, with a total of 80,599 commits, is the second largest, while Hbase, with around 37,807 commits, is recorded as the third-largest cluster.

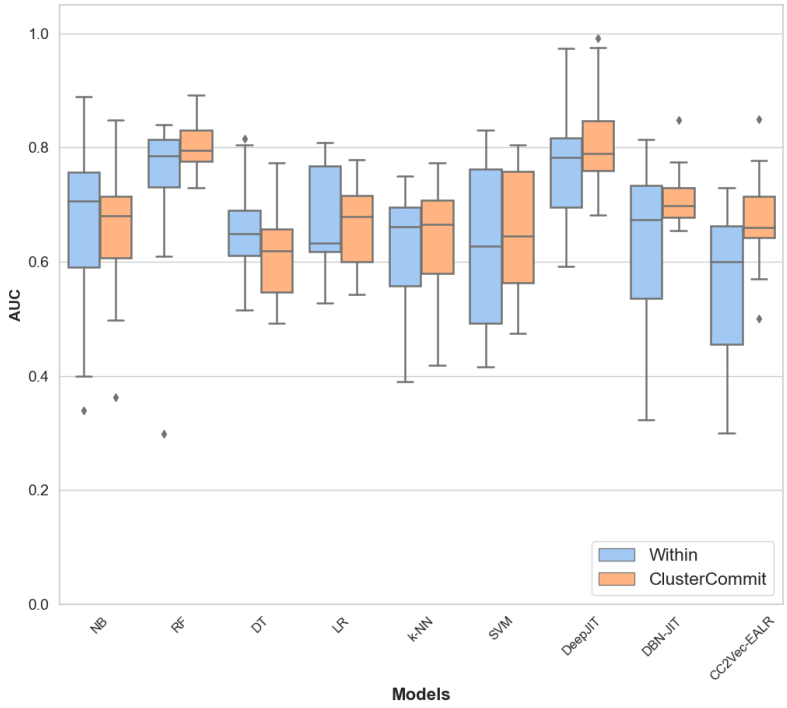


Figure 5.6: The results of 9 JIT models (Hadoop cluster).

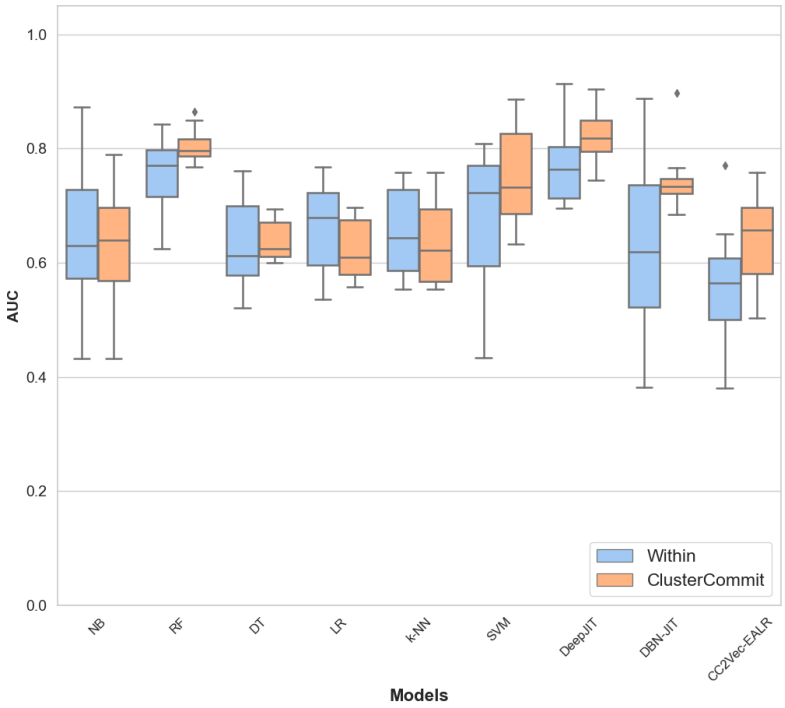


Figure 5.7: The results of 9 JIT models (Hive cluster)

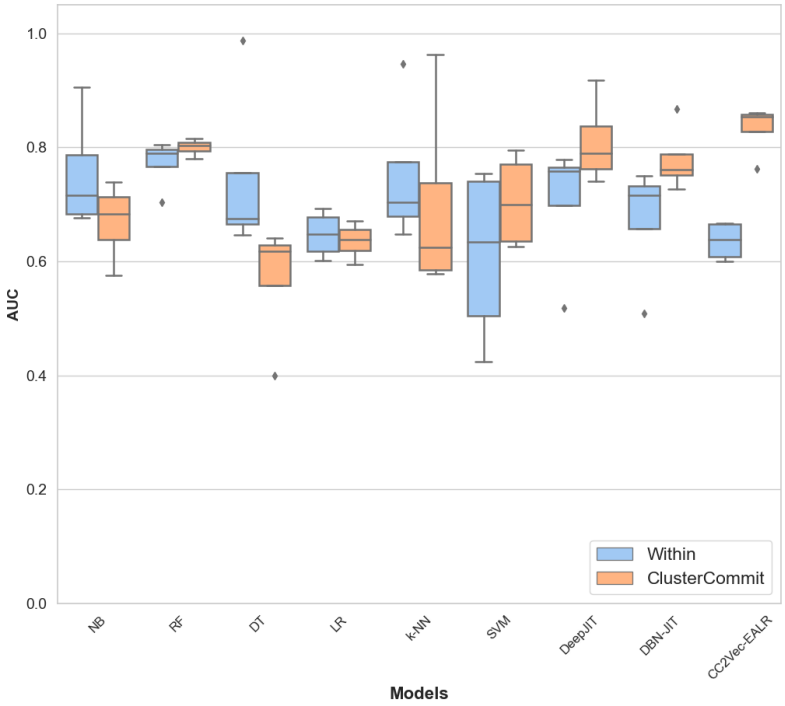


Figure 5.8: The results of 9 JIT models (Hbase cluster)

Figures 5.9 and 5.10 illustrate the outcomes for the Avro and Cocoon clusters. Interestingly, the performance of all ML models experiences a decline when utilizing the ClusterCommit approach in these two clusters. Conversely, all DL models show improvements, highlighting the effectiveness of the ClusterCommit approach in enhancing the performance of DL models across diverse clusters. It highlights the precision and dependability of predictions. Furthermore, the results indicate that incorporating ClusterCommit enhances the accuracy of DL models and reduces variance, resulting in more assured predictions [20, 21]. On the contrary, the impact of cluster size directly influences the performance of ML models, with more complex models performing better in larger clusters. This phenomenon leads to a non-linear data distribution as the cluster size increases [19, 20].

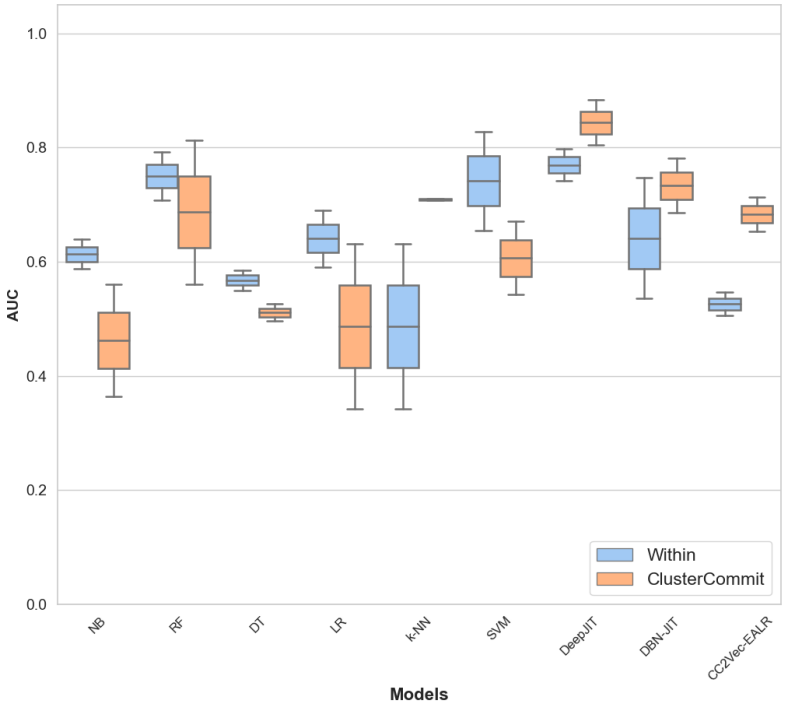


Figure 5.9: The results of 9 JIT models (Avro cluster)

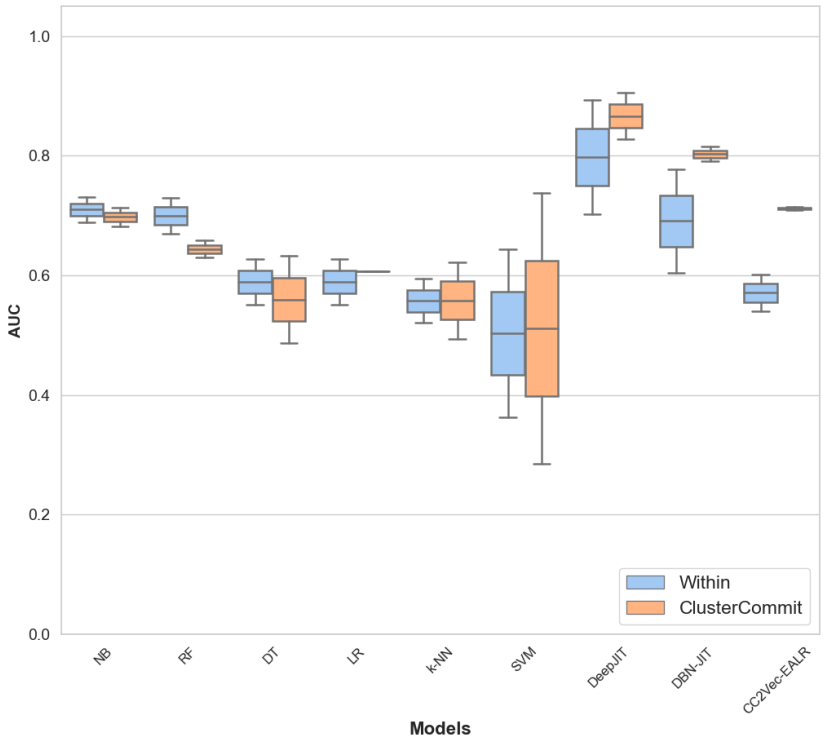


Figure 5.10: The results of 9 JIT models (Cocoon cluster)

Finding RQ2: In general, the ClusterCommit approach consistently increases the performance of deep learning models with no effect based on the cluster size. However, the extent of improvement is contingent on the specific model and the cluster size under consideration. In comparison to ML, more complex models such as RF and SVM yield better results when exposed to data from larger clusters, such as Hadoop, Hive, and Hbase. Conversely, the cluster's size directly impacts the classifier's efficacy. The results underscore that straightforward models like NB, DT, LR, and k-NN do not show improvement when utilizing Cluster of projects in both large and small clusters.

5.3.3 RQ3: What are the main factors that need to be considered using the ClusterCommit approach?

The improvement in performance observed with the ClusterCommit approach across different clusters can be attributed to two factors: data size complexity and cluster characteristics, which are explained further in what follows:

Data size and complexity

Data distribution often conforms to a non-linear and highly skewed pattern, a characteristic commonly associated with imbalanced data distributions [3, 5, 15]. Figures A.1, A.2, A.3, A.4, and A.5 (in Appendix) visually represent the distribution of 14 process features for the Hadoop project. Process features are illustrated using histograms, while syntactic and semantic features (CC and CM) are depicted as scatter charts (refer to Figure A.6 in Appendix). The distribution of normal and buggy data exhibits significant overlap, particularly in features like AGE, Entropy, NDEV, etc., resulting in complicated patterns that pose challenges in detection [19].

To this end, applying the ClusterCommit approach leads to an expansion in both data size and dimensions, a critical factor influencing various ML models [22, 84] which also known as heterogeneous data issue [22]. Simpler models such as NB, LR, DT, and k-NN experience a decline in performance when confronted with merged data. These classifiers are sensitive to data distribution and encounter limitations in processing and detecting patterns within extensive datasets. Conversely, more sophisticated models like RF and SVM, especially when leveraging the Radial Basis Function (RBF) Kernel for SVM, exhibit enhanced performance in handling the complexities of this data structure [19]. A similar trend is observed in DL models, which effectively operate with intricate data, revealing hidden patterns within it [21].

Cluster Characteristics and Bug Reports (BR)

As mentioned earlier, external factors like BR can influence the effectiveness of the ClusterCommit approach. Variability in the quality and quantity of BRs across clusters might contribute to differing degrees of improvement.

To dig deeper, we examined sample buggy commits and their corresponding bug reports of Bigtop, Curator, and Reef. We found that the description of many Bigtop bug reports mentions issues related to Hadoop and other systems that are supported by Bigtop, such as Spark. This is because Bigtop is not a standalone tool. Data scientists use it as an infrastructure to test and configure big data components such as those built with Hadoop and Spark. Bugs in these components affect Bigtop as well. For example, in the description of bug report BIGTOP-901[1] we can read:

> **Example 01:** "Oozie smoke test uses mapred.job.tracker and fs.default.name to find hostnames of the master daemons. In the **Hadoop** 2.x these names are **yarn**.resourcemanager.address and fs.defaultFS"

Another example would be the description of bug report BIGTOP-2288 [2] that indicates:

> **Example 02:** "Failed to start **Hadoop** timelineserver. Return value:1"

The same applies to Curator, which is a Java Virtual Machine for Apache ZooKeeper used to maintain configuration information for distributed synchronization of services. We found some bug reports of the Curator project, such as bug reports CURATOR-52 and CURATOR-409 refer to failures in ZooKeeper and Hadoop. For example, the bug report CURATOR-409 is linked to Hadoop bug report HADOOP-15974 and Zookepper bug report ZOOKEEPER-3181 and ZOOKEEPER-2355. Finally, the Reef system provides a

[1] https://issues.apache.org/jira/browse/BIGTOP-901
[2] https://issues.apache.org/jira/browse/BIGTOP-2288

library for developing portable applications for cluster resource managers such as Apache Hadoop YARN. Examples of Reef bug reports refer to Yarn are REEF-1787 and REEF-204.

In the Hive cluster, Accumulo is an application that is used to store and manage large data in clusters. It uses HDFS files, which are needed to be configured with Hive. The description of Accumulo bug report ACCUMULO-4672 [3] refers to Hive as we can see below:

> **Example 03:** "Eventually, the same class tries to extract the samplerConfiguration from this tableConfig (after noticing it is not present in the InputSplit), and this throws an NPE. Somehow, the tableConfig was null. It very well could be that **Hive** was to blame, I just wanted to make sure that this was captured before I forgot about it.".

In the same bug report, the reporter of the bug included the following comment when replying to a user who faces the issue:

> **Example 04:** "...no I haven't dug into it more. I don't see any **HIVE** jira issue open yet. Do you have something that I can follow your progress on? Also, out of curiosity, what problem are you trying to solve via upgrading the dependency? The last time I looked at this, all recent Accumulo 1.7 and 1.8 versions were compatible..."

We found similar cases with Ambari and Oozie projects. For example, the description of bug report OOZIE-3563 [4] of the Oozie project refers to the inability to initiate Hive as the cause of the problem:

[3] https://issues.apache.org/jira/browse/ACCUMULO-4672
[4] https://issues.apache.org/jira/browse/OOZIE-3563

Example 05: ”**Hive** example in pseudo-distributed mode is failing with the following error message: ACTION[0000008-191121145732587-oozie-mart-W@hive-node] Launcher exception: java.lang.RuntimeException: Unable to instantiate **org.apache.hadoop.hive.ql.metadata.** SessionHiveMetaStoreClient”.

Finally, with Hbase cluster, we found bug report HBASE-1562 [5] that contains comments:

Example 06: ”@Purtell: Can you put this in some testcode and see what you get? Properties sysProps = System.getProperties(); System.out.println(”arc ” + sysProps.getProperty(”sun.arch.data.model”)); @Stack, haha yeah that is a great pointer. The reason that we are seeing some of these problems are the code taken from **Derby** used to decide the arc size, but we can make this much better if we don’t need to run gc and try to figure it out that way, just want to see what result Andrew gets.”

We could not locate many examples for the smaller clusters (Avro and Cocoon) as we had previously demonstrated for the clusters (Hadoop, Hive, and Hbase), as illustrated in Table 5.5. This table presents information about each cluster and the total number of overlapping bug reporters, where overlapping reports indicate that each project in a cluster mentioned other projects from the same cluster.

Table 5.5: The Overlapping reports for each cluster.

Cluster Name	Total of Bug reports	Total of Overlap bug reports	Ratio of Overlapping reports
Hadoop	14,177	3,418	24%
Hive	3,916	3,383	86%
Avro	32	-	0%
Cocoon	218	-	0%
Hbase	57	41	72%

[5] https://issues.apache.org/jira/browse/HBASE-1562

The performance of JIT-SDP models using ClusterCommit depends on the size and characteristics of the dataset. Deep Learning, RF, and SVM models succeed with larger datasets due to their complexity and capacity to capture complicated patterns. Conversely, LR, NB, DT, and k-NN may perform better with smaller datasets, where simpler models and fewer data points allow them to avoid overfitting and make reasonable assumptions about data relationships. The choice of model should consider both the dataset size and the heterogeneous issue.

Finding RQ3: Two primary factors significantly impact the model's performance when utilizing the ClusterCommit approach. Firstly, the data size and complexity increase due to the data merging process, which is attributed to the high overlap between normal and buggy commits. This phenomenon introduces a heterogeneous data issue. Secondly, the characteristics of the clusters and the extent of overlapping Bug Reports play a crucial role. It was observed that a higher ratio of overlapping bug reports within a cluster could mitigate the effects of the heterogeneous issue, as evidenced in clusters like Hive and HBase, where the ratio of overlapping bug reports is above 60%.

5.4 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 known to perform well in various classification tasks and 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 rigorously tested the scripts to ensure they work properly. We also make all the data and scripts available online[6] 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.

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, particularly industrial proprietary systems to which we did not have access. Furthermore, we employed the implementation of RA-SZZ provided by the authors[7] to categorize the dataset into normal and buggy commits. While RA-SZZ is a robust labeling technique, it is important to acknowledge the possibility of errors in the implementation, which could potentially influence our results.

[6]https://github.com/wahabhamoulhadj/opencommit

[7]RA-SZZ GitHub repository: https://github.com/danielcalencar/ra-szz

5.5 Conclusion

In this chapter, we explore applying the ClusterCommit approach in JIT-SDP models. To investigate its efficacy, we conducted experiments using six machine learning algorithms—NB, RF, DT, LR, k-NN, and SVM—as well as three widely used deep learning algorithms, namely DeepJIT, DBN-JIT, and CC2Vec-EALR. These models have been established in the JIT domain. We compared their performance with their single-project classifiers, employing Time-aware Validation data-splitting. Our findings indicate that for simpler machine learning algorithms (NB, DT, LR, and k-NN), single-project classifiers outperform ClusterCommit. However, for more complex models like RF and SVM, ClusterCommit demonstrates improved performance, particularly with large clusters such as Hadoop, Hive, and Hbase. Moreover, all deep learning algorithms demonstrate enhancement when using the ClusterCommit approach.

Two key factors emerge as critical for machine learning and deep learning models. Firstly, the growth of the heterogeneous data issue poses a significant challenge. Secondly, the ratio of overlapping bug reports between cluster projects significantly influences model performance.

Future directions in research should address several key aspects. Firstly, the exploration of the ClusterCommit approach in a cross-project context, where JIT models are trained on a group of projects from the same cluster and tested individually on target projects. Our study indicates that utilizing ClusterCommit can mitigate the heterogeneous data issue [22]. Community detection algorithms could further reduce variance between projects by incorporating additional aspects into the community graph. Secondly, the impact of graph clusters should be studied using a more comprehensive methodology, particularly in the context of social network analysis [76, 85].

Chapter 6

Commit-Time Defect Prediction Using One-Class Classification

6.1 Introduction

In this chapter, we investigate using One-Class Classification (OCC) algorithms in JIT-SDP, where we train a 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 [28, 29]. 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. [5]. More specifically, we compare the performance of the following OCC algorithms: Isolation Forest (IOF) [86],

One-Class k-Nearest Neighbors (OC-k-NN) [87], and One-Class Support Vector Machine (OC-SVM) [28] 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 [13, 14]. We decided to maintain consistency within our research framework and focus on specific 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 [20, 88]. 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. [13] and Pornprasit and Tantithamthavorn [10], 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 chapter: The next section reviews software defect prediction and techniques for learning from imbalanced data. Section 6.2 describes three one-class classifiers, which will be used in our experiments. Section 6.3 describes methods and experimental protocol used for conducting the experiments. In Section 6.4, we present the results to provide answers to the research questions. Potential threats to validity and our mitigating actions are presented in Section 6.5, followed by the conclusions and future work in Section 6.6.

6.2 One-class classification

OCC techniques rely on data from the majority (negative) class to train the machine learning model instead of binary classifiers, which need labeled data from both positive and negative classes [26]. 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 6.1) and then detect deviations from this expected behavior as anomalies or outliers [89].

In this chapter, 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.

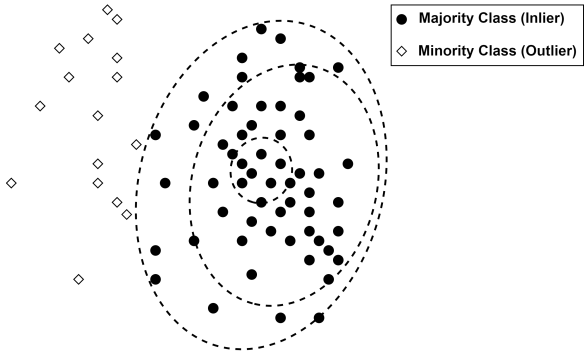


Figure 6.1: An illustration of OCC approach learning from the majority class and detecting deviations as anomalies or outliers.

6.2.1 One Class Support Vector Machine (OC-SVM)

Support Vector Machine (SVM) is a binary supervised machine learning approach that separates classes based on the maximum margin hyperplane [19, 63]. 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 [19]. OC-SVM is a version of SVM adapted to the OCC approach that only learns from the majority class [84]. 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. 6):

$$\min_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j K(x_i, x_j) \tag{6}$$

$$\text{subject to } 0 \leq \alpha_i \leq \frac{1}{\nu n}, \sum_{i} \alpha_i = 1 \tag{7}$$

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. 8):

$$f(x) = sign(\sum_{ij} \alpha_i \alpha_j K(x_i, x_j) - \rho) \tag{8}$$

where ρ denotes the offset of the separating hyperplane.

6.2.2 One Class k Nearest Neighbors (OC-k-NN)

k-NN is a supervised machine learning approach that uses lazy processing to classify the data [90]. 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 [91]. 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 [91].

The distance between the data point and training points is measured using Minkowski distance as shown in Equation (9). The Minkowski distance d is the generalized formula of both (Manhattan for $p = 1$ and Euclidean for $p = 2$) distances [91]. After selecting the distance measure, we only need to tune the k value, the number of the closest neighbors to the new incoming point.

$$d = \left(\sum_{i=1}^{n} |p_i - q_i|^p \right)^{\frac{1}{p}} \tag{9}$$

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 [92].

6.2.3 Isolation Forests (IOF)

The Isolation Forests (IOF) is a tree-based ensemble algorithm, the OCC counterpart of the Random Forests (RF) binary classifier [93]. 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) [86].

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 it is classified as abnormal (belonging to the minority class) [86]. Finally, the anomalous score of a particular point x is calculated as shown in Equation (10):

$$s(x, n) = 2^{-E(\frac{h(x)}{c(n)})} \tag{10}$$

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.

6.3 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 3.2. 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 6.4.1. In total, we trained 408 models ($(3$ **binary models**$*3$ **balancing methods**$+3$ **OCC models**$)*34$ **projects**) tested 30 times.

We use the PyOD library[1] 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 [94]. It supports over 40 anomaly detection algorithms and has been used in various academic research and commercial products [92]. We used the well-known Scikit-learn [94] library for binary classifiers, which is widely used in this field.

A classification model is built in three steps: training, validation, and testing. The initial

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

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

6.3.1 Cross-validation approach

This chapter uses 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 6.2.

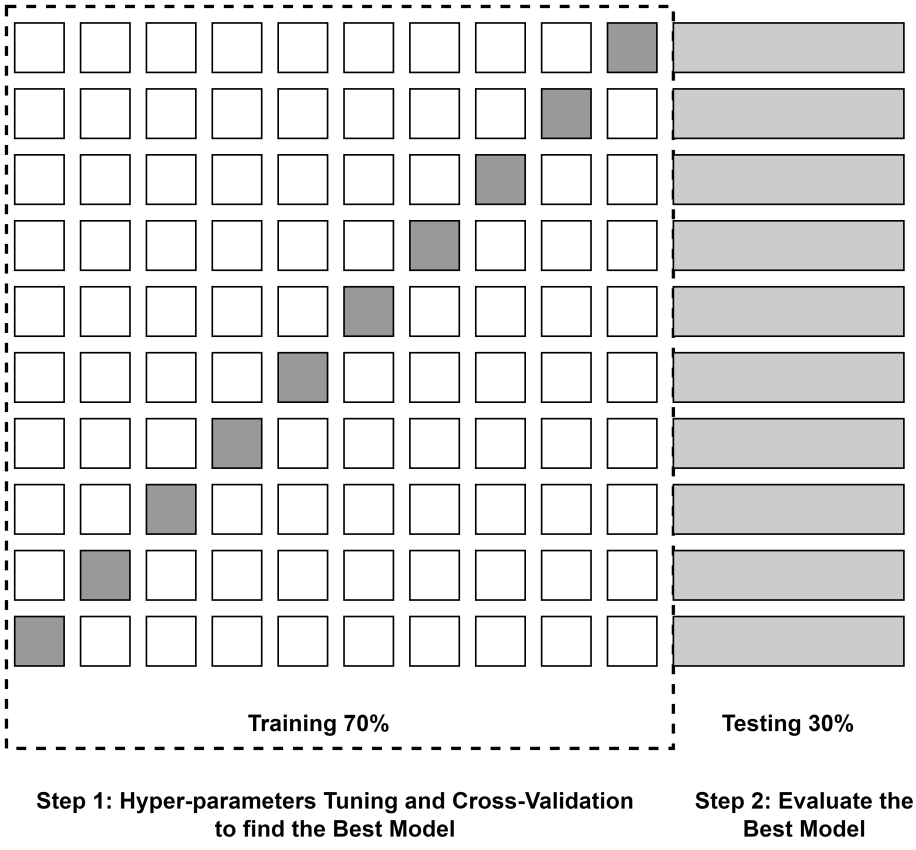


Figure 6.2: Cross-Validation Approach.

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 6.2 (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 6.2 (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 6.2 (Step 1). From Lines 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 6.2 (Step 2).

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,

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% */
$Data_{Tuning}, Data_{Testing} \leftarrow split\_data(Data)$
$folds \leftarrow Generate\_Folds(Data_{Tuning}, 10)$
/* Evaluate the binary model without balancing the data */
for $index = 1; index < Size(folds); index+ = 1$ **do**
    $Data_{Training} \leftarrow All\ Folds\ Except\ folds[index]$
    $Data_{Validating} \leftarrow folds[index]$
    $Model_{Imbalance} \leftarrow Get\_Best\_Model(Data_{Training}, Data_{Validating})$
$Results_{Imbalance} \leftarrow Model_{Imbalance}.Evaluate(Data_{Testing})$
/* Evaluate the binary model after balancing the data */
$ImbalanceMethods \leftarrow \{OverSampling, DownSampling\}$
foreach $Method \in Imbalance\_Methods$ **do**
    $Data_{Tuning} \leftarrow Balancing(Data_{Tuning}, Method)$
    for $index = 1; index < k; index+ = 1$ **do**
        $Data_{Training} \leftarrow All\ Folds\ Except\ folds[index]$
        $Data_{Validating} \leftarrow folds[index]$
        $Model_{Balanced} \leftarrow Get\_Best\_Model(Data_{Training}, Data_{Validating})$
    $Results_{Balanced} \leftarrow Model_{Balanced}.Evaluate(Data_{Testing})$

the use of cross-validation is highly recommended in order to build more reliable models, avoid overfitting, and improve generalization to unseen data [95, 96]. 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 [95, 96]. 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 6.3).

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% */
$Data_{Tuning}, Data_{Testing} \leftarrow split\_data(Data)$
$folds \leftarrow Generate\_Folds(Data_{Tuning}, 10)$
for $index = 1; index < k; index+ = 1$ **do**
    $Data_{Training} \leftarrow All\ Folds\ Except\ folds[index]$
    $Data_{Validating} \leftarrow folds[index]$
    $Data_{normal}, Data_{buggy} \leftarrow DataFilter(Data_{Training})$
    /* Merge the buggy data with Validation set */
    $Data_{Validating} \leftarrow Merge(Data_{Validating}, Data_{buggy})$
    /* Train the OCC model only with normal data */
    $Model_{Best} \leftarrow Get\_Best\_Model(Data_{normal}, Data_{Validating})$
$Results_{One-class} \leftarrow Model_{Best}.Evaluate(Data_{Testing})$

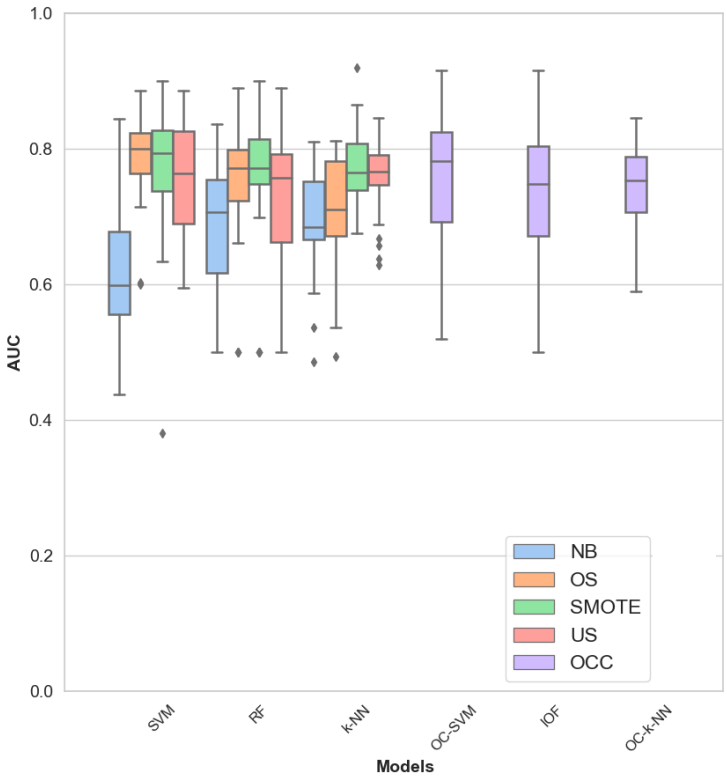


Figure 6.3: Overall performance of binary and OCC models using average AUC for all projects (Cross-Validation).

6.3.2 Time-sensitive validation approach

Tan et al. [16] proposed a time-sensitive validation (TV) approach to training 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 6.4 (Step 1). In Lines 3 and 4, the dataset is divided into three distinct parts: training (50%), validation (20%), and testing (30%) sets (Figure 6.4, 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 6.4). This data is then used for model training and parameter validation, using the validation set as shown in Figure 6.4 (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).

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 6.4, Step 2). In Line 5, the training data is filtered as normal and

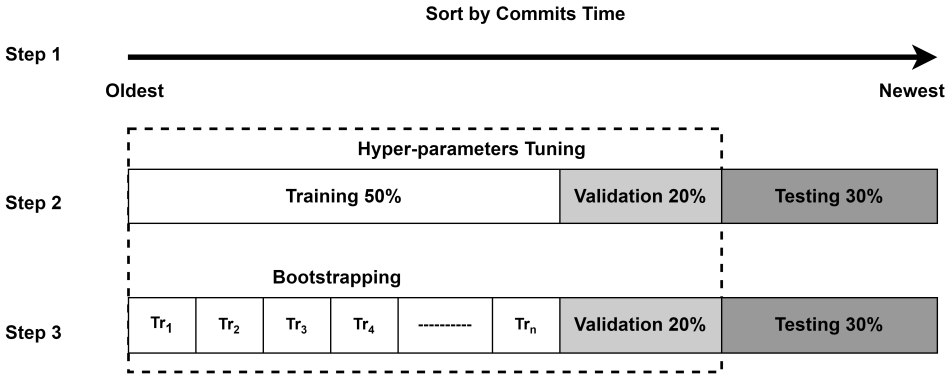


Figure 6.4: Splitting data using the time-sensitive validation Approach.

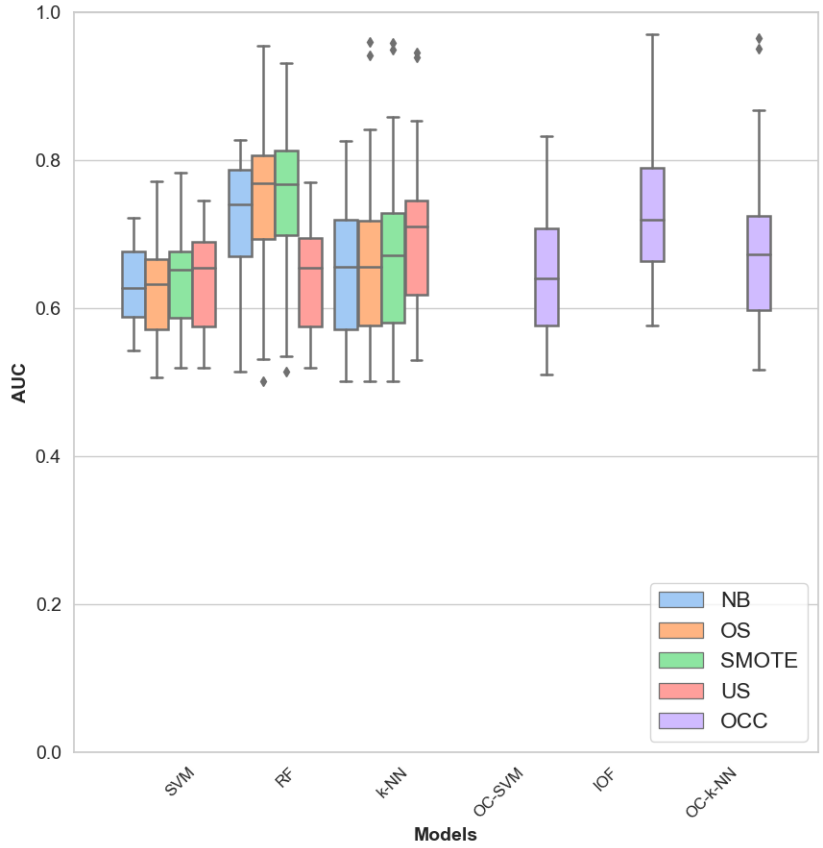


Figure 6.5: Overall performance of binary and OCC models using average AUC for all projects (time-sensitive validation).

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

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

Data: $Data$
Result: $Results_{imbalance}, Results_{Balanced}$

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

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%).

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

Data: $Data_{Tuning}$
Result: $Results_{One-class}$

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

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 6.5. This approach aims to assess the model's performance and stability by generating multiple samples that mimic the characteristics of the original training data.

6.4 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.

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

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 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 6.1 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 6.1 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 6.1: 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 6.2 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 6.2 shows results of F1-score where SVM outperforms 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.

Table 6.2: 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%

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 6.3, 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 6.3: 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 6.4 exhibits the outcomes of both OCC and binary models, incorporating balancing techniques (OS, SMOTE, and US), following the implementation of a time-sensitive 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.

Table 6.4: 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%

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 3.3, 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 3.3). This threshold aligns, by default, with the point closest to the upper left corner of the ROC curve, as depicted in Figure 3.2. 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 6.6. For instance, the ROC curve in Figure 6.6 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).

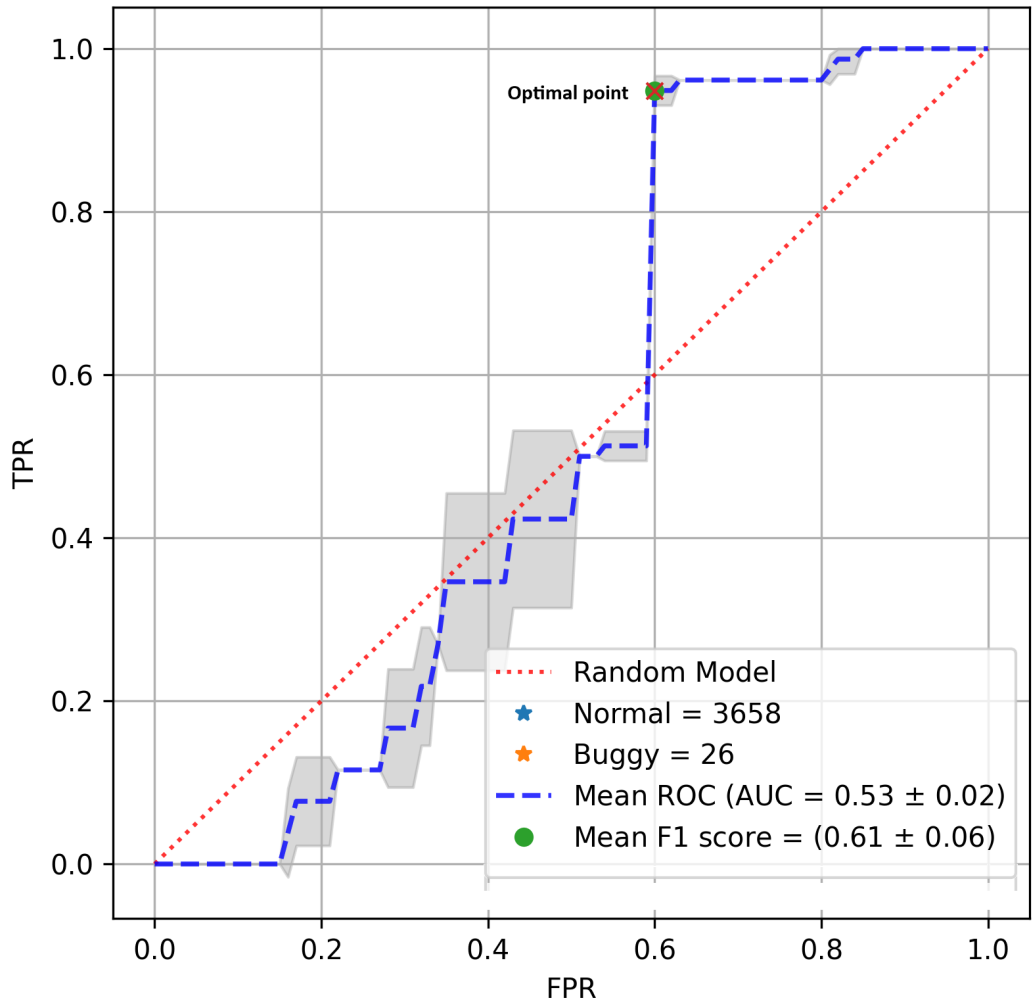


Figure 6.6: An example of testing for a project to display F1-score and AUC based on the ROC curve.

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.

6.4.2 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.

RQ2.1: Results using cross-Validation

Table A.1 (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 6.7 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

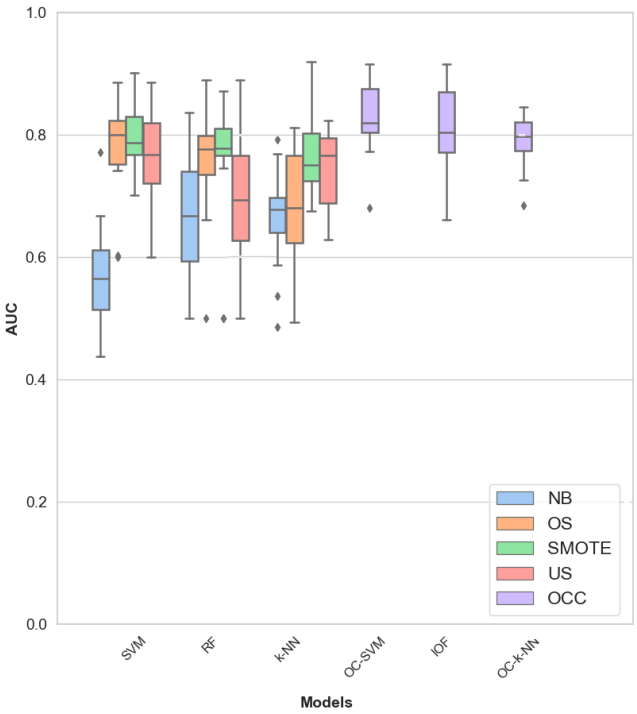


Figure 6.7: Average AUC of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (Cross-Validation).

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.

For projects with low IR (a total of 20 projects out of 34), the results show that binary classifiers usually perform better than OCC methods (see Figure 6.8). Although the results

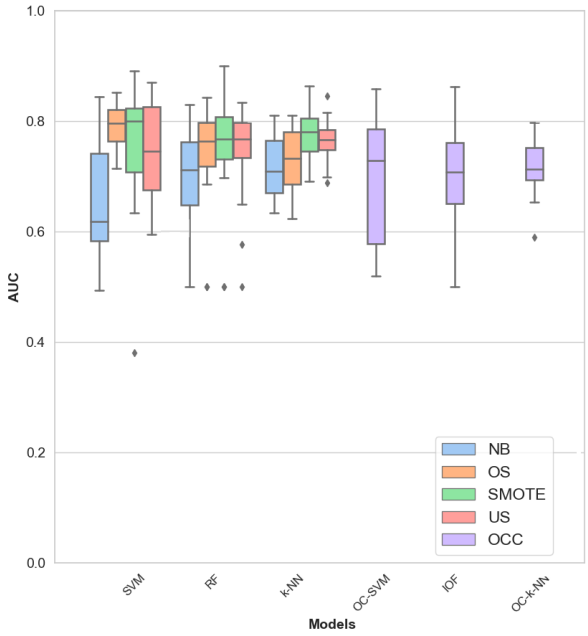


Figure 6.8: Average AUC of binary and OCC models for projects with low IR (IR<22) (Cross-Validation).

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 (sometimes 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 a balancing technique is also recommended as it was already stated in related work (see Song et al. [15]).

Table 6.5 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 [71, 73].

Table 6.5: 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 6.6 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 is small in all cases.

Table 6.6: 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 6.9 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 A.2 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 6.10 when IR is low.

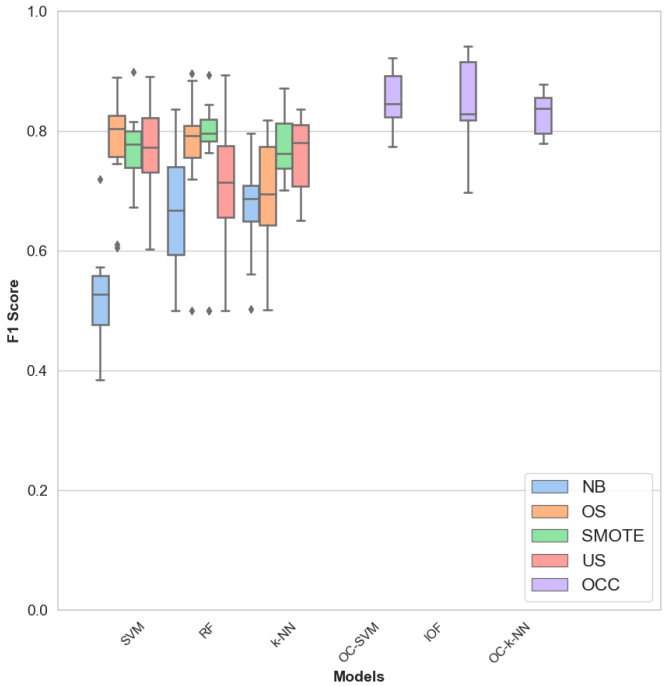


Figure 6.9: Average F1-score of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (Cross-Validation).

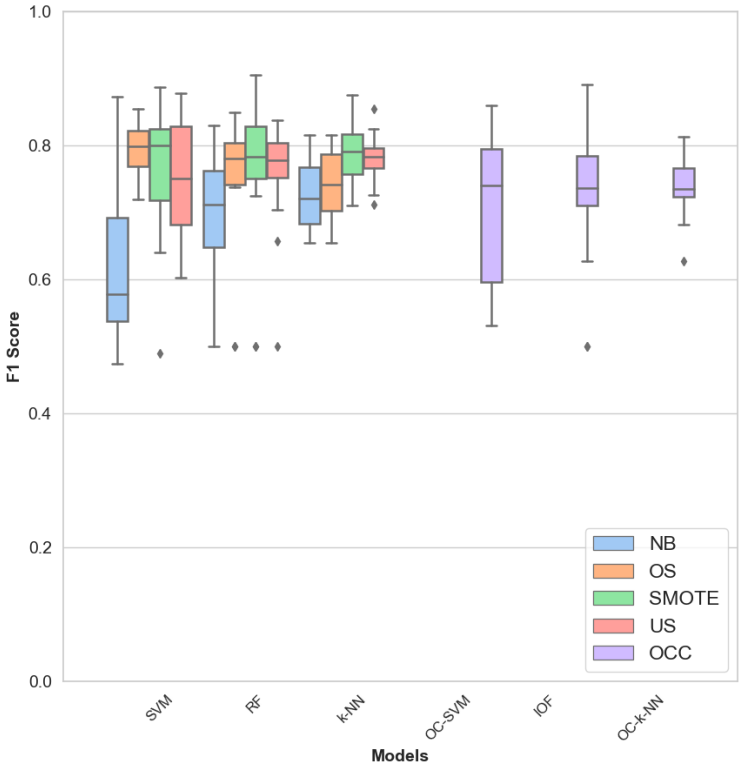


Figure 6.10: Average F1-score of binary and OCC models for projects with low IR (IR<22) (Cross-Validation).

Table 6.7 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 6.8, 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 6.7: 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 6.8: 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.

RQ2.2: Results using time-sensitive validation

Table A.3 (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 (6.4.2) elaborates further on these cases. This observation persists even when applying balancing techniques such as OS, US, and SMOTE.

Figure 6.11 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 6.12 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 of 29 ones and OC-k-NN had 15 worst results out of a total of 29 projects.

Table 6.9 represents 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 $\delta \geq 0.474$). 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

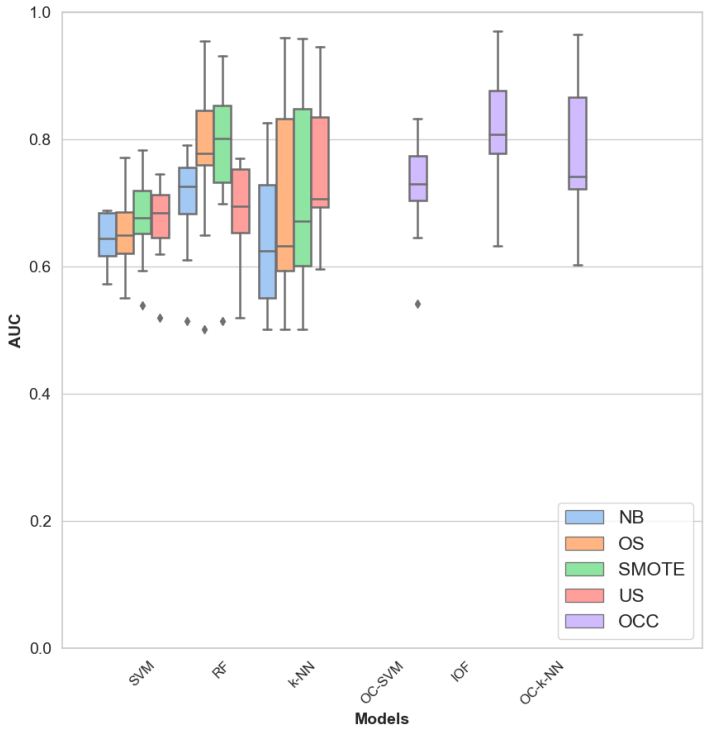


Figure 6.11: Average AUC of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (time-sensitive validation).

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 6.9: 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 6.10 presents the Cliff's δ values for projects characterized by low IR. All the

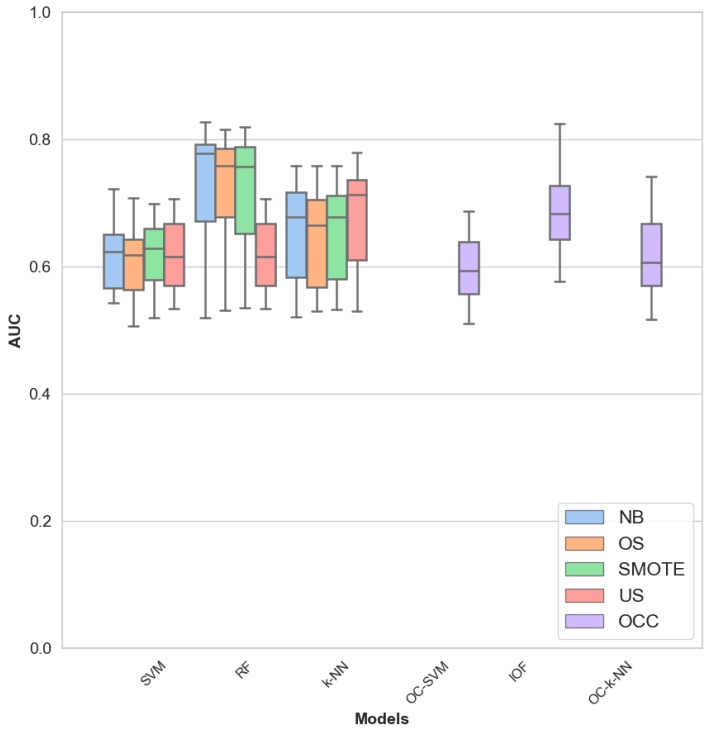


Figure 6.12: Average AUC of binary and OCC models for projects with low IR (IR<22) (time-sensitive Validation).

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 6.13 illustrates the average results using the TV approach through boxplots for projects characterized by medium and high IR. The detailed F1-score

Table 6.10: 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

results for TV can be found in Table A.4 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, and SMOTE balancing methods, while k-NN-US demonstrates similar performance to the OC-k-NN model.

Table 6.11 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 6.14, 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 6.12, 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.

Table 6.11: 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

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 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 into

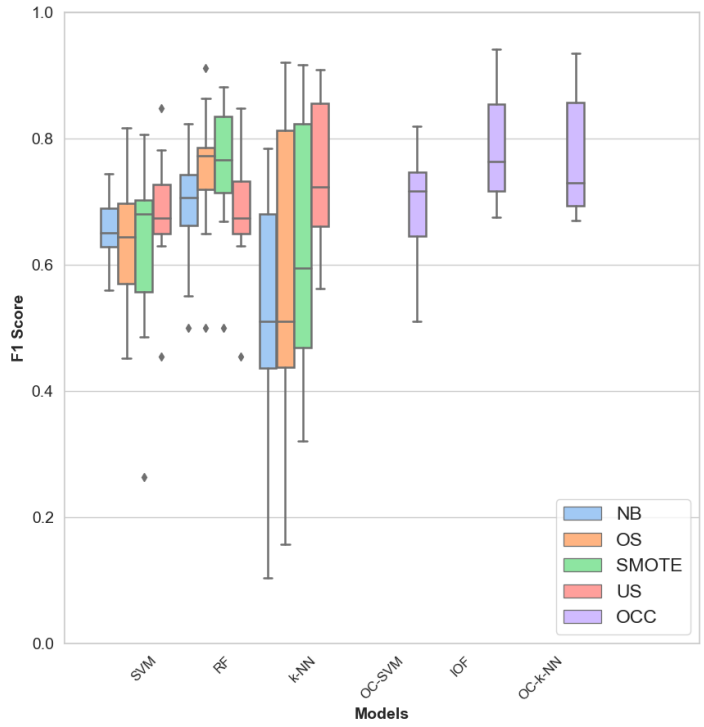


Figure 6.13: Average F1-score of binary and OCC models for projects with medium to high data imbalance ratio (IR>=22) (time-sensitive Validation).

Table 6.12: 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

three parts: training, validation, and testing (see Figure 6.4), 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

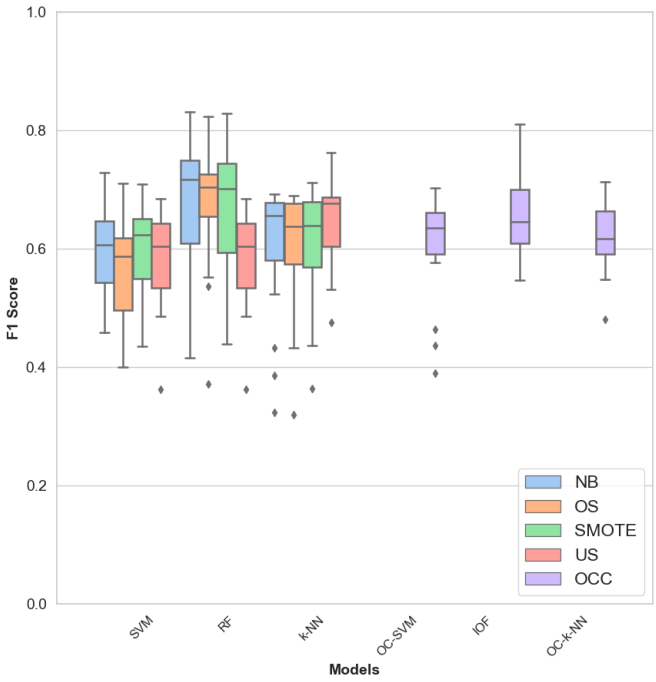


Figure 6.14: Average F1-score of binary and OCC models for projects with low IR (IR<22) (time-sensitive validation).

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. [15]. 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.

6.4.3 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 3.2. 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 [97]. Table 6.13 lists the 14 features in the descending order of importance. Table 6.14 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 the top 9 features are used. We only kept the top 9 features shown in Table 6.13 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 6.15 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 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

Table 6.13: 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 6.14: 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	-	-	-

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 [98].

Table 6.15: 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**	-	-	-

We also do the same process with the time-sensitive validation protocol. Table 6.16 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 6.16: 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

Table 6.17 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 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 6.17: 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 6.18 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.

Table 6.18: 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**	-	-	-

6.5 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[2] 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.

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 [3] to label the dataset into normal and buggy commits. Although RA-SZZ is a powerful labeling technique, errors in the implementation may occur, which can impact our results.

6.6 Conclusion

In this chapter, 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,

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

[3]RA-SZZ GitHub repository: https://github.com/danielcalencar/ra-szz

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 $IR \geq 22$), 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 will 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., [29, 32]). 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.

Chapter 7

JITBoost: Boosting Just-In-Time Defect Prediction Performance Using Boolean Combination of Classifiers

7.1 Introduction

In this chapter, we propose a framework called JITBoost, which uses BCC [30] to predict buggy commits. BCC leverages Boolean functions to create classifiers that combine the decisions of individual classifiers with the aim of improving the overall prediction accuracy. Specifically, we investigate the use of three BCC algorithms (discussed in more detail in section 7.2): Brute-force Boolean Combination (BBC) [30], Iterative Boolean Combination (IBC) [28], and Weighted Pruning Iterative Boolean Combination (WPIBC) [29]. These algorithms are used in the field of anomaly detection (e.g., [29] [28]) and have been shown to perform better than single classifiers. We propose JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC and compare their performance with individual JIT-SDP algorithms.

To evaluate the performance of the JITBoost framework, we conduct experiments using a dataset comprising 34 projects and a total of 259k commits. Our objective is to compare the effectiveness of JITBoost-BBC, JIT-Boost-IBC, and JIT-Boost-WPIBC algorithms against existing JIT-SDP techniques, which use individual traditional ML methods as well as DL algorithms.

Our research aims to address the following three research questions:

- **RQ1**: How does the performance of JITBoost algorithms compare to JIT-SDP models that use traditional machine learning algorithms?
- **RQ2**: How does the performance of JITBoost algorithms compare to a deep learning JIT-SDP algorithm?
- **RQ3**: How does the combination of traditional JIT-SDP models and deep learning models affect the performance of the JITBoost algorithms?

For RQ1, we combine the decisions of six traditional classifiers (see section 7.2) using BBC, IBC, and WPIBC and compare the results with that of each individual classifier. We found that all Boolean combination algorithms perform better than the single ML algorithms. For RQ2, we compare the Boolean combination classifiers of RQ1 with the newly proposed JIT-SDP DL algorithm DeepJIT [13]. We found that the combination of traditional ML algorithms performs better than when using DeepJIT. In the last question, RQ3, we compare the combination of six traditional classifiers and another combination of the same classifiers and DeepJIT. The objective is to see if the DL algorithm improves the accuracy of the combination. Our findings show that the accuracy is only improved when using JITBoost-WPIBC.

These findings have important implications for practitioners in software development, as they suggest that simpler ML models may be just as effective as more complex DL models. This is also inline with the recent finding of Zeng et al. [13], which showed a

simple ML method can outperform CC2Vec [25] and DeepJIT [14] when applied to very large datasets.

This study can benefit researchers and practitioners by proposing a JIT-SDP framework that can improve the accuracy of predicting buggy commits, leading to more reliable and accurate tools for defect prediction at the commit level.

The structure of the chapter is as follows: The next section provides a review of software defect prediction techniques using ML and DL. In section 7.2, we present the Boolean combination algorithms. Section 7.3 describes the study setup, including the datasets, features, evaluation metrics, and the algorithms. In section 7.4, we present the results that address the research questions. Section 7.5 outlines potential threats to validity and the actions taken to mitigate them. Finally, the chapter concludes with section 7.7, which presents the conclusions and highlights future research directions.

7.2 Boolean Combination of Classifiers

The Boolean combination of classifiers is an approach that uses Boolean logic operators, such as AND, OR, and NOT, to combine the decision of multiple classifiers into a single classifier. These classifiers can be of any type, including decision trees, Support Vector Machines, logistic regression, etc. The approach works by first generating a set of classifier predictions based on the available features in the dataset. Then, it combines these predictions on the Receiver Operator Characteristics ROC curve space using Boolean operators to form the final new classifier.

Consider the decision vectors of two classifiers A and B. We can combine the decisions using six Boolean operators: $\wedge$, $\vee$, $\neg$, $\oplus$, $\neg\wedge$, $\neg\vee$, $\equiv$. There exist 10 possible ways to combine these decisions, namely $A \wedge B$, $\neg A \wedge B$, $A \wedge \neg B$, $\neg(A \wedge B)$, $A \vee B$, $\neg A \vee B$, $A \vee \neg B$, $\neg(A \vee B)$, $A \oplus B$, and $A \equiv B$. Each of these combinations results in a new classifier, which may or may not improve the accuracy of the individual classifiers. The idea of a Boolean

combination of classifiers is to explore the space of all possible combinations in order to find the combination that provides the best accuracy on the ROC curve [30]. Doing so, however, may result in computational overhead as the number of classifiers increases. In this chapter, we experiment with three different Boolean combination algorithms, namely Brute-force Boolean Combination (BBC) [30], Iterative Boolean Combination (IBC) [32], and Weighted Pruning Iterative Boolean Combination (WPIBC) [29].

Figure 7.1 illustrates an example of two models, A and B, in the context of a Boolean combination of classifiers. The dashed line in the plot represents the result of combining the two models using a Boolean function. Each point on the dashed line corresponds to the combination of a point from model A and another point from model B. The BBC algorithm explores all possible Boolean combinations to plot the dashed line in the ROC space. This approach allows for optimizing the performance of the combined classifier and finding the best trade-offs between true positive and false positive rates.

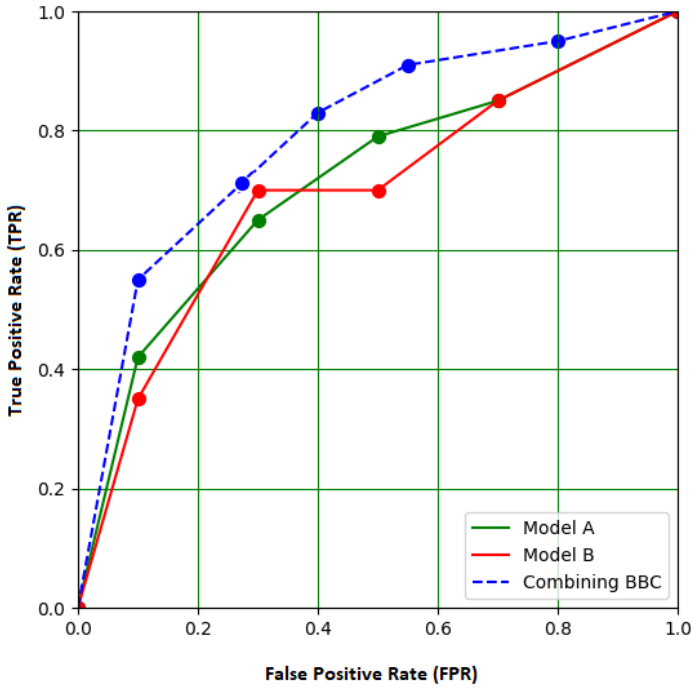


Figure 7.1: Example of combining two models in the ROC space

Brute-force Boolean Combination (BBC)

The Brute-force Boolean Combination algorithm is an exhaustive search algorithm that generates all possible combinations of Boolean operators. This approach tests all combinations of the individual classifier outputs to find the combination with the highest classification accuracy [99].

Suppose we have three individual classifiers; each classifier produces a binary output (either 0 or 1) for a given input. The BBC approach explores all $2^3 = 8$ possible combinations of the three binary outputs (000, 001, 010, 011, 100, 101, 110, 111) to see which combination results in the highest classification accuracy. Boolean logic operators are applied to the outputs (predictions) to create a final prediction. Then, evaluate the classification accuracy of each combination by comparing the predicted output to the true output for a set of test labels [99].

The combinations produce the highest classification accuracy to be selected as the optimal combination for the given classifiers. However, this approach can become computationally expensive as the number of individual classifiers increases since the number of possible combinations grows exponentially with the number of classifiers [99].

Iterative Boolean Combination (IBC)

IBC is another approach for combining multiple classifiers into a single classifier using Boolean operators. Unlike the BBC approach that tries all possible combinations of classifiers, IBC combines classifiers iteratively until a satisfactory level of accuracy is achieved [31].

The IBC algorithm operates by first selecting an initial subset of classifiers, such as the top-performing classifiers in terms of AUC or accuracy. Then, IBC iteratively combines this subset of classifiers using Boolean operators (such as AND, OR, and NOT) to

generate a new complex classifier. The performance of the new classifier is evaluated using a validation set, and the process is repeated until a satisfactory level of performance is achieved [31].

The main difference between IBC and BBC is that IBC is more efficient as it does not try all possible combinations of classifiers. Instead, it starts with an initial subset of classifiers and iteratively combines them until a satisfactory level of performance is achieved. Suppose we want to synthesize a boolean function that satisfies the following properties: 1) It has three input variables (A, B, and C). 2) It outputs 1 if and only if exactly two of its inputs are 1. Using the BBC method, we would need to consider all possible combinations of the input variables ($2^3 = 8$ combinations) and evaluate the output of the function for each combination. We could then use these evaluations to construct a truth table and derive the boolean expression that satisfies the desired properties. This approach can be time-consuming and impractical for larger functions with many input variables [30, 32]. On the other hand, the IBC algorithm could be used to synthesize the function more efficiently [31]. We can start with a set of initial functions that satisfy some of the desired properties, such as: $(A \wedge B, A \wedge B, B \wedge C)$. We can then iteratively combine these functions to generate larger functions that satisfy more of the desired properties. For example, we can combine the first two functions using the OR operator to obtain: $(A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C)$. This function satisfies two of the desired properties: it outputs 1 if and only if exactly two of its inputs are 1, and it outputs 0 if all inputs are 0.

This function satisfies all of the desired properties and can be expressed using only three Boolean operators. The IBC algorithm is able to synthesize this function much more efficiently than the BBC, which would have required evaluating all possible input combinations [31].

Weighted Pruning Iterative Boolean Combination (WPIBC)

Weighted Pruning Iterative Boolean Combination (WPIBC) is an extension of IBC that aims to improve the performance of the model. WPIBC uses a weighted kappa score. The weighted kappa score takes into account the similarity between the predictions of the classifiers, as well as the degree of difficulty in making the prediction. The kappa score is measured using Equation 11.

$$kp = \frac{2 * (TP * TN - FN * FP)}{(TP + FP) * (FP + TN) * (TP + FN) * (FN + TN)} \tag{11}$$

The WPIBC algorithm starts by generating an initial set of classifiers using IBC. Then, for each classifier in the ensemble, the weighted kappa score is calculated using the similarity predictions. The classifiers with the similar weighted kappa scores are pruned from the ensemble steps, then the process is repeated iteratively until no more classifiers can be pruned. The remaining classifiers are then combined using boolean operators to generate the final classifier. The main advantage of WPIBC over previous algorithms is its ability to identify and remove classifiers that are not contributing to the performance of the ensemble. By using the weighted kappa score as a pruning metric, WPIBC can identify classifiers that are making poor predictions and remove them from the ensemble, improving the overall performance of the model [29].

In comparison, the BBC algorithm generates all possible combinations of classifiers, which can lead to a large number of classifiers and computational complexity. On the other hand, IBC generates classifiers iteratively, which can be computationally efficient but may not identify and remove poorly performing classifiers [29]. WPIBC combines the benefits of both approaches by generating classifiers iteratively while also identifying and removing poorly performing classifiers using the weighted kappa score, resulting in a more effective and efficient ensemble model [29].

7.3 Study Setup

This section represents the overall configuration for our study. First, we present the dataset description and feature extraction. Next, we discuss the data labeling and splitting approaches. After that, the evaluation metrics are used to measure accuracy. Then, the algorithms used to build JITBoost are presented.

7.3.1 Datasets Description and Features Extraction

To assess the effectiveness of our approach, we conducted a study on 34 open-source projects from the Apache Foundation. A comprehensive overview of the dataset is provided in Chapter 3, Table 3.1. We also used the 16 features proposed by Kamei et al. [5] and Hoang et al. [14], which are widely used in the JIT-SDP area (e.g., [17] [13] [54] [3]). Table 3.2 presents the 16 features we extracted from the projects. All the data used in this work is made available online[1].

7.3.2 Data Splitting and Preparation

We used two approaches, Cross-Validation (CV) and Time-aware Validation (TV), for training and evaluating the models [14]. In CV, the dataset is divided into a training set (70%) and a testing set (30%), with the testing set kept hidden during model training. This ratio was chosen to compare the data sizes between CV and TV approaches. It was observed that the buggy commits in the dataset occurred in the last 30% of commits based on sorting by commit time (Figure 7.2) [13].

For this research, both CV and TV approaches were employed, following the methodology of Zeng et al. [13]. We used a ten-fold CV to hyper-tune the model parameters, where nine folds were used for training and one fold for validation. These steps are done

[1]https://doi.org/10.5281/zenodo.8206280

only with the training set (70%). This procedure was repeated ten times, as suggested by Zeng et al. [13], to ensure fair testing, and the average of the ten tests was recorded as the CV output. The best model was then evaluated using the previously hidden (30%) testing set. The use of CV is recommended to build more reliable models, prevent overfitting, and enhance generalization to unseen data [95,96]. In this chapter, the testing set was randomly selected without replacement and held out from the training process.

The CV method, as mentioned in Tan et al. [16], does not take into account the temporal aspect of commits when selecting samples for training and testing. To address this limitation, other JIT-SDP studies, such as McIntosh et al. [17], Hoang et al. [14], and Lomio et al. [3], have adopted a time-sensitive (TV) approach for data splitting. The TV approach considers the time-sensitivity of changes, where JIT-SDP models are trained using earlier data to predict buggy commits in later ones.

In our study, we followed a chronological sorting of the data and employed the TV approach for JIT-SDP models. The dataset was divided into training and testing sets using a split point determined by a 70% and 30% ratio, respectively, as shown in Figure 7.2. In addition to the TV approach, unlike previous studies such as McIntosh et al. [17], Hoang et al. [14], and Lomio et al. [3], we also performed a ten-fold CV approach using the early 70% of the data, similar to the approach conducted by Zeng et al. [13]. The number of repetitions in TV approach is one due to its restrictions. We applied the same setting as Hoang et al. [14] and Zeng et al. [13] to our dataset.

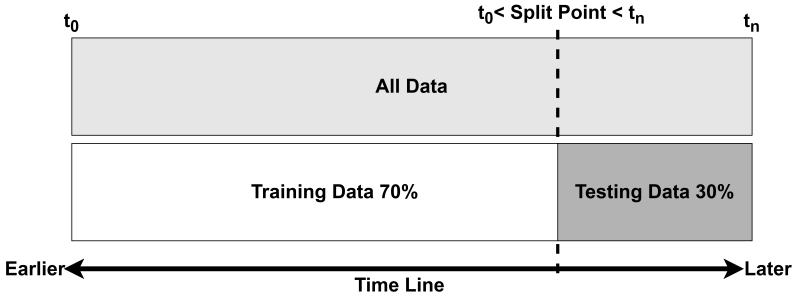


Figure 7.2: Splitting dataset using time-aware validation

7.3.3 Evaluation Metrics

To evaluate the performance of classification models, we use the Area Under the ROC Curve (AUC) [19, 20]. More details are explained in the previous Chapter 3, section 3.3.2.

7.3.4 Algorithms

We propose three JITBoost models (JITBoost-BBC, JITBoost-IBC, JITBoost-WPIBC) that use three different BCC techniques to enhance the JIT-SDP accuracy. We use six traditional ML classification methods and one DL algorithm to create JITBoost (see Figure 7.3).

The traditional ML algorithms are: Naive Bayes (NB), Random Forest (RF), Decision Tree (DT), Support Vector Machine (SVM), Logistics Regression (LR), k-Nearest Neighbors (k-NN). We chose these algorithms because they are used extensively in the field of JIT-SDP (e.g., [4, 5]).

We used the end-to-end deep learning framework (DeepJIT) [13]. Unlike other approaches such as DBN-JIT [14] and CC2Vec [25], which only employ deep learning models to extract and build semantic information and syntactic structure from commit messages and code changes, DeepJIT takes a more comprehensive approach. DeepJIT not only utilizes a deep learning model for extracting semantic information and syntactic structure but also trains the model using a Convolutional Neural Network (CNN) algorithm [13]. Moreover, the DeepJIT is specifically selected for evaluation because it outperforms other DL models, such as DBN-JIT and CC2Vec, and its status as an end-to-end deep learning framework [13].

Figure 7.3 shows the process of combining these algorithms to create JITBoost models using BBC, IBC, and WPIBC. For RQ1 and RQ2, we combine the six traditional algorithms to create JITBoost algorithms. For RQ1, we compare the combined algorithms to each traditional ML algorithm. For RQ2, we compare the combination to Deep JIT. As for

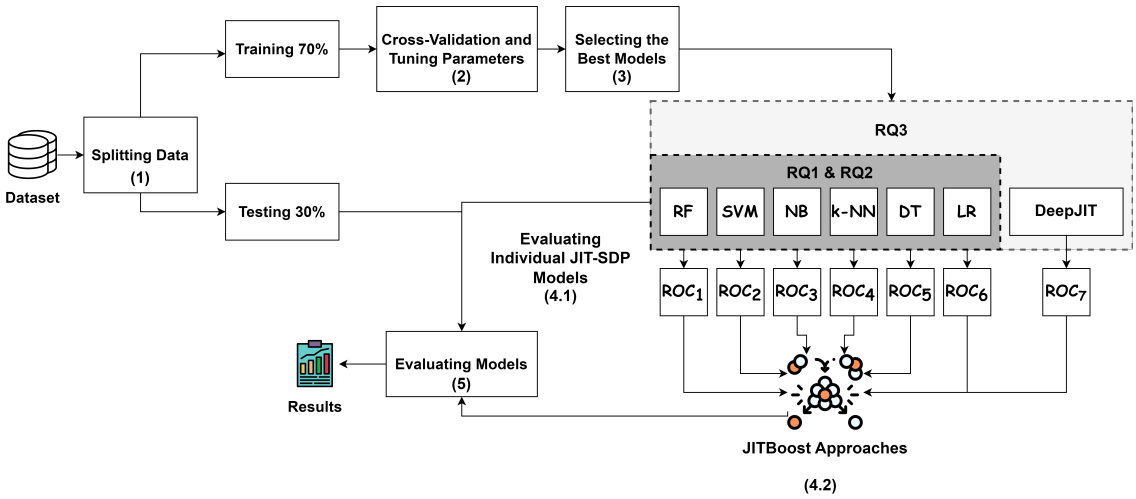


Figure 7.3: The JITBoost Overall Approach

RQ3, we combine the six traditional ML algorithms and DeepJIT and compare that to a combination of only traditional ML.

7.4 Results Analysis and Discussions

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

7.4.1 RQ1: How does the performance of JITBoost algorithms compare to JIT-SDP models that use traditional machine learning algorithms?

Figure 7.4 shows the average AUC results of JITBoost models (JITBoost-BBC, JITBoost-IBC, JITBoost-WPIBC) and that of the six traditional ML methods (SVM, LR, etc.), using both CV and TV data splitting approaches. The results indicate that when using the CV approach, all JITBoost models perform better than the ML models. JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC achieve average AUC values of 0.891, 0.879, and

0.886, respectively, while the best ML model (RF) achieves an average AUC of 0.857. When the TV data splitting approach is used, the performance of all ML models declines even further. The best ML model (RF) achieves an average AUC of 0.776, whereas all JITBoost algorithms still maintain a higher average AUC (the worst result is 0.854 by JITBoost-IBC) compared to the ML models.

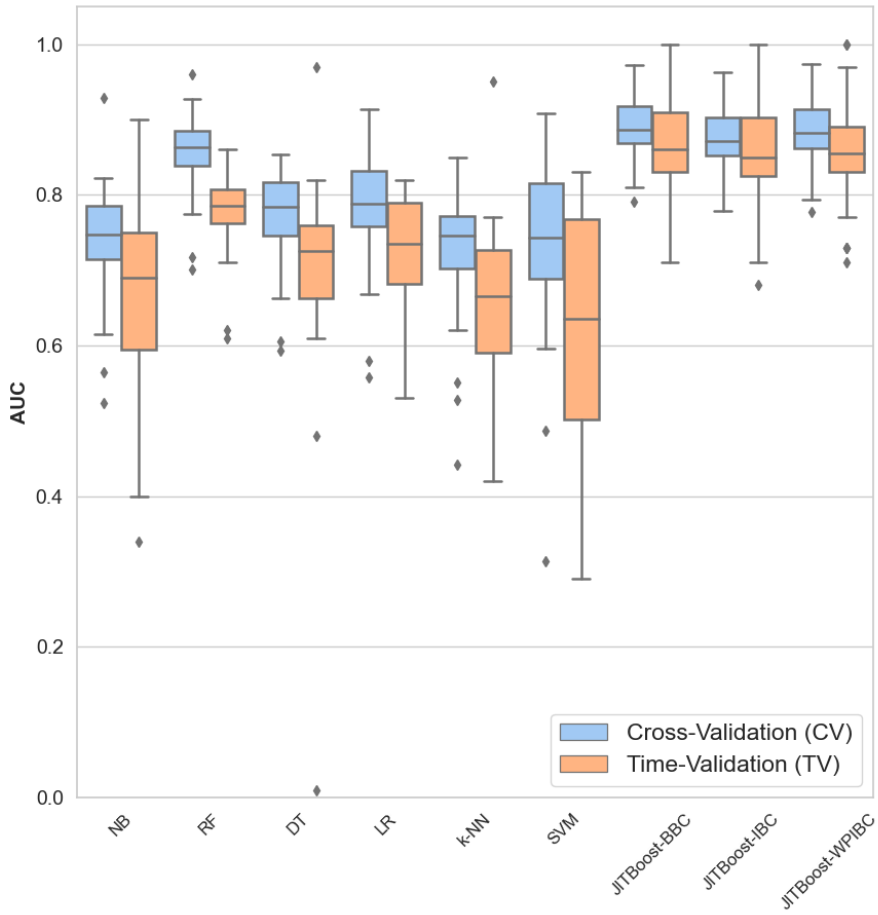


Figure 7.4: Comparison of JITBoost models with ML models.

We used the Mann-Whitney U test [69] [70] to determine the statistical significance of the model's results. The null hypothesis (h_0) assumes that the results of the models are not statistically different, while the alternative hypothesis (h_1) suggests that the model's results are statistically different. The null hypothesis is rejected when the p-value is less than 0.05 (95% confidence interval) [67].

Additionally, we used Cliff's δ effect size to quantify the magnitude of the difference between the two groups. These tests are explained previously in Chapter 3, section 3.3.3.

To compare the performance based on the data splitting approach, the results using CV and TV are examined in Table 7.1. The table presents the average and standard deviation of AUC for all models, the improvement ratio (IM) measured as $(AUC_{CV} - AUC_{TV})/AUC_{CV}$, and the results of the Mann-Whitney U test and Cliff's δ. The findings indicate that JITBoost's performance is slightly improved by 3% with the TV approach compared to CV. However, the p-value suggests that the results of JITBoost models using CV and TV are not statistically different because p-value > 0.05, so we can not reject h_0, although a small effect size is observed.

Table 7.1: The statistical analysis for models with different data splitting approaches (CV and TV).

Classifier	AUC_{CV} **($\mu \pm \sigma$)**	AUC_{TV} **($\mu \pm \sigma$)**	**IM%**	**Cliff's δ**	**p-value**
NB	0.742 ± 0.074	0.669 ± 0.136	10%	0.381	0.007
RF	0.857 ± 0.052	0.776 ± 0.056	10%	0.801	0.000
DT	0.767 ± 0.067	0.696 ± 0.146	9%	0.472	0.001
LR	0.783 ± 0.074	0.721 ± 0.082	8%	0.469	0.001
k-NN	0.724 ± 0.085	0.657 ± 0.102	9%	0.482	0.001
SVM	0.733 ± 0.118	0.626 ± 0.156	15%	0.392	0.006
DeepJIT	0.737 ± 0.101	0.774 ± 0.105	-5%	0.230	0.104
JITBoost-BBC	0.891 ± 0.041	0.863 ± 0.071	3%	0.267	0.058
JITBoost-IBC	0.879 ± 0.044	0.854 ± 0.077	3%	0.266	0.060
JITBoost-WPIBC	0.886 ± 0.045	0.857 ± 0.074	3%	0.262	0.063

On the other hand, the p-value of the ML models is less than 0.05, indicating statistically different results. The effect sizes vary, with RF and k-NN exhibiting large effects and NB, DT, LR, and SVM showing moderate effects. These results suggest that the data splitting approach (i.e., CV and TV) has a limited impact on the performance of JITBoost models, but it significantly affects the performance of ML models, with some models showing large or moderate effect sizes.

Next, we examine the differences in performance based on the classifiers. Specifically,

we compare each JITboost model individually to the other six models, using the same data splitting approach. The same comparison is performed for all other models used in this research question. Table 7.2 presents the performance of the models compared to others, using both the CV and TV approaches. All ML models perform lower than the JITBoost models.

Table 7.2: Effect size by type of classifier

	CV		**TV**	
Classifier	**Cliff's δ**	**p-value**	**Cliff's δ**	**p-value**
NB	-0.408	0.000	-0.017	0.000
RF	-0.453	0.000	-0.413	0.000
DT	-0.325	0.002	-0.050	0.000
LR	-0.320	0.002	-0.341	0.001
k-NN	-0.197	0.009	-0.262	0.012
SVM	-0.524	0.000	-0.552	0.000
DeepJIT	-0.368	0.000	-0.015	0.000
JITBoost-BBC	0.577	0.000	0.531	0.000
JITBoost-IBC	0.473	0.000	0.477	0.000
JITBoost-WPIBC	0.529	0.000	0.494	0.000

Furthermore, all p-values are less than 0.05, indicating statistically significant differences in results for CV and TV data splitting approaches. The effect size is moderate for models such as NB, RF, DT, and LR, while the SVM model a large effect size and small for k-NN with CV approach. Using TV approach, models (NB, DT, and k-NN) have small size effect, while RF and LR has moderate effect size and SVM a large effect size. On the other side, all JITBoost models have p-values less than 0.05, which also indicates statistical differences with large effect sizes ($\delta > 0.474$). In summary, the results indicate that JITBoost-BBC models have statistically different results compared to ML models, with p-values less than 0.05 for CV and TV data splitting approaches.

Finding RQ1: Our findings show that JITBoost models outperform ML models by 18%, 17%, and 17% for JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC, respectively, when using CV. Additionally, when using TV, JITBoost models show even better performance with improvements of 36%, 35%, and 35% for JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC, respectively, compared to ML models. While the choice of data splitting approach had a small impact on JITBoost's performance, it significantly affected ML models, with improvements of at least 8%.

7.4.2 RQ2: How does the performance of JITBoost algorithms compare to a deep learning JIT-SDP algorithm?

This research question compares JITBoost models to DeepJIT models. Figure 7.5 visualizes the results of JITBoost models created using six ML models only and a model created using DeppJIT. It can be seen that all JITBoost Models outperform DeepJIT. The JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC achieve an average AUC of 0.890, 0.880, and 0.890, respectively, with the CV approach compared to an average AUC of 0.737 for DeepJIT. Also, they achieved an average AUC of 0.860, 0.850, and 0.860, respectively, with the TV approach, compared to 0.774 for DeepJIT. The JITBoost models achieve better results and lower variance (see the standard deviation in Table 7.3. In other words, JITBoost lead to a more confident prediction, reducing prediction errors that may be caused by overfitting [19, 100].

We used the Mann-Whitney U test and Cliff's δ size with the results of the JITBoost and DeepJIT models. As shown in Table 7.1, DeepJIT using TV performs better than when using the CV data splitting approach (an improvement of 5%). However, the p-value of DeepJIT is greater than 0.05, so we can not reject the h_0. It means the results are not statistically different with a small effect size. Therefore, we cannot claim that the DeepJIT

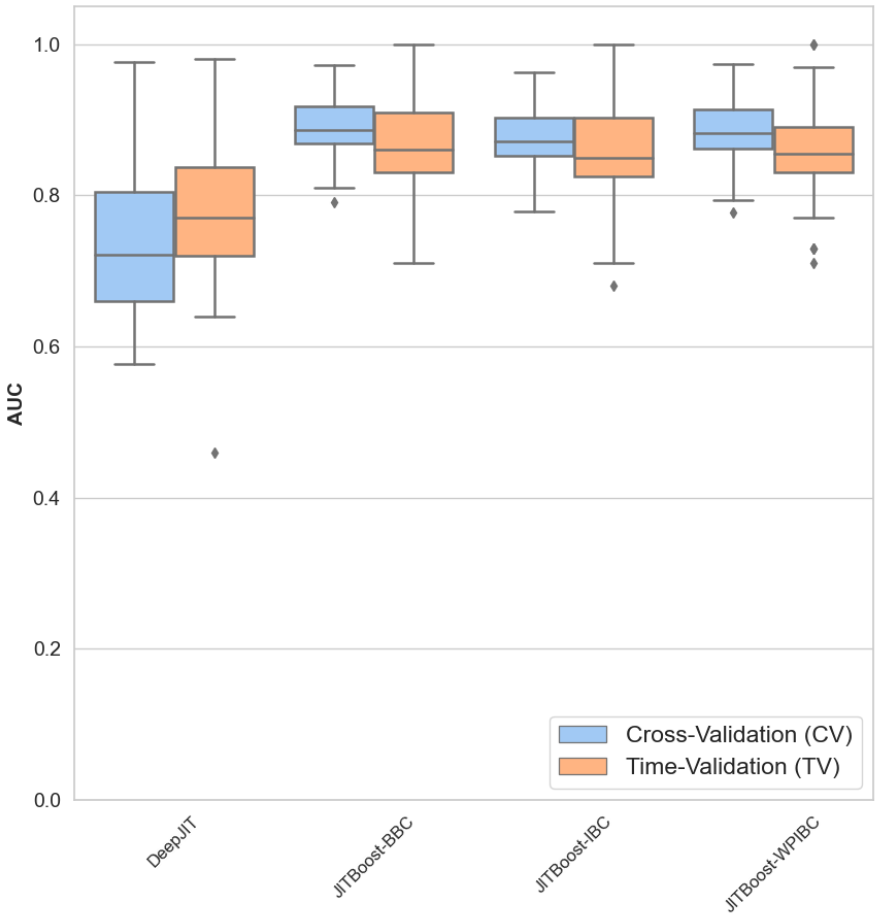


Figure 7.5: Comparison between JITBoost models and DeppJIT models using both data splitting approaches CV and TV.

model is affected by the data splitting approaches. This also applies to JITBoost models.

We statistically analyze the results of JITBoost models with DeepJIT models. Table 7.2 shows the effect size and p-value for each JITBoost model compared to DeepJIT. The p-value of all models is less than 0.05 with CV and TV data splitting approaches, meaning there is a significant difference between JITBoost accuracy and DeepJIT. The effect size of DeepJIT is moderate using CV and small with TV.

Finding RQ2: Our findings show that JITBoost models exhibit superior performance compared to DeepJIT. Using CV, JITBoost-BBC, JITBoost-IBC, and JITBoost-WPIBC outperform the accuracy of DeepJIT by 16%, 15%, and 15%, respectively. Similarly, with the TV approach, JITBoost models surpass DeepJIT model by 10%. It is worth noting that both JITBoost models and DeepJIT show only small effects in performance based on the choice of data splitting approach. These results indicate that JITBoost models consistently outperform DeepJIT, regardless of the data-splitting method employed.

7.4.3 RQ3: How does the combination of traditional JIT-SDP models and deep learning models affect the performance of the JIT-Boost algorithms?

This research question examines the impact of integrating DeepJIT predictions into JITBoost models to enhance their performance. The predictions generated by DeepJIT are combined with ML models for this purpose. However, it is worth noting that previous discussions have already addressed the influence of data-splitting approaches on JITBoost models. Hence, our objective is to measure the extent of improvement achieved by incorporating DeepJIT.

The overall performance of JITBoost models, with and without DeepJIT, is presented in Figure 7.6. We found that including DeepJIT as the seventh classifier into JITBoost does not affect JITBoost-BBC models using both CV and TV techniques. The JITBoost-IBC models improved by 1% when we included DeepJIT with CV, but the performance decreased by -1% using TV. Finally, The JITBoost-WPIBC is improved by 4% and 2% using both CV and TV, respectively, when the DeepJIT predictions are included. The p-value for all models is greater than 0.05, leading us not to reject the null hypothesis (h_0)

and indicating no statistically significant differences in the results. Additionally, the effect size, as indicated by Cliff's δ, is observed to be small.

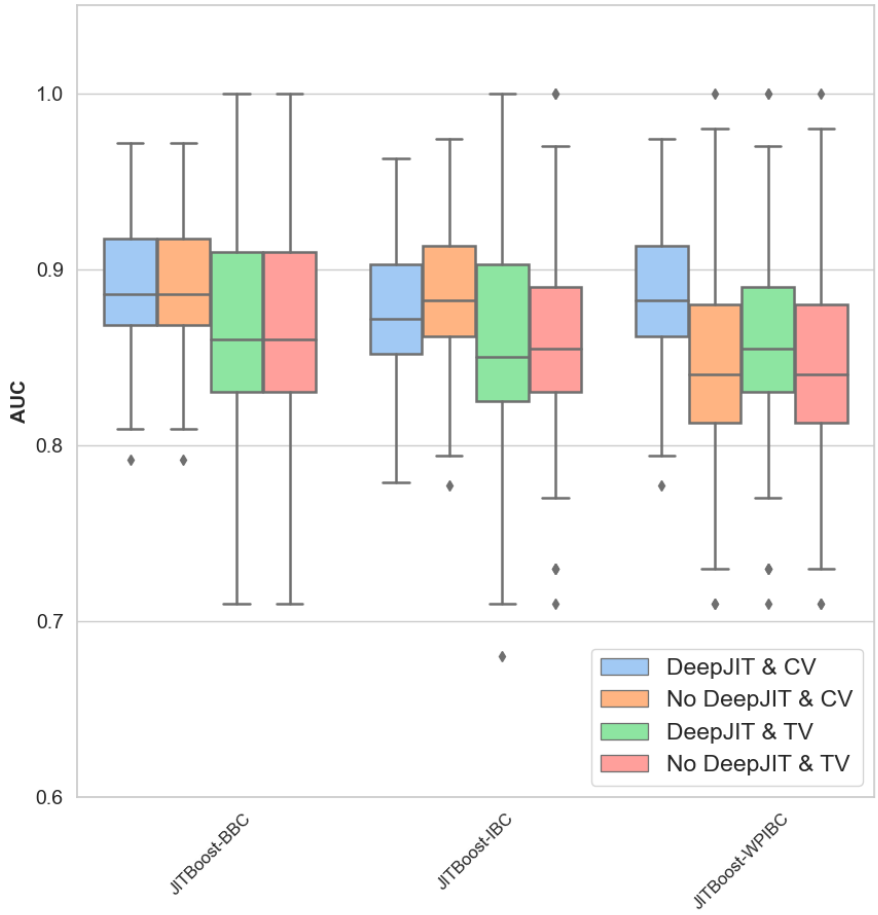


Figure 7.6: Comparison of JITBoost models with DeepJIT to JITBoost models with and without DeepJIT.

As discussed in section 7.2, the JITBoost-BBC algorithm generates all possible combinations of boolean functions leading to the best output, which is not affected when incorporating the DeepJIT model compared to the JITBoost-IBC and JITBoost-WPIBC models. Also, the DeepJIT does not outperform the RF model in the previous results. However, it is important to acknowledge that the BBC algorithm's major drawback is its high processing complexity. On the other hand, the IBC and WPIBC algorithms optimize their processing time by pruning possible cases and accelerating the combination of Boolean functions [30].

This difference in approaches explains the improvement observed when adding DeepJIT as a classifier to JITBoost-WPIBC models because WPIBC drops out the similar vectors from the predictions.

Table 7.3: The effect size of improvement gain after combining all seven models.

Model	IM_{CV}	IM_{TV}	**Cliff's** δ	**p-value**
JITBoost-BBC	0%	0%	0.267	0.058
JITBoost-IBC	1%	-1%	0.266	0.060
JITBoost-WPIBC	4%	2%	0.262	0.063

Moreover, it is important to consider the computational resources required for training DL models. As illustrated in Table 7.4, this study uses different hardware configurations, including 2 CPUs and 3 GPUs, for tuning, training, and testing the DeepJIT model. The optimal hardware configuration in this study was identified as C_2 and G_2. These powerful hardware setups were specifically employed to accelerate the processing time of the DL model. Conversely, a simpler hardware configuration, such as C_1, was sufficient for the ML model.

Table 7.4: Hardware specifications utilized for deep learning model.

ID	**Hardware**	**Specifications**	**Release Date**
C_1	CPU Ryzen 9 5900X	12 cores, 3.7 GHz, 32GB RAM-DDR4	Nov-2020
C_2	CPU Intel Xeon	48 cores, 3.8 GHz, 64GB RAM-DDR4L	Jun-2019
G_1	GPU GTX 970	1664 CUDA cores, 4 GB GDDR5 RAM	Sep-2014
G_2	GPU RTX 8000	4608 CUDA cores, 48 GB GDDR6 RAM	Aug-2018
G_3	GPU A100 NVIDIA	6912 CUDA cores, 40 GB HBM2 RAM	May-2020

Finding RQ3: We found that JITBoost models outperform DeepJIT when compared to other ML models. Even when considering DeepJIT as a seventh classifier alongside the JITBoost models, the observed improvement is not statistically significant. It is important to note that training the DL model like DeepJIT requires substantial computing resources and time. While JITBoost models demonstrate better performance without such resource-intensive training requirements, the difference in performance between JITBoost and DeepJIT is not statistically significant.

7.5 Threats to Validity

In this section, the threats to the validity of our results and recommendations are discussed.

Internal Validity: Internal validity threats refer to factors that could potentially influence our findings. One potential threat is the choice of algorithms. To address this, we employed robust algorithms that have a strong track record in various classification tasks and are widely used in research across different fields. Another concern relates to the datasets we used. Although we conducted experiments on 34 different Java Apache projects, incorporating additional datasets comprising different programming languages would enhance the generalizability of our results. Furthermore, comparing cross-validation with time validation might be impacted by the number of tests conducted. Time validation has certain limitations that prevent multiple tests similar to cross-validation. Additionally, the choice of the number of folds in cross-validation can also have an influence.

External Validity: External validity pertains to the extent to which our findings can be generalized. Our experiments encompassed 34 datasets from diverse software projects. However, it is important to note that we do not make claims regarding the generalizability of our results to all projects, especially industrial or proprietary systems to which we did

not have access.

7.6 Replication Package

All the data, scripts, and results discussed in this chapter are available on Zenodo:

https://doi.org/10.5281/zenodo.8206280

7.7 Conclusion

In this study, we examined the effectiveness of Boolean Combination Classifiers (BCC) for Just-In-Time Software Defects Prediction (JIT-SDP) models. Our experiments involved three BCC algorithms: Brute-force Boolean Combination (BBC), Iterative Boolean Combination (IBC), and Weighted Pruning Iterative Boolean Combination (WPIBC), using 34 datasets. We compared their performance against state-of-the-art JIT-SDP models that utilize machine learning (ML) and deep learning (DL) approaches.

Our findings revealed that, in our specific case, DeepJIT was unable to outperform certain ML models, such as Random Forest (RF). However, when combining ML models within the JITBoost framework, the JITBoost algorithms outperformed all state-of-the-art JIT-SDP models employing ML and DL classifiers. Notably, including DL models alongside JITBoost algorithms enhanced the performance of the JITBoost-WPIBC algorithm. Furthermore, we observed that the choice of ML models significantly impacted data-splitting approaches, such as cross-validation and time-aware validation. In contrast, DL and JITBoost models exhibited minimal effects in this regard.

Chapter 8

Conclusion and Future Work

Software maintenance encompasses a range of tasks, including fixing bugs, adjusting a system to evolving conditions, and integrating new functionalities in response to customer needs. Recently, the adoption of machine learning and deep learning to aid software maintenance tasks has seen considerable growth. One particular influential area of research centers on Just-in-Time Software Defect Prediction (JIT-SDP) techniques, the topic of this thesis. These techniques allow for predicting bugs at the level of code commits before integrating the changes into the central code repository. Through the analysis of commits, which signal the completion of particular tasks, it becomes feasible to detect and rectify undesired code changes that may introduce bugs. In addition, JIT-SDP techniques offer the benefit of delivering immediate feedback to developers to address problems in the submitted code so they can change it immediately. Furthermore, JIT-SDP techniques can be readily integrated into the DevOps workflow, as they can be incorporated as a part of a code versioning system.

8.1 Research Contributions

In this thesis, we introduced three novel contributions that extend the frontiers of JIT-SDP and present promising opportunities for enhancing software maintenance and code quality. The first contribution is ClusterCommit, an automated approach that groups projects based on their shared dependencies. The main idea is to increase the size of the training data, allowing JIT-SDP models to learn from different patterns of commits originating from similar projects. ClusterCommit achieves this by selecting projects with the same dependencies and functional requirements. ClusterCommit was designed, implemented, and evaluated with six ML and three DL models. For ML models, ClusterCommit is affected by the cluster size and type of classifiers. For instance, RF and SVM only improved with large clusters. In contrast, the single project is more efficient for all ML models when the cluster size is small. For the DL models, the ClusterCommit approach increases the performance of all classifiers with both small and large cluster sizes. These results indicate that ClusterCommit is more effective when used with SVM, RF, and DL algorithms.

The second contribution is a novel JIT-SDP method that is based on one-class classification, where we train a model only using the majority class (in our case, the normal commits). By doing so, we do not need to use the minor class instances (i.e., buggy commits), which occur less frequently than normal commits. Also, the distribution of buggy commits over the timeline shows limited or unavailable buggy data. In this scenario, it is often challenging to use binary classifiers. We showed through extensive experimentation that the OCC-JIT-SDP approach outperforms binary classifiers when the imbalance ratio of normal commits to buggy commits is medium to high.

Finally, the third contribution is the JITBoost framework. The main idea is to combine multiple JIT-SPD models using Boolean Combination of Classifiers (BCC). The JITBoost framework increases the performance of JIT-SDP models by fusing multiple traditional ML and DL classifiers in the ROC curve. The results of JITBoost, when applied to multiple

ML/DL algorithms, show better performance than using these algorithms individually.

All these contributions have been implemented in open-source tools with the objective of fostering open science. The links to these tools are as follows:

(1) **ClusterCommit**[1]: is a tool that uses graph database (Neo4j), Maven dependence manager for Java, sklearn, and Keras (TensorFlow interface).

(2) **OCC-JIT-SPD**[2]: a tool using sklearn and PyOD (Outlier Detection and Anomaly Detection) library.

(3) **JITBoost framework**[3]: is a framework containing the implementation of three BCC algorithms: Brute-force Boolean Combination (BBC), Iterative Boolean Combination (IBC), and Weighted Pruning Iterative Boolean Combination (WPIBC).

To evaluate our research, we used a substantial dataset derived from 34 open-source projects, with a cumulative total of 259k commits sourced from the Apache Foundation[4]. This extensive dataset allowed us to assess the effectiveness of our approaches in a diverse and representative open-source context, further affirming their utility in large-scale software maintenance scenarios.

8.2 Opportunities for Further Research

8.2.1 Exploring additional features for clustering projects

Currently, ClusterCommit relies solely on the Maven dependency manager to identify projects that are related to each other. In the future, we would like to add more features that would improve the clustering algorithms. Examples of features include bug reports,

[1]https://github.com/wahabhamoulhadj/OpenCommitBeta
[2]https://github.com/wahabhamoulhadj/jit-occ
[3]https://github.com/wahabhamoulhadj/bcml
[4]https://www.apache.org/

developments teams, functional requirements, etc. This way, we can create strong clusters that would improve the accuracy of clusterCommit.

8.2.2 Applying the proposed techniques to cross-projects

Another area for future exploration is to apply the proposed methods to cross-projects. The idea is to train JIT-SDP models on multiple projects to predict defects in other projects. For example, we can build models using OCC-JIT-SDP on a selected set of projects and examine if the trained model can be used to predict buggy commits in a completely separate project. However, a significant challenge with this cross-project approach is the difference between training and testing datasets, known as heterogeneity [22]. Also, the data imbalance problem may become more prevalent when using a cross-project approach. We clearly need to conduct further studies to assess how the techniques proposed in this research can be improved to address the data heterogeneity and imbalance issues.

8.2.3 Experimenting with a diverse set of systems

One limitation of our research is related to the selected projects that are used for evaluation. These projects are mainly developed in Java for the Apache foundation. They share common libraries and programming conventions. We should experiment with a more diverse set of systems, such as those written in other programming languages, industrial systems, microservice-based systems, mobile applications, etc., to claim the generalizability of the findings.

8.2.4 Applications to Defect Localization and Recommendation

Defect localization is another research area that has been attracting considerable attention in the last decade [18]. After identifying a buggy commit, there is a need to locate parts of the code that caused the bug. Yan et al. [18] proposed a two-phase approach for

JIT-SDP bug localization. They used the RA-SZZ algorithm to label the dataset, which can find the faulty line locations later during model inference. However, the data distribution in the timeline directly affects the accuracy of JIT-SDP predictions, creating higher false positives [3, 17]. To address this, we believe that we should invest in explainable machine techniques to identify the causes of bugs. To do so, we believe that using simpler ML algorithms can make this task easier than using deep learning. When experimenting with JITBoost, we found that combining simple ML algorithms yields better performance than DL algorithms. In other words, one future direction would be to design explainable ML methods that can reveal the causes of bugs, without compromising accuracy.

The ability to locate bugs opens the door to designing techniques for recommending fixes. We can train models that learn from previous fixes to address a buggy commit. One promising area of research would be to use Large Language Models (LLM) [101], which are instrumental in bug resolution. Various LLM models including Microsoft Copilot[5], Llama [6], and different versions of ChatGPT [7], can generate code and offer suggestions to streamline bug-fixing and reduce the efforts of developers.

8.3 Closing Remarks

This research has ventured into the ever-evolving landscape of JIT-SDP, an important activity in software maintenance and evolution. We have proposed three innovative contributions, ClusterCommit, OCC-JIT-SDP, and JITBoost, that we believe represent a step forward in enhancing software maintenance and code quality through the lens of JIT-SDP. We hope this research will further encourage software practitioners to adopt JIT-SDP techniques and spark fresh insights that inspire fellow researchers to explore new horizons in this important discipline.

[5]https://developer.microsoft.com/en-us/copilot
[6]https://ai.meta.com/blog/code-llama-large-language-model-coding/
[7]https://openai.com/blog/chatgpt

Bibliography

[1] R. Pressman, *Software Engineering: A Practitioner's Approach*. USA: McGraw-Hill, Inc., 2009.

[2] P. Grubb and A. A. Takang, *Software Maintenance: Concepts and Practice*. USA: World Scientific Publishing Co., Inc., 2003.

[3] F. Lomio, L. Pascarella, F. Palomba, and V. Lenarduzzi, "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)*, vol. 1, pp. 1–10, 05 2022.

[4] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR'18)*, pp. 153–164, 2018.

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

[6] G. Catolino, D. Di Nucci, and F. Ferrucci, "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, 2019.

[7] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen, "Improving defect prediction with deep forest," *Information and Software Technology*, vol. 114, pp. 204 – 216, 2019.

[8] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), p. 297–308, Association for Computing Machinery, 2016.

[9] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 318–328, 2017.

[10] C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, vol. 1, pp. 369–379, 2021.

[11] W. Zheng, T. Shen, X. Chen, and P. Deng, "Interpretability application of the just-in-time software defect prediction model," *Journal of Systems and Software*, vol. 188, p. 111245, 2022.

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

[13] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: How far are we?," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, (New York, NY, USA), p. 427–438, Association for Computing Machinery, 2021.

[14] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "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, 2019.

[15] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1253–1269, 2019.

[16] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering Volume2*, ICSE '15, p. 99–108, IEEE Press, 2015.

[17] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, pp. 412–428, May 2018.

[18] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[19] C. M. Bishop, *Pattern recognition and machine learning*. Information science and statistics, New York, NY: Springer, 2006.

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

[21] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning: The mit press, 2016, 800 pp, isbn: 0262035618," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, 2018.

[22] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 309–320, 2016.

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

[24] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 17–26, 2015.

[25] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, vol. 1, pp. 518–529, 2020.

[26] C. Bellinger, S. Sharma, and N. Japkowicz, "One-class versus binary classification: Which and when?," in *2012 11th International Conference on Machine Learning and Applications*, vol. 2, pp. 102–106, 2012.

[27] X. Deng, W. Li, X. Liu, Q. Guo, and S. Newsam, "One-class remote sensing classification: one-class vs. binary classifiers," *International Journal of Remote Sensing*, vol. 39, no. 6, pp. 1890–1910, 2018.

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

[29] M. S. Islam, W. Khreich, and A. Hamou-Lhadj, "Anomaly detection techniques based on kappa-pruned ensembles," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 212–229, 2018.

[30] W. Khreich, E. Granger, A. Miri, and R. Sabourin, "Boolean combination of classifiers in the roc space," in *2010 20th International Conference on Pattern Recognition*, pp. 4299–4303, 2010.

[31] W. Khreich, E. Granger, A. Miri, and R. Sabourin, "Iterative boolean combination of classifiers in the roc space: An application to anomaly detection with hmms," *Pattern Recognition*, vol. 43, no. 8, pp. 2732–2752, 2010.

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

[33] W. Khreich, E. Granger, A. Miri, and R. Sabourin, "A survey of techniques for incremental learning of hmm parameters," *Information Sciences*, vol. 197, pp. 105–130, 2012.

[34] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 78–88, 2009.

[35] H. Zhang and X. Zhang, "Comments on "data mining static code attributes to learn defect predictors"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 635–637, 2007.

[36] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, (New York, NY, USA), p. 13–23, Association for Computing Machinery, 2008.

[37] W. Rhmann, B. Pandey, G. Ansari, and D. Pandey, "Software fault prediction based on change metrics using hybrid algorithms: An empirical study," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 4, pp. 419–424, 2020. Emerging Software Systems.

[38] K. E. Bennin, K. Toda, Y. Kamei, J. Keung, A. Monden, and N. Ubayashi, "Empirical evaluation of cross-release effort-aware defect prediction models," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 214–221, 2016.

[39] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, p. 46–57, IEEE Press, 2019.

[40] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206 – 220, 2017.

[41] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, "500+ times faster than deep learning: (a case study exploring faster methods for text mining stackoverflow)," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 554–563, 2018.

[42] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*,

ESEC/FSE 2017, (New York, NY, USA), p. 49–60, Association for Computing Machinery, 2017.

[43] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 763–773, Association for Computing Machinery, 2017.

[44] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "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*, FSE 2016, (New York, NY, USA), p. 157–168, Association for Computing Machinery, 2016.

[45] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 72–83, Association for Computing Machinery, 2017.

[46] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.

[47] M. Kiehn, X. Pan, and F. Camci, "Empirical study in using version histories for change risk classification," in *Proceedings of the 16th International Conference on Mining Software Repositories (MSR'19)*, p. 58–62, 2019.

[48] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22 – 36, 2019.

[49] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "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, 2014.

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

[51] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22 – 36, 2019.

[52] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[53] R. van Dinter, C. Catal, G. Giray, and B. Tekinerdogan, "Just-in-time defect prediction for mobile applications: using shallow or deep learning?," *Software Quality Journal*, vol. 31, pp. 1281–1302, 2023.

[54] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1559–1586, 2021.

[55] E. C. Neto, D. A. da Costa, and U. Kulesza, "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, 2018.

[56] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, (New York, NY, USA), p. 1–5, Association for Computing Machinery, 2005.

[57] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 81–90, 2006.

[58] D. Silva and M. T. Valente, "Refdiff: Detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 269–279, 2017.

[59] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[60] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC genomics*, vol. 21, no. 1, pp. 1–13, 2020.

[61] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006. ROC Analysis in Pattern Recognition.

[62] J. Yao and M. Shepperd, "Assessing software defection prediction performance: Why using the matthews correlation coefficient matters," in *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, (New York, NY, USA), p. 120–129, Association for Computing Machinery, 2020.

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

[64] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, 2014.

[65] Q. Zhu, "On the performance of matthews correlation coefficient (mcc) for imbalanced dataset," *Pattern Recognition Letters*, vol. 136, pp. 71–80, 2020.

[66] J. Huang and C. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[67] G. D. Ruxton, "The unequal variance t-test is an underused alternative to Student's t-test and the Mann–Whitney U test," *Behavioral Ecology*, vol. 17, pp. 688–690, 05 2006.

[68] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff´s delta calculator: A non-parametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2010.

[69] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[70] A. Bucchianico, "Combinatorics, computer algebra and the wilcoxon-mann-whitney test," *Journal of Statistical Planning and Inference*, vol. 79, no. 2, pp. 349–364, 1999.

[71] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "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*, 2006.

[72] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, no. 3, p. 494–509, 1993.

[73] J. Cohen, "A power primer," *Psychological Bulletin*, vol. 112, pp. 155–159, 1992.

[74] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?," *Empirical Softw. Engg.*, vol. 23, p. 384–417, Feb. 2018.

[75] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, p. 036106, 2007.

[76] D. Knoke and S. Yang, *Social network analysis*, vol. 154. Sage Publications, 2019.

[77] Tin Kam Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, pp. 278–282, 1995.

[78] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: an overview," *Bioinformatics*, vol. 16, pp. 412–424, 05 2000.

[79] A. M. Shehab, A. Hamou-Lhadj, and L. Alawneh, "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)*, vol. 1, pp. 1–5, 2022.

[80] M. Tsvetovat and A. Kouznetsov, *Social Network Analysis for Startups: Finding connections on the social web*. " O'Reilly Media, Inc.", 2011.

[81] D. Lin, C. Tantithamthavorn, and A. E. Hassan, "The impact of data merging on the interpretation of cross-project just-in-time defect models," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[82] S. van Berkum, S. van Megen, M. Savelkoul, P. Weterman, and F. Frasincar, "Fine-tuning for cross-domain aspect-based sentiment classification," in *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pp. 524–531, 2021.

[83] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade: Second Edition*, pp. 421–436, Springer, 2012.

[84] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the Support of a High-Dimensional Distribution," *Neural Computation*, vol. 13, pp. 1443–1471, 07 2001.

[85] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[86] S. Hariri, M. C. Kind, and R. J. Brunner, "Extended isolation forest," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1479–1489, 2021.

[87] M. Yousef, S. Jung, L. C. Showe, and M. K. Showe, "Learning from positive examples when the negative class is undetermined-microrna gene identification," *Algorithms for molecular biology*, vol. 3, no. 1, pp. 1–9, 2008.

[88] B. Butcher and B. J. Smith, "Feature engineering and selection: A practical approach for predictive models," *The American Statistician*, vol. 74, no. 3, pp. 308–309, 2020.

[89] P. E. Hart, D. G. Stork, and R. O. Duda, *Pattern classification*. Wiley Hoboken, 2000.

[90] J. Wang and J.-D. Zucker, "Solving multiple-instance problem: A lazy learning approach," 2000.

[91] L. Jiang, Z. Cai, D. Wang, and S. Jiang, "Survey of improving k-nearest-neighbor for classification," in *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, vol. 1, pp. 679–683, 2007.

[92] Y. Zhao, Z. Nasrullah, and Z. Li, "Pyod: A python toolbox for scalable outlier detection," *Journal of Machine Learning Research*, vol. 20, no. 96, pp. 1–7, 2019.

[93] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*, pp. 413–422, 2008.

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

[95] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[96] M. Feurer and F. Hutter, *Hyperparameter Optimization*, pp. 3–33. Cham: Springer International Publishing, 2019.

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

[98] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014. 40th-year commemorative issue.

[99] M. Barreno, A. Cardenas, and J. D. Tygar, "Optimal roc curve for a combination of classifiers," *Advances in Neural Information Processing Systems*, vol. 20, 2007.

[100] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[101] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Thirty-seventh Conference on Neural Information Processing Systems*, pp. 1–20, 2023.

Appendix A

Table A.1: 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 A.2: 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.

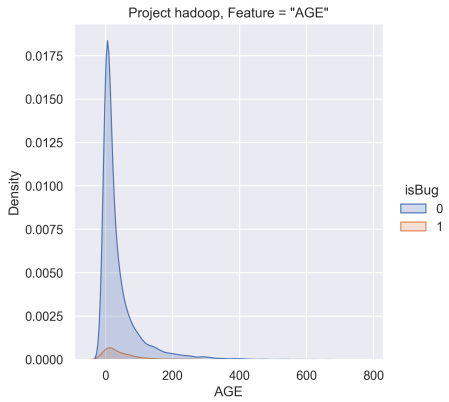
		SVM					RF					k-NN				
Project name	**IR**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**
Drill	1.4	0.756	0.803	0.722	**0.810**	0.597	0.755	0.779	0.500	0.779	0.743	0.788	0.782	0.797	0.788	0.745
Flume	1.7	**0.873**	0.834	0.706	0.832	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 A.3: The relationship between JIT-SDP model accuracy using AUC and the data imbalance ratio (IR) with time aware-validation. NB stands for No balancing, OS stands for Over-sampling, US stands for Under-sampling, and SMOTE stands for Synthetic Minority Oversampling Technique.

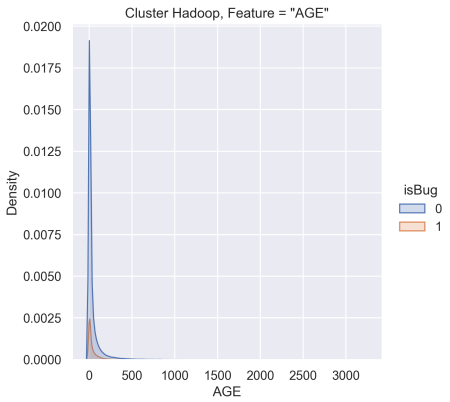
		SVM					**RF**					**k-NN**				
Project name	**IR**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**
Drill	1.4	0.651	0.622	0.599	0.654	0.604	0.789	0.788	**0.789**	0.654	0.683	0.706	0.710	0.704	0.711	0.616
Flume	1.7	0.557	0.572	0.699	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 A.4: The relationship between JIT-SDP model accuracy using F1-score and the data imbalance ratio (IR) with time aware-validation. NB stands for No balancing, OS stands for Over-sampling, US stands for Under-sampling, and SMOTE stands for Synthetic Minority Oversampling Technique.

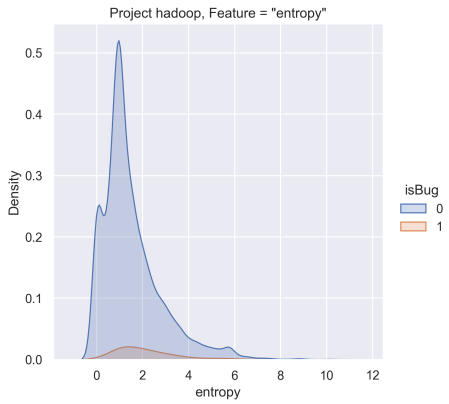
		SVM					**RF**					**k-NN**				
Project name	**IR**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**	**NB**	**OS**	**SMOTE**	**US**	**OCC**
Drill	1.4	0.648	0.637	0.629	0.602	0.624	0.720	**0.727**	0.724	0.602	0.654	0.680	0.685	0.680	0.681	0.630
Flume	1.7	0.545	0.400	0.671	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***



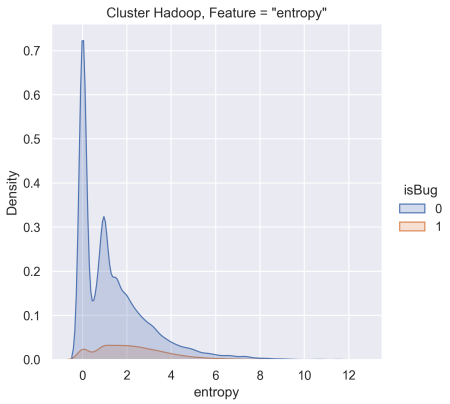
(a) Single-Project (Age)



(b) Cluster-based (Age)

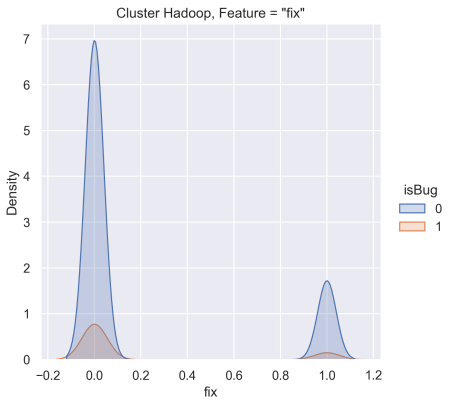


(c) Single-Project (Entropy)



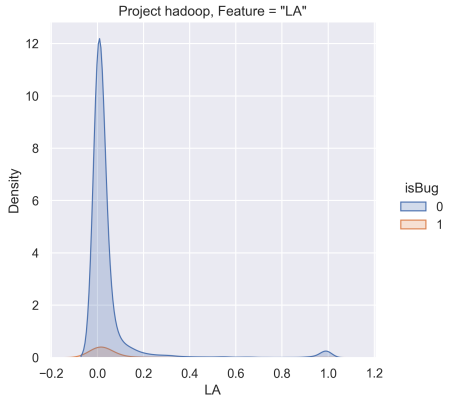
(d) Cluster-based (Entropy)

(e) Single-Project (Fix)

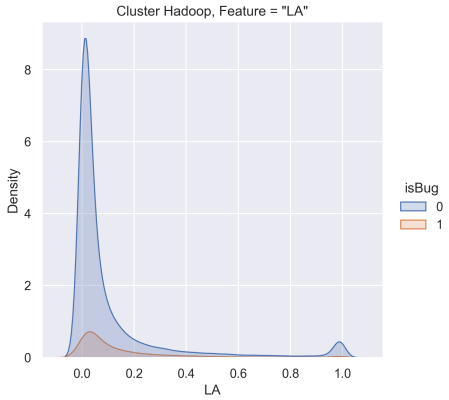


(f) Cluster-based (Fix)

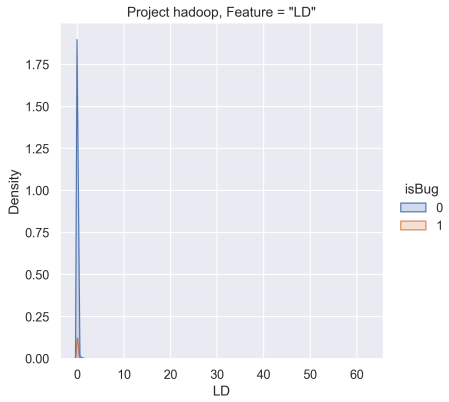
Figure A.1: Example figures of three features (Age, Entropy, and Fix) distribution of (Hadoop set 01)



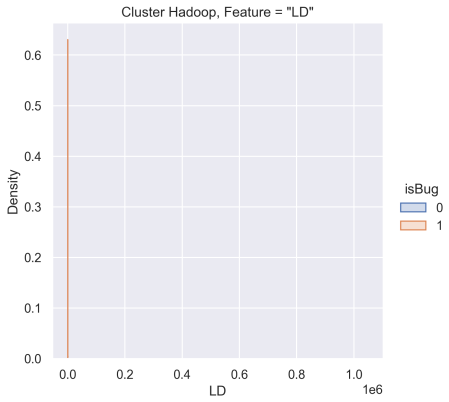
(a) Single-Project (LA)



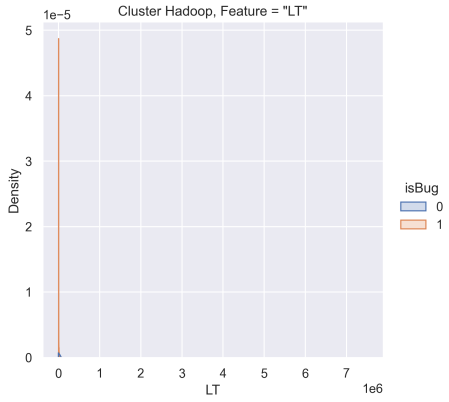
(b) Cluster-based (LA)



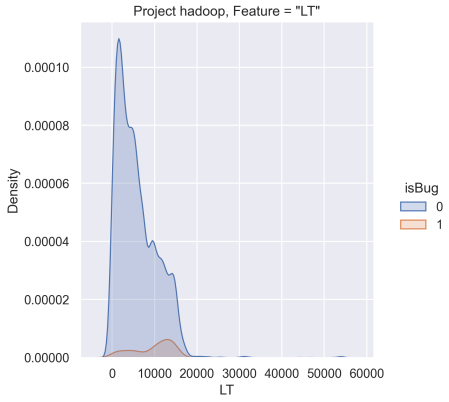
(c) Single-Project (LD)



(d) Cluster-based (LD)

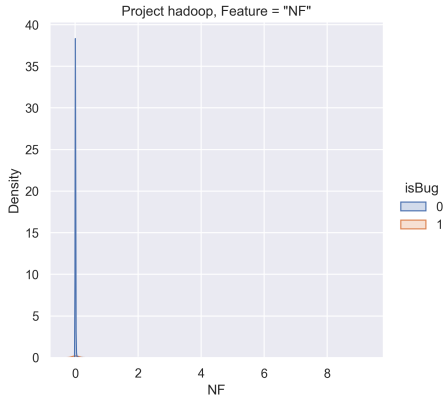


(e) Single-Project (LT)

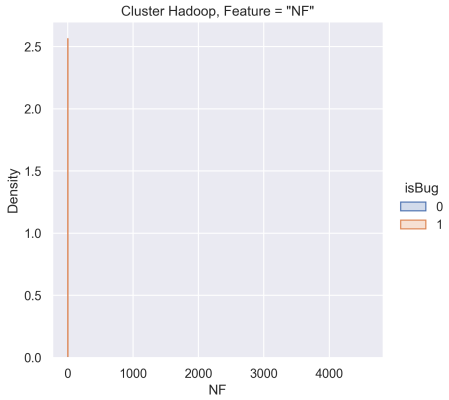


(f) Cluster-based (LT)

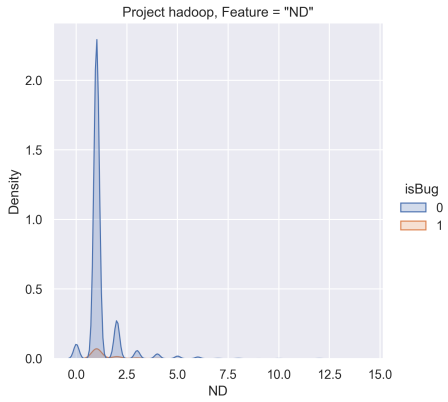
Figure A.2: Example figures of three features (LA, LD, and LT) distribution of (Hadoop set 02)



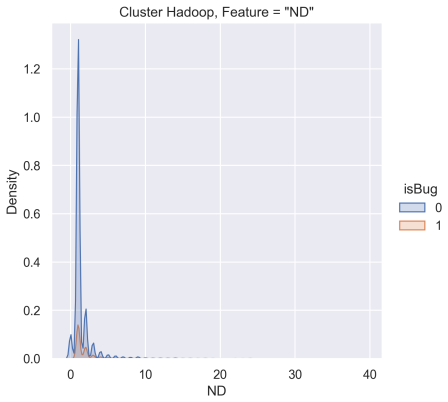
(a) Single-Project (NF)



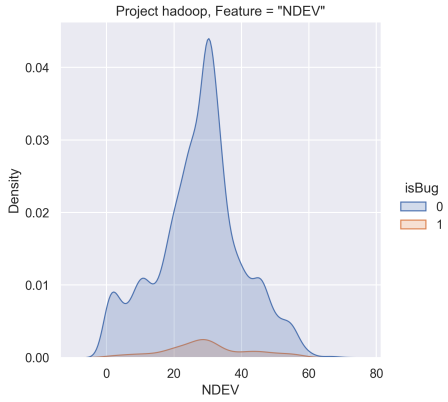
(b) Cluster-based (NF)



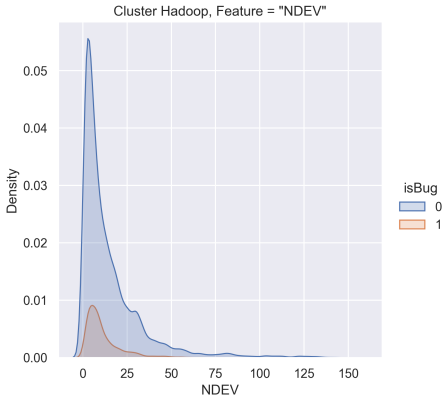
(c) Single-Project (ND)



(d) Cluster-based (ND)

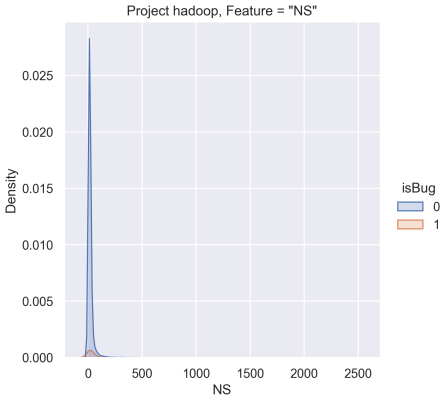


(e) Single-Project (NDEV)



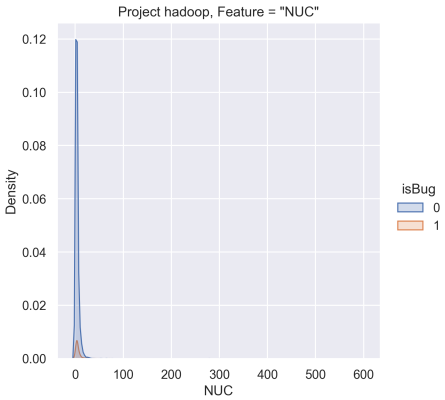
(f) Cluster-based (NDEV)

Figure A.3: Example figures of three features (NF,ND, and NDEV) distribution of (Hadoop set 03)

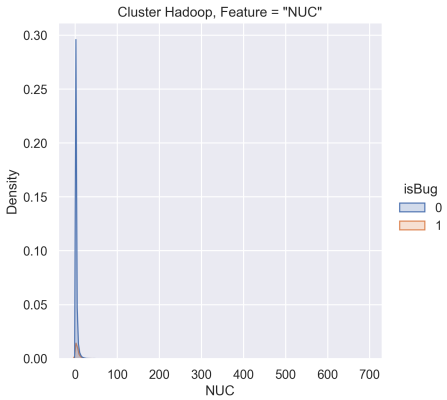


(a) Single-Project (NS)

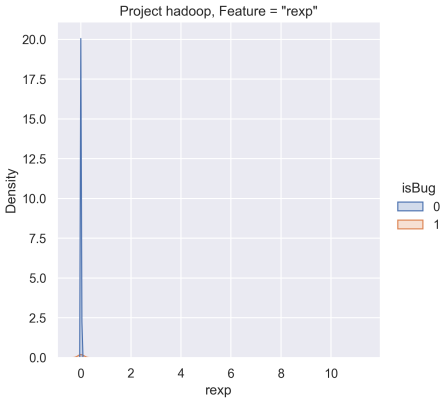
(b) Cluster-based (NS)



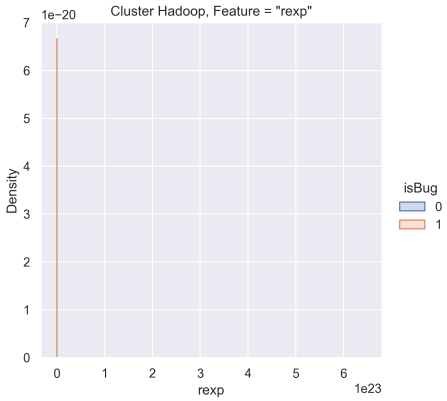
(c) Single-Project (NUC)



(d) Cluster-based (NUC)

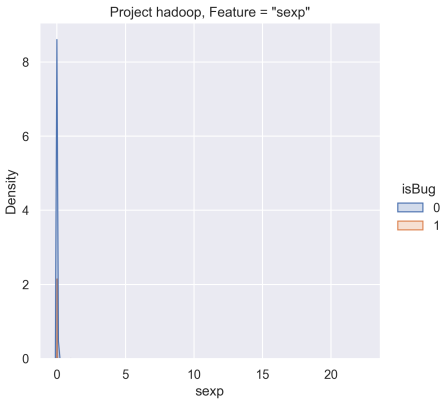


(e) Single-Project (REXP)

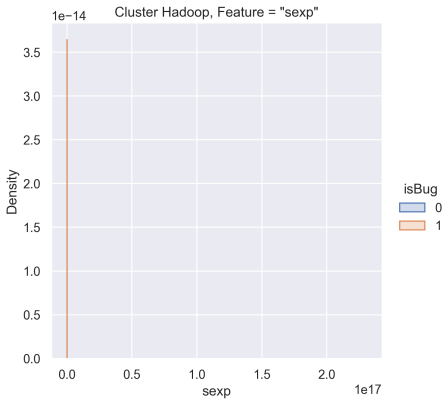


(f) Cluster-based (REXP)

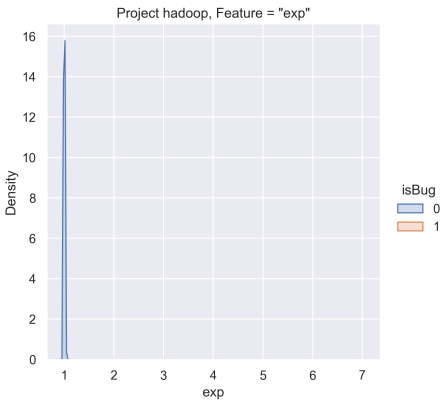
Figure A.4: Example figures of three features (NS, NUS, and REXP) distribution of (Hadoop set 04)



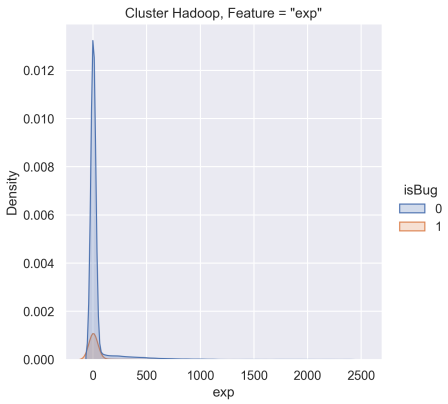
(a) Single-Project (SEXP)



(b) Cluster-based (SEXP)

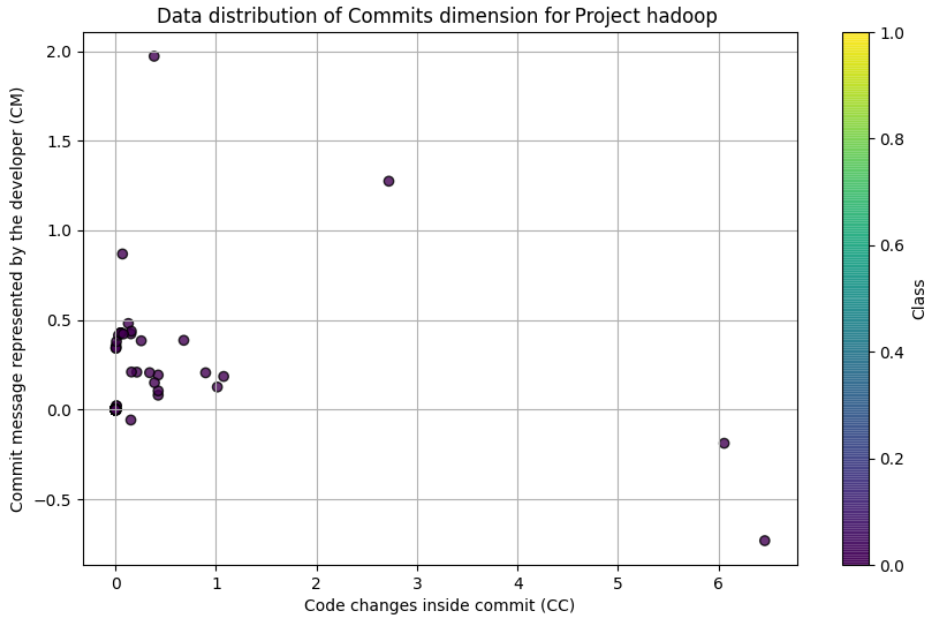


(c) Single-Project (EXP)

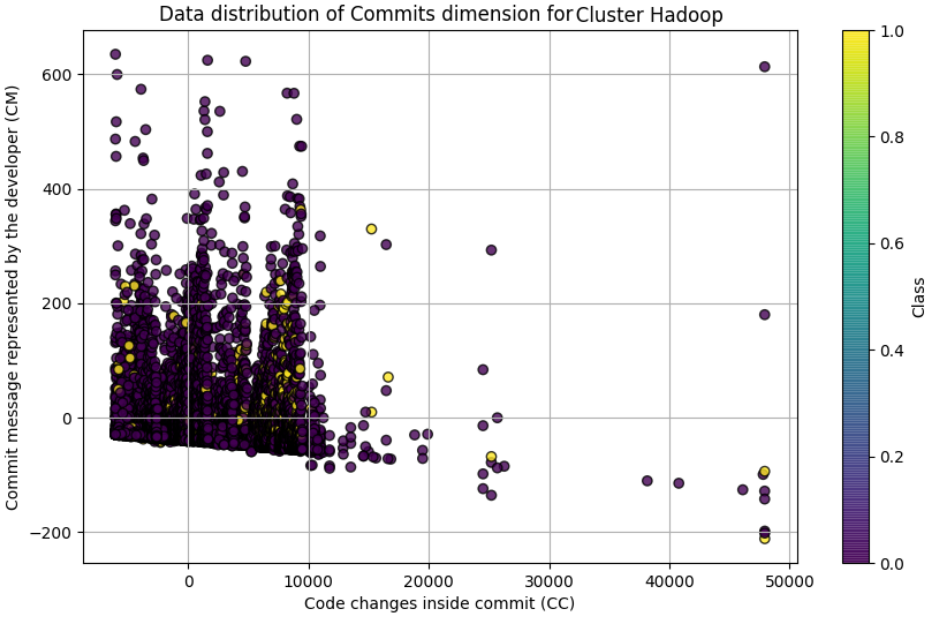


(d) Cluster-based (EXP)

Figure A.5: Example figures of two features (SEXP and EXP) distribution of (Hadoop set 05)



(a) Single-Project (CM and CC) features



(b) Cluster-based (CM and CC) features

Figure A.6: Example figures of syntactic and semantic features (CM and CC) distribution of (Hadoop set 06)