DEEP Q-NETWORK FOR JUST-IN-TIME SOFTWARE DEFECT PREDICTION

AHMAD MUHAIMIN BIN ISMAIL

FACULTY OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY
UNIVERSITI MALAYA
KUALA LUMPUR

2023

DEEP Q-NETWORK FOR JUST-IN-TIME SOFTWARE DEFECT PREDICTION

AHMAD MUHAIMIN BIN ISMAIL

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2023

UNIVERSITI MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: Ahmad Muhaimin Bin Ismail

Matric No: 17202117/1 /WVA190005

Name of Degree: Doctor of Philosophy

Title: Deep Q-Network for Just-in-Time Software Defect Prediction

Field of Study: Software Quality

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This Work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date:

Subscribed and solemnly declared before,

Witness's Signature Date:

Name:

Designation:

DEEP Q-NETWORK FOR JUST-IN-TIME SOFTWARE DEFECT PREDICTION

ABSTRACT

Mitigating software defects at code level at early stages allows for long-term maintenance of software quality. According to IBM's report, the cost of fixing an error rises exponentially as software moves forward in software development lifecycle. The cost to fix defects after software release is up to 15 times more than the fixing cost for defects uncovered during the initial software development phase. Quality assurance relies on code reviews to identify and fix software defects. Apart from code optimization and formal inspection, software defect prediction makes use of limited resources as part of the code review process to identify the most cost-effective way to discover defects. A software defect prediction approach is conducted at three levels of granularity: modules, files, and changes. Change level prediction, also referred to as Just-in-Time software defect prediction, assists in reducing the amount of code coverage without inspecting the entire file or package. Nevertheless, an inaccurate model of Just-in-Time software defect prediction impedes both prevention and recovery of defects. The accuracy of prediction is mainly adversely affected by imbalanced class distributions and rate of false results. Accordingly, the focus of this study is on the problems of ineffective oversampling in imbalanced class distributions and high false-positive rates in effort-aware software defect prediction. This study proposes a reliable framework for Just-in-Time software defect prediction to accurately predict software defects during the code change process using Deep Q-Network (DQN). The proposed framework consists of two modified parts: 1) rebalancing class distribution within training datasets by kernel-based cross oversampling, and 2) using DQN as a defect classifier for accurate prediction. The proposed framework is further validated by

checking the constructed prediction model for efficiency in effort cost and prediction accuracy in open-source software projects. Validation of the prediction model is performed through within-project prediction, cross-project prediction, and timewise prediction to ensure model reliability. The quality assurance team can improve software defect localization by prioritizing testing based on Just-in-Time software defect prediction.

Keywords: Software quality, code review, just-in-time software defect prediction framework

Q-RANGKAIAN MENDALAM UNTUK RAMALAN KECACATAN PERISIAN SECARA TEPAT MASANYA

ABSTRAK

Mengurangkan cela perisian pada tahap kod di peringkat awal membolehkan pemeliharaan jangka panjang terhadap kualiti perisian. Menurut laporan IBM, kos untuk memperbaiki suatu cela perisian meningkat secara eksponen seiring dengan peredaran perisian dalam kitaran pembangunan perisian. Kos untuk memperbetulkan cela perisian selepas perisian dikeluarkan adalah sehingga 15 kali lebih tinggi daripada kos memperbaiki cela perisian yang ditemui semasa fasa pembangunan perisian awal. Jaminan kualiti bergantung kepada semakan kod untuk mengenal pasti dan memperbaiki cela perisian. Selain daripada pengoptimuman kod dan pemeriksaan formal, ramalan cela perisian menggunakan sumber terhad sebagai sebahagian daripada proses semakan kod untuk mengenal pasti cara yang paling berkesan dari segi kos untuk mengesan cela perisian. Pendekatan ramalan cela perisian dijalankan pada tiga tahap granulariti: modul, fail, dan perubahan. Ramalan peringkat perubahan, juga dikenali sebagai ramalan cela perisian "Just-in-Time," membantu mengurangkan jumlah liputan kod tanpa menyemak keseluruhan fail atau pakej. Walaubagaimanpun, model ramalan cela perisian "Just-in-Time" yang tidak tepat menghalang pencegahan dan pemulihan cela perisian. Prestasi ramalan terjejas oleh taburan kelas yang tidak seimbang dan kadar keputusan yang palsu. Oleh itu, tumpuan kajian ini adalah pada masalah memperbanyak sampel yang tidak berkesan dalam taburan kelas yang tidak seimbang dan kadar positif palsu yang tinggi dalam ramalan cela perisian yang peka terhadap usaha. Kajian ini mencadangkan satu kerangka yang berkesan bagi ramalan cela perisian "Just-in-Time" untuk meramalkan cela perisian secara tepat semasa perubahan kod menggunakan Deep Q-Network (DQN). Kerangka yang dicadangkan terdiri

daripada dua bahagian yang telah diubahsuai: 1) mengimbangi taburan kelas dalam dataset latihan melalui memperbanyakan sample secara silang berasaskan kernel, dan 2) menggunakan DQN sebagai pengklasifikasi cela perisian untuk ramalan yang tepat. Kerangka yang dicadangkan ini kemudian disahkan melalui pemeriksaan model ramalan yang dibina untuk kecekapan dalam kos usaha dan ketepatan ramalan dalam projek perisian sumber terbuka. Pengesahan model ramalan dilakukan melalui peramalan dalam projek, peramalan secara silang projek, dan peramalan berdasarkan perubahan masa untuk memastikan kebolehpercayaan model. Pasukan jaminan kualiti boleh meningkatkan penyetempatan cela perisian dengan mengutamakan pengujian berdasarkan ramalan cela perisian "Just-in-Time".

Kata kunci: Qualiti perisian, semakan kod, rangka kerja ramalan cela perisian secara tepat masanya

ACKNOWLEDGEMENTS

In the name of Allah, the Most Gracious, the Most Merciful, I begin this acknowledgment with praise and gratitude to Allah the Almighty, whose guidance and blessings have illuminated my path throughout this Ph.D. journey. Completing a Ph.D. thesis is a journey that requires the collective effort, support, and encouragement of many individuals. I am deeply grateful to all those who have played pivotal roles in this academic endeavour. Foremost, I extend my heartfelt gratitude to my esteemed supervisors, Assoc. Prof. Dr. Siti Hafizah Ab Hamid, and Dr. Asmiza Abdul Sani. Your wisdom, guidance, and scholarly insights have been the cornerstone of my research journey. Your dedication to my intellectual growth, patience in addressing my queries, and unwavering support throughout this rigorous process have been invaluable. I am truly fortunate to have had the privilege of your mentorship. I would like to express my profound appreciation to my friends and family who have stood by me with unwavering support and understanding. To my beloved spouse, Allya Nadira Abdul Rashid, your love, patience, and belief in my abilities have been a constant source of inspiration. You provided the emotional support that sustained me through the challenges of this Ph.D. journey. To my dear mother, Mariam Awang, your sacrifices, and unending encouragement have been a driving force in my pursuit of knowledge. Your unwavering faith in me has been a source of strength. This Ph.D. thesis represents the culmination of years of dedication and hard work, and it is a testament to the collaborative spirit of those who have contributed to this endeavour. Each of you has played a significant role in shaping my academic journey, and for that, I am profoundly thankful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

DRL	:	Deep Reinforcement Learning
DQN	:	Deep Q-Network
JIT-SDP	:	Just-in-Time Software Defect Prediction
KCO		Kernel Crossover Oversampling
KPCA		Kernel Principal Component Analysis
LOC	:	Line of Code
PCA		Principal Component Analysis
SQA	:	Software Quality Assurance
SDP	:	Software Defect Prediction

CHAPTER 1: INTRODUCTION

COVID-19 pandemic affected many businesses, small to large enterprises are forced to quickly reorganize working processes and accelerate IT priorities and technology roadmaps. In a recent report by Accelerated Strategies Group, 63.3% of business respondents noted that they accelerated digital transformation as a priority for their companies (Gartner, 2021). Their primary focus is based on contactless services, cloud migration, and DevOps activities. Since digital products determine the creation of sustainable and adaptable businesses, the development of software systems plays a critical role in building a better post-pandemic world.

1.1 Research Background

Software systems are becoming increasingly complex and are used in everything from mobile devices to space shuttles. The increasing importance and complexity of software systems in our daily lives make software quality a critical, yet extremely difficult issue to address. A well-developed software system increases the organization's reputation, promotes customer trust in software products, improves workflow efficiency, and provides a real competitive advantage (Ramler *et al*., 2019). Therefore, it is imperative to ensure that the software being built is reliable and fulfills its quality objectives. The quality and reliability of the software depend on the software defects that existed in the system. The higher number of defects decreases the reliability of the software, and a lot of effort is required to maintain the software quality. Gartner's 2012 report states that 20% - 28% of failure potentially happen to software projects ranging from small to large size because of the complexity and low quality of the requirements blueprint (Alami, 2016), which requires 60% - 80% correct effort (Ebert, 2007).

Software Quality Assurance (SQA) provides a set of activities that ensure software meets a specific quality level and takes up a large amount of the maintenance effort. In SQA, the code review process helps developers to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills (Beller *et al*., 2014). The code review process intends to systematically inspect source code for improvement and defects. Nevertheless, code review often involves repetitive and tedious tasks that increase the mental burden on reviewers and hamper their effectiveness (Singh *et al*., 2017). Code review is a widely used approach to support software quality (Kononenko *et al*., 2016). In a code review, large teams of authors and reviewers take turns creating and reviewing source code, sharing knowledge, proposing advice, fixing bugs, and promoting excellence. One of the most challenging aspects of code review is the ability to predict defects in the code. The researchers propose alternative techniques to improve the code review process, specifically by examining how to support developers and reduce the required cognitive effort. One solution aims to support code review with artificial intelligence aimed at maximizing reviewers' efficiency without increasing the cost of their review. Promising practice in this sense is predictive analysis, which aims to automatically predict the areas of source code that are most likely to be problematic, thus drawing the reviewer's attention.

Software defect prediction (SDP) allows for more efficient code review by predicting the defects prone to a software project in a cost-effective manner. SDP enables software defects to be predicted before they are observed by looking at the underlying properties of project artifacts. A software project in the context of SDP is a collection of procedures for the development of an intended software product with software versions by the related software artifacts. The software version contains an abundance of historic software project development information stored in software repositories. Two ways in

which the development information is stored: 1) within-project and 2) cross-project. Within-project version consists of the historical software development information extracted from prior software project releases. The cross-project version consists of the software development information extracted from version releases of other similar software projects. The historic software project development information consists of three sets of information: 1) a set of software metrics, 2) defects information, and 3) meta-information about the software project. This information is critical to project managers in improving their software development practices, especially in tracking and fixing defects in software releases (Tosun *et al*., 2010).

Current fast advancement technologies place project managers constantly to respond faster to technological changes and new requirements by releasing new software versions in a limited amount of time and budgets. To handle such situations, other than static code analysis and software defect localization, SDP is a solution that enables the identification of future defects in an optimized and cost-effective way for the software project at early stage. It is also capable to provide feedback on software defects which only detected in future software releases. To date, the extensive research on SDP has driven the involvement of more industries to participate in bringing more additional resources toward open-source software projects (Li *et al*., 2018). For that reason, the research on the SDP approach is expected to be growing more in upcoming years due to the availability of more public access software projects. The SDP approach is available to be deployed at several levels in open-access software projects. The SDP approach is performed at three granularity levels: 1) module, 2) file, and 3) changes level. Module-level SDP involves the prediction of the defect-prone modules before the testing phase. File-level SDP is conducted before a software release to act as a quality control step for software releases. Change-level of SDP is a continuous activity of quality control for each submission of code changes.

Changes level of SDP were first proposed by Mockus and Weiss (2002) recommending which code changes to software projects need to be inspected first based on the risk of introducing defects. Information within code changes is critical to be understood by the developers to carry out tasks on features addition, defects fixing, performance improvement, troubleshooting, and code maintenance (Misirli *et al*., 2016). To date, code changes level prediction is also known as Just-in-Time SDP (JIT-SDP) (Kamei *et al*., 2013). JIT-SDP enables the prediction process to be done once source code changes are committed in the version control system. The prediction process helps increase our understanding of software defect patterns in the early development phase which is exploited further for the quality control scheme. Accordingly, this research proposes an approach to improve the performance of JIT-SDP in more effective ways. JIT-SDP offers two advantages over module and file prediction levels. First, it reduces the amount of code coverage without having to examine whole files or packages during code review. Secondly, code change level prediction also is used to identify whether a certain change causes a defect at check-in code transaction. Thus, the developers are able to allocate the limited test resources in a more efficient way for practical application.

1.2 Motivation

As with any research work, several factors motivate the purpose of producing the research. Similarly, the following three significant factors motivated this research.

(a) *Difficulty in software defect management.* Managing the number of defects is an important aspect. Finding and fixing the defects cost lots of money. The data from the Gartner report in 2018, American companies spent around 42% money spend on software defects in IT products. Usually, software developers find and detect software defects through the process of testing. However, this is an expensive

process that takes a long time and unworthy the cost before release. Ultimately, most of the testing only happens at a later stage of software development. During testing, if the defect rate is higher than the acceptable level, the software development team is faced with a dilemma: to postpone the software release to fix these defects or to release the software products containing defects.

(b) *Cost-effective process*. The interests of software engineering in quality assurance are activities such as testing, verification and validation, defect tolerance, and software defect prediction. Software defect prediction effectively reduces the testing effort by identifying early signs of the potential of defect-inducing source code. Therefore, the identified defect source code can easily be fixed by developers, which reduces testing effort.

(c) *Automated code review tools*. Software defect prediction attracted the attention of large companies, which began experimenting with augmented code review tools (Gray *et al*., 2011; Tosun *et al*., 2010; Yan *et al*., 2020). For example, Google developers evaluated FixCache (Sadowski *et al*., 2011) which is a well-known tool for the defect prediction model. The evaluation of SDP is conducted in a typical working environment as a code review tool for the company (Lewis *et al*., 2013). FixCache uses a newly developed concept of defect locality which provides excellent results within controlled environments with minimal interactions with external factors. The developers of Google found, however, that the defects result generated are too imprecise to work in practice as code review recommendations. It is imperative that more research is conducted to improve the prediction results. In the context of this research, deep reinforcement learning with a quality-balanced training dataset provides an effective prediction of software defects based on the characteristics of code change metrics provided.

Despite many research attempts, all the factors mentioned above remained relevant and strongly motivated the production of this research toward the advancement of the software defect prediction domain.

1.3 Statements of Problem

The primary focus of this research is the issue of inaccurate prediction. Accurate prediction of software defects is important to ensure the quality of software during the software development process before software failure occurs. It helps developers to check and locate defects immediately at the time they are introduced. However, an inaccurate defect prediction causes the generation of false positives on non-defective instances that are predicted as defective labels and false negatives on defective instances that are predicted as non-defective labels. Getting false alarms wastes resources during the code review process. Particularly, the developer's effort and time to review the false result thus, cause frustration to the developers (Lewis *et al*., 2013). Primarily two problems that severely impact the prediction accuracy are identified, 1) ineffective oversampling in imbalance class distribution and 2) high false-positive rate in effort awareness evaluation.

1.3.1 Ineffective Oversampling in Imbalance Class Distribution

Software project datasets tend to have highly skewed class distributions (Song *et al*., 2018). In a skewed data distribution, the majority of data is in the clean class and a small portion of data is in the defect class. A skewed distribution is also called imbalanced data. To balance the number of instances in the minority class of the defect class, oversampling generates more synthetic instances. However, the defect data generated by oversampling are often duplicated or overlapped within the spatial distribution (Li *et al*., 2018). A key characteristic of SDP imbalance datasets is the lack

of variation and the lack of information about the distribution of data (Bird *et al*., 2009; Chen *et al*., 2018). Figure 1 illustrates an example of spatial distribution with many overlapped data points. The problem of overlapped data instances negatively impacts prediction models that utilize oversampled data. Figure 2 illustrates the performance of the Logistic Regression classifier in JIT-SDP using oversampled data from oversampling techniques (Barua *et al*., 2014; Chawla *et al*., 2002; Haibo He *et al*., 2008; Han *et al*., 2005; Lunardon *et al*., 2014). All baseline techniques, however, fail to distinguish their accuracy performance in imbalanced datasets which result in similar performance. This observation is influenced by the limited number of empty spaces available within the minority class (defect). A complex boundary line resulted in a small distance between the new and old data (Bellinger *et al*., 2016; Han *et al*., 2023). Accordingly, defects data are generated into clean class data spaces, which accounts for the problem of overlapping class spatial distributions.

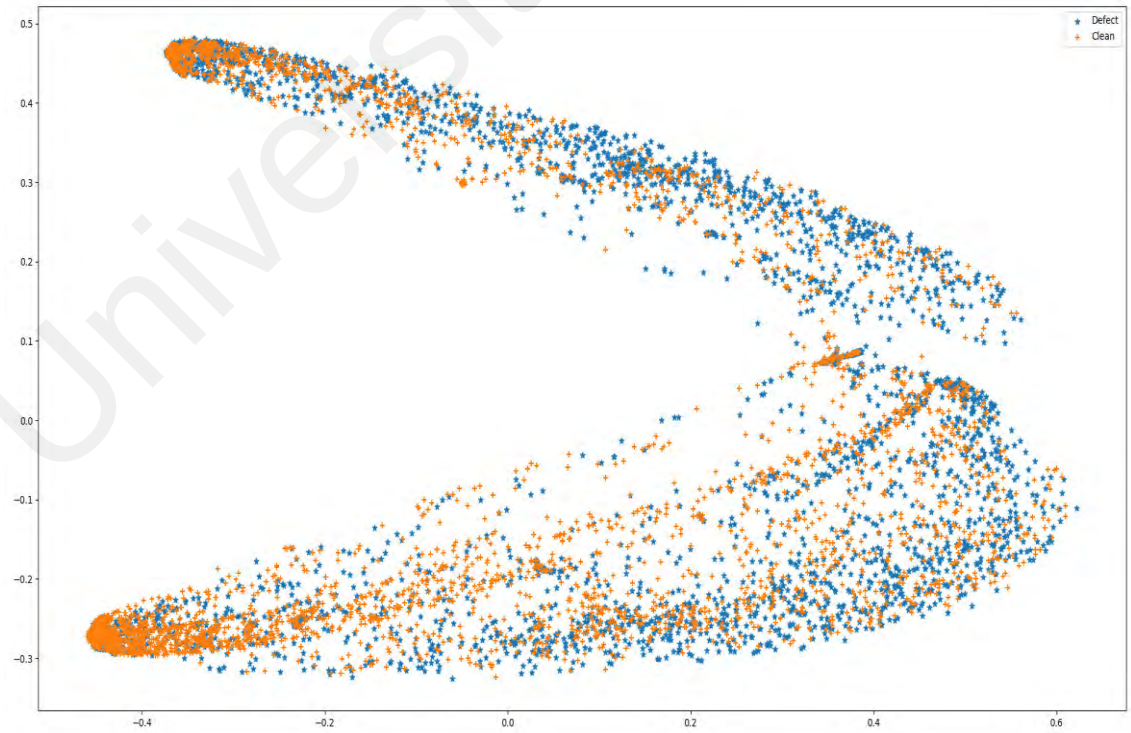


Figure 1: High overlap data instances in imbalance distribution of Eclipse-JDT

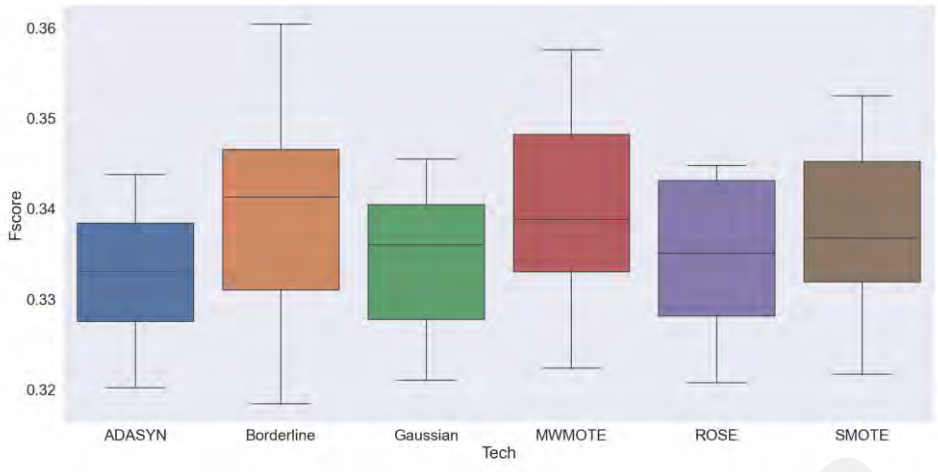


Figure 2: Accuracy of JIT-SDP model across 10-folds of oversampled datasets

1.3.2 High False Positive Rate in Effort Awareness Evaluation

For the prediction model to be cost-effective, the costs associated with quality assurance (QA) efforts such as code inspection and defect fixing must always be measured (Arisholm *et al*., 2010). Without QA efforts, the cost-effectiveness of the JIT-SDP model is uncertain. QA teams are particularly interested in determining how much time and effort it will take to fix a specific software defect (Feng *et al*., 2021). For instance, cost-benefit analyses are used to determine whether code inspections are worth the effort of fixing a defect. Accordingly, the accuracy of the prediction should be reflected in the effort awareness of the JIT-SDP model. A high rate of false predicted defects is associated with the performance of the effort-aware JIT-SDP model (Huang *et al*., 2019) . In Figure 3, more than 50% of false positives were generated by different effort-aware JIT-SDP models. A high false positive rate indicates that the effort awareness in JIT-SDP is still insufficient to ensure cost-effectiveness. The current effort-aware metric (ACC) may identify a wrong best JIT-SDP model that does not benefit the user to the maximum extent (Çarka *et al*., 2022). Thus, the high false positive rate of the effort aware JIT-SDP model hinders the practical adoption of prediction models in the industry.

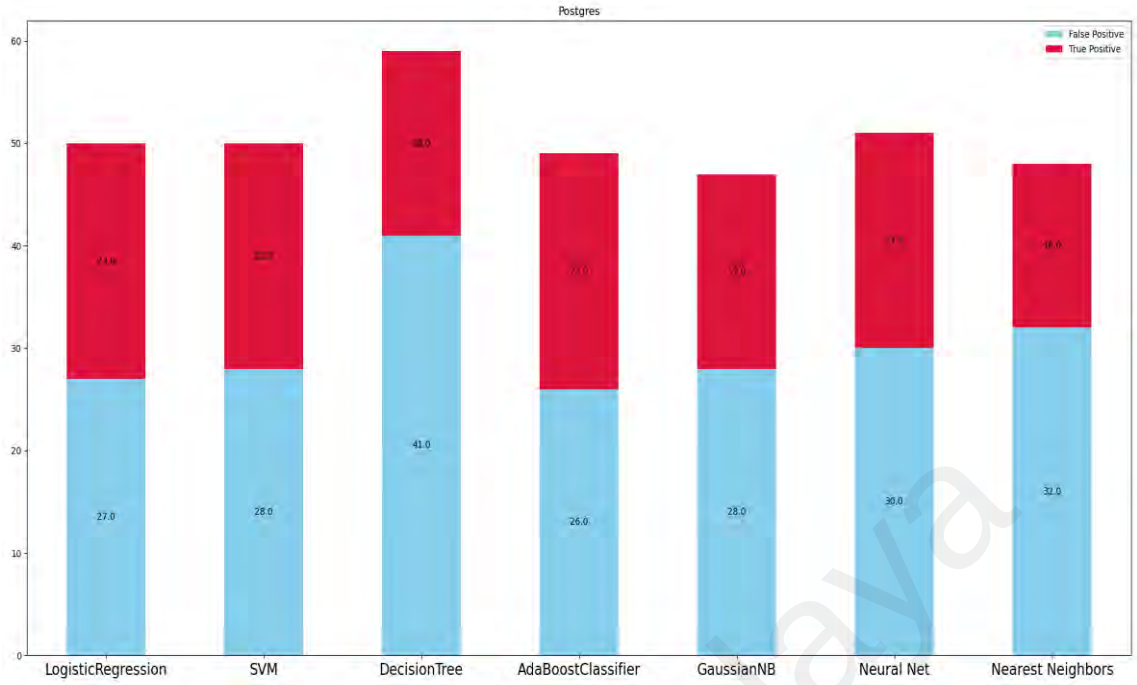


Figure 3: Accuracy of JIT-SDP models based on 20% of inspection effort

1.4 Scope of Research

This research is focused on utilizing Deep Q-Network (DQN) to address inaccurate software defect prediction. Based on the modern code review perspective, the research focuses on the prediction of software defects. The prediction of software defects is performed during the code transaction phase (JIT level). Data from three sources is utilized in this study in order to predict software defects, which include within-project, cross-project, and timewise datasets. Within-project data provides information regarding historical software development as gathered from previous software project releases. In cross-project data, software development information is extracted from the versions of other similar projects that have been released. A time-based data set is compiled based on the development timeframe of software. It is important to note that the primary concern in this prediction is the issue of inaccurate defect prediction. In Figure 4, the highlighted boxes illustrate the overall focus of this research project.

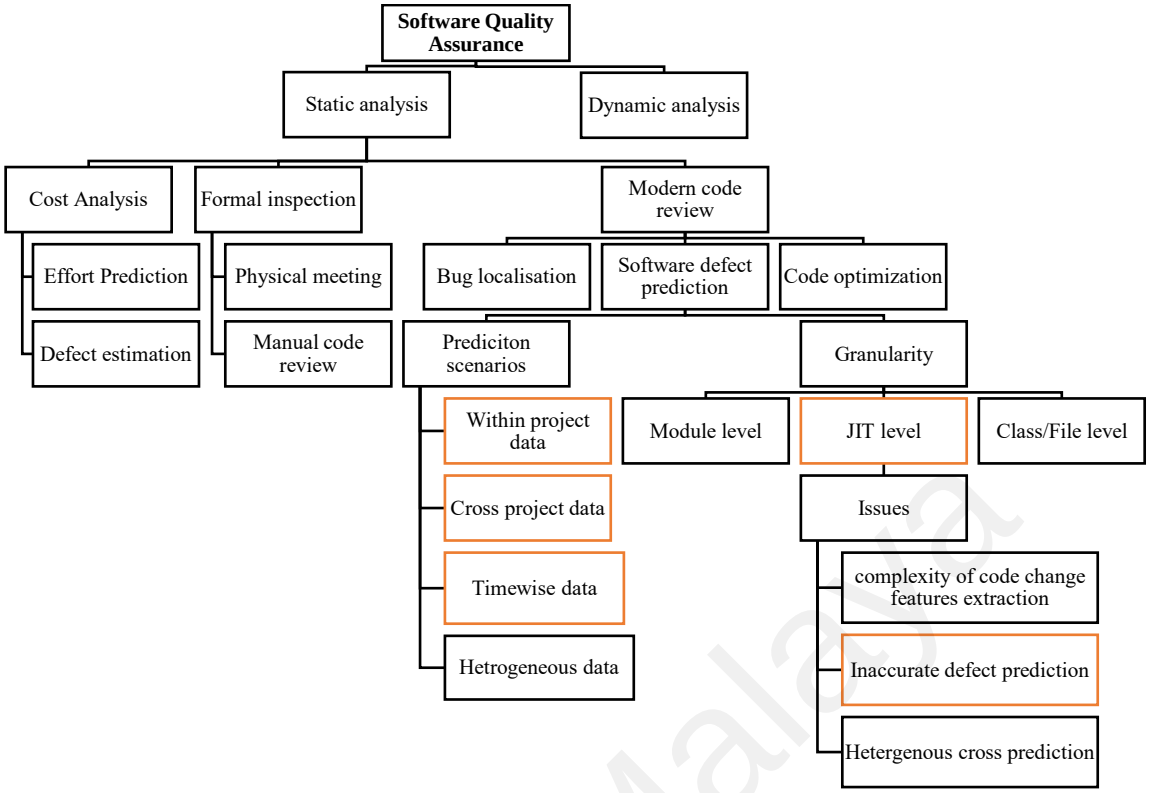


Figure 4: Scope of this research work

1.5 Research Objectives

This research aims to develop a Just-in-Time software defect prediction framework that enhances the accuracy performance with an increase of effort-aware prediction. To achieve this aim, this research aligns with the following set of objectives:

1. To determine the similarities and limitations of the existing Just-in-Time software defect prediction frameworks.
2. To design balanced datasets using an oversampling technique based on kernel analysis and cross-over interpolation.
3. To develop a model for Just-in-Time software defect prediction using Deep Q-Network.

4. To evaluate the prediction performance of the proposed framework using the proposed oversampling and Deep Q-network for within-project validation, cross-project validation, and time-sensitive validation.

1.6 Research Methodology

This research is conducted according to four phases corresponding to the four research objectives as shown in Table 1. Figure 5 shows the mapping of research objectives, methodology, questions, and outcomes using SMART method. The phases of the research methodology are outlined as follows.

Table 1: Mapping of research objectives, methodology, and outcome

PHASE	RESEARCH OBJECTIVES	METHODOLOGY	OUTCOME
Phase 1	To determine the similarities and limitations of the existing Just-in-Time software defect prediction framework (RO1)	• Comprehensive literature review (Sections 2.2 to 2.8) • Experimental setup (Sections 3.1.1 and 3.2.1)	1.Review of existing framework (Chapter 2): • Taxonomy of the literature in JIT-SDP frameworks • Limitation of existing JIT-SDP frameworks. 2.Comparison of experimental results for existing frameworks (Chapter 3)
Phase 2	To design balanced datasets using an oversampling algorithm based on kernel analysis and cross-over interpolation (RO2)	• Data collection (Section 4.1.1) • Experiment setup (Section 5.1.3) • Statistical analysis (Section 5.1.5)	• Kernel crossover oversampling algorithm (KCO) (Section 4.2) • Balanced class dataset generation with an increase in data diversity (Section 5.1)
Phase 3	To develop a model for Just-in-Time software defect prediction using Deep Q-Network (RO3)	• Model design (Section 4.3) • Experiment setup (Section 5.2)	• A JIT-SDP model for higher prediction accuracy and effort-awareness contexts (Section 5.2.6) • Framework of DQN with KCO (Section 5.2)

Phase 4	To evaluate the prediction performance of the proposed framework using within-project validation, cross-project validation, and time-sensitive validation (RO4)	• Comparative evaluation (Section 5.2.4) • Model reliability (Sections 5.2.4 and 5.2.5)	• Results of performance comparison with existing approaches and validation with software projects (Sections 5.2.6 and 5.2.7)

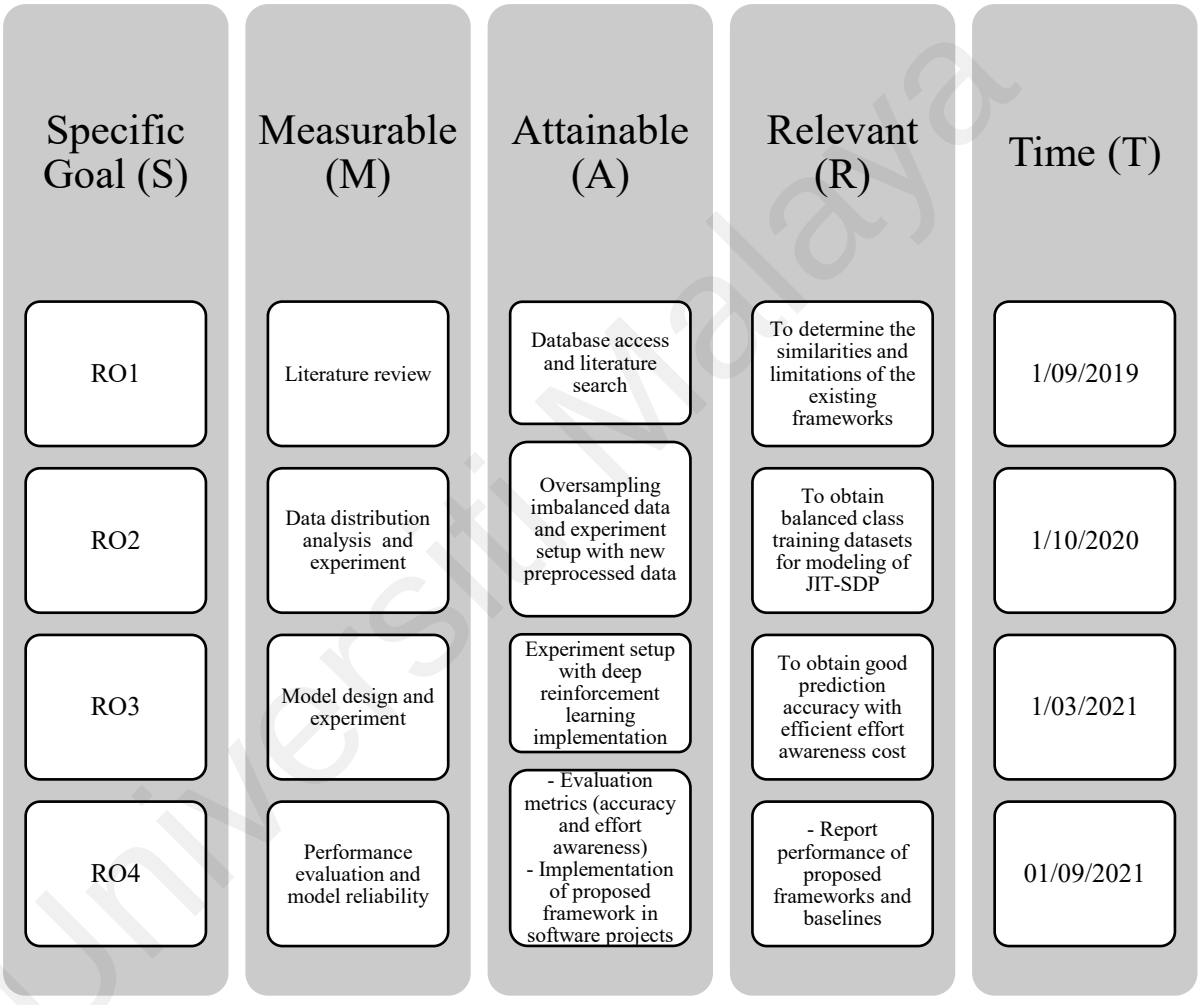


Figure 5: SMART method

1.6.1 Phase 1

The first phase of the research methodology according to the research question (RO1). Specifically, the research seeks to identify and investigate the research gap affecting inaccurate SDP predictions; with a focus on JIT-SDP. A comprehensive

literature review is conducted using popular database engines, including IEEE, Springer, ScienceDirect, ACM, and Google Scholar. In this review, the state-of-the-art techniques within existing frameworks of JIT-SDP are examined from 2013 to 2023, since JIT-SDP was first introduced by Kamei in 2013. Figure 6 depicts JIT-SDP trends over the last few years. An analysis of research issues and trends is presented in the review. Figure 7 illustrates an overview of the review during this phase.

According to the extensive literature review, two research problems contribute to inaccurate software defect predictions. The identified problems are based on imbalances in class distribution and effort awareness context. Further analysis of the identified problem by comparing the performance of baselines JIT-SDP in two separate experiments. The first experiment addresses the problem of ineffective oversampling in imbalance classes. This experiment examined which oversampling techniques perform better under different imbalanced class settings. This experiment aims to determine whether oversampling techniques deliver different predictions when dealing with overlapping class distributions that vary in characteristics depending on data characteristics. Various oversampling techniques are compared, including SMOTE, SMOTE-Borderline, ADASYN, GAZZAH, MWMOTE, ROSE, and MAHAKIL. The second experiment compares baseline classifiers in JIT-SDP against false positive results associated with effort awareness evaluations. Effort awareness of the JIT-SDP model needs to be consistently reflected in the quality of predictions. Therefore, the false positive rate is considered in the evaluation of the effort-aware model to assess the efficacy of using machine learning methodology concerning classifier accuracy performance.

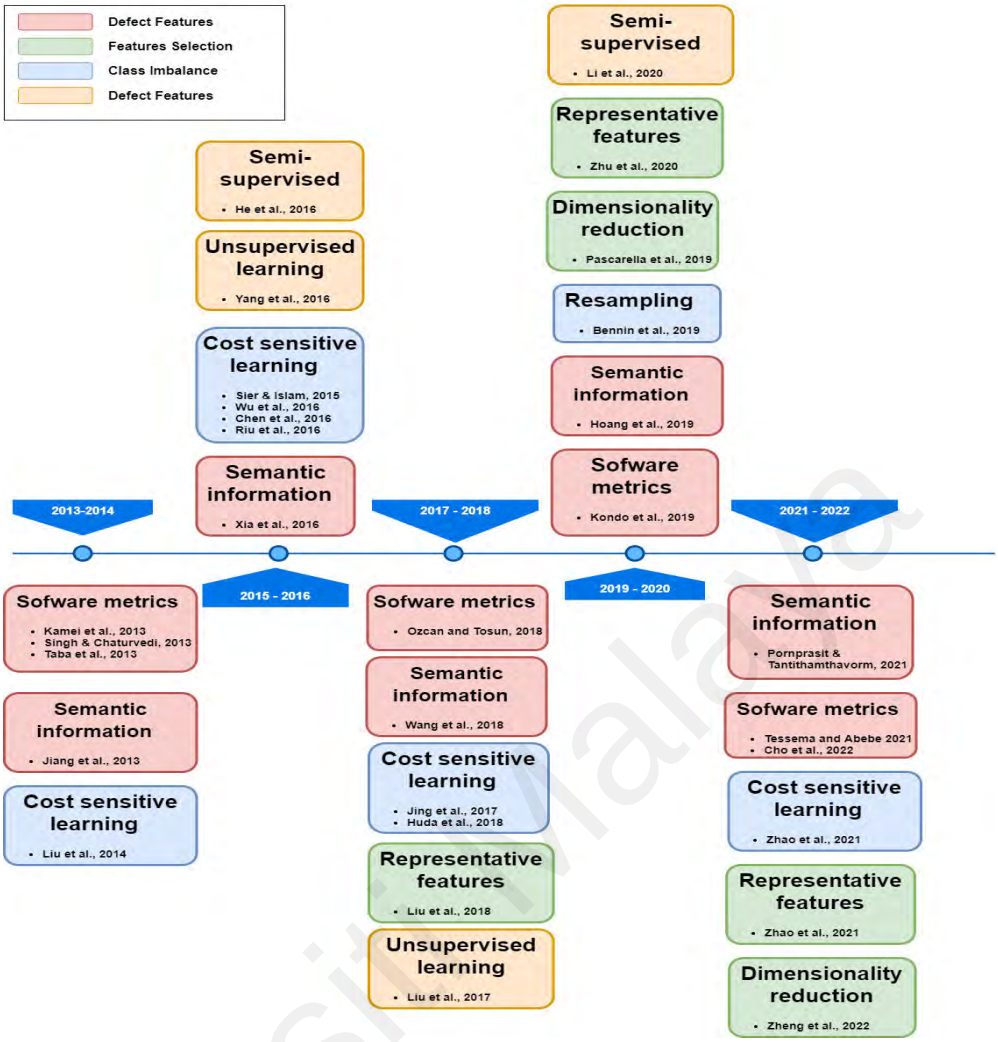


Figure 6: Progress of new approaches in JIT-SDP works

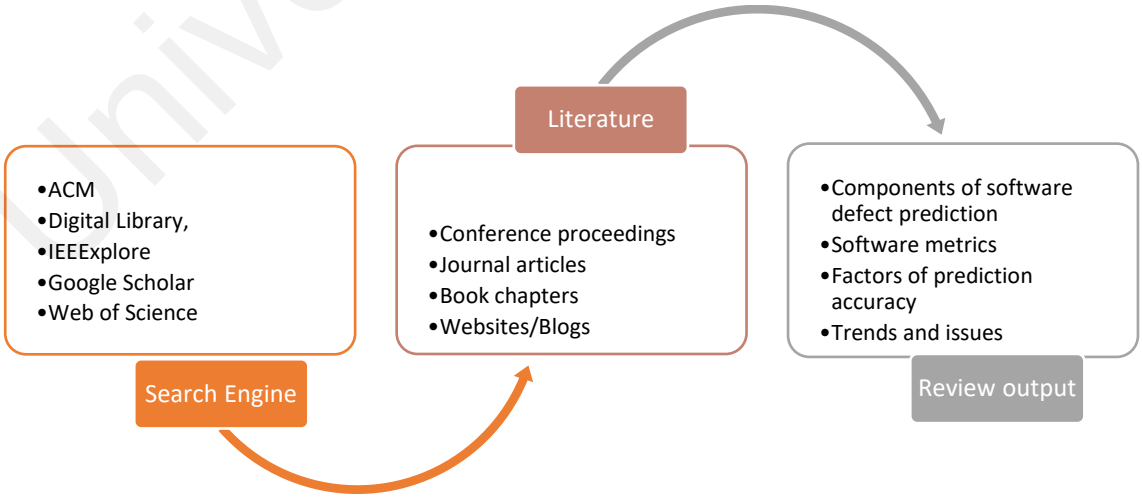


Figure 7: Overview of literature review

1.6.2 Phase 2

The second phase (RO2) of the research methodology focuses on the development of a new oversampling technique for balancing the class distribution of target datasets. Oversampling presents a challenge since existing techniques generally introduces duplicate or overlapped instances into the distribution of the existing data (Zhao *et al*., 2023) . Zhang *et al*. (2021) considering spatial distribution of samples in oversampling. Spatial distribution causes the boundaries between different types of samples to become blurred. Several important aspects to consider when analyzing the spatial distribution of samples, including class imbalance severity, clustering, overlap class and distribution shape (Lorena *et al*., 2020). For imbalance severity, a highly imbalanced dataset where the majority class significantly outnumbers the minority class produces class imbalance bias. As a result, minority class predictions are less accurate as the model tends to predict the majority class more frequently. Second, grouping or clustering instances belonging to the same class impacts the performance of a machine learning model. In densely grouped classes, the model has difficulty separating instances from those of other classes. In class overlap, the extent to which instances of different classes overlap or intermingle with each other. If instances are tightly clustered and overlap heavily, the model may have difficulty distinguishing between classes. Lastly, the distribution shape of the spatial distribution of instances across classes can also impact the performance of a machine learning model. For example, a dataset with instances spread evenly across a region may perform better than a dataset with instances tightly clustered in a few areas.

Motivating from spatial distribution, this study improves the ability to coop with the characteristic of spatial distribution by proposing Kernel Cross-oversampling (KCO). During this phase, several processes are conducted to ensure that the training datasets have the desired quality. The first step involves extracting the code change data

into software metrics. The extracted metrics are then used to analyze the distribution of classes. To determine the most effective experimental settings, the model parameter parameters are tuned based on an analysis of the data distribution for each of the software project datasets. Next, synthetic data of the defect class are generated using the proposed oversampling technique. The proposed technique includes three components: kernel-based principal component analysis (KPCA), spectral clustering, and crossover interpolation. As part of the proposed oversampling process, the first part is devoted to representing multidimensional features into two-dimensional features by employing KPCA. In this way, correlations between data instances are distinguished with visualization of data distribution. The second component of the proposed oversampling consists of deploying spectral clustering to explore the distribution of data for the plotted data distribution of the first component. The spectral clustering method allows for the separation of data distribution sources into several data clusters. Each of these clusters is measured based on the proportion of clean class data within the cluster. A candidate region for the generation of synthetic defect data is chosen from three clusters with the lowest number of clean class data. The selection is based on the premise that clusters with a low percentage of majority classes yield good neighborhoods and low-occupied space for the generation of synthetic data. The generation of new defect data by cross-interpolating according to template parents within the selected data clusters. Interpolating for new data continues to iterate until a balanced distribution of data classes is achieved. Figure 8 provides the overall work in balancing the class distribution of the training data. The balanced training data are now ready for deployment into the machine learning algorithm.

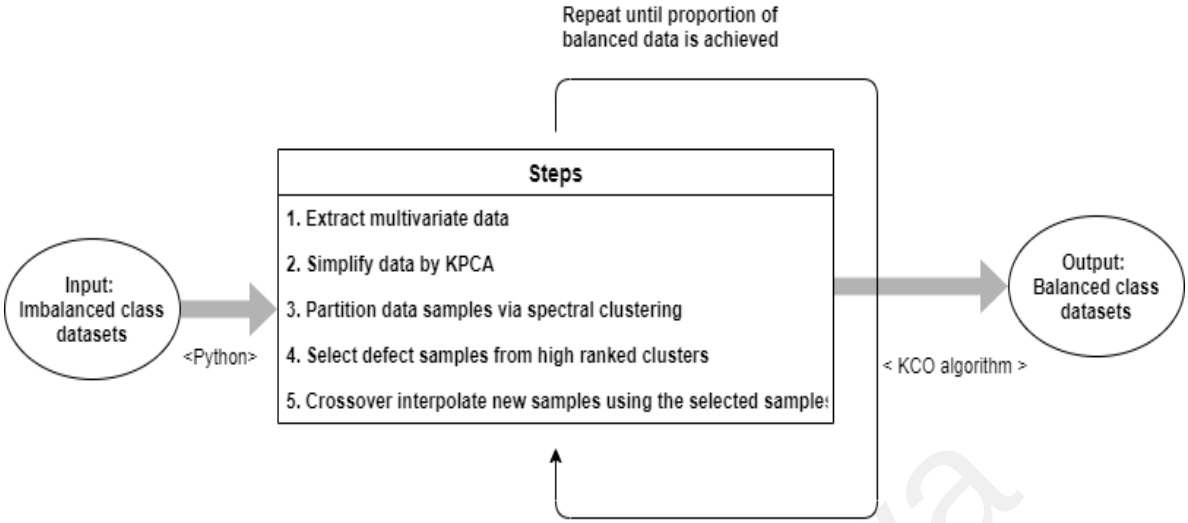


Figure 8: Process of balancing imbalance class distribution

1.6.3 Phase 3

The third phase of the research methodology (RO3) focuses on developing a prediction model utilizing DQN to enhance the accuracy of the defect classifier by taking effort into account when classifying software changes. Defect classifiers suffer from poor prediction when data drift occurs within the properties of code changes (Tabassum *et al*., 2023). One of the main challenges associated with JIT-SDP is the high rate of false positive predictions (Quach *et al*., 2021). Improvements needed to be made regarding the handling of data drift which source of false positives prediction. It is advisable to explore specific aspects of handling data drift, including continuous model refinement (online classification learning), dividing the data by period, and measuring the evolution of defect-inducing change patterns (Tabassum *et al*., 2023; Tan *et al*., 2015). Therefore, JIT-SDP will need an effective classifier that can reduce the effect of false positives prediction, such offered by DQN. For this phase, the pre-processed training data generated in Phase 2 is used to construct a DQN framework for JIT-SDP. DQN framework consists of two parts, namely environment, and agent. In the environment, code reviewing is formulated as a virtual environment in which DQN can

interact and earn rewards based on actions taken during the learning process. As part of training the model, the environment stores training samples and rewards or punishes correct and incorrect predictions.

The second component, namely the agent, is composed of two subcomponents: the deep learning model and action memory. In the agent of DQN, Q-network model is fed with training data from action-memory using a mini-batch learning mechanism. Q-network is based on a neural network algorithm. Q-network consists of many hyperparameters and finding the best combination of parameters is treated as a search problem. The values of hyperparameters cannot be determined from a regular learning process. Consequently, hyperparameters must be tuned before DQN training begins. This study adopts hyperband tuner strategy (Li *et al*., 2018) for hyperparameter tuning. For the action-memory mechanism is based on the decayed epsilon policy. The final output of this phase is a deep learning model for JIT-SDP. The constructed deep learning model is now ready for evaluation with other techniques to achieve a better balance between accuracy and effort awareness. Figure 9 shows the overview of training of DQN as JIT-SDP model.

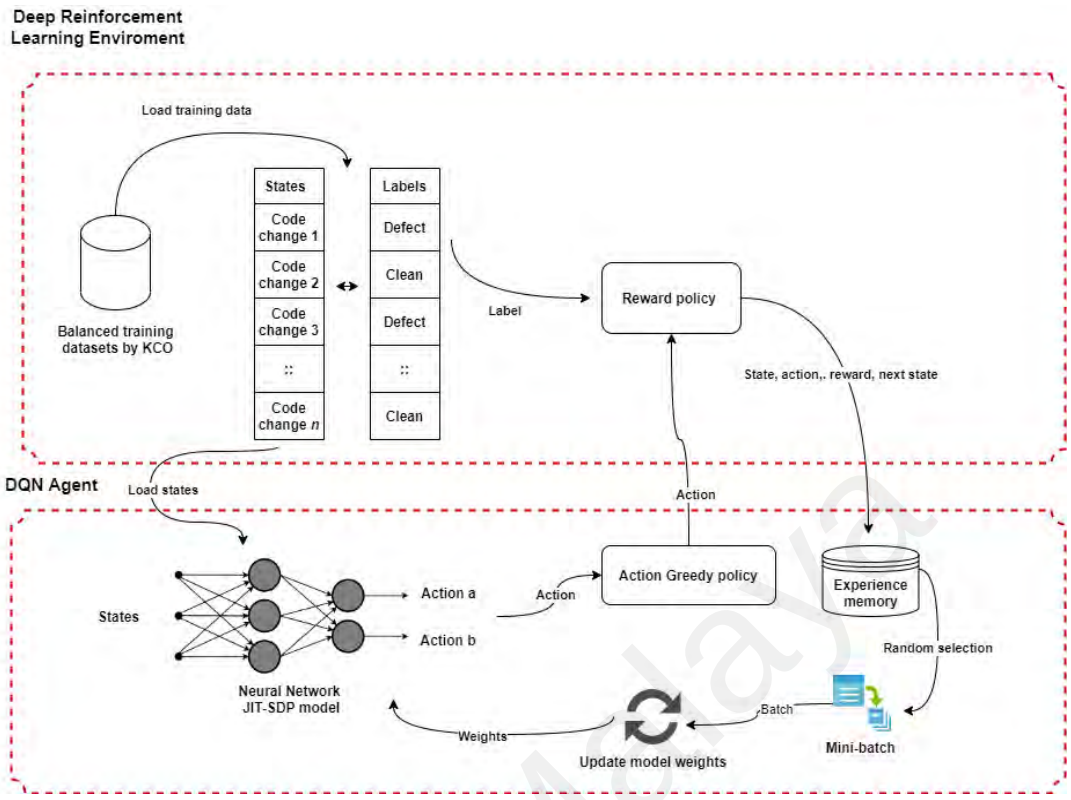


Figure 9: Mapping of agent and environment of DQN

1.6.4 Phase 4

The last phase of this methodology (RO4) focuses on the evaluation and comparison of the proposed framework with baseline frameworks. In this respect, Phase 4 evaluates the classifiers in JIT-SDP based on two performances such as prediction accuracy in F-scores and effort awareness. To test the model reliability, the proposed framework is implemented in software projects such as Columba, Bugzilla, Postgres, Mozilla, Eclipse.Platform and Eclipse.JDT. The phase further compares the performance of the proposed framework and baseline frameworks in three prediction scenarios namely within-project prediction, cross-project prediction, and time-wise prediction. Within project-prediction is performed within the same software project data. In this context, StratifiedKFold is used in this scenario to ensure that the class distribution in the datasets is kept in the training and test splits. The datasets are randomly divided into ten folds, with eight folds serving as training data and the

remaining fold serving as test data. Each fold is only used as a testing dataset once in cross-validation. Furthermore, the data need to be folded in such a way that each fold consists of the same proportions as the original dataset. The average result is recorded using StratifiedKFold to strengthen the reliability of the experiment outcomes. Timewise-validation is also performed within the same project, which takes into account changes in chronological sequence. The chronological order of the changes data for each software project is ordered based on the commit date The changes made during the same month period are then aggregated. Assume that the modifications in a project are divided into n components. For example, the datasets ($1 \leq i \leq n$ - 5) for training data and testing data consist of changes committed for two consecutive months. To predict testing data for parts i+4 and i+5, a prediction model m is developed using a combination of part i till part i+1 as training data. Cross-project prediction provides the predictive performance according to prediction across different software projects. The training data set on one project is used to predict defect-proneness in another project as the testing data set. For a set of n projects, this method produces n * (n - 1) prediction effectiveness values (Zhu *et al*., 2020). For this research, six projects are used as the subject projects. Accordingly, each prediction models produces 6 × (6 - 1) = 30 prediction effectiveness values.

1.7 Research Significance

The research provides efficient effort-awareness and higher accuracy in the prediction of software defects based on given code changes. This research believes that with appropriate solutions, an accurate and effective JIT-SDP model is achieved. Throughout this research, the importance of considering the advancement of classifier techniques, and the impact of having a set of quality defect datasets are given. The proposed framework reduces the efforts during code review by helping to uncover more

risky changes in the software project that not be reachable by the regular testing process. The proposed framework provides better results compared to other state-of-the-art frameworks in JIT-SDP overall performance.

This research provides valuable findings for both researchers and practitioners. For researchers, this research provides a new baseline model that need to be used in future JIT-SDP studies for evaluation of accuracy and effort-aware performance. For practitioners, this research provides an accurate prediction model for effort-aware JIT-SDP. The prediction model developed by the proposed framework is benefiting QA teams to help prioritize test cases and enhance static defect localization. Moreover, the solution provided by this research potentially be utilized in different research domains. For instance, the proposed oversampling technique is possible to apply for imbalance learning in other research domains such as static code analysis, development effort prediction, and code vulnerability prediction. Especially, the studies that utilize software metrics as the features or independent variables of the research problem. In addition, further application of the proposed deep reinforcement learning technique for JIT-SDP provides more depth analysis available for defect localization and production cost analysis studies.

1.8 Thesis Structure

Chapter 2 introduces the JIT-SDP frameworks for identifying the similarities and limitations of existing works following Phase 1 of research methodology. The review provides detailed background on JIT-SDP to allow a better understanding of the current research landscape. It also goes into detail on the progress in JIT-SDP approaches, starting from the software metrics until the most recent modelling of the prediction model by machine learning-based classification. The discussion maps the technique to a clear chronology to uncover the advantages and limitations of prior techniques. Finally,

the chapter also presents open issues in the prediction of software defects relevant to the current state-of-the-art in the domain. The output of this chapter enables investigation of the available classification taxonomies of current approaches and highlights their limitations.

Chapter 3 discusses Phase 1 of the research methodology by analyzing the limitations of existing work in the context of the identified problem statements. In the first section, the factors that contributed to the research problem of ineffective oversampling in imbalanced class distribution are analysed. The analysis provides classification results for the prediction of software defects based on balanced datasets provided by state-of-the-art oversampling techniques. The results of the analysis provide a deeper understanding of how the distribution of data in imbalance class datasets affected the performance of oversampling techniques. In the second section, the problem of predicting false positives in effort awareness evaluation is analysed and discussed in order to comprehend the current state of classifier selection. The analysis reveals how the selection of classifiers plays a crucial role in minimising false alarm results. Several baseline classifiers in JIT-SDP are evaluated and compared throughout the analysis to provide an overview of the performance of effort-aware models based on these classifiers.

Chapter 4 presents the development of the proposed JIT-SDP framework, aligned with Phases 2 and 3 of the research methodology. The chapter is organized into three sections reflecting the stages in the development of the proposed framework. The first section provides an overview of the process involved in developing the JIT-SDP model. JIT-SDP model development is divided into three phases: data extraction, data pre-processing, and training and prediction of the model. Following the second section, which corresponds to Phase 2, a new oversampling technique is developed to handle the

problem of overlapping spatial distribution within imbalanced datasets during data pre-processing. The technique relies on kernel analysis and spectral clustering to facilitate crossover interpolation for new samples of data. A key benefit of this technique is the improvement of the quality of training data for modelling the JIT-SDP model. The third section, which is part of the training and prediction process, is concerned with the development of DQN algorithm in order to generate a prediction model with a focus on reducing false positive predictions which corresponds to Phase 3. Application of DQN algorithm as JIT-SDP classifier provides more depth learning for capturing the pattern of software defects during code changes. The framework utilizes incremental learning with help from DQN and improvises on the existing classifier chain approach to achieve the objectives

Chapter 5, which is align with Phase 4 of the methodology, presents the results and discussion of the proposed solution in two sections. For the first section, evaluations of the proposed oversampling technique along with baseline techniques are done in the interests of both compare and showcase the robustness of the proposed solution. In the second section, experimental results from the application of DQN into JIT-SDP are discussed. The experimental evaluation compares the existing frameworks and the proposed framework with DQN embedded as a classifier.

Chapter 6 presents the conclusion to the research. This chapter revisits the contributions and maps them to the initial objectives of this study. Furthermore, it highlights the limitations of this study and makes valuable suggestions for future research. The chapter concludes by briefly discussing future directions and efforts to expand the research boundaries in software defect prediction.

CHAPTER 2: OVERVIEW OF SOFTWARE DEFECT PREDICTION

Software defects are more likely to be introduced over time. The domain has experienced significant evolution over the years due to various solutions that create a heterogeneous landscape. To understand the domain, this chapter discusses the literature on Software Defect Prediction (SDP). The literature review pays special attention to Just-in-Time Software Defect Prediction (JIT-SDP) context central to this research. Initially, this chapter presents some details regarding the introduction to the code review, which prompted the development of JIT-SDP. The second section provides background information about JIT-SDP and explains it in detail. The next section provides further information regarding the change level of software metrics. The section that follows discusses factors that influence the effectiveness of software metrics. Following is a discussion of existing oversampling techniques for imbalanced defect datasets. Following are further details regarding machine learning approaches and effort-aware models in JIT-SDP. In the preceding section, the need for advanced classifier and predictor approaches is discussed, along with the possibility of applying deep reinforcement learning to software engineering. Lastly, several issues have been identified to the extent that these issues require further discussion.

2.1 Automated code review

An important step in making a high-quality, secure software application is to implement automated code review. Developers tend to make mistakes during software development, thus by using best practices for a systematic code review is an effective way to improve the quality of software. Despite a manual review using the knowledge and skills of the code review team, security threats in source code that are meant to hide from users still at risk. Manual code review adds fresh perspectives from experts to identify logic errors, confirm the code works, and hold the developer accountable.

Having a team of experts check newly written code for source code purpose and logic is invaluable during manual code review. However, manual code review focuses more on the logic and intent of source code. Automated code review supplemented with manual review results in a safer and more efficient application much faster than manual review alone. Automated code review provides faster speed, higher accuracy, and better defect detection during the software development process.

During an automated code review, the source code is compared to a standard set of rules for common mistakes or security risks. Figure 10 illustrates the code review process, which consists of four primary steps:

1. **Upload code changes by developers**. Code authors or developers submit and upload code changes into code review tool for code inspection. Developers then invite the reviewer to perform the code review.
2. **Examine and review the changes by reviewers**. Reviewers evaluate the technical parts of the proposed change and provide feedbacks to the developers. Reviewers also give a score to show whether approve (positive value) or disapprove (negative value) for the proposed changes.
3. **Revise the proposed changes**. The developers make changes to the proposed changes based on the comments and then upload a new version to the code review tool.
4. **Integrate the approved changes**. Steps 2 and 3 are repeated until reviewers determine that the quality of the code change is sufficient for integration. The proposed changes then automatically upload and integrate into the code base.

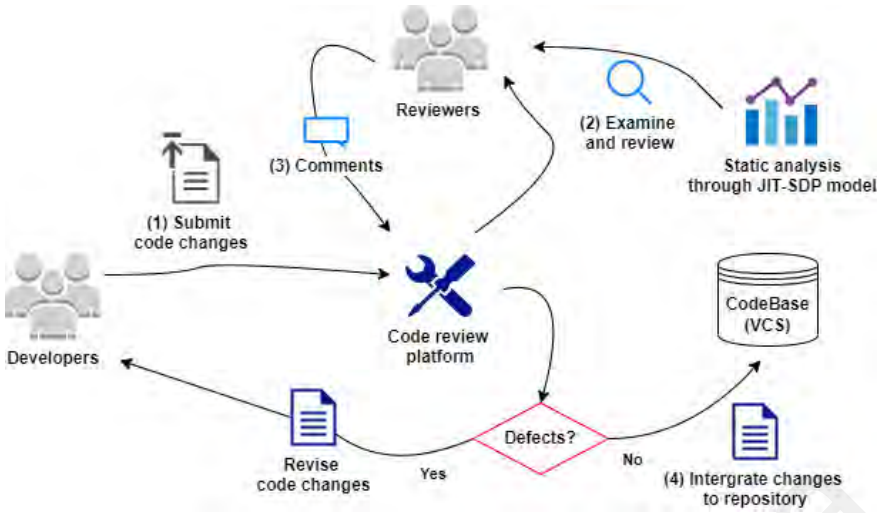


Figure 10: Overview of code review

The code review process often results in false alarms, but it still provides resistance to the software and improves its overall performance (Kononenko *et al.*, 2016). Removing false alarms from automated code reviews is an effective method of increasing both the quality and security of source code. A major challenge in the code review process is the effort of reviewers to conduct code inspections. To communicate an issue, a code reviewer needs to examine hundreds of lines of code and make comments regarding any possible defects. Tools such as Gerrit (https://www.gerritcodereview.com), Collaborator (https://smartbear.com/product/collaborator), and diff-styles (https://git-scm.com/docs/git-diff) have proven effective at detecting design flaws and coding violations. By integrating static analysis processes into code review, particularly JIT-SDP, the code reviewer reduces the amount of time and effort that they must devote to reviewing the code. Static analysis assists in automatically detecting coding standards violations and style violations. Consequently, reviewers concentrate more on important tasks, such as identifying logical flaws and optimizing code. However, code reviewers are still expected to go over the analysis results and identify relevant issues that were

not picked up by the analysis, as well as comment on any issues that the analysis missed.

Code review tools are generally divided into generations. According to Hedberg (2004), the current (5^{th}) generation is expected to offer increased flexibility in terms of supporting documents and processes. The current generation of tools however has several limitations, which are described as follows.

1. Developer experience plays a significant role in variation in defect detection in code reviews, as it depends heavily on the experience of the reviewers for the artifacts (code changes) under review. According to several studies (Kononenko *et al*., 2016; Lewis *et al*., 2013; Mockus, 2016; Sikic *et al*., 2021), the number of defects discovered during the review process correlates with the level of expertise of reviewers. Nevertheless, recommending and manually selecting appropriate reviewers is difficult for large developer teams.
2. Understanding the code under review is required. Effectiveness of the review process affects the ability of the reviewer to comprehend proposed changes (Mantyla & Lassenius, 2009). Without a detailed code analysis, reviewers are unable to comprehend proposed changes, resulting in greater variation and ineffective defect detection.
3. Manual selection of relevant change subsets for large change sets or code fragments within code changes is challenging. When dealing with large changes, code review often resorts to reviewing a large number of small changes (Baum & Schneider, 2016). Reviewing only small changes generates high overhead and duplicate effort. It is important that the change under review meet certain quality requirements rather than the size of the change.

Towards higher review effectiveness, cognitive support code review tools, which are called sixth generation tools, need to provide more flexibility with better cognitive support (Baum & Schneider, 2016). Thus, implementing JIT-SDP model within code review will provide a better understanding of defect-proneness and software quality. A mapping of current limitations on code review with JIT-SDP opportunities is shown in Figure 11.

Current limitation factors	Possibilities via JIT-SDP
• Developer experience • Depth understanding • Large changes	• Reviewer recomendation • Defect proneness • Prioritizing effort aware changes

Figure 11: Limitations and recommendation of current code review

2.2 Just-in-Time Software Defect Prediction

In SDP, software defect proneness is predicted without executing software parts using the underlying characteristics of a software project in order to predict the likelihood of defects. In software engineering, SDP has been a major research area for the past four decades (Wan *et al*., 2018). Akiyama (1971) conducted the first study on SDP in 1971 in order to estimate the number of software defects by assuming complex source code was prone to introducing software defects. A simple prediction model for software complexity was proposed based on lines of code as an indicator of complexity. Nevertheless, simply relying on this metric is insufficient to represent software complexity. In turn, modern SDP approaches used a variety of software metrics to represent the complexity of software projects (Meiliana *et al*., 2017; Piotrowski & Madeyski, 2020; Punitha & Chitra, 2013; Radjenović *et al*., 2013; A. Singh *et al*., 2018).

The previous SDP approaches are impractical for large-scale software development since module granularity is often set as a file or method. Consequently, the JIT-SDP approach is introduced to handle the prediction process on code changes that are more detailed. Due to its ability to yield practical results during check-in time (Kamei *et al*., 2013), JIT-SDP is argued to be superior to other SDP (module and class level). Companies such as Avaya, BlackBerry, Cisco, Ubisoft, and Google (Lewis *et al*., 2013; Mockus & Weiss, 2002; Nayrolles & Hamou-Lhadj, 2018; Shihab *et al*., 2012; Tan *et al*., 2015) have implemented JIT-SDP frameworks to improve their software project reliability. In the course of software development and maintenance, developers may submit code changes for various reasons, including fixing defects, extending functionality, refactoring codes, and improving system performance. In JIT-SDP, these changes are classified into two groups: defective changes and clean changes. A defective change is a change that is prone to introducing one or more defects, while a clean change is a change that is not likely to introduce any defects. In general, these changes are quantified in the form of software metrics.

2.2.1 Software Metrics

Software projects require measurements for quality assurance, performance, debugging, management, and cost estimation. Measurements are also crucial to discovering defects in software components. Software metrics are the most commonly used type of measurement. Software metrics are used to predict software defects. A prediction model is constructed on the basis of software metrics which is intended to predict the maximum number of software defects. Generally, software repositories such as version control systems and issue tracking systems provide software metrics based on data gathered from software development. Figure 12 summarizes the process of extracting software metrics from these repositories.

Software metrics are broadly divided into two types: code metrics and process metrics. A code metric reflects the complexity of the source code. Based on the hypothesis that source code with higher complexity is more likely to contain defects. In contrast, process metrics provide insight into many aspects of the software development process, including changes in source code, ownership of source code files, developer interactions, dependency analysis, and project team organization (Li *et al*., 2018). Process metrics are more useful than code metrics in building a prediction model due to the stagnation of code metrics (Rahman & Devanbu, 2013). A study showed that various process metrics have been utilized in recent years to model JIT-SDP, particularly those involving code change levels (Son *et al*., 2019). The discussion of change level software metrics continues in Section 2.3.

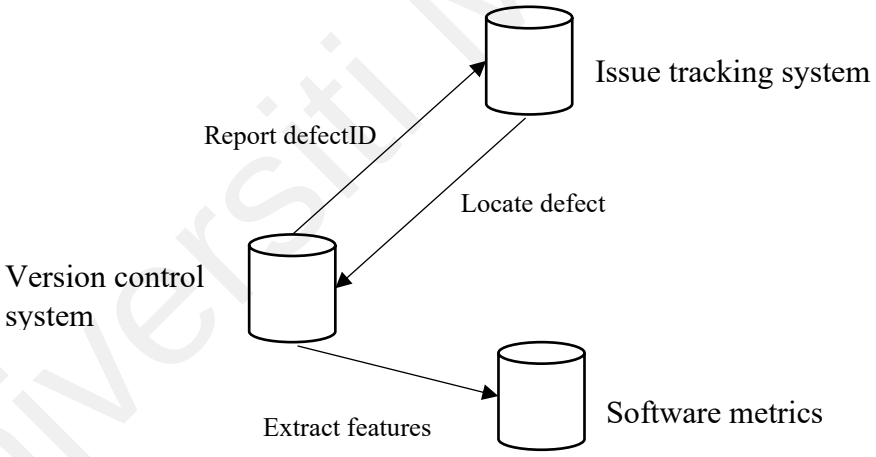


Figure 12: Extraction process of software metrics

2.2.2 Software Defect Dataset

Software defects are classified into two types: functional defects and maintainability defects. Functional defects are defects discovered in functional requirements that result in direct software failure. Maintainability defects, on the other hand, are found in design specification, implementation, and maintenance. However, since maintainability defects

are generally less expensive to fix than requirement defects, JIT-SDP more focusing on design defects.

Application of software metrics, in general, is a robust predictor to train a JIT-SDP model from software defect datasets. JIT-SDP employs both public and private datasets. Public datasets are usually open-source data extracted from primary and secondary sources of code repositories. Primary source data typically are mined from VCS (e.g. GIT) integration code repositories such as GitHub, Bitbucket, Gitlab, Jenkins, and Codebase. These data are extracted from software projects such as Mozilla, AgroUML, Eclipse.platform, EclipseJDT, Columba, PostgreSQL, Linux kernel, Bugzilla (Buz), Lucene (Luc), Jackrabbit, Xorg, ArgoUML, GWT, Jaxen, JRuby, Xstream, SWT, QT, OPENSTACK, Hadoop, Camel, Gerrit, Osmand, Bitcoin and Gimp. Practically, the primary source data first needs to be pre-processed which involves the metrics data extraction and data labeling, before being ready to be used as a set of training data for the prediction models.

Data from a secondary source is directly used as a training dataset due to software metrics that are already calculated or already available from its software artifact. Private datasets are those that are created for industrial applications that are not made public. Most companies/organizations are concerned about the privacy preservation aspect of their software projects, therefore extracting these datasets is often difficult. Private datasets are therefore mainly accessible via restricted or closed-access repositories for internal research.

Interestingly, prior studies (Gray *et al*., 2011; Jiarpakdee & Hassan, 2011; Mockus & Weiss, 2002) criticized the poor quality of existing defect dataset that is often used to train defect prediction models which leads to biased prediction performance. Since the classification algorithms used to train the prediction model are insignificantly impacted

the performance of the prediction result (Ghotra *et al*., 2015), various data preprocessing is beneficial to detect and mitigate biases in the defect datasets. The biases that existed are attributed to the complexity of a classification problem, especially in JIT-SDP. The complexity of a classification problem (Lorena *et al*., 2019) is mainly based on three main factors. These are, 1) the ambiguity of the classes, 2) the sparsity and dimensionality of the data, and 3) the complexity of the boundary separating the classes.

For ambiguity of class boundaries, it occurs when the characteristics of data classes in a dataset are improperly represented. As a result, overlapping and unbalanced data instances are introduced (Chen *et al*., 2016). To enhance the JIT-SDP model's classification accuracy, it is critical to consider how change information is represented as model features within datasets. In the absence of improper data representation, defective change examples are likely to be hidden by non-defective examples due to the complex distribution of data. There is a curse of dimensionality associated with sparsity and dimensionality data, which results in overfitting that is caused by having too many features for the datasets (Shivaji *et al*., 2013). Instead of selecting the entire set of features, which is indeed costly in terms of classification costs, it is imperative to select only meaningful subsets of features to represent datasets. In the context of complexity boundaries, the area surrounding class boundaries where defects and non-defect classes often overlap is considered the complexity boundary. Nevertheless, whether this information should be considered noise or as informative remains worth of investigation. Noise instances are presumably reduced when class decision boundaries are enhanced

In previous experiments on JIT-SDP, experiments only examined the risk of code changes in commercial or open-source projects. Usually, the experiments evaluated

code changes data from public repositories. The datasets are associated with six software projects: Columba, Bugzilla, Postgres, JDT, Platform, and Mozilla. Table 2 shows a summary of the selected datasets. Columba is a Java-based email client with a user graphical interface, wizards, and internalization support. Bugzilla is a web-based bug tracking system and testing tool. PostgreSQL is a powerful, open-source object-relational database system. Eclipse-JDT (JDT) is an IDE supporting the development of any Java application which includes features like syntax highlighting, content assistance, refactoring support, and debugging tools. Eclipse-Platform (Platform) is an open-source integrated development environment for programming and supports plugins that allow developers to extend its functionality. Datasets used in this experiment are derived from the extractor of code changes by SZZ algorithm. The features are based on change metrics by Kamei *et al*. (2013) that are associated with code and process metrics. Table 3 provides the features detail according to change metrics.

Table 2: Description of software project datasets

Project	Language	Description	No of changes	Period	Defect %
Bugzilla (BUG)	Java	Web-based bug tracking system	4620	08/1998–12/2006	37
Columba (COL)	PERL	Email client written in Java	4455	11/2002–07/2006	31
Eclipse JDT (JDT)	C++	Java development tool	35,386	05/2001–12/2007	14
Eclipse platform (PLA)	Java	Integrated development environment for programming language	64,250	07/1996–05/2010	15
Mozilla (MOZ)	Java	Web browser application	98,275	08/1998–12/2006	5
PostgresSQL (POS)	C++	Object-relational database system	20,431	11/2002–07/2006	25

Table 3: Dimensions of change metrics

Dimension	Name	Definition	Description
Diffusion	NS	Number of modified subsystems	Change modifying many subsystems are more likely to be defect-prone
	NM	Number of modified directories	Many directories in a change are more likely to be defect-prone
	NF	Number of modified files	Change touching many files is more likely to be defect-prone
	Entropy	Distribution of modified code across each file	Changes with high entropy are more likely to be defect-prone due to developers' need to recall changes across files
Size	LA	Line of code added	More lines of code added are likely to introduce defects
	LD	Line of code deleted	More lines of code deleted are likely to introduce defects
	LT	Line of code in a file before the change	The larger the file, the more likely a change introducing defects
Purpose	FIX	Whether or not the change is a defect fix	Fixing a defect indicates that an area where errors are more likely to occur
History	NDEV	Number of developers involved in the changes	The larger number of developers is more likely to introduce defects because files revised gave many different designs thought and coding styles
	AGE	The average time interval between the last and current change	The lower AGE tends to introduce defect
	NUC	Number of unique changes to modified files	The larger spread of modified files, the higher complexity
Experience	EXP	Developer experience	More experience developers are less likely to introduce defects
	REXP	Recent developer experience	A developer modified the files recently is less likely to introduce defects
	SEXP	Developer experience on a subsystem	Developers are familiar with subsystems modified are less likely introduce defects

2.2.3 Machine Learning

JIT-SDP adopts two primary approaches to machine learning: 1) Data pre-processing, and 2) Modelling of defect classifiers. It is considered optional to use machine learning for data pre-processing in the JIT-SDP framework. It is a method for preparing data to increase the reliability and consistency of raw software defect datasets. Contrary to machine learning for software defect classifiers, it is an iterative process of fitting the available data into machine learning algorithms to construct the model. Figure 13 illustrates the workflow of the machine learning approaches in modeling defect prediction.

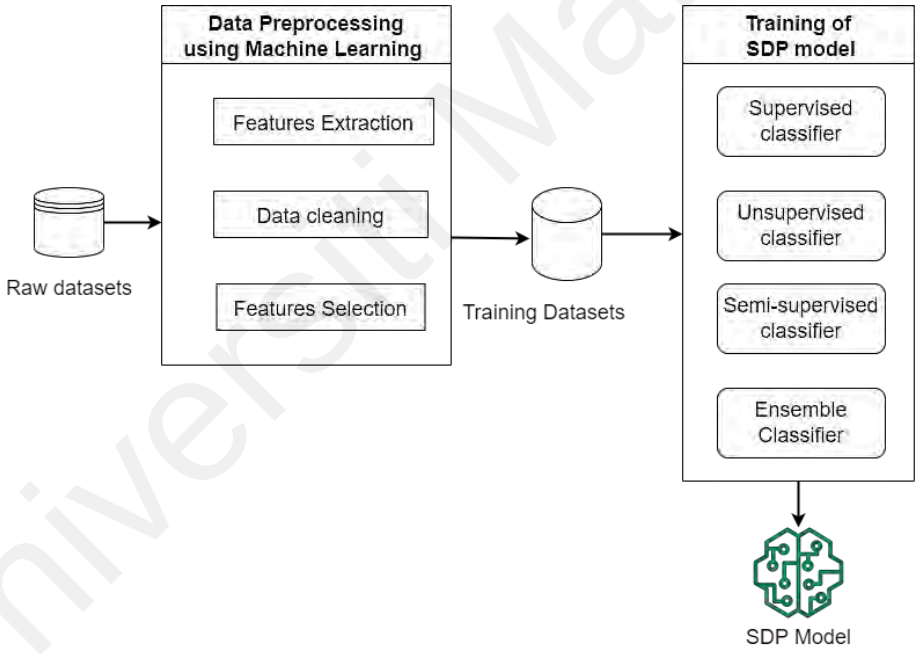


Figure 13: Workflow of machine learning in software defect prediction

For JIT-SDP, data preprocessing is as important as machine learning. In fact, most of the work required to create an effective machine learning model for software defect prediction consists of preparing and managing the data used to train the model (Bowes *et al*., 2018). Software defect datasets are known to be incomplete, class imbalanced, and redundant (Kim *et al*., 2011; Meiliana *et al*., 2017; Pandey *et al*., 2021). This means

that data extracted from the software under development produce inconsistent and unsatisfactory training datasets for JIT-SDP modeling. Data preprocessing becomes increasingly important in software defect prediction, many researchers start developing machine learning techniques for preprocessing software defect datasets (Akmel *et al*., 2018). Some of the most popular machine learning techniques for data preprocessing in software defect prediction include feature extraction (Fan *et al*., 2021; Malhotra & Khan, 2020; Rosen *et al*., 2015), data cleaning (Gray *et al*., 2011; Mockus, 2016), and feature selection(Hosseini *et al*., 2018a; Huda *et al*., 2017; Laradji *et al*., 2014; Shivaji *et al*., 2013).

The most popular methodology for modeling the JIT-SDP classifiers involves machine learning. In the literature on JIT-SDP, four categories of machine learning are found: supervised, unsupervised, semi-supervised, and ensemble classifiers. Supervised learning refers to the utilization of all labeled training data when developing the prediction model. A semi-supervised learning method uses a limited number of labeled training data and a large number of unlabelled data in order to construct the JIT-SDP models. In unsupervised learning, unlabelled data are used in modeling the prediction model without the need for labeled training data. Ensemble learning involves combining the predictions from two or more models. Detailed discussion is provided in Section 2.7.

Machine learning is indeed widely used for software defect prediction; however, the preprocessing of data and the training of the classifier need for improvement to handle inaccurate issues. In a review of various studies, we found that many have focused primarily on data preprocessing as part of their experimental design. A limited amount of attention is paid to the generalized perspective of the proposed models. Although various advanced classification methods are proposed for the JIT-

SDP, the true potential of classification methods is not yet fully explored. The reason is that the preprocessing aspect of training data is still inadequate. For this reason, the classifier of the prediction model produces results biased toward noisy data. To this end, it is necessary to conduct a proper preprocessing of the raw data with the aid of algorithms. In addition, it is necessary to use advanced machine learning algorithms as the model classifier for achieving unbiased results.

2.3 Change Level Software Metrics

Software metrics from various factors are available as model features for JIT-SDP to assist in predicting whether a code change will introduce future defects. Figure 14 illustrates how prior studies utilized code and process metrics. In terms of code metrics, these metrics indicate the complexity of the source code, whereas process metrics define the complexity of the development process. Code metrics are generally categorized as measures of size, complexity, and object-oriented features. Both size and complexity metrics are based on file-level measurements, whereas object-oriented measurements are based on class-level measurements. Source code analysis tools are capable of calculating these code metrics automatically to allow for the automatic calculation of these code metrics. However, a common threat associated with analysis tools is that the same metrics calculated by different tools often produce different values for the same source code files. The use of code metrics alone as a prediction feature is not sufficient to represent the actual characteristics of source code during a code change. As a result, improvement in prediction performance is mostly driven by a combination of both code and process metrics.

Several JIT-SDP works have combined code metrics with process metrics to represent features of code changes, including dependency network metrics (Herzig *et al*., 2016; Zimmermann & Nagappan, 2008), change burst metrics (He *et al*., 2016;

Nagappan *et al*., 2010), change metrics (Kamei *et al.*, 2013; Yang *et al.*, 2016; Yang *et al.*, 2017;Chen *et al*, 2018, Qiao and Wang, 2019; Huang *et al.*, 2019), complexity entropy metrics (Singh & Chaturvedi, 2013), antipattern metrics (Taba *et al*., 2013), periodic experience metrics (Ozcan and Tosun, 2018), and context metrics (Kondo *et al.*, 2019). Table 4 provides a brief description of each of these metrics.

- *Dependency network metrics*: Zimmerman and Nagappan (2008) proposed information flow between code entities modeled by code dependency graphs. The metrics allow the identification of source files that are more prone to introducing defects. They showed that interactions between files resulting strong defect prediction capabilities. A set of network metrics comprises of three groups of dependency graphs. The first are the ego metrics, which calculate the properties of complexity neighborhoods within the local network within the dependency graph. The second group of metrics in the dependency graph relates to structural metrics which measure the size of the sub-networks that are connected to each of the data nodes in the graph. Lastly, centrality metrics describe how many nodes are dependent on each other. A node with a large number of dependencies is more prone to defects. On the basis of semantic interaction features within a code change, the metrics proposed by them enable the prediction of defects.
- *Change burst metrics*: Nagappan *et al.* (2010) introduce the concept of change bursts and extract them from a series of changes. A change burst is a sequence of consecutive changes. It is defined by two parameters, namely the gap size and the burst size. Burst size is the minimum number of changes in a burst. Increasing the gap size yields longer bursts and increasing burst size eliminates shorter bursts. The metrics include four main group metrics: change metrics (measured by the size and extent of the changes), temporal metrics (measured by when change bursts occurred), developer metrics (measured by the properties of developers involved

during the changes), and code churn metrics (measured by the number of lines added, deleted, or modified during the changes). The proposed metrics allowed the prediction of defects to be performed despite the lack of information regarding the software requirements (i.e., relying only on the change history). A change with a higher amount of change burst indicates the source code is likely to produce more defects.

- *Change metrics*: Kamei *et al.* (2013) proposed software metrics for change measures to predict whether a change introduces a future defect or not by considering fourteen factors grouped into five dimensions (i.e. diffusion, size, purpose, history and experience). Diffusion refers to the number of files a change involves where a highly distributed change is more complex and harder to understand. Size indicates the size of LOC within code churn operation. Purpose gives the number of defect-fixing changes. History provides the number of previous changes and defect fixes. Experience describes experience information about developers. These five dimensions are the metrics combination of code metrics (i.e. LOC and code complexity) and process metrics (i.e. code churn, code ownership, and context of change). Recently, the measurement of these change metrics able to be generated by using a web application called *CommitGuru* which (Rosen *et al*., 2015) provide publicly. Change metrics enable the prediction on the risk of code changes by predict defect proneness at the time of submitting commits.
- *Periodic developer metrics*: Ozcan and Tosun (2018) proposed the measurements of periodic developer experience considering the contextual knowledge of developers on files and directories during commit time. Three aggregation methods (minimum, maximum and average) are used to measure characteristics of files of the related commit for developer experience. The proposed metrics aiming to capture experience of developers (previous knowledge) on files, commits and

activities. Different developers have different/similar knowledge at the end of code development. Thus, measuring their experience periodically give more generalized characteristics developers-based defect prediction model. The periodic developer experience is found to be more effective capturing the defect proneness compared to activity-based metrics (code churn activity).

- *Context metrics*: Kondo *et al.* (2019) proposed context metrics which involves counting keywords and word in the context of a change. The intuition of counting words due to a context with more words is likely to be more complex than a context that has less words. As for 'keyword', it refers to the keyword defined in the programming language of the source code. The number of keywords in the context gives indication of the nested degree of a change. A change with a larger number of keywords is likely to more complex than a change that has fewer keywords around it. Higher complexity of the change indicates the more likelihood that the change is a defective change.
- *Aggregated change metrics*: Šikić *et al.* (2021) describe the chronological order of the changes by aggregating the data of all changes made to the software between two versions. The proposed metrics aggregated previous existing change metrics by representing a chronology of commits. The metrics comprises of fourteen different aggregated change metrics which are extended from change metrics by Kamei *et al.* (2013)). Aggregated change metrics consider sequential and chronological order of all changes to capture more generalized defective change characteristics during the development process.

Vast majority of researchers in JIT-SDP works tended to focus on utilizing software metrics by Kamei *et al.* (2013). Over the past five years, only a few applications of other software metrics as the features of the prediction model: Dependency metrics, Change burst metrics, Periodic developer metrics, Context metrics, and Aggregated

metrics. One of the increasing concerns on the current software metrics is that existing metrics for JIT-SDP have reached a performance limit. Moreover, the emergence of various object-oriented approaches during software development required more aspects of consideration for change-based metrics. Thus, the development of new metrics is required

The effectiveness of available software metrics varies across different software project datasets. This is largely because software metrics selection is influenced by software projects in the existing study (Xia *et al*., 2014). Therefore, it is imperative to choose software metrics since they aid in improving prediction performance. Summary of the related work in JIT-SDP involving usage change level software metrics is presented in Table 5

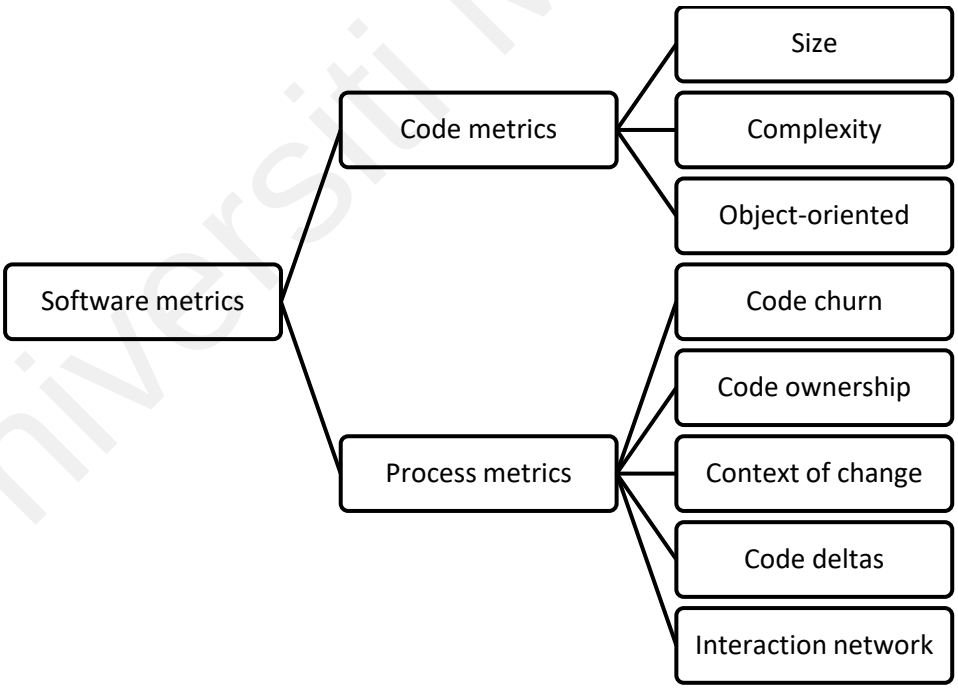


Figure 14: Existing software metrics

Table 4: Software metrics for JIT-SDP

Software Metrics	Change Attribute	Description	Metric Name	Limitations
Dependency network metrics (Herzig *et al*., 2016; Zimmermann & Nagappan, 2008)	Dependency	Information flow between code entities in code dependency graph	Size, Ties, Pairs, Density, WeakComp, nWeakComp, TwoStepReach, ReachEfficency, Brokerage, nBrokerage, EgoBetween, nEgoBetween, EffSize, Efficiency, Constraint, Hierarchy	Developer factors are neglected in the network metrics, but humans are the ones who introduce defects
Change burst metrics (He *et al*., 2016; Nagappan *et al*., 2010)	Sequence	Sequence of changes factor by extracting series of consecutive changes information.	NOC, NOCC,NOCB, TBS, MaxCB, NOCE , NOCL, TFB, TLB, TMB, NDEV, CT, TCB, MCB	The gap between sequence of changes needed to be fine adjusted to obtain a good prediction result. Distribution of defects across sequence of changes is ignored
Complexity Entropy metrics (Singh & Chaturvedi, 2013)	Complexity	Entropy of the complexity of code changes over a period of time	HCPF, HCM	The value of entropy parameters for decay function needs to be adjusted properly to obtain best results

Antipattern metrics (Taba *et al*., 2013)	Antipattern	Antipattern properties in a file	NAP, ANA, ACM, ARL	Performance of metrics is highly dependent on the number of antipatterns computed with DÉCOR tool.
Change metrics (Chen *et al*., 2018; Huang *et al*., 2019; Kamei *et al*., 2013; Yang *et al*., 2015, 2017)	Diffusion	Distribution of a change as a highly distributed change have more complex and harder to understand	NS, ND, NF, Entropy	Correlation analyses are required because the problem of multicollinearity (redundant) often is found on these metrics
	Size	Size of a change	LA, LD,LT	
	Purpose	Number of changes to fix defect	FIX	
	History	History of previous changes and defect fixes	NDEV, AGE, NUC	
	Experience	Experience information about developers	EXP, REXP, SEXP	
Periodic experience metrics (Ozcan and Tosun, 2018)	Periodic	Developer's prior knowledge on files, commits and activities periodically calculated	ExpLocAvg,ExpDirectoryLocAvg, ExpComAvg, ExpDirectoryComAvg, EditFreqAvg, ExpBuggyComAvg ExpImprovComAvg, ExpNewFeatureAvg, ETotalLocAvg, NumOfDeveloperAvg	Insignificant in case of software project that only have few source code revisions (limited incremental developer information)

Context metrics (Kondo *et al.*, 2019)	Context	Information in the lines that surround the changed lines of a commit (i.e. code churn based on keywords)	NCW, NCKW	The metrics assumption is that the number of keywords (context) in changed regions indicate the nested change which contribute to defect-proneness. As time goes on, source code eventually achieve maturity and nested changes are unlikely to occur frequently

Table 5: Change level software metrics

Related Works	Software metrics	Classifier algorithm	Project datasets	Size of data	Types of prediction
(Singh & Chaturvedi, 2013)	Complexity entropy metrics	LR and SVR	Public: Moz	Total:17992 *All defective changes	Within project
(Taba *et al.*, 2013)	Antipattern metrics	LR	Public: AgroUML and Eclipse	Total:168881 (Defect:56078)	Cross and within project
(Kamei *et al.*, 2013)	Change metrics	LR	Public: Buz, Col, JDT, Pla, Moz, and Pos Private: 5 java projects	Total:260519 (Defect:27015) *Only available for public dataset	Within project

(Yang *et al.*, 2015)	Change metrics	DBN	Public: Buz, Col, JDT, Pla, Moz, and Pos	Total:227417 (Defect:27015)	Within project
(Herzig *et al.*, 2016)	Dependency network metrics	LR,knn, RP, and SVM	Public: ArgoUML GWT, Jaxen, JRuby and Xstream	Total:36050 (Defect:7202)	Within project
(He *et al.*, 2016)	Change burst metrics	RF	Public: Eclipse, JDT, and SWT	Total:18251 (Defect:11269)	Within project
(Yang *et al.*, 2017)	Change metrics	RF	Public: Buz, Col, JDT, Pla, Moz, and Pos	Total:227417 (Defect:27015)	Cross and within project
(Chen *et al.*, 2018)	Change metrics	LR	Public: Buz, Col, JDT, Pla, Moz, and Pos	Total:227417 (Defect:27015)	Cross and within project
(Ozcan and Tosun, 2018)	Periodic developer metrics	LR, NB, kNN, J48, and RF	Public: Luc and Jackrabbit	Total:5422 (Defect:2234)	Within project
(Huang *et al.*, 2019)	Change metrics	LR, RF, SMO, kNN, J48, and NB	Public: Buz, Col, JDT, Pla, Moz, and Pos	Total:227417 (Defect:27015)	Cross and within project
(Kondo *et al.*, 2019)	Context metrics	LR and RF	Public: Hadoop, Camel, Gerrit, Osmand, Bitcoin and Gimp	Total:137062 (Defect:27317)	Within project

2.4 Inaccurate Factors Affecting Software Metrics

Same metrics may perform well in one organization while failing miserably in another organization (Chen *et al*., 2021). Performance issues with software metrics are also compounded by data preprocessing issues. Figure 15 shows the factors contributing to the inaccuracy of JIT-SDP performance related to software metrics as model features.

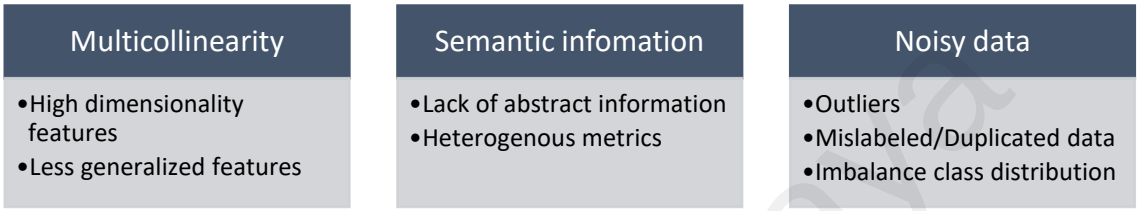


Figure 15: Issues of change level metrics

2.4.1 Multicollinearity Features

The relationship between metrics of code changes and the defect prediction output is very complex (Qiao & Wang, 2019). In addition, software metrics often show multicollinearity between features, making it imperative to eliminate highly correlated features. As a result, multicollinearity reduces the number of features available for prediction models, resulting in a lack of coverage of code change properties. In consequence, the accuracy of the prediction models suffers. Classification of multicollinearity among features in prior studies can be attributed as Figure 16

Figure 16: Taxonomy of handling multicollinearity features

1) Dimensionality Reduction

High dimensionality features contributed to high correlation among software metrics which often occur in SDP. It is a condition that negatively impacts the generalizability of the prediction model (Mamun *et al*., 2017; Jiarpakdee & Hassan, 2011). Curse of high dimensionality is mainly attributed to irrelevant or correlated features that existed within JIT-SDP datasets. In these datasets, two forms of correlation among metrics are commonly existed: collinearity and multicollinearity. For collinearity, one metric is linearly predicted by another metric. In contrast to multicollinearity, it is a condition in which one metric is linearly predicted by a combination of two or more metrics. Dimensionality reduction identifies and removes correlated/irrelevant features to ensure that only discriminant features are selected as the training datasets for the classification models. The reduction of features in the context of JIT-SDP is based on two criteria: high correlation filter and rank of information gain.

- High correlation filter is a process of reducing the relevance of features by their correlation with dependent variables. Filter-based approaches provide faster features selection due to not requiring training of the models. It reduces the complexity of a model which makes it easier to interpret. Thus, the removal of highly correlated features increases the speed of learning algorithms, decreases bias measures, and higher interpretability of model or simpler model.

- Rank of information gain is a process of sorting features regarding the amount of information gained through the selected feature whether the change data is defective or clean. The lower-ranked features are selected as irrelevant features that needed to be removed.

For high correlation features, the reduction is based on the analysis of correlation among individual software metrics. Two variants of correlation analyses are found in JIT-SDP works, which are described as follows:

- Spearman correlation analyses measure the correlation between two metrics. The pair of metrics with correlation coefficients above the specified threshold level is considered highly correlated features. The aim is to find the best subset of metrics that have the highest correlation with the classification outcome while having a low correlation between themselves. Spearman correlation test is known to be resilient toward abnormal distributions as commonly present in defect datasets
- Variance inflation factor (VIF) helps to find the multicollinearity among metrics by constructing a regression model to predict metrics based on a combination of other metrics. The measurement of VIF scores is calculated through the model fitting error between a regression model constructed by other metrics with metrics under examination. A metric is considered multicollinearity with other metrics in case of VIF score reaches a specified threshold value.

For high rank of information gain, prior works focus on using information gain algorithm (Quinlan, 1986), which enables the measurement of gain provided by each metric toward defects prediction. The algorithm quantifies the entropy value of the prediction model in case of excluding the individual metrics as the subset of selected features. Each metric is sorted according to its contribution to the decision of the model. The highest expected reduction of information gained is ranked at the top of the list.

The output of this algorithm provides a subset of features that are capable of positively influencing the relationship between selections of certain metrics with defect-proneness changes.

The correlation between software metrics makes it difficult to identify precisely which features are responsible for the predictive power of the SDP model. In addition, these highly correlated features increase model training time, reduction in model accuracy and performance due to overfitting toward correlated features (Hawkins, 2004). Therefore, the first step in SDP is to identify and remove correlated/irrelevant software metrics which ensuring that only discriminant features are selected as the training datasets for the classification models. According to Lorena *et al*. (2019) and Shivaji *et al*. (2013), the accuracy of prediction remains unaffected using a small number of features, and even the performance is improved in some cases.

For JIT-SDP, filter-based selection techniques are widely used to reduce the dimensionality of features in training datasets. Most studies utilized a high correlation filter (i.e. spearman test) to remove redundant/irrelevant metrics for their training data as illustrated in Table 6. In contrast to Pascarella *et al.* (2019) works, they utilized information gain algorithm to eliminate the less informative metrics and only selected the higher informative gain metrics for their prediction model. Nevertheless, the selection of the best subset metrics according to a filter criterion produce differently for the other filter criteria. (Jiarpakdee *et al*., 2018) argued that low consistency among filters-based techniques is the result of different evaluation criteria producing different subsets of metrics. Moreover, better performance not always achievable via metrics reduction. Occasionally, the previously removed metrics become important in future revisions causing lower prediction quality.

Table 6: Dimensionality reduction in JIT-SDP works

Related Works	Software Metrics	Dimensionality Reduction
Yang *et al.* (2016)	Change metrics (Kamei *et al.*, 2013)	High correlation filter
Yang *et al.* (2017)		
Cho *et al.* (2018)		
Huang *et al.* (2018)		
Chen *et al.* (2018)		
Yang *et al.* (2019)		
Cabral *et al.* (2019)		
Li *et al.* (2020)		
Zheng *et al*. (2021)		
Pascarella *et al.* (2019)		Rank of information gain
Ozcan & Tosun (2018)	Periodic developer metrics (Ozcan & Tosun, 2018)	High correlation filter
Sikic *et al.* (2021)	Aggregated metrics (Sikic *et al.*, 2021)	High correlation filter

2) Metrics Representation

For metrics representation, the features are provided with mapping data (set of features) to learn the representation of itself. Representative learners are mainly based on deep learning approaches of finding a representation of the basic features into abstract deep semantic features by the integration functions. Figure 17 describes the detailed workflow of the metrics presentation process. Different integration functions are available in the literature. In particular, functions such as the sequence of file versions (Liu *et al.*, 2018), and minimizing reconstruction errors (Zhu *et al*., 2020) are reported in related studies of JIT-SDP. In summary, each of these criteria is as follows:

- **Sequence of file versions** provides information regarding the historical changes of source files across version sequences. It describes how source files/code change over project evolution. The information is useful for representing traditional software metrics in continuous software versions as defect predictor features.
- **Reconstruction errors** measure the difference cost between input software metrics and the consequence of reconstruction of compressed features by network algorithm. The minimized reconstruction errors are regarded as a compressed representation of software metrics. The pre-processed metrics by this criterion have robust features representation and more generalized capability.

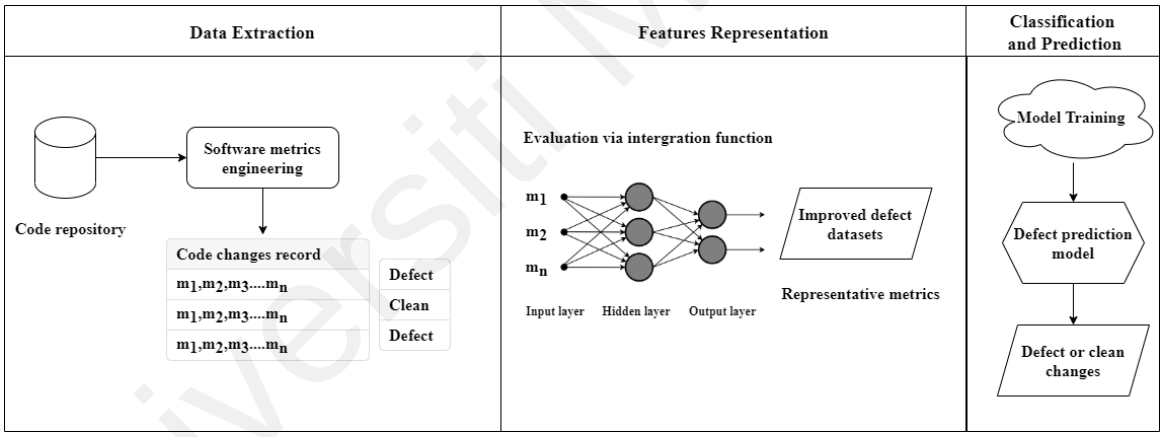


Figure 17: Metrics representation process

Instead of removing the correlated metrics, the prior works (Liu *et al.*, 2018; Zhu *et al.*, 2020) employ metrics representative approaches to provide a better selection of metrics for building a defect classifier. They enhanced the robustness of software metrics, thereby the represented metrics provide greater generalization ability and are more robust in constructing the prediction model. The usage of metrics representation approaches enables the construction of basic code change features into deep abstract

semantic features. The main drawback of these techniques, however, producing a higher risk of overfitting than filter techniques and are computationally expensive (Li *et al.*, 2018; Wu *et al.*, 2016). These techniques apply only to datasets of reasonable features dimensionality. In case of data dimensionality is very high, the number of weights in the network overly larger to find a near-optimal setting of the network

Autoencoders are used for representation learning by utilizing neural networks. It comprises of multilayer (input layer, hidden layers, and output layers) feed-forward neural network. The design is based on neural network architecture such that a compressed knowledge representation of the original input is produced by integration functions. It extracts deep representations from the traditional software metrics. Autoencoders usually have a high number of features connections. Therefore, it converges slowly and is likely to get stuck in local minima. Autoencoder-generated features are often used to replace the original features in deep learning to produce better results. To the best of our knowledge, the application of autoencoders is very limited in the field of SDP.

Historical Version Sequence of Metrics (HVSM) helps to highlight the trend of code changes throughout version sequence information of files (Liu *et al.*, 2018). It provides a representation of changing information by joining code and process metrics in a specific number of continuous historical versions. In contrast, existing process metrics only consider the change information between two adjacent versions. Therefore, the discovery of sequence historical information of whole revisions is unavailable for extraction. HVSM requires an efficient neural network classifier in handling the sequential data, which is capable of training in data with different lengths of input. Even though with consideration of only code metrics, the metrics representation of this approach is claimed to outperform baseline classifiers trained with both code and process metrics.

2.4.2 Semantic Information

Existing software metrics are unable to distinguish programs with different semantics because of the inability to capture abstract information within source code during the code change process. Figure 18 shows the types of semantic features extracted from the source code. The details of each feature are presented in Table 7.

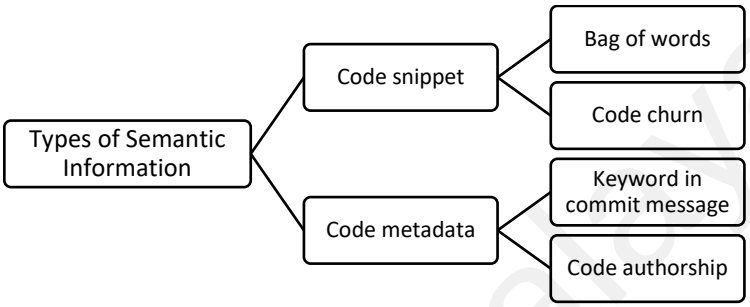


Figure 18:Types of semantic features

To bridge the gap between semantic information and SDP, it is necessary to use source code semantic representations. The semantics features are collected by applying a deep learning algorithm to a representation of source code in the form of an Abstract Syntax Tree (AST) or Control Flow Graph (CFG). The construction of semantic features is based on three main steps: 1) Parsing source code/commit message for changes into token vectors in form of AST or CFG, 2) Mapping and normalizing each type of vector in numerical vectors and 3) training the deep learning algorithm with the input vectors to generate features.

Table 8 provides a brief description of previous JIT-SDP studies utilizing semantic features. Semantic-based features allow for different contextual information of the same source code characteristics to be distinguished. The extraction of these features heavily relies on the choice of deep learning architecture used to learn the semantic representation of the source code. To analyze ASTs of source code during code

changes, different deep learning approaches adopted different architectures and learning processes (Wang *et al*., 2018). Thus, it prevents the extracted semantic features from having the same properties value. The biggest advantage of deep learning based semantic features is its resistance to nonlinear combination relationships between features compared to conventional software metrics.

Prior studies of JIT-SDP have often ignored the semantic information within code changes. The explanation for this condition is that semantic information is usually buried deep within the source code. Simple deep learning approaches such as DBN, CNN, and RF are commonly used to learn the context of code changes. Nevertheless, the advancement of various modern pre-processing data techniques such as noise reduction, data tagging algorithm, and untangling change algorithm in these recent years has made it possible to produce more reliable data input for deep learning process. These techniques help to provide more opportunities for more complex machine learning approaches to be applied in capturing more information in the context of code changes. More advanced machine learning requires higher data amount and quality compared to other conventional approaches. Besides, JIT-SDP studies primarily concentrate on predicting functional defects. However, in code review practice, developers find more reliability defects rather than functional defects (Mantyla & Lassenius, 2009). The context of code changes in a different type of defect remains unexplored. Further studies on the context of various classes of defects during code changes are significant in the modeling of JIT-SDP.

Table 7: Context of source code information during code changes

Semantic features	Description	Intuition	Limitation
		Changed code snippets	
Vector of words	Information occurrence of each individual word in ASTs	Defect cause by calling wrong words (class, function, and variable) and LOC with more words is likely to be more complex	Number of words in LOC became less informative due to refactoring code and code optimization process often occurred as software development reached maturity across time
Code churn	Information regarding the context of code churn in LOC	Syntax information is often incomplete in code snippets and changes also have different locations for added and deleted lines.	Code churn snippets are project specific features which are rarely or never appear in changes from different project. Therefore, it is unsuited for cross project defect prediction.
		Commit metadata	
Keywords in commit message	Number of keywords occurrence in a commit message	Occasionally, developers write defect identifier in a commit message and more keywords indicate more complex changes	Number of keywords are limited in case of having a short message and defect identifiers not always written on commit logs
Authorship	Developer information regarding history commit activities	Developer's unique defective change patterns possibly to be captured	Insignificant for software development with limited developer collaboration

Table 8: Previous studies of JIT-SDP using semantic-based features

Context/Research		Jiang *et al.* (2013)	Xia *et al.* (2016)	Wang *et al.* (2018)	Hoang *et al.* (2019)	Pornprasit & Tantithamthavorn (2021)
Code snippets	Bag of words	✓				
	Code churn	✓	✓	✓	✓	

Commit metadata	Commit message	✓	✓	✓	✓
	Authorship	✓	✓		✓

2.4.3 Noisy Data

To determine whether a file or change is defective or clean, many researchers examine the defect database and version archives for open-source systems (Wahono, 2015). However, recent studies (Bird *et al*., 2009; Hosseini *et al*., 2018b) have demonstrated that data gathered from mining software repositories contain a high level of noise. The presence of noise in the data adversely affects the accuracy of defect prediction. Defect dataset noises indirectly influence prediction performance in a significant manner. In fact that the prediction performance decreases significantly when the dataset contains more than 35% of both false positives and false negatives (Kim *et al.*, 2011), especially for machine learning algorithms that lack robust noise resistance. In order to mitigate noise in input data, researchers have employed noise handling approaches for several sources of noise. We have categorized noise handling methods into three categories, as shown in Figure 19, as follows: 1) removing outliers, 2) reducing mislabelled data, and 3) resampling imbalanced class data.

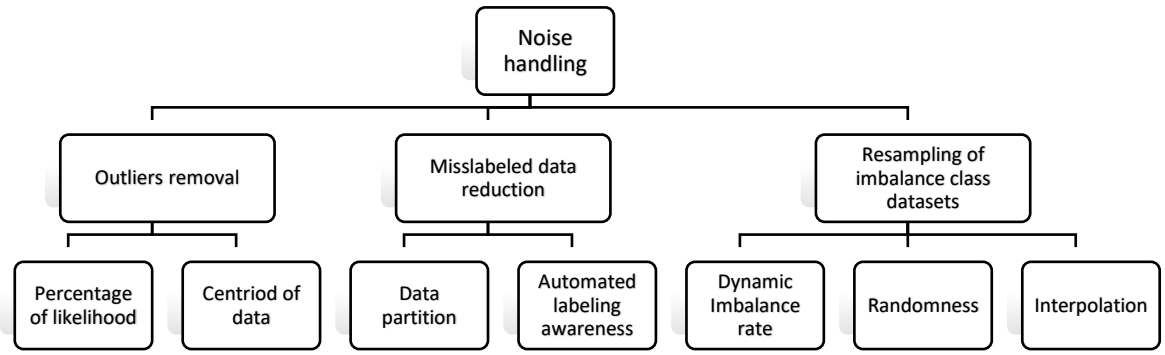


Figure 19: Categories of noise handling approach

1) Removal of Outliers

In a defect dataset, outliers are the instances that significantly deviated from the general observations of the dataset. The outliers are considered as noisy instances that seriously downgraded the performance of a defect classifier algorithm. Nevertheless, the removal of instances by considering them as outliers negatively leads to biased learning due to significant loss of defect information carried by those instances (Tang and Khoshgoftaar, 2004). Therefore, careful detection and removal of outliers are particularly important in the SDP since it is an uncertainty line between the outlier instances and normal instances. The detection of outliers is mainly done based on two criteria in the existing studies, which are the percentage of likelihood of instances to be outliers and data dispersion from the centroid of data distribution. Details of these two criteria are as follows:

- ***Percentage of likelihood*** measures the possibility of an instance becoming an outlier by evaluating the likelihood measurement such as Euclidean distance between the test instances with neighboring opposite class instances. The test instances are tagged as outliers when the percentage values reach a specific threshold.
- ***Centroid of data*** guides the selection of outliers according to the distance between the test instance and the centroid of its closest large cluster generated by a clustering algorithm. The top-ranked instances in each class are tagged as potential outliers.

To remove outliers in defect datasets, (Tang & Khoshgoftaar, 2004) provided Clustering-based Noise Detection (CBND) that utilized the centroid of data distribution to find the outliers. The outliers are identified according to the distance between test instances and the centroid of the nearest cluster which is generated from K-means

algorithm. The algorithm consists of two hyperparameters which are the noise factor or threshold value and the number of clusters within data classes. This algorithm selected the top-ranked samples for each cluster as the potential outlier instances. The selected samples are according to large difference in distance from the centroid.

In addition, Kim *et al.* (2011) proposed Closest List Noise Identification (CLNI). It is capable to detect outlier instances and it is possible to eliminate them for cleaner defect datasets. CNLI algorithm measures the likelihood of instances in the datasets to be the opposite label. The percentage of likelihood is based on the distance of all instances that are close to the examined instance. The distance ratio between the neighboring instances and the examined instance is calculated. The examined instance is considered as a noise (outliers) instance in case of the percentage of likelihood exceed a specified threshold value. Thus, CLNI is capable to remove the outliers and excluding them from consideration for training data.

Chen *et al*. (2016) proposed neighbor cleaning learning (NCL) to eliminate outliers from the majority class of defect datasets by identifying class overlap instances. The main idea is to identify the potential overlapped instances from majority class by locate the nearest neighbor of opposite labeled instances. The overlapped instances are easily identified especially in the case of the larger number of neighbor instances and the opposite instances are loosely clustered. It also utilized Euclidean distance to evaluate the difference between the test instances with neighboring instances. The removal of overlapped instances that consider the outlier instances has a greater impact on imbalanced datasets.

In the context of JIT-SDP, the effect of outliers is increasing concern due to potentially generating more false alarm results, especially in the case of adopting oversampling approaches. Surprisingly, it is found that only recent works by Wang *et*

al. (2018) utilized an outlier removal technique by employing CLNI. One of the possible explanations for this lack of application is the outlier tolerant capabilities of advanced classifier algorithms such as ensemble learning and deep learning approaches. Nevertheless, outlier detection and removal techniques still are relevant to reduce the potential impact of false alarm instances in training data.

2) Mislabelled data reduction

Identifying defect-inducing changes from historical changes in a software project is a key task for JIT-SDP. It is inefficient to manually identify defect-inducing changes in the projects with a large number of historical changes. Thus, automated labeling approaches especially SZZ algorithms are preferable to identify defect-inducing changes. Prior studies observed that conventional automated labeling is affected by a large amount of noise (e.g., changes that only modify code comments or blank lines), which results in mislabeled changes (Fan *et al*., 2019; Herzig *et al*., 2013). The mislabeled changes include false positives and false negatives. In this context, false positives refer to changes that do not introduce any defects but are labeled as defective changes, and false negatives refer to the changes that are labeled as clean instead of defective changes. The mislabeled changes contributed to the wasted of the developer's effort to inspect false-positive changes. Mislabeling data severely impacts the defects count and overall performance of the prediction models (Li *et al*., 2018). It is challenging to obtain defect datasets in JIT-SDP with the absence of mislabeled changes that truly clean datasets without noise. Many prior studies (Herzig *et al*., 2016; Pascarella *et al*., 2019; Trautsch *et al*., 2020) utilized different mislabeled data reduction by considering two contexts: which are data partition and awareness in automated labeling. Table 9 provides some brief information on the context of mislabeled treatment.

With respect to JIT-SDP, data partitioning during data pre-processing is associated with tangled code changes which frequently occur during the submission of code changes to revision control systems. The tangled changes contribute significantly to the difficulty of identifying lines of code based on the defect identifier in the issue report. In addition, tangled changes are found to increase the number of files associated with defects. Since the number of files is affected by the number of defects, ignoring tangled changes in the defect datasets generates an amount of noise that substantially impacts the estimated defects. Accordingly, Herzig *et al.* (2016) suggested dividing code change sets into smaller pieces to untangle the code changes. Each partition contains code changes with closely related instances. Data dependencies between code changes are considered when determining whether they are related or not. With the use of untangling algorithms, it is possible to simplify the process of untangling changes automatically, which reduces the significance of noise generated by tangled changes.

Prior JIT-SDP studies utilized SZZ algorithm (Śliwerski *et al*., 2005) to automatically generate the label for defect datasets. The standard version of the SZZ algorithm comprises the following steps: 1) Identify defect-fixing changes, 2) Identify buggy lines, 3) Trace potential defect-inducing changes, and 4) Filter incorrect defect-introducing changes. Due to the foundational role of the SZZ algorithm, researchers have raised concerns about the quality of SZZ-generated data (Fan *et al.*, 2019). SZZ algorithm is known to be affected by a large amount of noises, which results in false labeled data. Noises in the automated data labeling process are generated due to modifications such as in code format, refactoring, comment lines, and meta-changes. The variants of the SZZ algorithm in handling these noises are explained as given in Table 10.

Over the past recent years, there is little attention to the reduction of mislabeled data approaches. Although most of the studies have utilized automated labeling techniques by SZZ algorithm, a few studies (Herzig *et al*., 2016; Pascarella *et al*., 2019; Zhu *et al*., 2020) utilized mislabeled treatment approaches in cleaning false labeled instances in the generated datasets. In fact, the impact of mislabeling insignificantly affected the overall precision of the constructed defect prediction model (Tantithamthavorn *et al*., 2019). Nonetheless, ignoring these mislabeled data lead to additional waste of inspection effort, and the interpretation of the model is also negatively affected (Fan *et al*., 2019).

Table 9: Factors of mislabelled data treatment

Factors		**Context**	**Consideration**
Data partition		Developers often submit a single commit with multiple context/task changes at once. Consequently, overlapped files or code are found for each of these tangling changes. Thus, making the confusion about the actual label of these changes	Converting tangled changes into smaller partitions of data based on the context of changes (e.g. data dependency, operations and commit keywords) reduces the complexity of changes.
Automated labeling	Non-informative line	Format/indentation modifications and comment lines can cause automatic labeling to misidentify these lines as part of the defective lines.	Format/indentation modifications must be ignored as the behaviors of the code are unaffected by these lines. Thus, reducing the false positives instances
	Meta-changes	Meta-changes are branch change (e.g. copying code in a branch to a new branch), properties changes (e.g. file properties modification such as permission), and branch merge (e.g. from one branch to another). The source code in this modification is unchanged. Thereby, meta-changes are mistakenly regarded as defective lines in automated labeling	Ignoring meta-changes reduces the possibility of false positives.
	Refactoring	in case of defective changes are identified incorrectly as a part of defects due to the impact of refactoring modification (e.g. changes in function/file name). Refactoring modification is unlikely to involve defects fixing changes.	Refactoring lines should be ignored to avoid mislabelling defect-inducing lines

Table 10: Variant of SZZ algorithm

Variants of SZZ	**Biases factors**	**Identification of buggy lines**	**Identification of defective changes**
Standard SZZ (Śliwerski *et al*., 2005)	N/A	Lines of code related to defect-fixing changes are considered buggy lines	The latest code changes that involve modifications before defect-fixing changes are considered defective changes
AG-SZZ (Kim *et al.*, 2006)	Non-informative line	Non-informative lines related to defect-fixing changes are excluded from the identification of buggy lines	An annotation graph is used to record modification series of lines of code. A depth-first search algorithm is used to identify the defective changes from the annotation graph.
MA-SZZ (Da Costa *et al.*, 2017)	Meta-changes	Non-informative lines and those lines involving refactoring modification related to defect-fixing changes are excluded from the identification of buggy lines	Improve the AG-SZZ version by considering the meta-changes associated with buggy lines
RA-SZZ (Neto *et al.*, 2018)	Refactoring modification	Non-informative lines and those lines involving refactoring modification related to defect-fixing changes are excluded from the identification of buggy lines	Improve the MA-SZZ version by considering the refactoring modification associated with buggy lines

3) Imbalance learning from class distribution

In general, JIT-SDP datasets consist of one big problem which is a large amount of training data needed to train the model. Unfortunately, the required data is unavailable

in the initial phase of software development. For this reason, the available datasets are known to have a highly skewed distribution (Chen *et al*., 2016). In this situation, the clean class is dominant in the data set as compared to data of defect class data. The imbalance in the class distribution of data leads to biases in the learning of the prediction model toward the data of the clean class. Consequently, the prediction model yield misclassification results.

The dataset usually is prepared from the number of clean and defective classes of a software project. In prior studies, researchers used a lot of balanced and imbalanced datasets to predict the defect. The performance of the defect prediction model by using balanced and imbalanced datasets makes a big impact on software testing. The class imbalance problem is well-recognized as one of the major causes of the poor performance of software defect prediction models (Song *et al*., 2018). In summary, Figure 20 shows the imbalance learning can be classified into resampling techniques, classification learning, and ensemble learning.

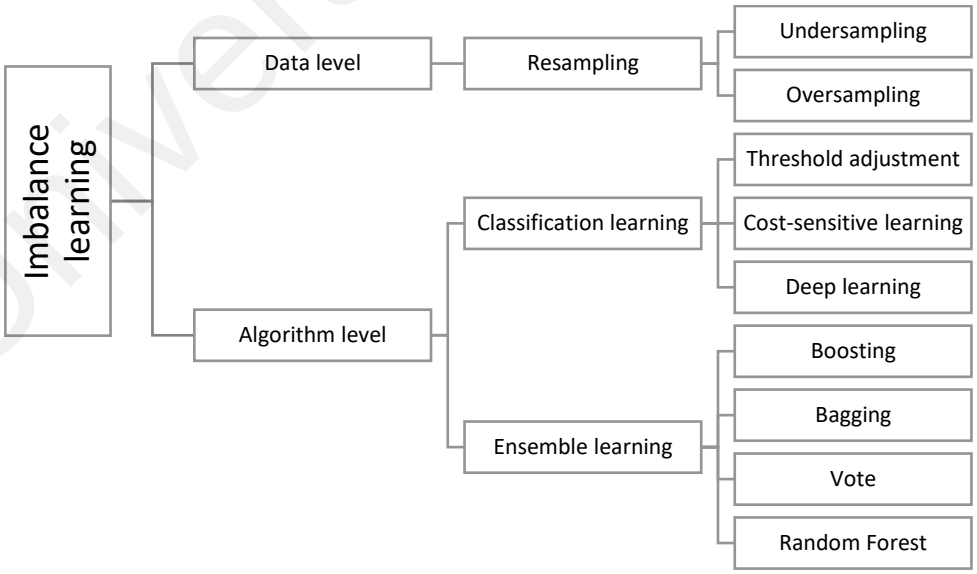


Figure 20: Classification of imbalance learning in SDP

Although several class-imbalance learning methods are presented in software defect prediction, there still exists room for improvement. Specifically, the resampling technique techniques usually need to remove or append lots of samples to achieve the class-balanced state, which leads to addition of noise due to insignificant synthetic instances. For cost-sensitive learning methods, how to set the cost value is a problem not yet being effectively solved. In ensemble learning techniques, how to effectively guarantee and utilize the diversity of individual classifiers is not addressed carefully.

Majority of the imbalance learning adopted by previous JIT-SDP frameworks are resampling methods. (Chen *et al*., 2018; Huang *et al*., 2019; Kamei *et al*., 2013; Kondo *et al*., 2019; Qiao & Wang, 2019; Wang *et al*., 2018; Yang *et al*., 2015, 2017; Yang *et al*., 2016). However, their works limit to the weakness as shown in Table 11. Alternatively, Cabral *et al.* (2019) utilized ensemble-based imbalance learning focusing on data drift problems which are also known as class imbalance evolution. Data drift is a change in the input data generation process, affecting the underlying probabilities of the data. These previous studies showed the effectiveness of the proposed oversampling technique for imbalanced data. Existing imbalance learning, however, do not consider overlapping data within class distributions. This results in ineffective performance as data overlapping prevents the identification of suitable regions for selecting hard-to-learn samples.

Table 11: Imbalance learning strategies in SDP

RELATED WORKS	STRATEGY	FOCUS PROBLEM	STRENGTH	WEAKNESS
(Cabral *et al*., 2019)	Ensemble-based learning – bagging oversampling technique	Data drifting - data imbalance evolution due to the evolution or	Consider class imbalance evolution Suitable for JIT-	Base machine learner choices highly influence the sampling

		maturing process in software project	SDP	results
(Huda *et al*., 2018)	Ensemble-based learning – three bases oversampling technique incorporate with random forest algorithm	Bias of conventional sampling approaches	Reduce false-negative rate in imbalance data and improve cost-sensitive classification performance	Base machine learner choices highly influence the results
(Bennin *et al*., 2018)	Resampling – MAHAKIL (oversampling)	High false positives and less diverse data in oversampling	Generate new samples that have the characteristics of previous instances Diversity within the data distribution	Risk of duplicated data sampling instances with the same output value
(Jing *et al*., 2017)	Ensemble-based and cost-sensitive learning - ISDA	Solve normal imbalance problem	Suitable for both within and cross-version imbalanced data	Ignore the diversity of data distribution
(Ryu *et al*., 2016)	Cost-sensitive learning - multi-objective	multi-objective cost in imbalance learning	Maximize defect detection and minimize false alarm probability	Optimization algorithm choices highly influence the results
(Chen *et al*., 2016)	Ensemble-based learning – boosting with under-sampling (AdaBoost)	Solve normal imbalance problem	Improve random under-sampling results	Inconsistence results due to random sampling
(Wu *et al*., 2016)	Cost sensitive learning - cost-	Solve imbalance problem in	Maximize type II misclassification	Suitable only for module level

	sensitive local collaborative representation	collaborative representation classifier-based SDP		defect prediction
(Siers & Islam, 2015)	Ensemble-based learning – SMOTE incorporates with decision forest algorithm (oversampling)	Balancing majority and minority class instances for lower classification cost	Minimize classification cost	Inconsistence results due to random sampling
(Liu *et al*., 2014)	Cost-sensitive learning - Two stage cost learning (classification and features selection stages)	Solve normal imbalance problem	Improve the efficiency of both classification and features selection cost	Ignore the diversity of data distribution

2.5 Resampling in Imbalance Class Distribution

The performance of the defect prediction model by using balanced and imbalanced data sets makes a big impact on the discovery of future defects. The class imbalance problem is well-recognized as one of the major causes of the poor performance of the prediction models. Many preprocessing approaches are proposed to solve the class-imbalance problem, particularly by resampling approaches (Song *et al*., 2018), which are classified into under-sampling and oversampling approaches as illustrated in Figure 21. The under-sampling selects only a subset of majority class instances to ensure the equality of the instances for the majority and minority classes in model training. Oversampling generates more synthetic/duplicated instances for the minority class to balance with the number of instances in the majority class. In the literature, resampling data class distribution is conducted which involves several sampling factors as summarized in Table 12.

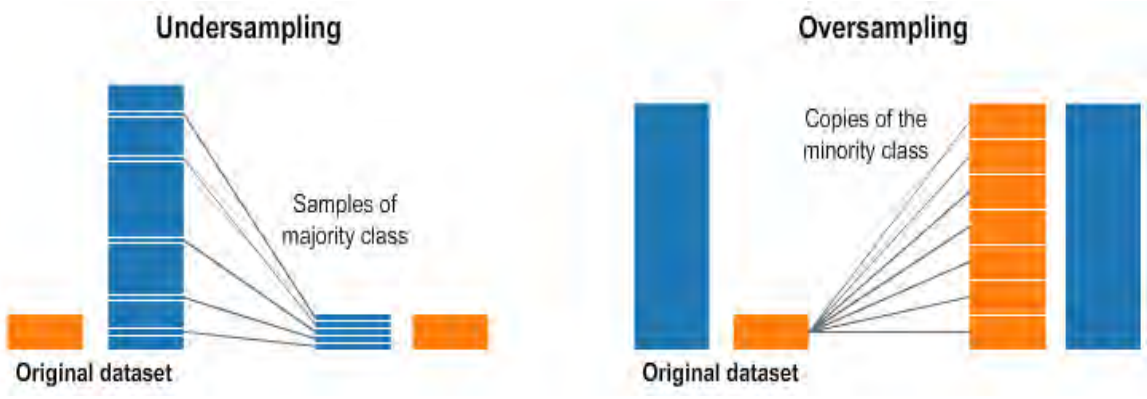


Figure 21: Resampling imbalance dataset into balanced datasets

Table 12: Factors of consideration on resampling imbalanced datasets

Factors	Description	Justification
Dynamic imbalance rate	Change in resampling rate throughout time	Imbalance ratios are found to be dynamic rather than a fixed rate for a whole dataset.
Randomness	Selection of random instances from a large population of defect/clean class	Equal chances for each instance to be selected for resampling and unlikely to be biased representation
Sequential-based evaluation	Selection of instances based on sequential evaluation	Exploring the significant level of each individual instance helps produce more quality samples.
Interpolation	Constructing new instances within the range of a discrete set of a few instances	New generated instances within a neighborhood of existing instances improve the generalization capacity of classification

Random under-sampling is the simplest and most common approach for resampling in imbalance defect datasets (Chen *et al*., 2016; Kamei *et al*., 2013). This approach ensures the majority class instances (non-defective changes) are randomly removed until the number of instances for the majority and minority classes is at the same level. It is known that the under-sampling approach provide a compact balanced set of training

data with reduction in cost for learning process. Conversely, random oversampling adopts the strategy of simply duplicating instances to increase the number of defective change instances until reaching the number of non-defective change instances. Applying random over-sampling, however, results in a higher risk of overlapped/duplicated labels for defect datasets. Despite the limitations of these random based resampling, the application in preprocessing imbalance defect datasets is easier and computationally inexpensive compared to other approaches such as cost-sensitive learning and hybrid techniques.

Synthetic minority over-sampling technique (SMOTE) is an improved technique of standard random oversampling (Chawla *et al*., 2002). It is a process of interpolation that synthesize new instances for the minority class. The new instances are created using random interpolation between several instances within a defined neighborhood. Thus, the generated instance is based on features value and their relationship instead of only considering the data distribution. SMOTE is considered as a foundation approach for the research community in class imbalance classification. For this reason, many extensions and alternatives are suggested since its release to increase its success in various scenarios (Fernandez *et al*., 2018).

Liu *et al*. (2008) introduced an under-sampling approach based on the iteration process by considering sequential evaluation to guide the sampling process for subsequent classifiers. The proposed approach sampling multiple subsets of majority class instances and training each of these subsets based on the ensemble classifier approach. For each iteration, the majority of class instances that are correctly classified by the current iteration are removed from consideration for the next iteration. Since several subset samples provide more details than a single subset, this approach provides better use of the majority class than traditional random under-sampling. Thus, an

efficient process of downsizing the majority class instance is achieved due to the fact it requires a shorter training time.

Cabral *et al*. (2019) recently proposed oversampling rate boosting (ORB) to cope with class imbalance evolution. They proposed adjusting the resampling rate over time rather than always using 1:1 ratio of the balanced defect dataset for resampling. Since the resampling rate does evolve throughout time, the proposed oversampling approach automatically readjusts the resampling rate according to the ratio of current instances class distribution. For example, in case of the prediction is considered biased toward the non-defective class, the resampling rate of the defect class need to be adjusted accordingly. The proposed approach is specifically useful in resampling defect datasets for online learning-based machine learning framework.

Some existing works adopted more advanced resampling approaches, such as under-sampling incremental-based evaluation (Chen *et al*., 2016), ORB (Cabral *et al*., 2019), and SMOTE (Chawla *et al*., 2002; Zhu *et al*., 2020). Despite the fact that greater predictive impact for resampling on the minority class than on the majority class, most of the recent works pre-processed the imbalance dataset by an under-sampling approach. The reason is attributed to the fact of under-sampling requires shorter training time and a simpler process compared to oversampling (Liu *et al*., 2008). Consequently, prior JIT-SDP research typically used under-sampling rather than oversampling (Zhao *et al*., 2022).

2.6 Oversampling for Imbalanced Datasets

Oversampling is an efficient and common technique for resampling imbalanced data. The purpose of oversampling is to make sure that the distribution of classes is balanced by increasing the number of samples of the minority class. The most common

method for predicting software defects is to oversample the minority (defective) samples. In practice, oversampling is useful for improving classification performance on datasets with an imbalance distribution. In this review, we summarize the components of oversampling techniques for SDP as shown in Figure 22. To achieve the desired distribution of data within SDP datasets, oversampling is composed of several components. The components are as follows:

i. Factors: consideration factors for oversampling focuses on data distribution
ii. Distribution analysis: analysis of individual instances according to measurements of the relationship between them.
iii. Intra-clustering: partitioning approaches within data distribution based on data classes
iv. Parent selection: selection of data template or guidance for interpolation of new synthetic data
v. Interpolate: interpolation techniques for the generation of new synthetic data

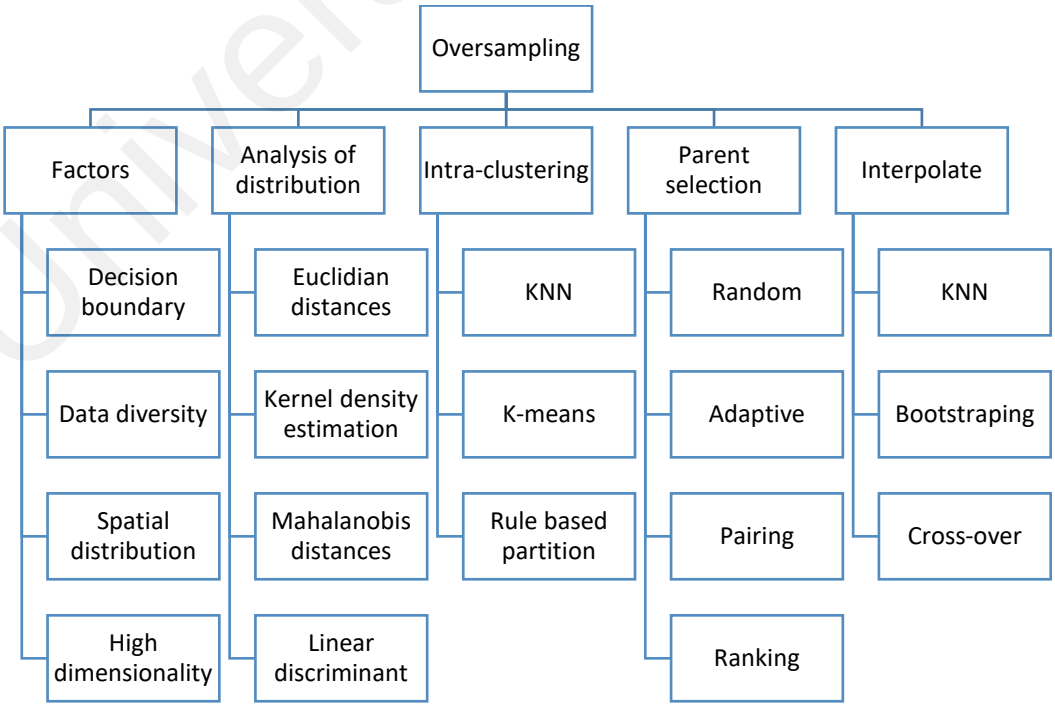


Figure 22: Taxonomy of oversampling in SDP

The fact that imbalanced class distributions adversely affect machine learning based models is well-established. Although deep learning proved to be a powerful tool in SDP, class imbalance distributions negatively affect the performance of machine learning algorithms since most classifiers are trained by overfitting on majority classes. In JIT-SDP, various oversampling techniques are designed to improve the performance of minority classes. Table 13 summarizes recent work related to oversampling.

Reviewing the oversampling of SDP as illustrated in Table 13, it is evident that most of the prior techniques attempt to create new data near the boundaries of the distribution. They assert that instances near the decision boundaries are likely to be more informative. For this reason, the empty spaces in the data distribution are less covered. On the contrary, several studies (Bennin *et al.* 2018; Li *et al*., 2019; Gong *et al*., 2019; Zhang *et al*., 2021) focus on increasing the occupied spaces in data distribution by increasing the diversity of data. The diversity of data within the distribution needs to be diverse concerning to minimize intraclass imbalance, especially for distributions with weak generalizability. Nevertheless, it is still true that diverse data widen the decision boundaries with respect to distribution. For this reason, some oversampling attempts to consider more than one factor such as spatial distribution and multidimensionality in the generation of new data (Liu *et al*., 2020; Feng *et al*., 2021; Zhang *et al*., 2021). For spatial distribution, samples within dense distributions are assumed to be difficult to learn, thus introducing duplicates into the original datasets. In the case of oversampling that is specifically associated with spatial distribution, the process highly dependent on the data partitioning algorithm and narrowing of boundary boundaries. For multidimensionality, potential downsides are the tendency to ignore the high informative instances and the bias toward high correlated features. Therefore, consideration of contributing factors remains a challenge for oversampling, especially for SDP data.

Table 13: Oversampling techniques in defect prediction

Technique	Focus	Distribution analysis	Intra partition	Data selection	Interpolation
SMOTE (Chawla *et al*., 2002)	Decision boundary	Euclidian distance	None	Random	KNN – nearest interpolate
Borderline-SMOTE (Han *et al*., 2005)	Decision boundary	Euclidian distance	None	Random - select instances close to borderline	KNN – nearest interpolate
ADASYN (He *et al*., 2008)	Decision boundary	Euclidian distance	None	Random - select instances hard to learn	KNN – nearest interpolate
MWMOTE (Barua *et al*., 2014)	Decision boundary	Euclidian distance	KNN	Random – select instance within cluster close to borderline	KNN – nearest interpolate
ROSE (Lunardon *et al*., 2014)	Decision boundary	Kernel density estimation	None	Random	Smoothed bootstrapping
MAHAKIL (Bennin *et al*., 2018)	Data diversity	Mahalanobis distance	Rule based	Pairing – inheritance at different level of parent	Cross-over interpolate – genetic algorithm
A-SUWO (Choirunnisa *et al*., 2018)	Decision boundary Spatial distribution	Linear discriminant analysis	KNN	Random - select instances within clusters that have minimum overlapped labels	KNN – nearest interpolate
ACWO (Zha *et al*., 2018)	Decision boundary	Euclidian distance	K-means	Adaptive - selection on centroid points of large	KNN – nearest interpolate

				clusters	
K-means based oversampling (Li *et al*., 2019)	Data diversity	Euclidian distance	K-means	Random – select within clusters	KNN – nearest interpolate
KMFOS (Gong *et al*., 2019)	Data diversity	Euclidian distance	K-means	Pairing – select two instances from different clusters	Cross-over interpolate – genetic algorithm
SDSMOTE (Liu *et al*., 2020)	Spatial distribution Data diversity	Euclidian distance	None	Ranking – select high difficulty instances	KNN – nearest interpolate
DVS (Zhang *et al*., 2020)	Data diversity High dimensionality	Kernel density estimation	None	Random – select according to eigenvalue of variance density	Cross-over interpolate – genetic algorithm
COSTE (Feng *et al*., 2021)	Data diversity High dimensionality	Multivariate correlation	None	Ranking – select instances with low complexity	KNN – nearest interpolate
K-means MAHAKIL (Zhang *et al*., 2021)	Spatial distribution Data diversity	Mahalanobis distance	Rule based K-means	Pairing – inheritance pairing	Cross-over interpolate – genetic algorithm

2.6.1 Impact of Oversampling

Simple oversampling adds duplicated samples from the original dataset, resulting to overfitting and numerical stability problem. In order to rebalance the class distribution of imbalanced data, advanced oversampling is required to avoid duplicating samples. The impact of advanced oversampling techniques depends on experimental settings, comprises of evaluation metrics, modeling classifiers and imbalanced ratio.

Evaluation metrics are critical in assessing classification performance by oversampling and guiding classifier modelling of defect prediction in imbalanced class data. For instance, the accuracy metric for an imbalanced classification problem is dangerously deceptive with respect to bias classification. This is because the accuracy metric is insensitive to datasets with a skewed distribution such in JIT-SDP datasets. Furthermore, as pointed by Tantithamthavorn *et al*. (2018) defect prediction models through oversampled datasets produce better Recall improvement but low in AUC performance. The issue arises due to the evaluation preference bias performance towards cases that are inadequately represented in the available data samples (Branco *et al*., 2016). Consequently, diverse evaluation metrics are necessary when working with an imbalanced classification.

In addition, the performance of oversampling techniques is contingent on the selection of modelling classifiers for JIT-SDP, which are based on supervised, unsupervised, and semi-supervised machine learning. Without transformation by oversampling in imbalanced data, machine learning algorithms learn more on the traits in the clean class data at the expense of learning the traits in the defect class data. Due to the fact that oversampling adds duplicate or similar data samples to the original, training datasets for the classifier eventually contain multiple overlapped samples. Later, resulting in overfitting for machine learning. Thus, oversampling especially in

within-project defect prediction models perform better than cross-project models. Classifier techniques like logistic regression, k-nearest neighbor and support vector machine are sensitive to oversampling techniques which affect the interpretation of defect prediction models. In other hand, random forest, and neural networks tend to be less sensitive (Tantithamthavorn *et al*., 2018). To conclude, the influence of oversampling on the interpretation of defect prediction models is highly dependent on the classifier techniques employed, indicating that oversampling techniques must be avoided when deriving knowledge or defect patterns from defect prediction models.

Highly imbalanced ratio is prominent in defect prediction datasets. Tantithamthavorn *et al*. (2018) found that 8% of the defect datasets consist of a defective ratio between 45%-55%. Indicating that only a small portion of defect dataset is based on small imbalanced ratio. In a dataset with highly imbalanced classes, if the classifier always predicts the majority class without any feature analysis. The results will still have a high rate of accuracy, which is obviously deceptive. In respect to oversampling in highly imbalanced datasets, techniques such as SMOTE, random oversampling and ROSE work well with different problems to a certain extent. Highly imbalanced datasets correlated with high false alarm rate during modelling (Menzies *et al*., 2007). Generating as much diverse synthetic data as possible restricted within the region of the defect class provides high recall and low false alarm rate (Bennin *et al*., 2018). When the oversampling technique generates synthetic samples that are widely dispersed but appropriately located within the decision boundary or region of the minority class, the false rate is reduced without compromising overall performance. In conclusion, the performance of oversampling varies greatly depending on the imbalanced ratio dataset.

2.7 Modeling Approaches for Defect Classifier

A variety of machine learning approaches for JIT-SDP studies is found in the literature. As reported by Catal *et al.* (2011), machine learning is proved as the most successful approach compared to statistical approaches. Machine learning approaches in modelling JIT-SDP exist based on either prediction of defect proneness of code changes (classification) or effort-aware prediction (regression). For prediction of defect proneness, JIT-SDP classifies the given code changes into defective or clean classes. On the other hand, the prediction based on regression refers to the prediction of a certain number of defects found in the given code changes information. This type of prediction model assigns an estimated number of defects for each of the code changes instead of classifying them into defective or clean change classes.

Modelling of JIT-SDP involves of formulating prediction of defect proneness or defect inducing at the granularity of code changes. Prior JIT-SDP typically utilised machine learning with batch learning to formulate such a model. Training instances are not required to be arranged sequentially in batch learning. Consequently, some studies (Cabral *et al*., 2019; Tan *et al*., 2015) contend that batch learning is unrealistic and that an alternative is to model online learning settings. Online learning necessitates the arrangement of training and testing instances in accordance with the arrival date of data in version control systems. Modelling of JIT-SDP consists primarily of two types of prediction projects: within-project prediction and cross-project prediction.

In general, machine learning algorithms are divided into three categories: supervised, semi-supervised, and unsupervised techniques. Supervised learning leverages the usefulness of defective or clean label information as the training datasets to build the prediction model. The techniques operate with supervision provided using the outcome of each training instance. In contrast, unsupervised learning enables the

development of the JIT-SDP model in the absence of defect data. Unsupervised learning ensures the modelling of JIT-SDP is done without requiring any labelling of code change information. Semi-supervised learning is the combination of supervised and unsupervised learning. The learning process only uses small amounts of defect data while utilizing a greater number of unlabelled code changes information. This technique is usually used when the prediction is related to the new software projects that have few version releases or less defect information. Table 14 summarize some of the works in JIT-SDP based on these categories.

Most commonly choices of modelling in JIT-SDP are by using supervised learning as tabulated in Table 14. Nevertheless, gathering enough data is a challenging process for new projects or projects with limited development history. Data collection is one of the known challenges in the supervised SDP model. Existing works (Yang *et al.*, 2016; Liu *et al.*, 2017) reported that unsupervised models produce a prediction performance superior to most supervised models. Unfortunately, the limitation of the unsupervised model is that it produces many false alarm results and poor prediction performance, especially in terms of F1-score (Huang *et al.*, 2019). Thus, semi-supervised learning is explored for JIT-SDP due to its capability to produce substantial improvement in learning accuracy with a small amount of labelled data. Interestingly, three studies (He *et al.*, 2016; Liu *et al.*, 2017; Li *et al.*, 2020) concerning the semi-supervised model in the context of JIT-SDP.

JIT-SDP looks at a wide range of classifier techniques, from standalone learners to ensemble-based learners, to find the best models. Commonly used standalone classifier includes Logistic Regression (Chen *et al*., 2018; Kamei *et al*., 2013; Taba *et al*., 2013), Naïve bayes (Jahanshahi *et al*., 2019), Support Vector Machine (Amasaki *et al*., 2021), Decision Tree(Yang *et al*., 2017), and Neural Network (Qiao & Wang, 2019). Whereas for ensemble-based learners range from single ensemble learner such as Random Forest (Sikic *et al*., 2021) to multi-layer ensembles (Wang *et al*., 2016; Yang *et al*., 2017).

Table 14: Machine learning in the JIT-SDP model

Types of classifiers	**Reference**	**Types of learning**	**Algorithm**
Standalone model	(Kamei *et al.*, 2013)	Supervised learning	LR
	(Singh and Chaturvedi, 2013)	Supervised learning	LR and SVR
	(Taba *et al.*, 2013)	Supervised learning	LR
	(Yang *et al.*, 2015)	Supervised learning	LR
	(Yang *et al*., 2016)	Unsupervised learning	-
	(Liu *et al.*, 2017)	Unsupervised learning	-
	(Chen *et al.*, 2018)	Supervised learning	LR
Ensemble model	(Jiang *et al.*, 2013)	Supervised learning	LR, NB and ADTree
	(Herzig *et al.*, 2016)	Supervised learning	kNN, LR, RP, and SVM
	(He *et al.*, 2016)	Semi-supervised learning	RF
	(Xia *et al.*, 2016)	Supervised learning	ADTree
	(Yang *et al.*, 2017)	Supervised learning	RF
	(Ozcan and Tosun, 2018)	Supervised learning	IBK, J48, LR, NB, and RF
	(Wang *et al.*, 2018)	Supervised learning	NB and LR
	(Qiao and Wang, 2019)	Supervised learning	NN
	(Kondo *et al.*, 2019)	Supervised learning	LR and RF

	(Hoang *et al.*, 2019)	Supervised learning	CNN
	(Huang *et al.*, 2019)	Supervised learning	CBT+
	(Li *et al*., 2020)	Semi-supervised learning	Ti-training
	(Zhu *et al*., 2021)	Supervised learning	DEA-CNN
	(Zheng *et al*., 2021)	Supervised learning	RF

2.7.1 Impact of Classifier Techniques

It is evident that different machine learning methods used in building the prediction model resulted in differences in the changes predicted as defective. As each classifier identifies distinct subsets of defects, certain features should be examined to determine if certain features are compatible with specific classifiers. Classifiers perform significantly better when combined with particular sets of features, such as reduced features or uncorrelated features (Bowes *et al*., 2018). Through the selection of the data features most relevant to the classification problem, it is possible to reduce the amount of noise in the data. The result makes it easier for the classifier to learn from the data. The selection of the most relevant features ensures that a model is less likely to overfit training samples, and more generalizable to new data. Furthermore, the selection of features impacts the effectiveness of classifiers by simplifying models. In addition to faster training and improved interpretation capability, the selection of features is also an optimization problem. Regardless of the selection approaches chosen, it is important to note that feature selection is an optimization problem. The result implies no guarantee that an optimal subset of features has been identified. A robust subset of features that performs well on the training data, however, requires careful tuning in order to achieve optimal subsets.

Effective prediction models require fine-tuning of the classifier. The purpose of parameter tuning is to find the optimal parameter values necessary to obtain the most optimal configuration of a classifier. The tuning process involves trying different combinations of parameters to determine the combination that achieves the highest level of accuracy for training data. Mahmood *et al*. (2018) observed that, despite tuning the parameters of the prediction models to improve performance, previous studies of JIT-SDP have generally failed to account for the tuning process. Nonetheless, tuning classifiers improves the overall performance of the prediction model significantly (Fu *et al*., 2016; Menzies *et al*., 2008). Classifiers with many parameters have an adverse effect on classification performance, which necessitates a more cautious selection of parameter values. In instances where there are a large number of datasets at scale, automatic tuning of classifiers is preferable to manual tuning. As a result of the classifier tuning analyses, it is indicated that attention should be focused on enriching the classifier optimization process to reach a more accurate JIT-SDP model.

2.8 Effort-Aware Model

Through the JIT-SDP model, the developers are capable of easily assigning the available test resources to defective parts to enhance the quality of software in the early stages of the development life cycle. For instance, if only 20% of the testing resources are available, the developers concentrate these testing resources on inspecting and fixing software parts that are more vulnerable to defects. Therefore, this provides an opportunity to deploy high-quality, low-cost, and maintainable software in a given time, resource, and budget. For this reason, it became a popular research topic in the software engineering field.

In the context of JIT-SDP studies, the effort-aware model refers to ranking the predicted software defect proneness according to a certain allocation of QA efforts. The

effectiveness of an effort-aware model of JIT-SDP is important to help the developers find more defects with less effort. As a result, the developers allocate limited testing or inspection resources to the most defect-prone code changes with the help of the effort-aware JIT-SDP model. To simplify, effort-aware models are considered as a direct extension of JIT-SDP. From the view of practice, it is more realistic and useful to apply effort-aware JIT-SDP models in the actual software development. Subsequently, improves production efficiency and quality, and reduces the development cost and software risk (Li *et al*., 2018).

Over the last decade, a few numbers of JIT-SDP studies mainly focus on improving the efficiency of effort awareness. Table 17 shows the related works on the effort-aware model of JIT-SDP. Since effort-awareness in JIT-SDP started in the year 2013, the application of the approaches is limited to a few open-source projects and not generalized well with other software projects. They used *ACC* and *Popt* for performance measurement. These metrics consider the inspection effort uniform for every LOC. In practice, however, one change in the LOC of a complex file requires more inspection effort compared to those changes that happen during initial code development. Therefore, the indication of these two metrics insufficient to show actual reflection of the performance of effort awareness for the JIT-SDP model. In addition, the previous effort-aware model has only a few prediction ranking factors such as defect density, risk of defect, size of changes, and the ratio between benefit and cost. These factors are used to prioritize which changes must be inspected first while considering limited resources. The choice of prediction ranking factors used for the effort-aware model resulting different effort awareness performances. To improve the effort awareness performance in the JIT-SDP model, previous works have employed a multi-objective approach (Chen *et al.*, 2018) and an ensemble-based model (Albahli, 2019; Li *et al*., 2020). Despite these solutions improving effort awareness performance, the trained models only

generalize well with the trained datasets and have no guarantee of finding the optimal balance between prediction accuracy and effort awareness performance. It is a conflict between prediction accuracy and effort awareness performances during the model construction phase of an effort-aware model (Chen *et al.*, 2018). More defects are found when more resources are spent on code inspection efforts.

2.9 Potential of Deep Reinforcement Learning in Software Engineering

Deep reinforcement learning (DRL) techniques are composed of two parts: a deep neural network that learns the state representation of the environment, and a policy network that selects actions. The deep neural network is used to approximate the value function, which is the expected long-term return from taking any action in any state. The policy network or network model uses this approximation along with feedback received after each action taken to learn what sequence of actions will result in maximum rewards.

Algorithms for solving the DRL problems that use models and planning are called model-based algorithms, as opposed to simpler model-free algorithms that are based on trial-and-error learners. Model-based algorithms use the deep neural network to approximate both the reward function and the transition distributions. The learned model is then employed in a variety of ways, including detection and prediction. Two main approaches are used to describe the policy as a planner or to utilize the model to generate synthetic transitions by augmenting the experience replay buffer. Some common model-based algorithms used for DRL include Deep Q-Network (DQN), Double DQN (DDQN), Dueling DQN, A3C, DDPG, TD3, and SAC. Table 16 provides a brief description of the algorithms.

DRL is promising for software engineering. One of the main reasons for its applicability in software engineering is that the field is constantly changing. New features are added to applications all the time, and it is difficult to predict how these will interact with one another. Recent works demonstrate the ability of DRL techniques to learn complex tasks in software engineering as reported in Table 15. These results demonstrate the great potential of DRL for improving software engineering tasks. However, as DRL is non-linear, several challenges need to be tackled for reaching its full potential. First, the design of the DRL environment for the problem simulation is needed to be adjusted properly. Then, a reward policy is required to be tuned for problem specifics. Lastly, hyperparameters for deep learning network need to be optimized according to given features. Thus, more research is needed before DRL techniques are ready to be widely adopted. To exploit the full potential of DRL for software engineering, future research needs to focus on:

i. Developing new architectures and learning algorithms specifically tailored for software engineering tasks.
ii. Investigating how best to represent program data and code structures to enable effective learning.
iii. Studying how different problem domains such as defects prediction, code optimization, and peer review prioritization provide benefits from DRL techniques.
iv. Evaluating the effectiveness of different DRL techniques on large real-world datasets.

In the context of modeling for the prediction, prior machine learning approaches for JIT-SDP are generally based on batch learning (supervised, unsupervised and semi-supervised model) context. Batch learning provides data learning on the entire training

datasets at once to learn the pattern of the introduction of defective changes. However, the reliability of SDP models is not yet sufficiently studied especially in the context of sequential learning. Sequential learning updates the training set incrementally to take advantage of the feedback from each run. In other words, it is an updatable classification. This is appropriate for JIT-SDP since it mimics how code reviews are done in practice. Therefore, it is useful to investigate aspects of sequential learning for improving the performance of the JIT-SDP model. Reinforcement learning is a type of learning which capable of iteratively learning optimal control from sequential data and is still unexplored. This provides an ideal opportunity to explore an alternative approach for the JIT-SDP model.

Table 15: DRL approaches in software engineering

Related works	DRL algorithm	Application
(Kim *et al*., 2018)	DQN	Generate test input for software under test
(Harries *et al*., 2020)	DQN	Functional software testing
(Hu *et al*., 2020)	DQN	Automated penetration testing framework
(Eskonen *et al*., 2020)	DQN	Automated and adaptive GUI testing

Table 16: Examples of model-based methods

Algorithm	Learning methods	Actions preference	Advantage
DQN	Value function	Discrete	Good in sparse rewards in highly dimensional input spaces
Double DQN	Value function	Discrete	Reduce overestimation of DQN
Dueling DQN	Value function	Discrete	Learn which states are advantage according to actions
A3C	Actor-critic	Continuous	Increase the convergence speed
DDPG	Actor-critic	Continuous	Direct policy learning
TD3	Actor-critic	Continuous	Reducing the overestimation bias of DDPG
SAC	Actor-critic	Continuous	Accelerate learning by preventing the policy from a bad local optimum

Table 17: Summary of previous effort-awareness JIT-SDP models

Related studies	**Priority factor**	**Classifier technique**	**Effort awareness evaluation**	**Software project datasets**
(Kamei *et al*., 2013)	Defect density	EALR	Popt = 61 and ACC = 35	Public: Buz, Col, JDT, Pla, Moz, and Pos Private: 5 java projects
(Jiang *et al.*, 2013)	Defect density	LR, NB, and ADTree	ACC = 41	Public: Linux, Pos, Xorg, Eclipse, Luc and Jackrabbit
(Yang *et al.*, 2015)	Defect density	LR	ACC = 51.04	Public: Buz, Col, JDT, Pla, Moz, and Pos
(Xia *et al.*, 2016)	Defect density	ADTree	ACC = 59 and effort-AUC	Public: Pla, JDT, Jackrabbit, Linux, Luc, Pos, and Xorg
(Yang *et al*., 2016)	Risk of defect	Unsupervised Models	Popt = 76.2 and ACC = 49.7	Public: Buz, Col, JDT, Pla, Moz, and Pos
(Liu *et al.*, 2017)	Size of changes	CCUM (code churn unsupervised model)	Popt = 89.3 and ACC = 73.6	Public: Buz, Col, JDT, Pla, Moz, and Pos
(Yang *et al.*, 2017)	Defect density	TLEL (Two-layer Ensemble of Random Forest)	ACC = 70.53	Public: Buz, Col, JDT, Pla, Moz, and Pos
(Wang *et al.*, 2018)	Defect density	LR and DBN + ADTree	ACC = 21.9	Public: Linux, Pos, Xorg, JDT, Luc, and Jackrabbit, Buck, Hhvm, Guava, Skia

(Chen *et al*., 2018)	Pareto front between defective probability and effort for defective changes	MULTI (LR + NSGA-II)	Popt = 88.9 and ACC = 73	Public: Buz, Col, JDT, Pla, Moz, and Pos
(Huang *et al*., 2019)	Size of changes	CBT+ (Logistic regression)	ACC: 35	Public: Buz, Col, JDT, Pla, Moz, and Pos
(Qiao & Wang, 2019)	Benefit-cost ratio (defective probability divided by sum of code churn)	Regression based Neural Network	Popt = 85.3 and ACC= 69.6	Public: Buz, Col, JDT, Pla, Moz, and Pos

2.10 Open Issues in Prediction of Software Defects

Several open issues related to the frameworks of the JIT-SDP model are identified as illustrated in Figure 23. These issues include cross-prediction on heterogeneous metrics, effort-aware prediction, optimization of the model, and latencies in data evolution.

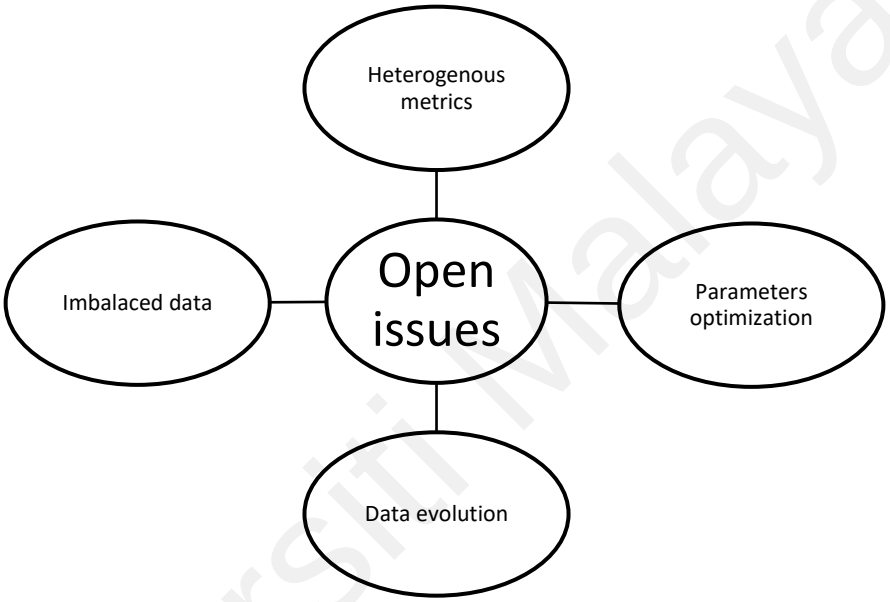


Figure 23: Open issues of JIT-SDP

2.10.1 Prediction of heterogeneous metrics

In practice, new start-up software projects often lack historical defect data. Therefore, researchers have utilized historical defect datasets from other projects to predict defects in a project that lacks historical data. This problem is called cross-project software defect prediction (CPSDP) which is also considered a part of the transfer learning problem. The present works on CPSDP mainly assume the same set of software metrics (i.e. homogeneous metrics) are used to measure the characteristics of a code change for both the source project and target project. Nevertheless, some metrics

especially object-oriented metrics are designed for specific programming languages and some features (e.g. features extracted from the commercial tool) are unavailable for other software projects. To address these problems, researchers have proposed many CPSDP approaches by employing heterogeneous metrics (Chen *et al.*, 2021). The focus of CPDP has shifted into heterogeneous data sources recently. Thus, transfer learning techniques especially for data manipulation components become the main interest and important in CPDP settings (Hosseini *et al.*, 2019). Researchers have overcome this problem by proposing various approaches such as distribution characteristics (Nam *et al.*, 2015), metrics selection (Xing *et al.*, 2015), and metrics representation (Jing *et al.*, 2015). In general, CPDP-based heterogeneous data involves three main steps as shown in Figure 24: 1) selection of relevant features by using appropriate features selection methods 2) finding of matching metrics according to the distribution of every possible combination of metrics from the source dataset and the target datasets 3) Training of model using the matched metrics and predict the defect outcome from the target project.

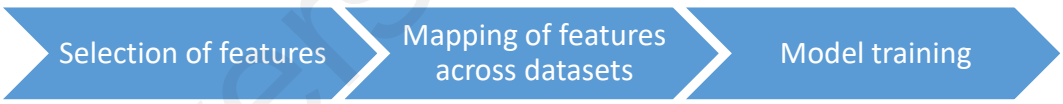


Figure 24: CPDP-based heterogeneous data workflow

The main challenge in the modelling of heterogeneous-based CPDP is that it required an appropriate metrics matching method to operate for different combination metrics across software projects. It is interesting to examine the application of deep learning in finding the informative features for metric matching between source project and target project. However, it is difficult to obtain accurate results due to the inability of conventional metrics often unable to distinguish data with different semantics (Wang *et al.* 2018). The emergence of approaches (Xia *et al.*, 2016; Wang *et al.*, 2018; Hoang *et al.*, 2019) in textual semantic features which are automatically learned by deep

learning provide beneficial semantics to tackle this problem. Thus, the combination of metrics representation learning and semantic features is believed to be complimentary for heterogeneous-based CPSDP and requires more investigation.

2.10.2 Model Optimization

Harman (2010) suggested that search-based optimization techniques are potentially useful to optimize the performance of the SDP model with multiple objectives. In particular, he suggested that the predictive model is capable to be built across multiple objectives such as predictive quality, cost-benefit, privacy, readability, coverage, and weighting aspect. Multi-objective in SDP is a relatively new application area. Canfora *et al.* (2013) presented an approach to building a logistic regression model with consideration of a compromise between defect-proneness and inspection cost using a genetic algorithm. Shukla *et al.* (2018) proposed a multi-objective optimization approach for the SDP model with consideration on minimize misclassification costs and minimizing inspection costs on defect-prone files. In the context of JIT-SDP, Chen *et al.* (2019) proposed a multi-objective optimization-based supervised approach that designed to maximize the number of identified defective changes and minimize SQA effort (i.e. code churn). However, much of the potential of the multi-objective approach is still unexplored in this context. Previous studies utilize only Genetic Algorithm (i.e. NSGA-II) in finding Pareto fronts across multiple objectives. In the future, the works on multi-objectives in SDP must be extended by considering other multi-objective Pareto-based optimization algorithms. In addition, considering different objective functions such as selection of software metrics, cost of imbalance learning, classification cost, and selection of data samples potentially also be exploited in future multi-objective based defect prediction.

Current development of SDP models commonly utilizes machine learning classification techniques. These techniques such as Naïve Bayes, Neural Network, KNN, and Logistic regression have hyperparameter settings that control the behaviors of the generated models. Hyperparameters are a set of parameters that need to be tuned before the training process began. For example, parameter settings in classification algorithms used during model training. Since the optimal parameter settings are unknown ahead of time, researchers often employ the default values for those settings (Tantithamthavorn *et al*., 2019). Default hyperparameter settings are known to be suboptimal configurations (Jiang *et al*., 2013). Thus, the prediction models are likely to underperform in case of these models are trained under suboptimal configuration. It is impractical to achieve the optimal configuration of hyperparameters by exploring all the possible configurations of a classification algorithm. Therefore, the use of any automated hyperparameters optimization technique is beneficial to achieving near-optimal configuration for an optimized defect prediction model. Automated hyperparameters optimization requires a large impact for improving the performance of classification techniques such as neural networks, decision trees, and Naïve Bayes that are parameter sensitive (Tantithamthavorn *et al*., 2019). For future works, more investigation of automated hyperparameters optimization on the performance improvement, performance stability, model interpretation, and ranking of defect prediction models must be done.

2.10.3 Latency of Data Evolution

JIT-SDP generally assumes that past defects resulting from changes are always similar to those that occur in the future (Tan *et al.*, 2015). It is imperative to note, however, that the characteristics of defect-inducing changes are evolving throughout the life cycle of the project. According to Mcintosh and Kamei (2018), code change

properties fluctuated over time. Defect classifiers suffer from poor prediction when data drift occurs within the properties of code changes. Improvements needed to be made regarding the handling of data drift. It is advisable to explore specific aspects of handling data drift in the future, including continuous model refinement (online classification learning), dividing the data by period, 91odelling the evolution of defect-inducing change patterns.

In addition to dealing with the problem of data drift, it is also important to address the problem of new software changes that are produced over time and appear during training. For example, sequential learning for JIT-SDP. Verification latency is also critical to prevent overly optimistic predictions for the JIT-SDP model (Tan *et al.*, 2015). Training instances are typically 91odellin after several changes have occurred. There is a delay before defect-causing changes are detected as defective changes, and it takes some time for non-defective changes to gain confidence and be viewed as clean changes. In this instance, it takes time to determine the actual label of each change. According to Cabral *et al.* (2019), a delay between the time a true label is received and the time it takes for the defect to be corrected is typically one to 11 years. By establishing a longer waiting period, more positive examples are identified for training. However, the risk of concept drift is directly proportional to the duration of waiting (McIntosh and Kamei, 2018). It is paramount to determine a reasonable compromise between waiting times and concept drift to obtain realistic results in the future.

2.10.4 High False Alarm in Imbalanced Dataset

Presence of false positives in class imbalance datasets during the learning process of machine learning is inevitable. The class imbalance causes false alarm results in the defect prediction process, which reduces the reliability of the model. If a false alarm is made, the cost of validating the defect in the software life cycle is increased. The cost of

defects is influenced by various factors such as the project's context, size, and location of the defect. Besides, the cost of validating software defects is also related to the application field of software engineering. During the software development process, it is usually a manual process to confirm the results of the defects discovered. Therefore, the false alarm increases the workload of testing and development, which ultimately consumes unnecessary costs.

To reduce disastrous results from the false alarm in imbalanced data of SDP, resampling technique mainly is considered by researchers (Li *et al*., 2018). Oversampling provides class rebalancing of the instances by increasing the percentage of positive instances in the dataset to obtain the class balance. The resampling technique mainly focuses on the size differences between majority and minority classes without focusing on independent instances. In this case, if the dataset is closely measured by software metrics, the performance of the classification model is closely affected by the near class boundary instances. Moreover, the interaction between the choice of oversampling techniques and the choice of classifiers is not well understood in the context of false alarm results. Similarly with the choice of dataset and input software metric types. Therefore, current techniques are still not ideal in practice.

2.11 Summary

This chapter presented a comprehensive review of JIT-SDP by elaborating on the fundamentals of the domain, including the different software metrics utilized to model the prediction of software defects, resampling strategies, and machine learning approaches. The review also highlighted the main issues faced in the domain over the years to uncover the evolution of approaches for JIT-SDP.

A clear understanding of the trends is essential to understanding the current state-of-the-art. Therefore, to fully understand the state-of-the-art, this chapter divided the components of modelling JIT-SDP into three different aspects: software metrics, resampling strategies, and machine learning. Each of the components is presented in a discussion including related works and current limitations. For depth discussion on software metrics, this chapter also laid down three main issues on factors affecting the accuracy of software metrics in representing the features in the context of JIT-SDP models. Following the discussion, resampling of imbalance class distribution in modelling of JIT-SDP which focusing more on oversampling process of defect data. Next, an in-depth discussion of JIT-SDP modelling in the context of machine learning is provided to provide current technologies toward effort aware JIT-SDP models. The potential of deep reinforcement learning for JIT-SDP in the software engineering domain is then presented, lighting up the opportunities for developing new deep reinforcement learning in JIT-SDP modelling based on effort awareness. In the last section, four open issues in the prediction of software defects are explained for a better understanding of the trends and limitations which hinder the progress of JIT-SDP research.

The findings of this review chapter demonstrate that the underlying research problems of imbalance class distribution and false alarm results need to be examined for further understanding the problem of inaccurate prediction. Review of imbalance class distribution shows that oversampling methods do indeed influence the performance of the JIT-SDP model. Additionally, the results of reviewing existing classifiers reveal that the selection of classifiers is highly correlated with the prediction results of software defects. Accordingly, the following chapter presents a comparative analysis of baseline approaches to the identified problems.

CHAPTER 3: EXPERIMENTAL ANALYSIS ON OVERSAMPLING AND EFFORT AWARENESS IN JIT-SDP

To establish problem statements of ineffective oversampling in imbalance class and high false positives in effort awareness, two experimental analyses are conducted. First, analysis of oversampling techniques in imbalance class datasets provides the observation of factors that affect the performance of oversampling. The analysis consists of two observation factors such as data distribution levels and choice of oversampling techniques. In the second analysis, a comparative experiment is conducted to revisit how the JIT-SDP model generates more false positives in effort awareness evaluation. This analysis helps in providing evidence of low prediction accuracy for current effort-aware JIT-SDP.

3.1 Oversampling for Imbalance Class Distribution

The experiment aims to determine whether oversampling techniques provide different prediction performances when dealing with overlapping class distributions that vary with data characteristics. To achieve the aim of the experiment, two objectives are conducted. Table 18 shows the mapping of research objectives with specific research questions. A comparative experiment is conducted to analyze oversampling techniques via different experimental settings. The experiment compares oversampling techniques including SMOTE, SMOTE-Borderline, ADASYN, GAZZAH, MWMOTE, ROSE, and MAHAKIL. For ease of explanation, seven techniques are divided into lightweight and heavy-weight techniques. The categories are based mainly on the complexity of the technique used in the oversampling techniques, especially the clustering algorithm which resides in the oversampling techniques. SMOTE, SMOTE-Borderline, ADASYN, GAZZAH, and ROSE are lightweight techniques. They are easy and fast to

implement for imbalanced datasets. MWMOTE and MAHAKIL are heavyweight techniques as they take longer to complete the oversampling process.

Table 18: Mapping of experimental objectives with research questions

Experimental objectives	**Research questions**	**Research variable**
To analyze the performance of oversampling techniques in different imbalanced data settings	RQ1: How does the oversampling performance depend on the characteristics of the datasets?	Imbalanced data characteristics
To evaluate and compare the feasibility of oversampling techniques in JIT-SDP	RQ2: Which oversampling techniques give the best performance in general?	Oversampling techniques

3.1.1 Experimental Setup

RQ1*: How does the oversampling performance depend on the characteristics of the datasets?*

Motivation: Comparison of oversampling performance is needed to observe different results in varying characteristics of software project datasets. Noted that the minority class (defect) instances used to generate synthetic instances are different at each running of oversampling for a certain dataset. This led to a high variance in the performance of prediction models. As a result, the prediction model producing high variance in predictions.

Approach: To examine the performance of oversampling techniques according to different levels of measurement for each feature, Gaussian noises are applied to the original dataset. In this setup, variance and overlapped spatial distribution in the training dataset are increased and this is a form of data augmentation. Furthermore, the

application of noise increases the randomness of training data, which means that the model is hard to learn from training samples. This is done to test the robustness and performance of an oversampling technique in the presence of known amounts of noise. Thus, modifying existing samples in the training samples to increase overlapped spatial distribution. The datasets used for oversampling vary in terms of the imbalance ratio and sample size. The datasets with an imbalance ratio of less than 15% are defined as highly imbalanced datasets and the remaining ones as mild imbalanced datasets. Severely imbalanced datasets imply that more artificial data instances are generated compared to the low imbalanced data when the application of oversampling techniques is conducted.

RQ2*: Which oversampling techniques give the best performance in general?*

Motivation: Little attention is paid to evaluating the stability of oversampling techniques especially in JIT-SDP. Oversampling in defect prediction is considered unstable, so we cannot be confident in the datasets oversampled by existing oversampling techniques. Thus, an analysis of oversampling performance for JIT-SDP model is conducted to analyse the stability of oversampling under different datasets conditions.

Approach: Oversampling techniques are utilized only for the training datasets. The experiments are conducted on each dataset after resampling with each oversampling technique. The oversampling procedure stops when it reaches 50% of the defect data in the training set. It assumes that a balanced dataset is achieved and that oversampling defect data to 50% of training data will achieve better results. (Fernandez *et al*., 2018). For the prediction performance, JIT-SDP is built based on Logistic Regression algorithm to evaluate the performance of each oversampling technique. Logistic regression is a widely used classifier in JIT-SDP similar to several studies (Kamei *et al.*,

2013; Taba *et al.*, 2013; Yang *et al.*, 2016; He *et al.*, 2018). The training data for Logistic Regression classifier are randomly selected using 2/3 of the sample size and the remaining 1/3 as the testing data for each dataset. The process of splitting the dataset is repeated 10 times to reduce the effect of bias throughout the experiments. The division is done using stratification such that the proportion of imbalanced class distribution is maintained. Figure 25 shows the steps followed for the empirical evaluation of the data oversampling techniques for each dataset. To evaluate the effectiveness of oversampling techniques, F-score is used which a commonly used metric for evaluating the accuracy of classification performance. F-score combine Precision and Recall that are derived from a confusion matrix. The confusion matrix consists of four possible prediction outcomes. If an instance predicted correctly as 'defective', it is considered as a true positive (TP); if an instance is misclassified as 'defective', it is a false positive (FP); if an instance is correctly classified as 'clean', it is a true negative (TN); if an instance is misclassified as 'clean', it is a false negative (FN). Using four numbers of confusion matrix, Recall, Precision and F-score are calculated. Recall is the ratio of the number of correctly predicted 'defective' instance to the number of actual 'defectIve' instance (Recall = TP/(TP+FN)). Precision is the ratio of correctly predicted 'defective' instances to all instances predicted as 'defective' (Precision = TP/(TP+FP)). Then, F-score is a hormonic mean of Recall and Precision (F-score = (2*Recall*Precision)/ (Recall + Precision)). The higher F-score value indicates better overall performance for the classification results by JIT-SDP model.

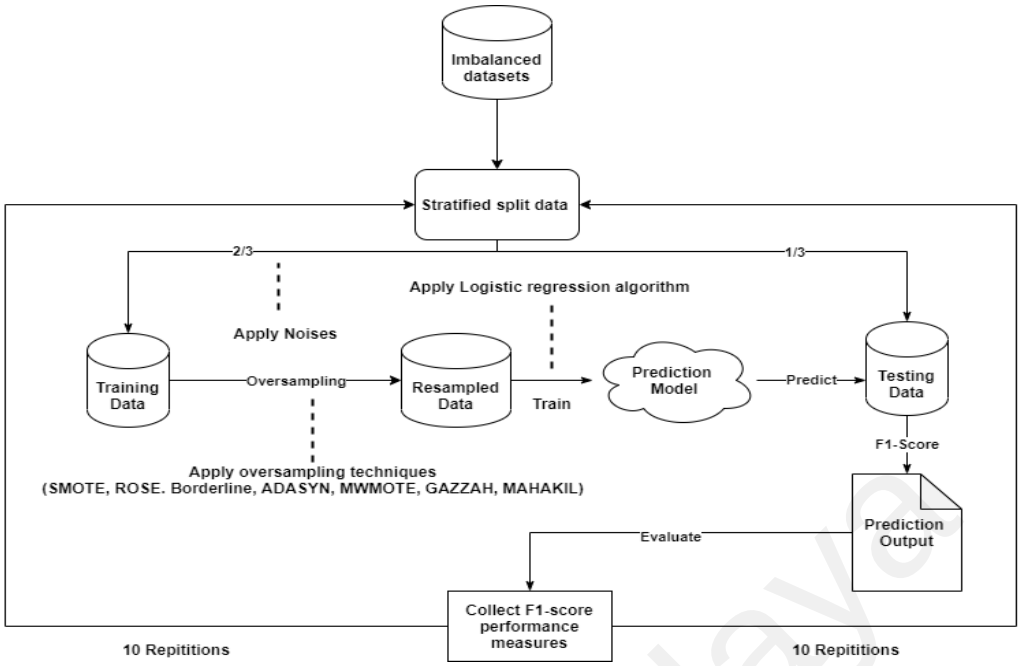


Figure 25: Procedure of comparison for oversampling techniques

3.1.2 Data Distribution

Distribution of imbalanced dataset in JIT-SDP consists of wide variability in the percent of defects that existed across software project datasets. Such different means that the geometry of the hyperspace boundary between different datasets varies in term of the overlapped class distribution. To illustrate this overlapped distribution, the imbalance datasets are transformed into two principle components (2D) representation by using Principal Component Analysis (PCA). Figure 26 illustrates the distribution of imbalanced datasets resulting in various overlapped spatial distributions.

The consistency of oversampling in different scenarios is examined by adding three Gaussian noise levels (25, 50, and 75%). Higher levels of noise increase the likelihood that class distributions overlap. In addition, this increases the diversity within the original datasets. Adding Gaussian noise allows for a variety of different datasets to be derived from original datasets, thereby allowing the evaluation of oversampling techniques in various class overlapping scenarios. With the addition of Gaussian noise,

the robustness and performance of oversampling techniques when exposed to known amounts of noise can now be tested. Here are some examples of the measurement of original software project datasets before and following the addition of noise, as shown Figures 27 to 32.

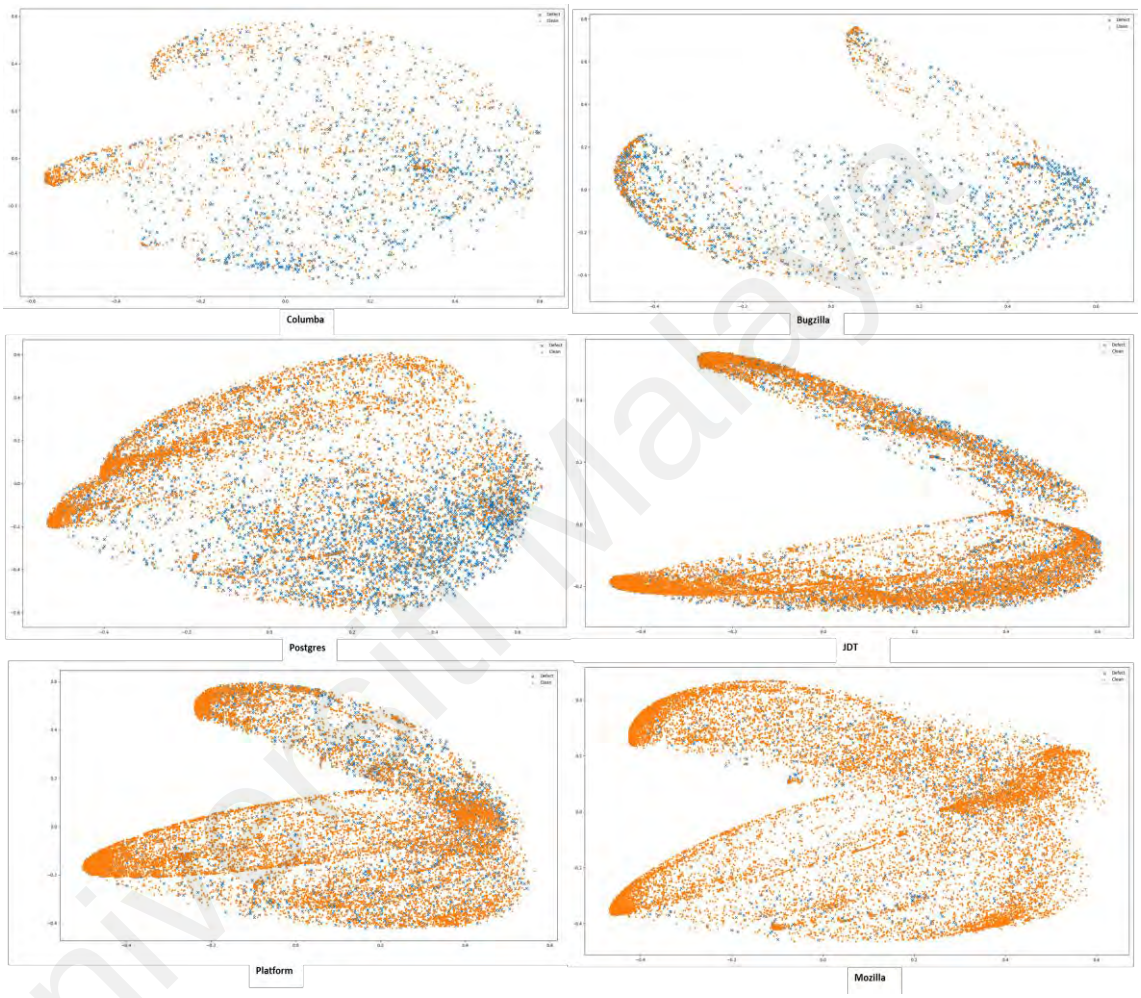


Figure 26 : Data distribution after transformation

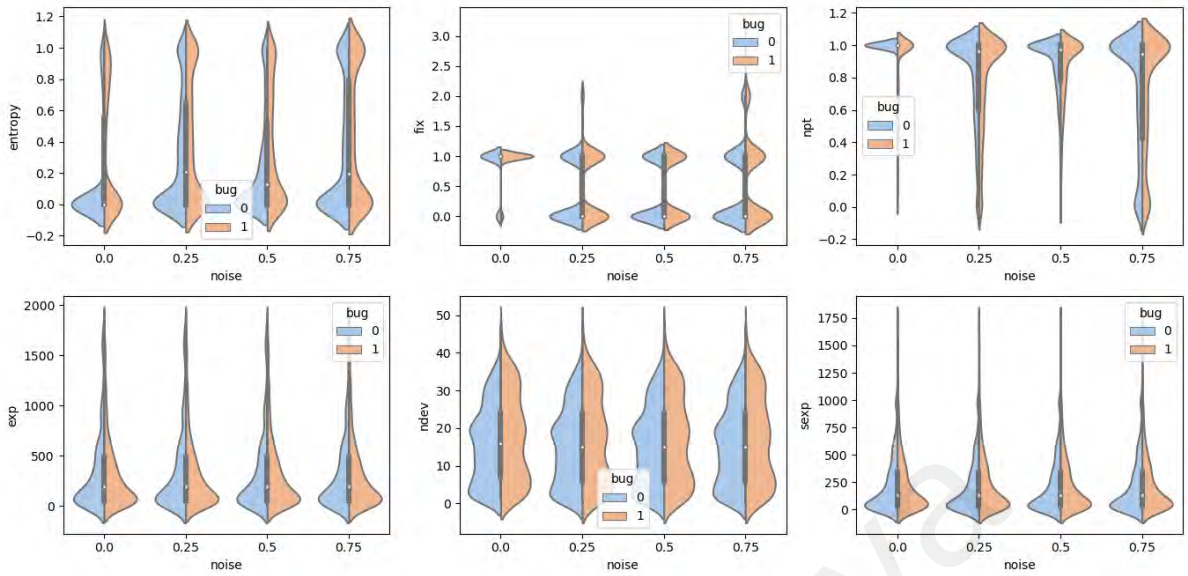


Figure 27: Addition of noise in Bugzilla dataset

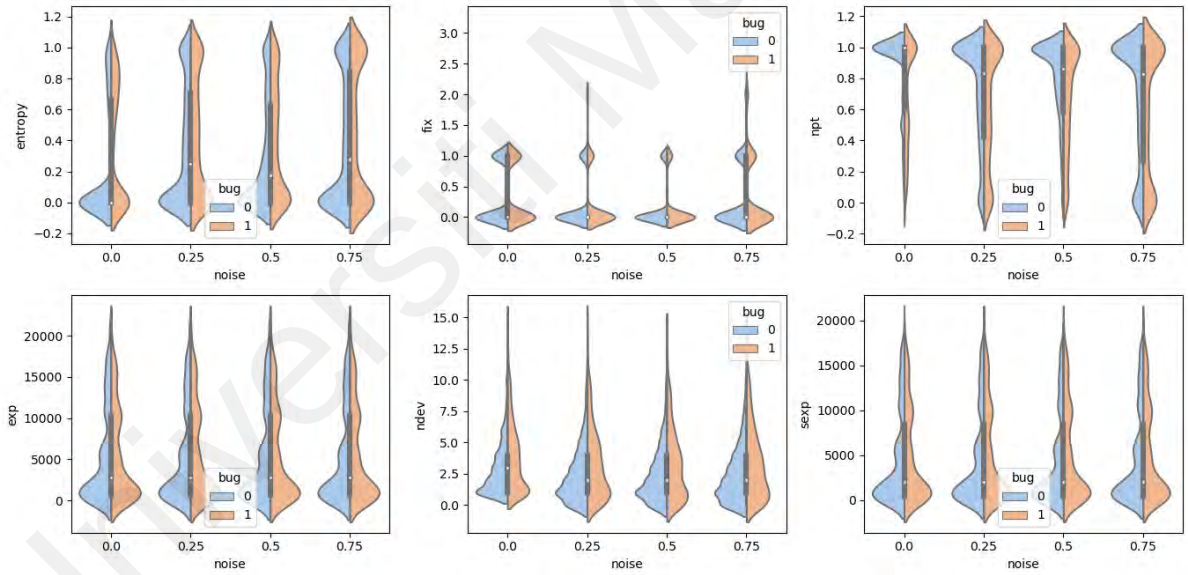


Figure 28: Addition of noise in Columba dataset

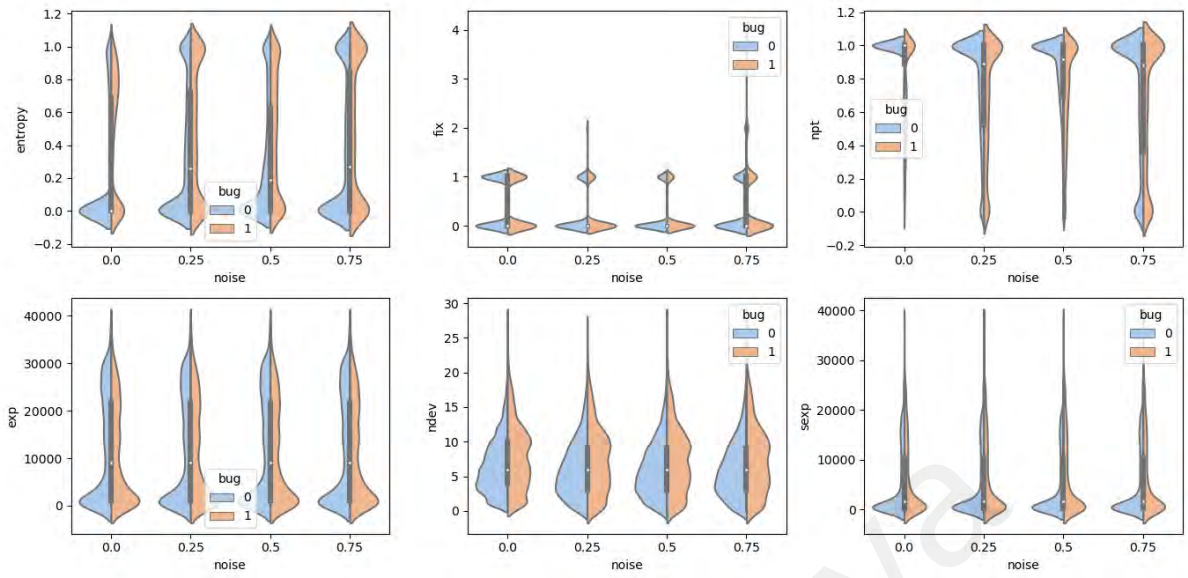


Figure 29: Addition of noise in Postgres dataset

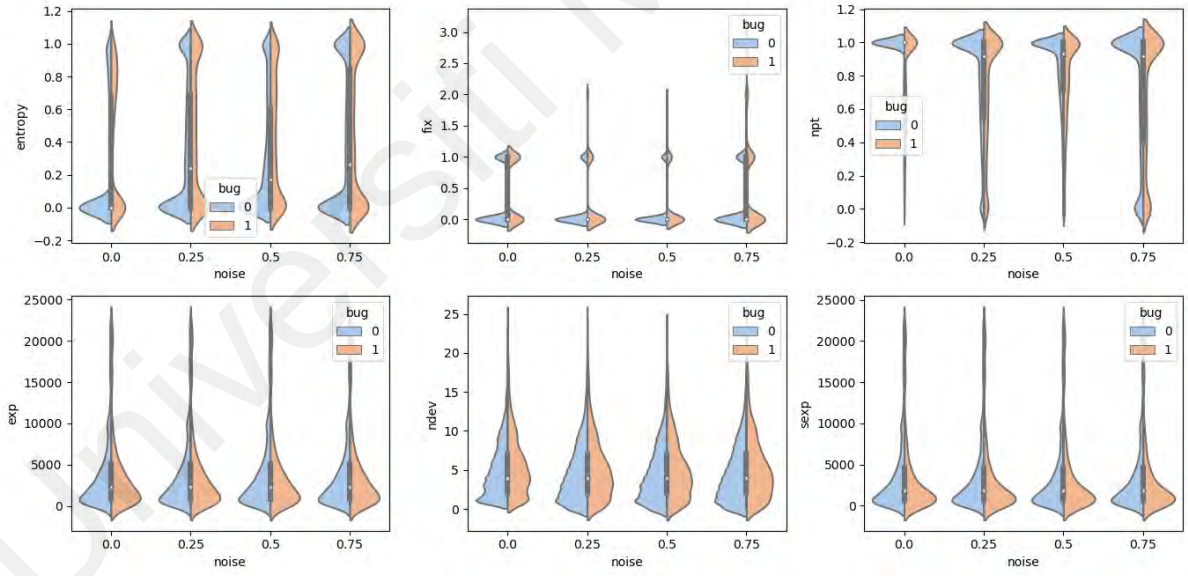


Figure 30: Addition of noise in JDT dataset

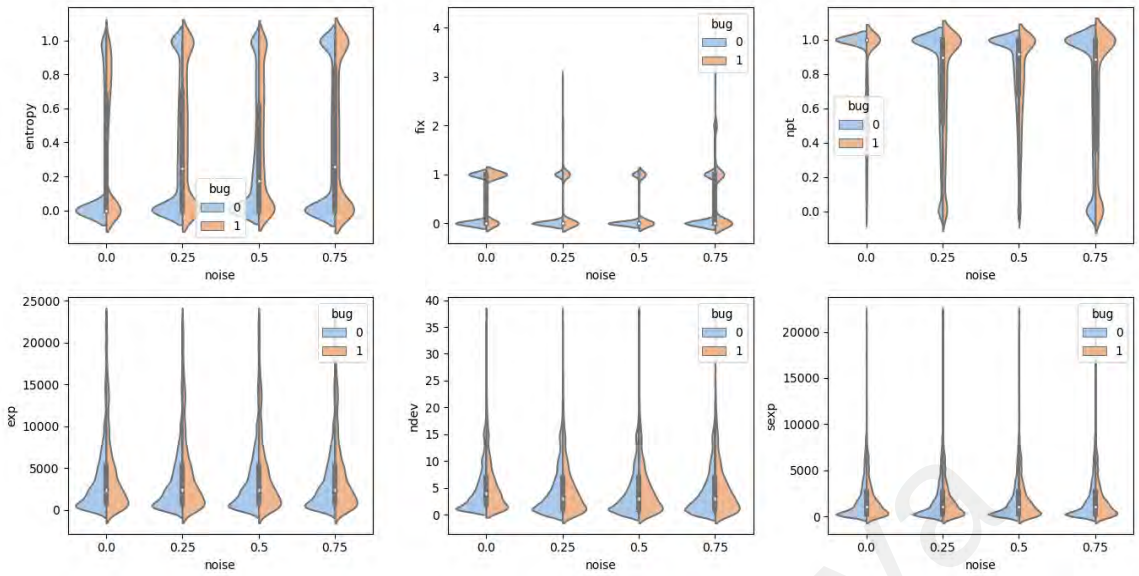


Figure 31 Addition of noise in Eclipse-Platform dataset

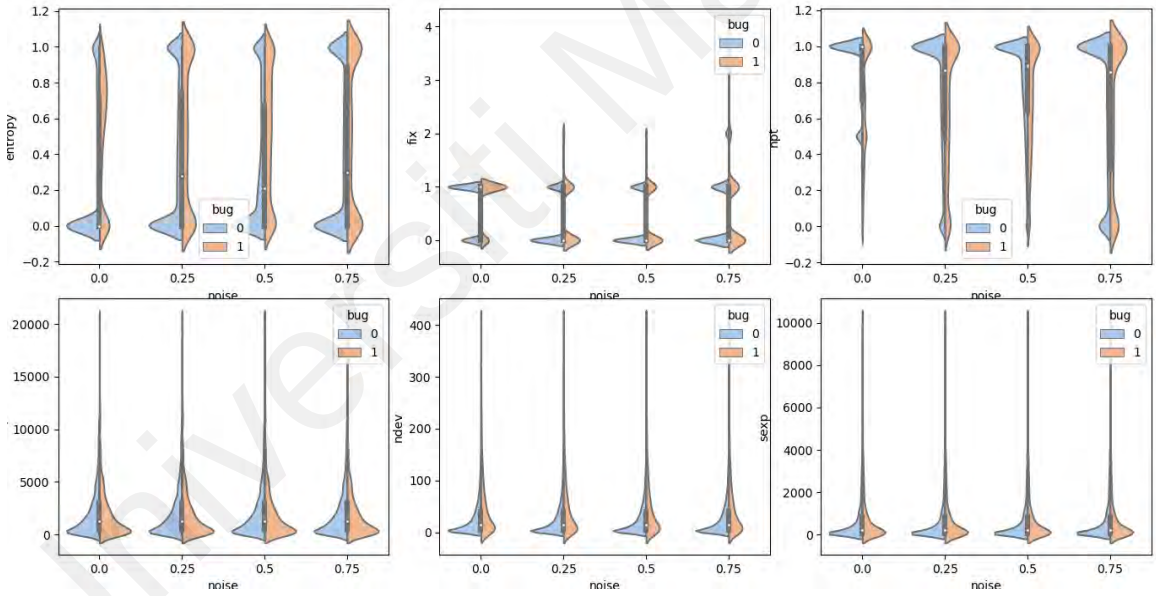


Figure 32: Addition of noise in Mozilla dataset

3.1.3 Baseline Techniques

SMOTE (Chawla *et al*., 2002) is a synthetic minority oversampling technique to overcome the problem of overfitting in which generated samples are exact replicates of observed samples. Using this technique, new samples are produced by linearly

interpolating an inferior sample with its k-Nearest Neighbors. In this approach, new samples are generated without considering the majority sample, which in turn lead to overlapping between majority and minority samples, thereby causing over-generalization as well as amplification of noise. Despite these drawbacks, SMOTE is widely adopted by researchers because of its simplicity.

ADASYN (He *et al*., 2008) adaptively generates minority data samples according to their distributions: more synthetic data is generated for minority class samples that are harder to learn compared to those minority samples that are easier to learn. Dynamically adjusts sample weight to reduce the bias in the imbalanced dataset by considering the characteristics of the distribution of the data. For each minority class sample, ADASYN incorporates a density distribution to determine the number of synthetic samples required. In this optimization process, it is induced to focus on the hard-to-learn (classify) examples within the minority class samples. As a result, the samples generated are not equal across all samples.

GAZZAH (Gazzah *et al*., 2015) is a hybrid approach that consists in oversampling the minority class using SMOTE star topology, and under-sampling the majority class. The under-sampling approach is based on selecting some feature vectors according to a distribution criterion. over-sampling a minority class by adding only a few synthetic instances and under-sampling the majority class by removing examples that are not relevant enough

MWMOTE algorithm (Barua *et al*., 2014) categorizes and identifies safety data, boundary data, and potential noise data from minority samples. It adaptively assigns the weights to the selected samples according to their importance in learning. The samples closer to the decision boundary are given higher weights than others. Similarly, the samples of the small-sized clusters are given higher weights for reducing within-class

imbalance. The synthetic sample generation technique of MWMOTE uses a clustering approach to partition datasets and uses the Euclidean distance similarity measure to find very close class samples and synthetically generate samples based on the weights assigned to the minority class samples.

Borderline-SMOTE (Han *et al*., 2005) is a modification of the SMOTE technique with a focus on cases of minority class data instances that are difficult to classify, otherwise known as borderline data instances. Before finding minority class instances, the algorithm finds minority class instances that have more majority class instances as closest neighbors than minority class instances

ROSE algorithm (Lunardon *et al*., 2014) reproduces already existing minority class instances at random, thereby increasing the number of minority instances. It is considered a smoothed bootstrap-based technique. Moreover, it is a simple and easy-to-implement method. The technique helps to generate synthetic data based on sampling methods and smoothed bootstrap approach.

MAHAKIL oversampling algorithm (Bennin *et al*., 2018) introduces the crossover operator of genetic algorithms to synthesize samples. The oversampling algorithm is based on the theory of inheritance and the Mahalanobis distance. The algorithm enables the data diversity within the minority class to increase by uniquely creating new synthetic minority instances based on having a small diversity measure distance value.

Table 19: Overview of oversampling techniques

Technique	**Advantage**	**Limitation**
SMOTE	Simplicity	Overgeneralized
ADASYN	Generating more data for harder to learn examples	Ignore minority samples close to the decision boundary
ROSE	Simplicity	Potential of leading to over-fitting toward near-duplicated instances

MAHAKIL	Increase the diversity of the synthetic samples	Potential of invading the majority of sample'' space
MWMOTE	Generating more data for harder-to-learn examples	Increase in complexity of the model
Borderline-SMOTE	Generating more samples near class boundaries	A less diverse sample generated
GAZZAH	Avoid irrelevant instances for generating a new instance	Inconsistence due to balancing policy

3.1.4 Result and Discussion

RQ1: How does the performance depend on the characteristics of the datasets?

1) Data without additional Gaussian noise

Figure 33 and Table 20 present F1 scores for different resampling techniques. For mildly imbalanced datasets of Columba, Bugzilla, and Postgres in the context of without additional noises, oversampling techniques are observed to insignificant improvements of F1-score except in the case of Bugzilla. Compared to other projects, Bugzilla is the least class imbalance ratio. In addition, Bugzilla distribution is also considered less diverse data since the number of data instances is among the lowest. As a result, sampling of data for machine learning becomes easier. However, the hybrid Gazzah technique results in the lowest performance when considered in terms of consistency and accuracy. As this technique entails under-sampling, some crucial information necessary to build an effective predictive model is lost during sample removal. In contrast, MAHAKIL achieves the highest median F1-score of 0.58 for this dataset, as a result of its focus on improving the diversity of data. This is one of the reasons why MAHAKIL's F1-score is more stable and more accurate.

For the high imbalanced dataset of JDT, Platform, and Mozilla, the consistency and prediction accuracy of models developed through oversampling data across techniques

is almost similar. This occurs because the dispersion in the data measurements already diverse before oversampling is conducted. Regarding this situation, as compared to mild imbalanced datasets, the size of the data within these datasets is larger and limited in terms of defects. With a large amount of data, more unique measurement instances are apparent in the distribution of data which indicates a diverse state. As a result, it is difficult for these oversampling techniques to improve the quality of the data when only limited empty spaces are available for the defect class without introducing new data into the region of the clean class. This phenomenon results in data resampling into overlapping spatial distributions.

2) Data with additional Gaussian noise

Figure 34 illustrates the prediction results obtained using baseline techniques in the presence of noise. The addition of noises to original data reduces the prediction model accuracy by all techniques. With an increase in the level of noise, the performance of oversampling techniques generally is downgraded. The addition of 25% noise is still a tolerable level of noise for most of the techniques as the accuracy of the model developed using these techniques is slightly reduced when compared to without additional noise. On the other hand, the addition of 50% and 75% noises shows the prediction accuracy is almost the same accuracy performance across the compared oversampling techniques. The observation indicates that the classifier built using logistic regression faces difficulty to learn from the training data. This is due to the effect of noises which affected the diversity of data and became denser in the data distribution. Consequently, more overlapped spatial distribution resides in the data regions with the dense distribution datasets.

RQ2: Which techniques give the best performance in general?

Generally, heavyweight techniques such as MAHAKIL and MWMOTE are more accurate and consistent in mildly imbalanced datasets. The justification for this observation is that these techniques enable the identification and exploitation of empty spaces within a scarcity of data by employing partitioning techniques. MAHAKIL utilized rule-based partitioning by dividing the ordered instances into two bins based on the midpoint of the distance matrix. While MWMOTE uses a clustering algorithm to partition data into clusters closer to the class boundary lines. The strengths of these techniques are not fully utilized when high imbalance and noisy data are present, as they are unable to oversample effectively in the low-density distribution of data. It is believed that this limitation is a consequence of the fact that oversampling in multivariate data requires a proper handling of the multi-dimensionality of the software metrics already included in JIT-SDP datasets. In particular, these two techniques rely upon distance-based measurements (i.e. Euclidean distance for MWMOTE and Mahalanobis distance for MAHAKIL) for diversity analysis, which is ineffective in dealing with multidimensional data. Concerning diversity analysis, distance-based measures have several limitations when dealing with multivariate data, such as the inability to recognize duplicate data, ineffectiveness in detecting outliers, and inefficiency in detecting covariance among data samples. Therefore, the multivariate aspect of the imbalanced data is improperly handled. This situation contributes to the challenging task of oversampling multivariate data in respect to address the spatial distribution problem effectively.

In contrast, lightweight techniques such as SMOTE, ROSE, Borderline, and ADASYN are approximately similar in terms of consistency for the accuracy of predictions. This similarity in performance is attributed to the focus of oversampling, in

which the main attribute of these techniques tried to exploit data near the boundaries of classes. The assumption in this context is that the data within these regions are harder to learn for the purposes of classification, and therefore the samples required to be selected for interpolation. Despite this assumption, the situation of data richness is rarely favored when dealing with the high imbalance and noisy data that result in a highly overlapped spatial distribution problem. A notable characteristic of JIT-SDP is that it lacks a scarcity of measurement in software metrics. Consequently, oversampling techniques are unable to identify feasible regions for the selection of hard-to-learn samples and result in ineffective performance. The partitioning of data is one possible way to identify feasible methods for oversampling that significantly decrease the effect of overlapped spatial distributions.

Table 20: Median of F1-scores after 10-folds stratified cross validation

Datasets/Technique	***SMOTE***	***ROSE***	***ADASYN***	***Borderline***	***MWMOTE***	***GAZZAH***	***MAHAKIL***
Columba	0.56	0.57	0.55	0.56	0.57	0.46	0.51
Bugzilla	0.46	0.46	0.48	0.49	0.46	0.46	0.58
Postgres	0.54	0.56	0.54	0.54	0.54	0.54	0.56
JDT	0.33	0.33	0.33	0.33	0.34	0.28	0.33
Platform	0.32	0.32	0.32	0.33	0.32	0.32	0.32
Mozilla	0.19	0.19	0.17	0.20	0.20	0.14	0.20

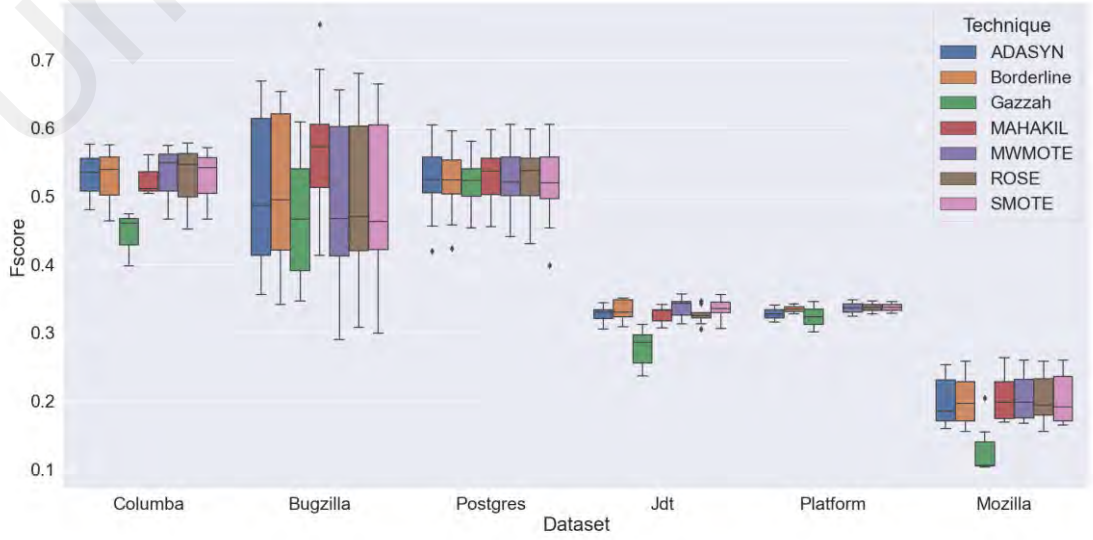


Figure 33: Without additional noise

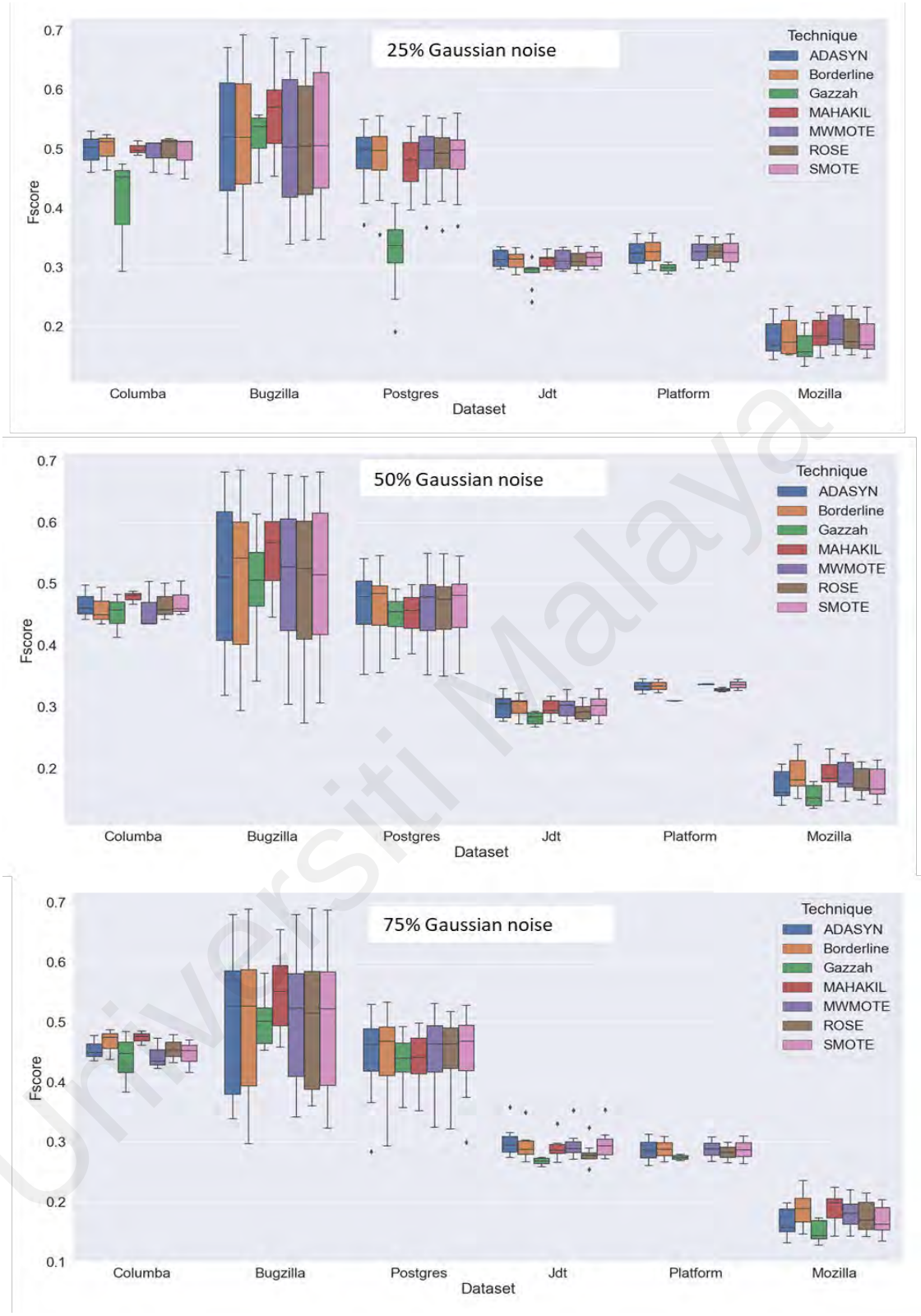


Figure 34: Performance of prediction with the addition of noise data

3.1.5 Threat to Validity

The first threat for this analysis is the unknown effect of the selection of classifier. This analysis used only logistic regression as the base classifier of JIT-SDP. The choice of logistic regression as the classifier technique is due to it being widely used in the previous JIT-SDP model. Nevertheless, the effectiveness of other classifiers remains unverified and needs to be studied in the future. Besides, six long-lived and widely used open-source software projects considered in this experiment are large enough to allow drawing statistically meaningful conclusions. The reported results may not be generalizable to other projects that have features different from the studied datasets. However, it is necessary to consider a wide variety of projects to replicate the analysis in the future to mitigate this threat. Removal of outliers in the original data is a potential threat to these experimental results. These outliers introduce additional noises to the distribution of original datasets. Nevertheless, some of the oversampling techniques especially for MAHAKIL and MWMOTE capable to handle outliers during data partition. Thus, to make fair comparison removal of outlier is excluded in the experiment. The software metrics considered for this analysis are a potential threat to the experimental results. By using a single set or type of metrics, generalization to other types of software metrics is unable to be concluded with the reported results. Nonetheless, code and process metrics are known to perform very well and are effectively used in several empirical studies on JIT-SDP. The reason is due to both types of metrics are easy to collect from any software once the code change transaction in VCS is available.

3.1.6 Conclusion

The class imbalance problem is a major challenge in JIT-SDP. The experiment examined which oversampling techniques perform best under different imbalanced class

settings based on six frequently used projects datasets. MAHAKIL performs better and is more stable in dense datasets. MAHAKIL is designed to enhance the diversity of data in small datasets. Therefore, MAHAKIL outperformed other baseline techniques, particularly in the Bugzilla dataset. However, all baseline techniques are unable to distinguish their performance in high imbalanced datasets and result in similar accuracy results. A factor that contributed to this observation is the limited number of empty spaces available for oversampling within the minority class (defect). After resampling the data, defects data are generated into clean class data spaces, reflecting the problem of overlapping class spatial distributions. To avoid interpolating defect class data into occupied spaces of clean class data, it is recommended that the diversity measurements be taken into account before oversampling in future work to overcome the problem of overlapping class distributions.

In terms of overall performance, heavyweight techniques are capable of producing better oversampled data for the JIT-SDP model, which results in improved prediction accuracy. Heavyweight techniques include a data partitioning component that assists in identifying suitable empty spaces for interpolating defect data. However, heavyweight techniques have difficulty handling data covariance, indicating their inability to perform effectively with data that is less diverse or with highly imbalanced class distributions. Thus, oversampling should allow for a better analysis of data diversity before identifying an area where data interpolation is feasible. Future works should consider alternative methods of measuring diversity in oversampling. In order to handle less diverse data, the application of similarities measurement is preferable to distance based measurements.

3.2 False Positives Prediction in Effort Awareness Evaluation

In the JIT-SDP model, machine learning is the main methodology for developing defect classifiers. Thus, this experiment aims to provide comparative analysis of baseline classifiers in JIT-SDP regarding false positives results in effort awareness evaluation. The experiment is incorporate with two main objectives which further investigate two research questions as given in Table 21. Effort awareness of the JIT-SDP model needs to consistently reflect the prediction accuracy. Accordingly, to address the effectiveness of using machine learning methodology concerning base classifier accuracy performance, the false positive rate is considered in the evaluation of effort-aware model.

Table 21: Mapping of research objectives and research questions

Analysis Objectives	Research Questions
To analyze the classification result of JIT-SDP for baseline classifier techniques	RQ1. Do different classifiers perform different classification results for JIT-SDP?
To evaluate the performance of effort-aware model	RQ2. Does any classifiers consistently fulfill the performance criteria of low rate of false positives

3.2.1 Experimental Setup

RQ1: *Do different classifiers perform different classification results for JIT-SDP?*

Motivation: The classifier used to classify defective changes represents a factor that strongly influences the classification results for the JIT-SDP model. In particular, Ghotra *et al*. (2015) discovered that the accuracy of a defect prediction model can increase or decrease by up to 30% depending on the type of classification used. Moreover, Panichella *et al*. (2014) demonstrated that despite similar prediction accuracy, the predictions of different classifiers differ in defect count.

Approach: For the requirement of the model building, training datasets need to undergo a class rebalancing process as recommended by Kamei *et al*. (2013). The training data is imbalanced because most of the changes are clean class whereas only a small percent of changes contain defects. The class imbalanced problem reduces the accuracy of the prediction. For this reason, the number of defects changes need to equal the number of clean changes in the training data. This experiment employs SMOTE (Chawla *et al*., 2002) to ensure the equality of the numbers of samples balanced for both classes. SMOTE generates new synthetic instances by combining certain defect class samples with previously defined *k* defect class nearest neighbor instances. The experiments ran 7 base learners of logistic regression (LR), support vector machine (SVM), decision tree (DT), Adaboost, Gaussian naïve Bayes (NB), artificial neural network (ANN), and k-nearest neighbor (KNN). All base learners are applied within stratified ten-cross validation settings. This setting divides the dataset into ten equal portions and uses each chunk once as the test set to evaluate the developed model using the remaining nine portions. The rationale behind using 7 classifiers is that each classifier consists of limitations and advantages. These 7 classifiers are mostly used in the literature for classification purposes. During the experiment, the prediction models are built by mapping the given software metrics to an output whose values are binary: clean and defect changes. The training data for all baseline classifiers are randomly selected using 2/3 of the sample size and the remaining 1/3 as the prediction data for each dataset. Then, the training dataset undergoes 10-fold stratified within project validation. The datasets are divided randomly into 10-folds, 9-folds serve as training data, and the remaining fold serves as test data. In cross-validation, each fold is used as a testing dataset only once. Additionally, the data are folded so that every fold consists of the same proportions as the original dataset. The highest prediction model among these folds is selected for the final prediction. The selected model is used to predict the

unseen data which is the prediction dataset. The final prediction result is recorded to show the credibility of the experiment results.

RQ2: *Does any classifiers consistently fulfill the performance criteria of low rate of false positives?*

Motivation: In an ideal scenario, a prediction model must have a high capacity for predicting defect proneness and a low false alarm rate. However, ideal cases are extremely rare. As highlighted by this condition, it is necessary to investigate the consistency of baseline classifiers' performance in dealing with false positives based on effort evaluation.

Approach: As the JIT-SDP model is to determine whether a code commit instance is defective change or clean change, some indicators for the binary classification task are used to evaluate the effectiveness of the base classifiers for comparison. In this analysis, a total of three indicators are used, including F-score, ACC, and false-positive rate (FPR). The details of these indicators are described as follows. The first indicator is F-score, which is the weighted harmonic average of Precision and Recall. F-score is intended to capture the prediction performance in imbalanced class distribution existing in the datasets. Furthermore, F-score provides a harmonic mean of precision and recall which gives a better measure of incorrectly classified cases than the accuracy metric. For evaluating the predictive effectiveness of a JIT-SDP model, the effort required to inspect those changes predicted as defect-prone is considered to find whether they are defective changes. Consistent with Kamei *et al*. (2013), the code churn which describes the total number of lines added and deleted by a change is used as a proxy for the effort required to inspect the change. Similar to Kamei *et al*. (2013) works, *ACC* is used to evaluate the effort-aware performance of the JIT-SDP models. *ACC* denotes the recall of defect-inducing changes when using 20% of the entire effort required to inspect all changes is implemented. Furthermore, this analysis also considered the rate of false

positives in 20% of inspection efforts (FPR@20%). Concerning the rate of false positives, a lower value is better. In contrast to *ACC* a higher value is better.

3.2.2 Result and Discussion

RQ1: Do different classifiers perform different classification results for JIT-SDP?

The results achieved running the baseline classifiers over all the considered software projects are reported in Table 22. As reported in the results, no single classifier is a clear winner in defects prediction. Indeed, the difference in terms of F-score achieved by the classifiers is quite small except for NB which is the lowest performance across six datasets. Despite this observation, the average F-score achieved by ANN is higher with respect to other classifiers: LR = +3%, SVM = +3%, DT = +7%, NB= 10%, kNN= +6%. In contrast, ANN and AdaBoost have an almost similar average of F-scores of 45.2% and 45.42% respectively. The results demonstrate how deep learning classifiers such as AdaBoost and ANN are superior classifiers compared to the baseline supervised classifiers. Such superiority is statistically significant when considering the differences between the performances of both AdaBoost and ANN with the ones achieved by other supervised classifiers on large-sized software projects such as JDT, Platform, and Mozilla. Here in these datasets, all classifiers achieve a poor F-score of less than 40%. The classifiers are trained using resampled data by SMOTE which still leads to the overgeneralization of the defect class because it only selects the nearest neighbor instances. For this reason, it is important to fine-tune the hyperparameters for both resampling and classifier algorithms according to the training datasets.

RQ2: Does any classifiers consistently fulfill the performance criteria of low rate of false positives?

Despite several studies (Kamei *et al*., 2013; Taba *et al*., 2013; Yang *et al*., 2015; Chen *et al*., 2018; Kondo *et al*., 2020) utilized LR as a suitable classifier in the context of effort-aware model in JIT-SDP. According to effort awareness in ACC given in Figure 35, findings however, LR is not significantly different with SVM and kNN with a magnitude of difference are less than 1% of ACC. In fact, ANN, DT, and AdaBoost perform equally or better than LR in all datasets. Nevertheless, based on the statistics shown in Figure 35, all classifiers are unable to produce good results in terms of false-positive rate within ACC score. The results show that none of the classifiers reduce FPR by less than 41%. In other words, none of these classifiers are capable of producing reliable effort awareness results when considering only 20% of efforts. Note that based on such experimental results, the choice between current classifiers provides no significant effect on reducing false positives in effort awareness. Therefore, due to such unreliable results for effort-aware of JIT-SDP, a good defect classifier algorithm is needed to compromise between reducing false positives and having reliable effort awareness in defect prediction.

3.2.3 Threat to Validity

The quality of the experimental results depends on the dataset used. Therefore, this analysis decides to use the dataset commonly used in JIT-SDP studies. Hence, the datasets are suitable for developing and validating models for identifying defects in code change transactions. Nevertheless, the experiments carried out in this analysis can also be performed with a different dataset. In addition, only six classifiers are adopted throughout this experiment and the parameters are set to the defaults. For the training datasets, only SMOTE with default parameters is considered in the data resampling

process. The performance of the prediction model with other classifiers, resampling techniques, or different parameters is not validated here. In the future, more oversampling techniques and more different machine learners are required to explore for the performance comparison.

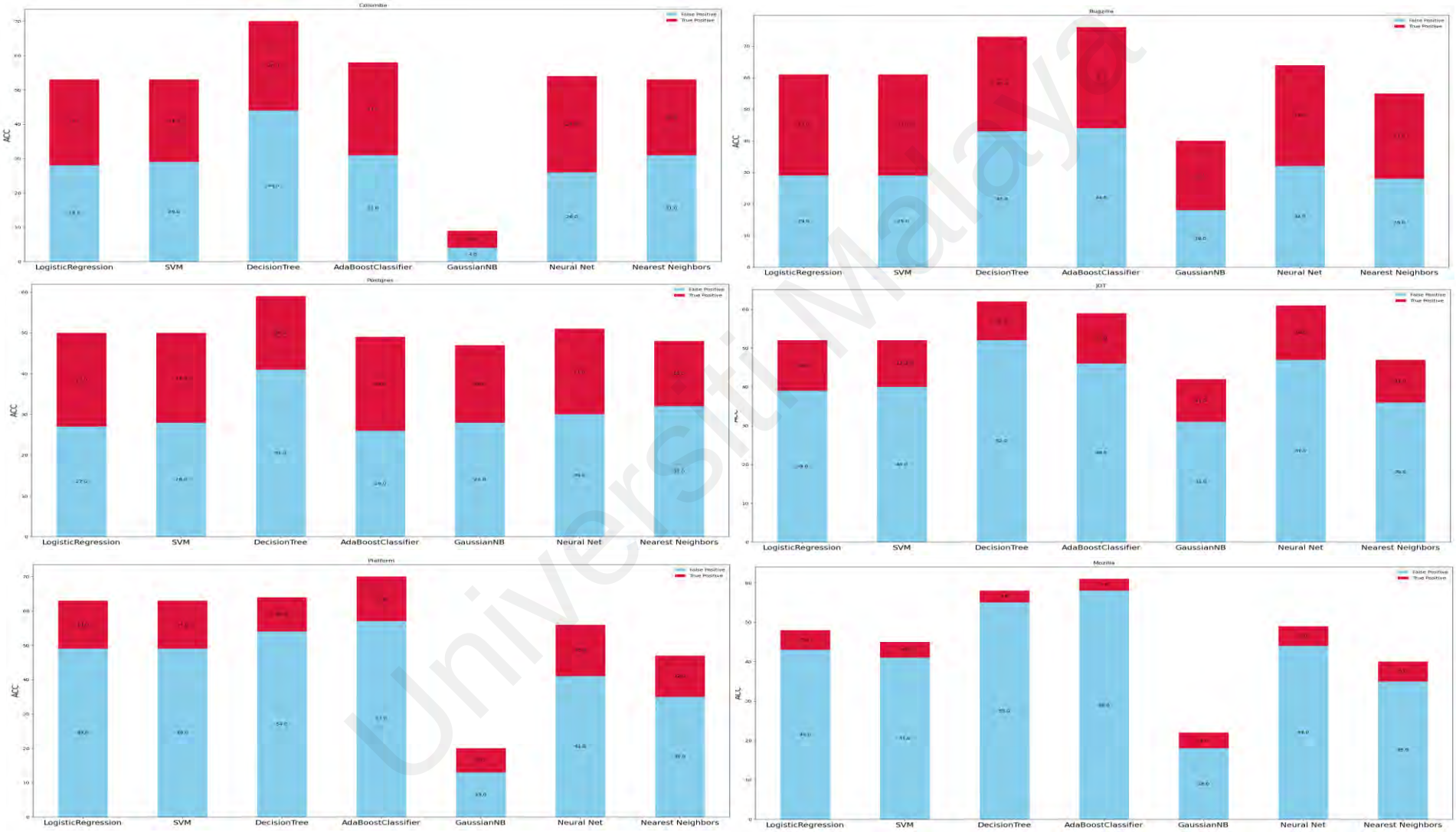
A total of three indicators are used, including F-score, ACC, and FPR performance measures are used to evaluate the effort awareness of the JIT-SDP models. These three selected performance measures are common in JIT-SDP. However, in case of other performance measures are adopted, different results are obtained. To reach a more general conclusion, more performance measures in future works are needed.

3.2.4 Conclusion

In the experimental analysis conducted on seven baseline classifiers for developing JIT-SDP model, we investigated which of the baseline classifiers is the feasible classifier for effort aware context. The results showed that the deep learning classifiers have edge advantage in predicting defect accurately. In addition, through evaluating different classifiers in effort awareness evaluation, no significant performance is achievable for the baseline classifiers except for deep learning classifiers (AdaBoost and ANN). Despite the fact that these deep learning classifiers predict more defects than other basic classifiers, the rate of false positives is still high and have not improved significantly in comparison to other classifiers. Current baseline classifiers are incapable of producing dependable results for effort-aware JIT-SDP. To improve the results of high false positives in defect prediction, an alternative advanced classifier capable of achieving a balance between reducing false positives and producing accurate predictions is urgently required.

Table 22: Accuracy performance of base learners

Datasets/ Technique	***Columba***			***Bugzilla***			***Postgres***			***JDT***			***Platform***			***Mozilla***		
	Fscore	ACC	FPR	Fscore	ACC	FPR	Fscore	ACC	FPR	Fscore	ACC	FPR	Fscore	ACC	FPR	Fscore	ACC	FPR
LR	52.1	52.4	53.1	59.6	61.8	47.5	54.3	49.6	54.4	34.7	51.7	75.4	34.4	63.3	77.4	20.7	47.3	90.2
SVM	51.9	53.1	53.9	59.3	61.2	47.7	53.7	49.5	55.5	34.1	52.4	76.7	33.9	63.1	77.6	19.2	45.4	91.3
DT	49.1	69.4	63.2	53.8	72.9	58.9	44.8	59.2	69	29.6	61.5	84.1	30.7	64.2	84.7	21	57.9	95.6
AdaBoost	54.7	58.3	53.5	60.1	75.9	58.4	56.1	48.5	53	37.5	58.9	78.1	38.1	70.1	81.8	24.8	61	94.7
NB	25	9	45.1	56	40	46	49.4	47.1	58.8	29.6	41.5	74.4	26.6	20.5	65.7	21.2	22.1	81.4
ANN	56.3	54.2	48	62	63.8	49.7	55.1	51.7	58.5	38	60.6	76.9	38.2	56.3	72.6	22.9	49.5	89.6
kNN	50.2	53.5	57.9	55.2	55.6	51.2	47.6	47.8	66.3	31.8	47.1	76.9	34	46.9	74	19.7	39.4	88.3



CHAPTER 4: DEVELOPMENT OF JUST-IN-TIME SOFTWARE DEFECT PREDICTION

To solve the identified research issue, this chapter provides details on the development of JIT-SDP. To begin, the proposed framework of JIT-SDP is provided to illustrate the detailed process of this development overview. The section explains data extraction, data pre-processing, and model training. In the following section, the development of the proposed oversampling technique is given in detail to address the class imbalance issue in SDP datasets. Next, the development of Deep Q-Network (DQN) algorithm in the JIT-SDP problem is explained in the following section.

4.1 Development Phases

JIT-SDP mainly comprised of three main phases: 1) data extraction, 2) data pre-processing, and 3) model training and prediction. Figure 36 depicts the overall process that is involved in the proposed JIT-SDP. The developed model using the proposed framework capable to achieve high accuracy and generalizability even for unseen data. The proposed framework helps to facilitate the modelling of JIT-SDP model according to robust accuracy and effort awareness performances.

4.1.1 Data Extraction

To conduct an analysis of the proposed framework, the research focused on open-source software projects. A total of six software project datasets were analyzed in this study which are used originally by Kamei *et al.* (2013). The datasets are extracted based on an analysis of code change characteristics that are represented as software metrics. The software metrics considered here are considered as change-level software metrics. The metrics comprise five dimensions. The diffusion dimension describes the distribution of a change. The assumption is that highly distributed changes are likely to

introduce defects. The size dimension refers to the size of a change. It is assumed that a complex change is expected in most cases, which introduces defects. In terms of the purpose dimension, a change intended to fix a defect led to the introduction of new defects. According to the history dimension, changes to the touched files or code are likely to introduce a defect if the files are modified by more developers. The experience dimension assumes that experienced developers are less likely to introduce defects when modifying source code. The metrics also are derived using the CommitGuru tool (Rosen *et al*., 2015), which automatically extracts the measurement for each of these metrics. Based on the five dimensions of metrics, the tool analyses code repositories by detecting changes within code change transactions. The process of identifying code changes is described as follows: first, code changes are extracted from the code repositories of the version control system (VCS) and issue tracking system (ITS). The data are analyzed based on the code changes characteristic as described in the preceding paragraph. To identify defective changes from clean ones, SZZ algorithm (Śliwerski *et al*., 2005) is applied to distinguish the defect-causing and non-defective changes. SZZ algorithm is an automated tool for identifying defects causing changes. Thus, the extracted changes are categorized into defects and clean changes classes. Across different projects, the proportion of defects and clean changes is different. During this data labeling, notably that all the datasets have imbalances. Mozilla is the largest imbalanced dataset of six datasets with defect rates containing only 5% defects whereas the Bugzilla dataset with the most balanced datasets with defects contains 36%. To generate good quality data, additional data preprocessing is required to ensure balanced class distributions and to eliminate any noise from the extracted data.

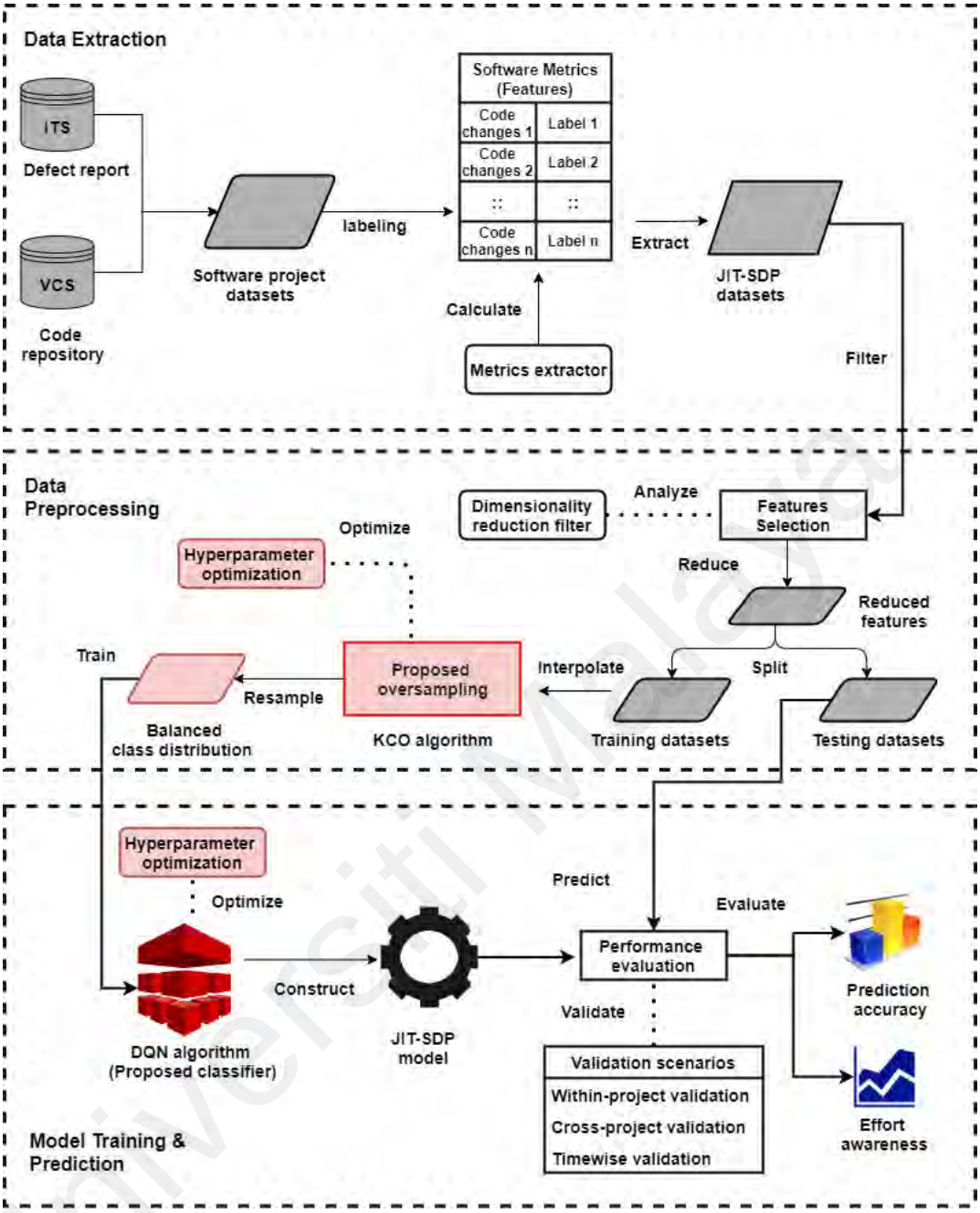


Figure 36: Conceptual framework of developing JIT-SDP model

4.1.2 Data Preprocessing

1) Logarithm and normalization

To ensure high quality data, four pre-processing operations are conducted on the training dataset. Firstly, skewed distribution datasets require log transformation. In defect pattern learning, the distribution in the dataset is extremely significant. Figure 37

illustrates the distribution of outliers present in the current dataset, which creates bottlenecks during the learning process. To make the distribution of the data more uniform, outliers need to be correctly dealt with or removed. According to the distribution, many outliers are present in the distribution, which makes it difficult to recognize data patterns. Thus, in this framework, the outliers must be properly handled by utilizing log transformation and normalization.

A natural logarithm of each change metric is applied to make patterns more visible and reduce variability. Due to the binary nature of FIX, the transformation does not apply to the metric. The natural logarithm is calculated as follows:

$$x' = \ln x \tag{4.1}$$

A data normalization process is then performed to limit the range of changing data values. The range of change metrics in this data set is not uniform. NS, NM, NF, NDEV, PD, RXP, REXP, and SXP are raw data with a range of values, whereas other metrics are normalized to [0, 1]. For unified data formats, these raw data are normalized using the Min-Max normalization method. This study used the Min-max normalization dataset because it has a high accuracy, low complexity and high learning speed. Normalizing features has several advantages including reducing prediction error, decreasing the likelihood of finding stuck upon local optimal solutions during training, and reducing the computational cost of training. (Qiao and Wang, 2019). In this scenario, Min-Max normalization is employed to linearly transform the original data. The formula for Min-Max normalization is as follows:

$$Norm(x) = \frac{x_i - \min(x)}{\max(x) - \min(x)} \tag{4.2}$$

Here, min (x) and max(x) represent minimum and maximum value of a change metric x. Min-max normalization maps a value xi to $Norm(x)$ in the range [0, 1]. By performing such a normalization, the relationship between raw data values is preserved. However, data normalization does not affect how the data is distributed.

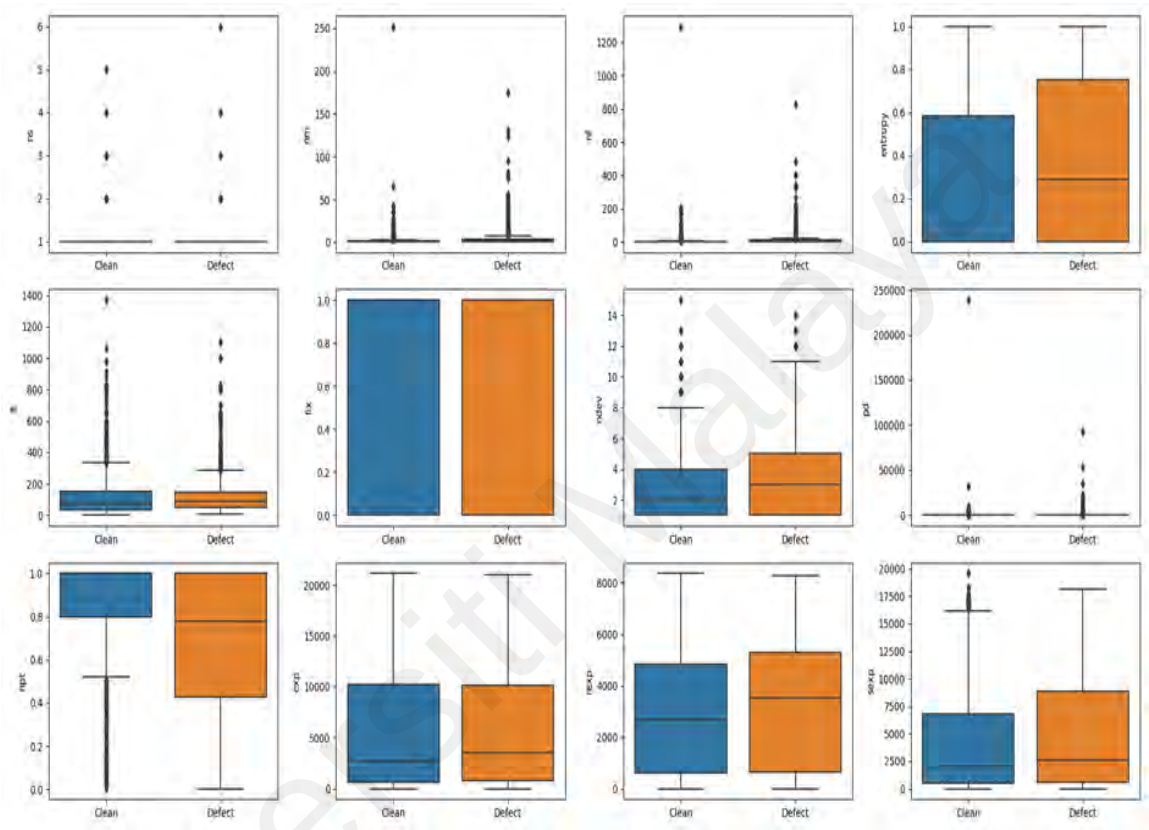


Figure 37: Skewness of data measurement across software metrics dimension

2) Features selection

The highly correlated metrics need to be removed, as suggested by Kamei *et al* (2013). In order to remove highly correlated measures, this study excludes NM and REXP metrics, as NF and NM, REXP and EXP are correlated. The exclusion of metrics is based on a manual selection of which collinearity features are eliminated to ensure only the unique features in the model are kept. Figure 38 shows the correlation analysis of software metrics. The analysis indicates a single feature could consist of one or more

features that are correlated. Incorporating highly correlated features decreases classifier accuracy (Shivaji *et al.*, 2013). Highly correlated features lead to serious multicollinearity problems. As part of this process, the metrics LA and LD are removed from consideration because both are used when evaluating efforts awareness later.

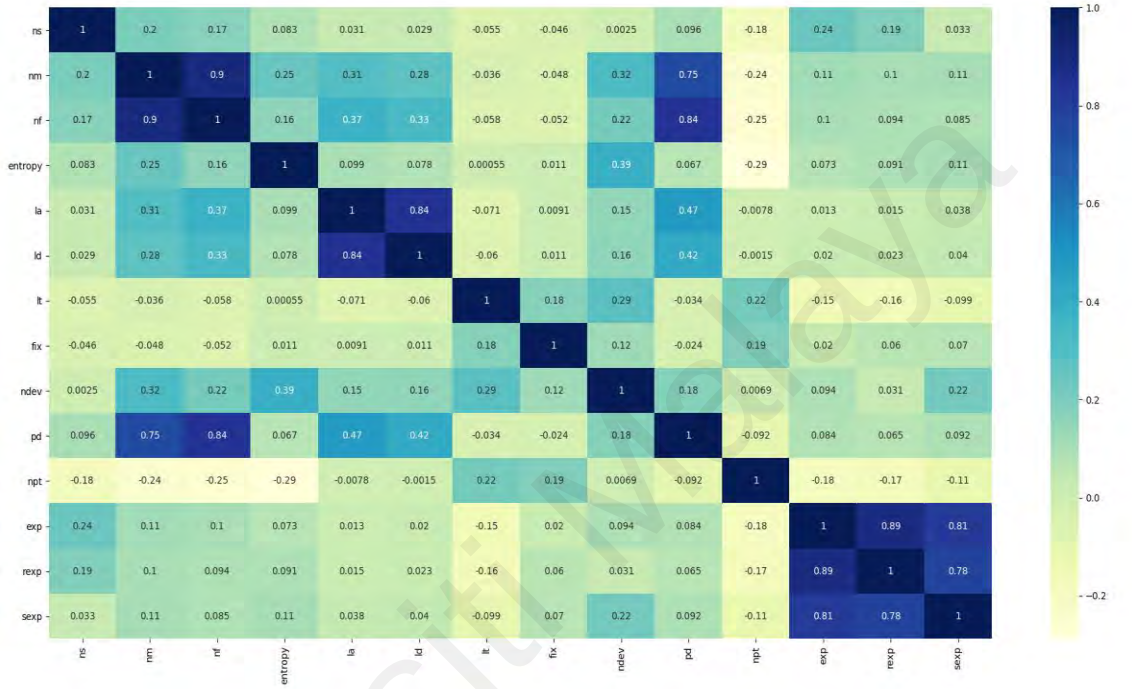


Figure 38: Correlation analysis of software metrics

3) Splitting training/testing data

To examine the proposed solutions for obtaining an adequate and realistic assessment, three prediction settings are conducted as shown in Figure 39. These validations are comprised of within-project prediction, cross-project prediction, and timewise prediction. The validations are performed using a ratio of training (80%) and testing (20%) data.

Within-project prediction is performed within the same software project data. In this validation, StratifiedKFold is used to ensure the class distribution in the datasets is

preserved in the training and test splits. The datasets are divided randomly into 10-folds, 8-folds serve as training data, and the remaining fold serves as test data. In cross-validation, each fold is used as a testing dataset only once. Additionally, the data is folded so that every fold has the same proportions as the original dataset. Using StratifiedKFold, the average result is recorded to improve the credibility of the experiment results.

Timewise-prediction is performed within the same software project, which considers changes in chronological order. Based on commit dates, the chronological order of changes data for each software project is ranked. Then, all the changes made within the same month period are grouped. Assume that the changes in a software project are grouped into n different parts. A prediction model *m* is built using a combination of fold *i* until fold *i* + 1 as training data to predict testing data for parts *i*+4 and *i*+5. According to this example, the datasets ($1 \leq i \leq n$ - 5) for training data and testing data consist of changes committed over a period of two consecutive months. Several factors account for this configuration. First, the release cycle of most projects is typically six to eight weeks. Second, it ensures that each training and test set receives a two-month interval between them. Thirdly, two consecutive months ensure that each training set has enough samples for supervised models, which is very significant. Lastly, it allows for enough training data for each project. Based on changes occurring over *n* months, the outcome of this method is *n* - 5 prediction effectiveness values for each model

Cross-project prediction is performed across different software projects. The training data set on one project is used to predict defect-proneness in another project as the testing data set. For a set of *n* projects, this method produces *n* * (*n* - 1) prediction effectiveness values (Zhu *et al*., 2020). For this study, six projects are used as the

subject projects. Accordingly, each prediction models produces 6 × (6 - 1) = 30 prediction effectiveness values.

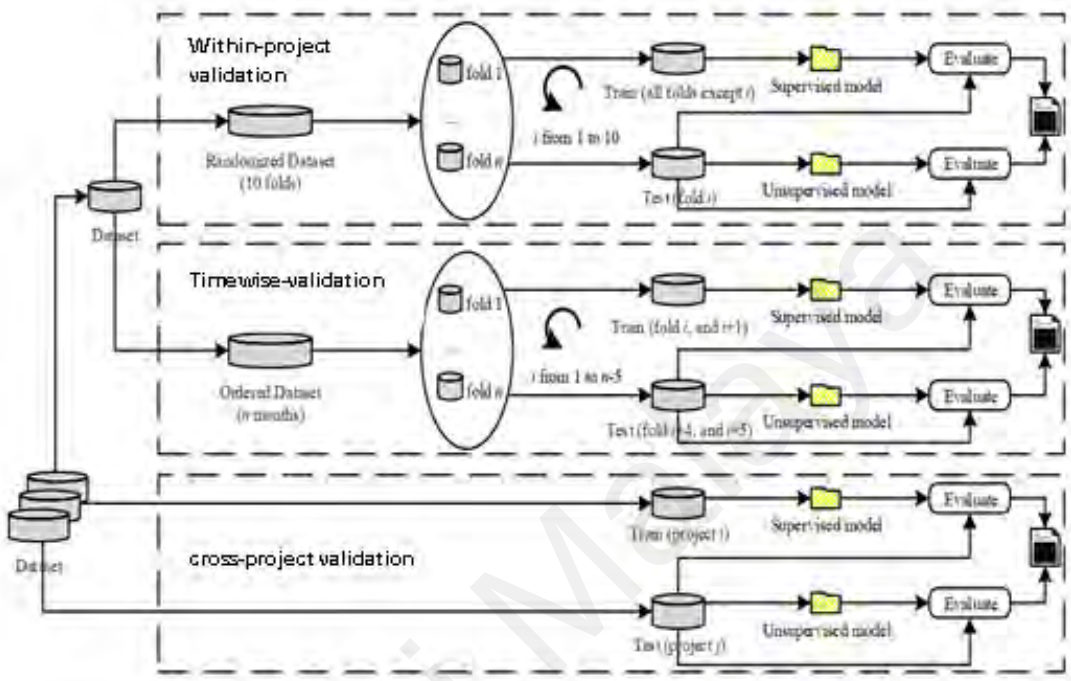


Figure 39: Scenarios of cross-validation

4) Oversampling in imbalanced datasets (RO2)

JIT-SDP datasets are relatively imbalanced, and defective changes represent only a small portion of overall changes. In the absence of adequate handling, this imbalance results in degradation of the predictive models' performance. Despite different researchers proposing various techniques to solve the class imbalance problem, no single method outperformed the others in all studies (Arora *et al.*, 2015). Conventional oversampling strategies do not consider overlapping data within the sample. In order to overcome the problem of overlapping data, this research proposes an oversampling algorithm based on kernel analysis and spectral clustering. In particular, the proposed

solution intends to address the following aspects as summarized in Table 23. Section 4.2 provides details regarding the implementation of the proposed oversampling.

Table 23: Mapping of proposed oversampling

Aims	**Problem**	**Solution**	**Contribute**
To reduce near-overlapping data	Complex boundary line led to a small distance between new and old data (Bellinger *et al*., 2016)	KPCA for measuring similarity between original samples. (Details in Section 4.2.1)	Capable to handle non-linear data distribution
To reduce the effect of high dimensional data	Presence of covariance among data samples (Rodríguez *et al*., 2022)	KCPA representing multidimensional data. (Details in Section 4.2.1)	Linearly represent multivariate data into lower dimension data while maximum variation is retained
To reduce the randomness introduced in oversampling procedure	Assume each minority instances are equally important (Sharma *et al*., 2022)	Spectral clustering with KPCA to select data template using ranking-based selection (Details in Section 4.2.2)	Capable to identified feasible spaces for interpolation
To avoid local optimal distribution in the overall dataset	Interpolation favor on using local information rather than overall data distribution (Han *et al*., 2023)	Cross-over interpolation by pairing different inheritance – a different level of parents. (Details in Section 4.2.3)	Diverse data distribution generated

4.1.3 Model Training and Prediction

1) Model training

In this research, binary classification is adopted to develop the JIT-SDP model. The classification produces the output values of the prediction function with {0,..,1}. Thus, if the value of the prediction function $f(x)$ is greater than or equal to 0.5, the change is classified as a defect, otherwise, the change is predicted as clean. The prediction function is defined as follows:

$$f(x) = \begin{cases} 0, & if\ Y(x) < 0.5 \\ 1, & if\ Y(x) \geq 0.5 \end{cases} \quad (4.3)$$

Where x is the given code change and $f(x)$ is the possibility for x to contain defects. As introduced in the data extraction section, a code change, x is represented as a set of metrics in this formulation.

$$x_0^n = \{NS, NF, Entropy, LT, FIX, NDEV, AGE, NUC, EXP, SEXP\} \quad (4.4)$$

Thus, the model of JIT-SDP is represented as:

$$Y(x) = g\big(x(n)\big) \quad (4.5)$$

The mapping of g refers to a machine learning function. Existing frameworks employ machine learning techniques to search g that gives the best fit labeled for a given data in software metrics $x(n)$. In this research, Deep Q-Network (DQN) algorithm is used to map the given data. The details of the implementation of DQN are given in Section 4.3.

2) Prediction

Predictive effectiveness of the JIT-SDP model is evaluated according to the prediction accuracy and effort aware measures. For prediction accuracy of the software defects, the performance measures are based on precision, recall, and F-score. All these measures are calculated using a confusion matrix as shown in Table 24.

Table 24: Confusion matrix

	Predicted defect	*Predicted clean*
True defect	True Positive (TP)	False Negative (FN)
True clean	False Negative (FN)	True Negative (TN)

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.6}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.7}$$

$$\text{F0.5} = \frac{1.25 \times \text{Recall} \times \text{Precision}}{(0.25 \times \text{Precision} + \text{Recall})} \tag{4.8}$$

For effort-aware measures, two widely adopted effort-aware performance of prediction models are considered, which are ACC and POPT. ACC indicates the recall of predicting defective changes when 20 percent of the effort is required to inspect all changes according to top-ranked changes. *Popt* is the normalized version of the efforts-aware performance indicator introduced by Mende and Koschke (2010). This measure is based on the concept of the code churn-based Alberg diagram. Figure 40 shows the relationship in the Alberg diagram between Recall achieved by a prediction model and the amount of inspected code. In Equation 4.9, P_{opt} is equal to $1 - \Delta_{opt}$, where is the area between the optimal model and the prediction model.

$$Norm(P_{opt}) = \frac{P_{opt} - worst(P_{opt})}{optimal(P_{opt}) - worst(P_{opt})} \quad (4.9)$$

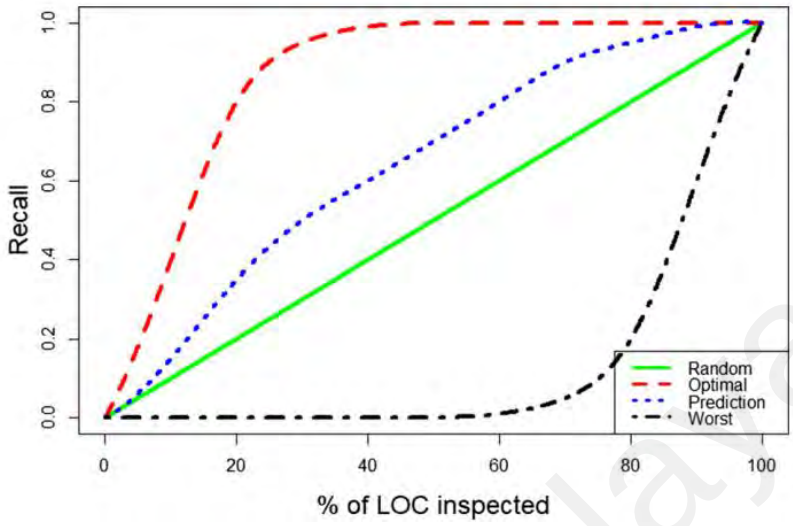


Figure 40: Alberg diagram based on *Popt*

4.2 Kernel Analysis and Crossover Oversampling Algorithm

Previous oversampling techniques cause generated minority samples to invade majority sample spaces, resulting in a lower recognition rate for minority samples. Using kernel analysis with spectral clustering and crossover interpolation as a method of oversampling, this study recommends Kernel Crossover Oversampling (KCO). The proposed algorithm attempts to generate new minority samples which combine features from two distinct data samples while at the same time being uniquely different. A basic intuition is that two instances, which are not necessarily close in distance, produce a new instance that is similar to both samples. The theory is derived from the chromosomal theory of inheritance, which considers the relevant features (software metrics) of defective changes as chromosomes. KCO attempts to produce a balanced class dataset with an increase in data diversity. The development of KCO is according to three fundamental phases as illustrated in more detail in Figure 41.

Algorithm 1 gives full procedure of KCO in producing the balanced class datasets. The first phase adopts KPCA to segregate the measurements for the minority samples as given in steps 1 to 6. In this process, KPCA transforms the original dataset into a simpler dimension dataset to analyze the occupied space in the data distribution. In the second phase which comprise of steps 7 to 11, spectral clustering divides the transformed data into several clusters. We then evaluate the fitness of each cluster based on the overlapped spatial distribution. By using the crossover operator of as in genetic algorithms, new samples are continuously synthesized to complete the oversampling of defect instances in the last phase as shown in steps 12 to 22. The newly generated samples combine with the initial data to produce a balanced dataset for training the JIT-SDP model. The following sections describe each phase in more detail.

Algorithm 1 Pseudo Code of Kernel clustering oversampling (KCO)

Input: Dataset of majority and minority class samples N; desired balanced proportion Pfp
Output: Balanced dataset at a set Pfp value
Procedure Begin

1) Split dataset N into majority class N_{maj} and minority class N_{min}
2) Compute the number of additional minority class to be generated T to attain Pfp
3) X_{new}: array for generated samples, initialized to 0
4) X_{newchk}: keeps count of the number of synthetic samples generated
5) Compute Kernel function of PCA for dataset N, KPCA = KernelPCA(n_components=2, kernel='rbf') where n_components = dimension of data and RBF = radial basis function
6) Transform dataset N into KPCA, $X_{tranformed}$
7) Create partitions of dataset $X_{tranformed}$ using Spectral clustering technique, $cluster = \{i \ldots 10\}$
8) For each $cluster_i$, sequentially compute spatial distribution fitness $F(cluster_i) = N_{maj} /(N_{maj} + N_{min})$
9) End for
10) Rank clusters according to spatial distribution fitness in increasing order
11) $Cluster_{best}$: Select top three clusters
12) While length of $X_{newchk} \leq$ size of N_{min}
13) Select samples $parent_a$, $parent_b$ from $Cluster_{best}$, where $parent_a$ and $parent_b$ are not equal
14) Generate a minority class synthetic sample X_i where X_i = average($parent_a$, $parent_b$)
15) Add X_i to X_{new} and increase X_{newchk} (i): $X_{newchk} = X_{newchk}$ (i) + 1
16) End while
17) While length $X_{newchk} \leq T$
18) Select samples $parent_a$, $parent_b$ from $Cluster_{best}$ and X_{new} respectively, where $parent_a$ and $parent_b$ are not equal
19) Generate a minority class synthetic sample, where $X_i = \boldsymbol{\lambda}\,(\boldsymbol{parent_a}) + (\mathbf{1} - \boldsymbol{\lambda})\boldsymbol{parent_b}$
20) Add X_i to X_{new} and increase X_{newchk} (i) : $X_{newchk} = X_{newchk}$ (i) + 1

21) End while
22) Add X_{new} to dataset N
23) Return N

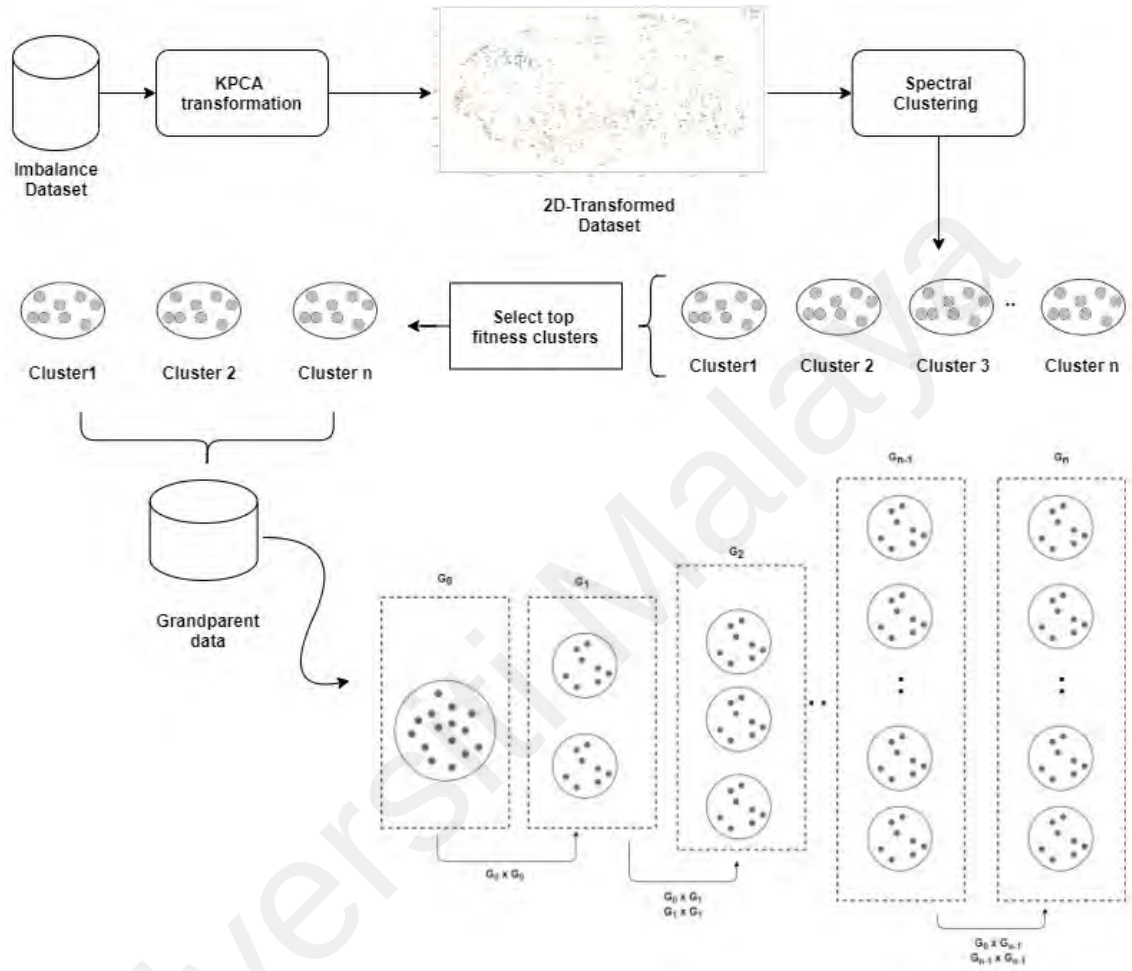


Figure 41: Overview of proposed oversampling technique

4.2.1 Phase 1: Diversity Measurement

Euclidean distance fails to be effective in nonlinear distributions (Xia *et al*., 2015) as presented in JIT-SDP datasets. JIT-SDP data typically exhibit a nonlinear distribution as a result of the uncorrelated relationship between software metrics. Several factors may affect the distribution, including clusters, non-convex shapes, or overlapping regions that are not accurately represented using a linear distance measure. The relationship between data points is unable to be well-represented by a straight line

calculated by Euclidian distance (Chen *et al*., 2022). Therefore, the measure does not accurately reflect the diversity of data points. Moreover, the JIT-SDP datasets duplicates as a result of the collection process for the metrics (Chen *et al*., 2016). Accordingly, the Euclidean distance measure unable to identify highly correlated or duplicated data samples within nonlinear distribution which failed to provide meaningful during information classifier training. As an alternative to handle highly correlated data, one may utilize feature engineering techniques such as Principal Component Analysis (PCA) (Lorena *et al*., 2019). PCA learns the original feature combinations linearly in new dimensional spaces. Nevertheless, PCA assumes that the learning data follow a linear separable Gaussian distribution. For real world data, particularly code changesets, linearly separated data is impractical due to the nonlinear structures of software metrics.

Prior studies have indicated that KPCA perform better than PCA for software engineering tasks (Zhao *et al*., 2021). Researchers have investigated the use of KPCA in software defect prediction, especially for the selection of features. Xu *et al*. (2019) found that basic classifiers including KCPA as a feature selection method achieve promising performance when compared to 41 baseline methods. Experimental results indicate that the framework outperforms PROMISE and NASA datasets, particularly in terms of F-measure, MCC, and AUC. Ho *et al*. (2022) utilized KPCA to reduce the dimensions of defect feature spaces from software metrics in order to extract essential information. A deep neural network (DNN) is then built to emphasize the semantic relations between software metrics so that defect data are distinguished from non-defect data using newly generated features from KPCA. Azzeh *et al*. (2023) examine the performance of nonlinear kernel functions and linear kernel functions in the context of different experimental parameters such as the granularity of the data, the imbalance ratio of the dataset, and feature subsets. According to their findings, RBF is the only

kernel function that exceeds linear and other nonlinear kernel functions. Nonetheless, reducing the dimensionality of a dataset did not often improve the accuracy of software defects prediction (Rosen *et al*., 2015; Śliwerski *et al*., 2005). Therefore, the KPCA should not be limited to measuring the similarity between features in software metrics. In other aspects of JIT-SDP, KCPA presents a promising alternative. As a result of KCPA, patterns in the data are identified that are not apparent by traditional methods of data representation, including handling high-dimensional datasets and capturing non-linear relationships among features. Therefore, the analysis of data distribution can be particularly important for data resampling.

This study employs KPCA to map multivariate of software metrics into a linear projection using a nonlinear kernel function. The process of data projection involves transforming the original data into lower dimension data. Data transformation process converts multivariate data into a new set of uncorrelated variables. Enabling efficient multidimensional scaling of JIT-SDP datasets with varying software metrics. In this way, the diversity analysis of JIT-SDP datasets by KPCA is independent of the data dimensions and becomes a scale-independent measurement. Therefore, the complex structure becomes easier to manage and allows the representation of features to be projected in a linear manner. Using a Radial Basis Function (RBF) kernel, KPCA provides a linear representation of the data while preserving the relative distances between pairs of data points that are close to the original space.

4.2.2 Phase 2: Data Partitioning

KCO makes use of spectral clustering that offers the advantages of simplicity while reducing complex multidimensional nonlinear datasets into clusters of data with similar characteristics in lower dimensions. Spectral clustering treats the data clustering problem as a graph partitioning problem without making any assumptions about the

shape of the clusters. Figure 42 shows the example of spectral clustering data distribution into several clusters.

The basic premise of spectral clustering in defect datasets is as follows: For a dataset with *n* samples $D = \{x_1, x_2, \ldots, x_n\}$ and each sample has variables $x_i = \{v_1, v_2, .., v_m\}$, where *m* is the number of software metrics. The clustering is based on dividing each sample into *k* clusters $C = \{C_1, C_2, .., C_k\}$. As a result, the samples in the clusters have a variance that is:

$$argmin_s \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\| \tag{4.10}$$

Where, $\mu_i$is the mean value of the samples in C_i.

For each cluster, the fitness is determined by calculating the number of samples for both the majority and minority classes. The intuition behind fitness evaluation for clusters is that regions with lower proportions of majority samples indicate lower overlapped spatial distribution. The following formula is used to calculate the fitness weight of each cluster:

$$Fitness(C_i) = \frac{Length\left(X_{maj}\right)}{Length\left(X_{maj} + X_{min}\right)} \tag{4.11}$$

Each cluster is evaluated in terms of its fitness, and the three best clusters are selected. In the selected clusters, more empty spaces are available, indicating areas that are suitable for interpolation. According to selected clusters, a pool of the most suitable templates for oversampling is identified.

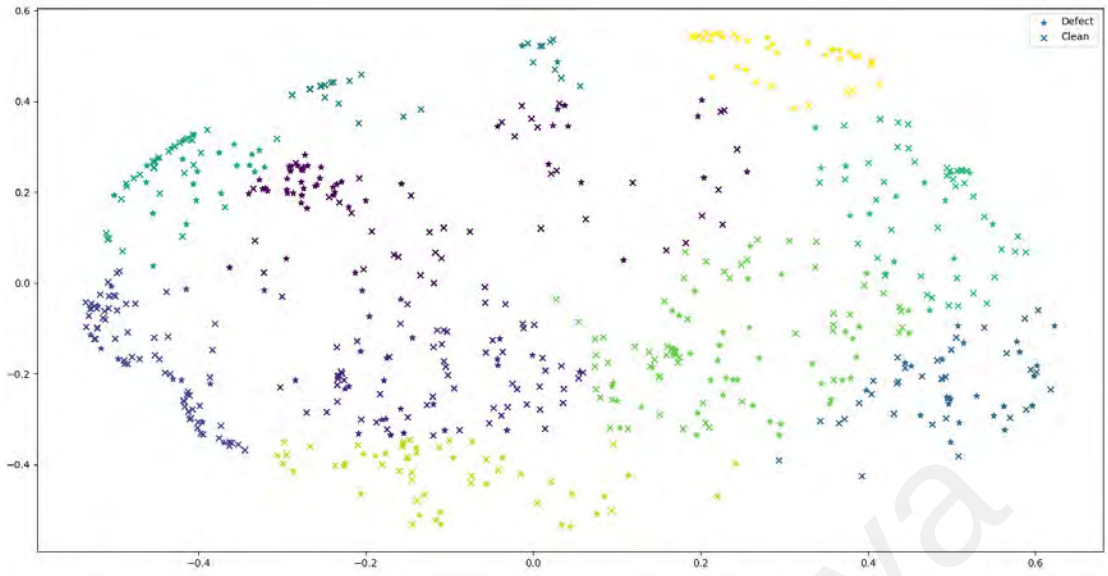


Figure 42: Spectral clustering within KPCA transformed data

4.2.3 Phase 3: Synthetic Data Generation

Interpolation in oversampling generates synthetic samples from existing minority class samples. One of the earliest methods for oversampling was the SMOTE, introduced by Chawla *et al*. (2002). SMOTE uses interpolation to generate synthetic samples from existing minority class samples. Even so, the use of SMOTE to develop prediction models still result in overgeneralization as it relies solely on the selection of nearest neighbour instances. Due to the limitations of SMOTE, a variety of modifications have been proposed, including Borderline-SMOTE (Han *et al*. 2005) and MWMOTE (Barua *et al*., 2014). Nevertheless, prior techniques unable to provide a diverse and balanced set of synthetic samples from datasets with high-dimensional input features. Cross-over interpolation provides an alternative way to generate synthetic samples by combining or "crossing over" the features of two existing minority class samples. Consequently, the generation new samples exhibit more representative and diverse to better reflect minority class distributions. In SDP, Bennin *et al*. (2018) first to propose crossover interpolation into oversampling process which named as MAHAKIL.

Mahalanobis distance is used to rank and divide instances into two groups. During the generation of new instances, two corresponding instances are chosen from each group. Synthetic instances tend to be more diverse when pairs of selected instances do not have a close distance between them. In comparison to SMOTE-based oversampling techniques, MAHAKIL offers superior performance and greater stability. Nevertheless, MAHAKIL fails to calculate the Mahalanobis distance when the number of instances of the minority class is smaller than the dimensionality of the sample. Thus, MAHAKIL does not function optimally when the number of minority class instances is lower than the number of metrics. Zhang et al. (2021) extended the work of Bennin et al. by adding K-means clustering to MAHAKIL to improve the recognition rate of positive samples. K-means is used to divide positive samples into clusters and then perform crossover interpolation to generate synthetic data. Nonetheless, K-means fails to generate an appropriate spherical partition of data in nonlinear datasets. Thus, an effective data partitioning such as that given by spectral clustering is needed to ensure the effectiveness of crossover interpolation. Spectral clustering produces clusters by partitioning the data based on the similarity of the data points and is useful for finding clusters in nonlinear datasets. Additionally, spectral clustering produces clusters with different shapes and sizes, which is advantageous in the context of crossover interpolation.

This study uses the crossover operator to generate new samples in the same manner as genetic algorithm. In this process, chromosome information contributes by two parents to generate a child. Chromosome information defined in this study as software metrics for JIT-SDP modeling purposes. In order to generate new samples, crossover operators combine the characteristics of two samples. Given two samples of $S_a^g = [a_1, \ldots$

., a_l] and S_b^g = [b_1, . . ., b_l] are two chromosomes crossed in *gth* generation and *l* is the length of chromosome or features, the child sample of $g + 1$ *th* generation is:

$$S_c^{g+1} = \lambda S_a^g + (1 - \lambda) S_b^g \tag{4.12}$$

Where λ is a random variable between a range of [0,1].

During the experiment, λ is set to 0.5 for generating the child samples. It means that the child samples inherit 50 percent of their characteristics from each of their parent samples. Figure 43 demonstrates an example of crossover operation during the generation of a new sample. In this context, the generation of new samples consists of a few steps.

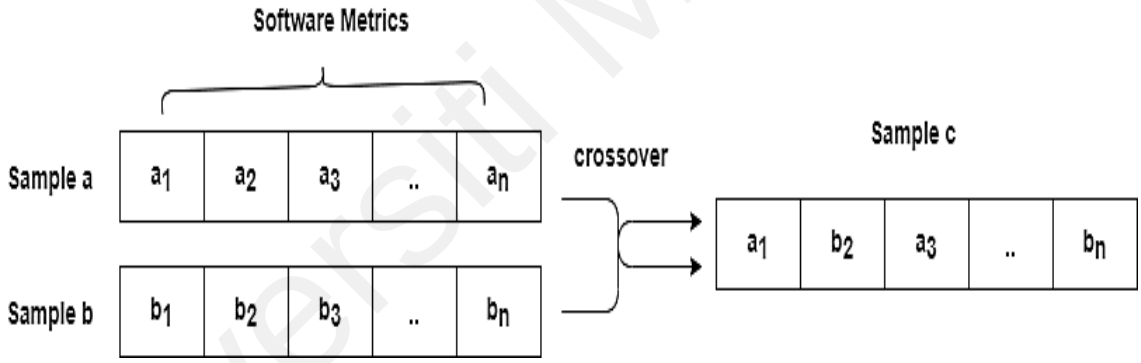


Figure 43: Example of multi-point crossover

Figure 44 illustrates the generation of new synthetic samples based on the level of inheritance. First, based on diversity measurements obtained from KPCA, the grandparent samples are identified, G_0. The samples from G_0 are then used to generate the G_1 set of new synthetic samples. To prevent new samples from entering the region of the majority class, the first parent node or grandparent act as a boundary such that all children produced in the future reside within the range of the parents. In the second generation G_2, samples from grandparent and samples from G_1 are selected as template

to generate new samples. In case of the interpolation at current generation is still not meet with the maximum samples, the process continues to crossover interpolate the samples within the previous generation until maximum number reach. The process of pairing the child nodes with older generations is repeated until the generated samples are sufficient (greater than or equal to the required number of samples). The pairing process is carried out using the sequential information inherited from the immediate parents of the instances beginning at G_1.

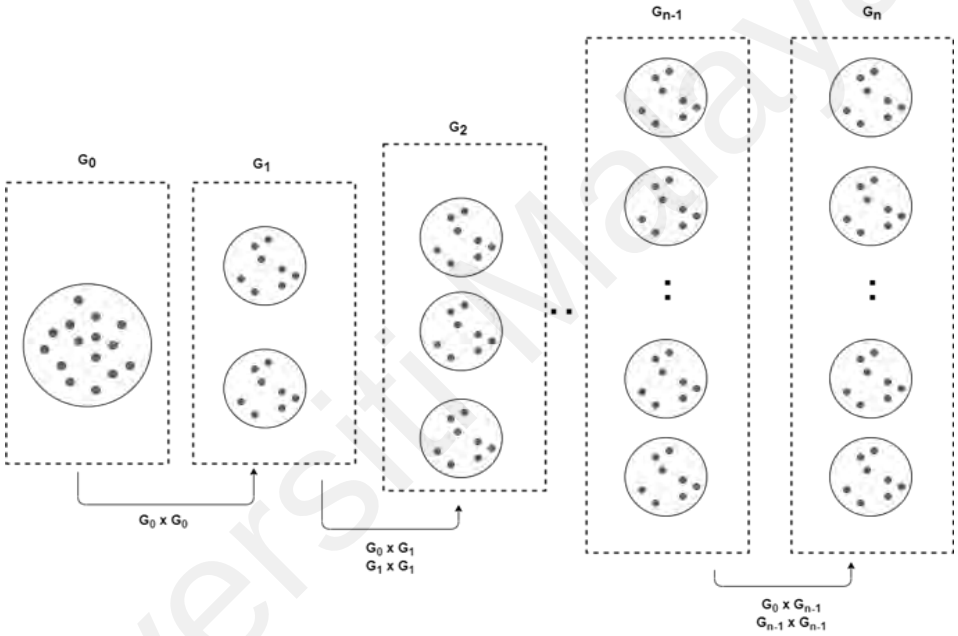


Figure 44: Crossover process across generations

In overall, the generation of new samples assumes that two samples that are not identical similar, as in being neighbors, and orderly merges two distinctive samples by considering them as parent samples. Child samples that are generated are distinctly unique but related to the original parent samples across generations. Thus, the newly generated samples are well distributed to occupied possible minority samples within selected data clusters. Ultimately, providing more information to the defect classifier. Additionally, by strictly working within the boundary of the minority class, crossover

operation helps in the prevention of data samples generated outside the decision boundary of the minority class. As a result, samples are derived from well-segregated parents that differ in the KPCA similarity measure, preventing duplicates.

4.2.4 Summary

Studies often rebalance samples by oversampling positive (defect) samples (Nam, 2014). However, Zhang *et al*. (2021) take the spatial distribution characteristics of samples into consideration. Overlapping data in spatial distribution will cause the boundaries between different types of samples to become blurred. As an extension of the above work, this work improves ability to cope with overlapped distribution based on KCPA, spectral clustering and cross interpolation. KCO is an alternative solution to improve classification performance when dealing with imbalanced data. KCO is incorporated in data pre-processing to enhance classification performance. Defect classifiers are expected to benefit from KCO by achieving better classifications. Further, KCO avoids generating erroneous or duplicate data instances that lead to high false positives by avoiding generating less diverse data points within the minority class.

4.3 Deep Q-Network in Just-in-Time Software Defect Prediction

To classify whether a change is a defective or clean change, existing classifiers employed supervised, unsupervised, and semi-supervised learning techniques. These techniques convert the JIT-SDP model into batch learning. Batch learning is learning on the entire training datasets at once to learn the pattern of the introduction of defective changes. However, the performance of these techniques is heavily affected in case of data drifting occurred in the software project datasets. Data drift is a change in the input data generation process, affecting the underlying probabilities of the data. Therefore, the

classifier technique is based on an alternative to batch learning approaches, by converting JIT-SDP into a sequential learning approach.

Reinforcement learning (RL) is a suitable technique to formulate sequential learning to learn the optimal prediction accuracy for agents interacting directly with an environment. To date, existing studies of JIT-SDP do not consider the RL technique to learn the pattern or behavior of defective changes in software projects since JIT-SDP is a recent emergent topic. Adopting RL into JIT-SDP is a challenging task due to the limited and imbalanced datasets in the software projects. Therefore, our work attempt to adopt Deep Reinforcement Learning (DRL) which enable the learning of software defect pattern through a combination of Q-learning framework and artificial neural network approach (DQN) by complex decision-making tasks throughout benefit and punishment policy. The mechanism of DQN is a learning process throughout trial-and-error, solely from rewards or punishments to produce the greatest reward.

4.3.1 Problem Definition

Given a defect dataset $D = \{(S_1, y_1), (S_2, y_2), (S_3, y_3), \dots, (S_n, y_n)\}$, where S_i is the feature vector for the i code change in the dataset and y_i represents the corresponding labels. Defect prediction forms the positive class in the datasets in case of $y = 1$ for further inspecting a defective code change and $y = 0$ for the accepted code change. The data is sorted with respect to time, preserving the sequential aspect and formulating the SDP problem as a sequential decision-making problem. The agent is given a series of code changes records, S_t at timestep t, and the agent takes an action of either approving the code change ($a_t = 0$) or inspecting the code change ($a_t = 1$). In return, the environment provides the agent with a reward based on the current classification performance and the next code change S_{t+1}. The agent is designed to minimize false-positive rates while maintaining a balance between prediction accuracy and false-

positive rate during classifying code changes. This is done using the reward function of *R*. Figure 45 illustrate the overall process of how the learning process works. Using a Markov Decision Process (MDP), the environment is represented as *S*, *a*, *R*, *T* with the following definitions:

States *S*: State S_t is the state of t_{th} change record where is called features vector x_t in the dataset.

Action *a*: The action space for this MDP is discrete given $A = \{0,1\}$. Where code reviewer during code review approves the code change as to be inspected ($a = 1$) or reject the code change for further inspection ($a = 0$).

Reward *R*: A reward r_t is a scalar which measures the fitness of the action a_t taken by the agent in the state s_t. Usually, the reward is positive value in case the agent chooses a preferable action and a negative value for the opposite action. In the context of this research, approving a defective change for further review process is preferred. Thus, the agent is rewarded positively by the environment. The reward mechanism for the MDP is explained in detail in the next subsection.

Transition probability *T*: The agent takes an action in the current state and environment gives back a new state. The transition from S_t to S_{t+1} is deterministic transition.

Episode *E*: An episode refers to an iteration of the agent interacting with the environment. This comprises of getting *S*, *a*, *R*, *T* until reaches a terminal state. During an episode, the agent making decision on each code change transaction one by one until reaches a terminal state ($L = 2000$). In this case, the agent takes action on $S_1, S_2, S_3 \dots, S_L$ in the first episode and $S_{L+1}, S_{L+2}, S_{L+3} \dots, S_{2L}$ in the second episode and continue the process until final episode ($E = 50$).

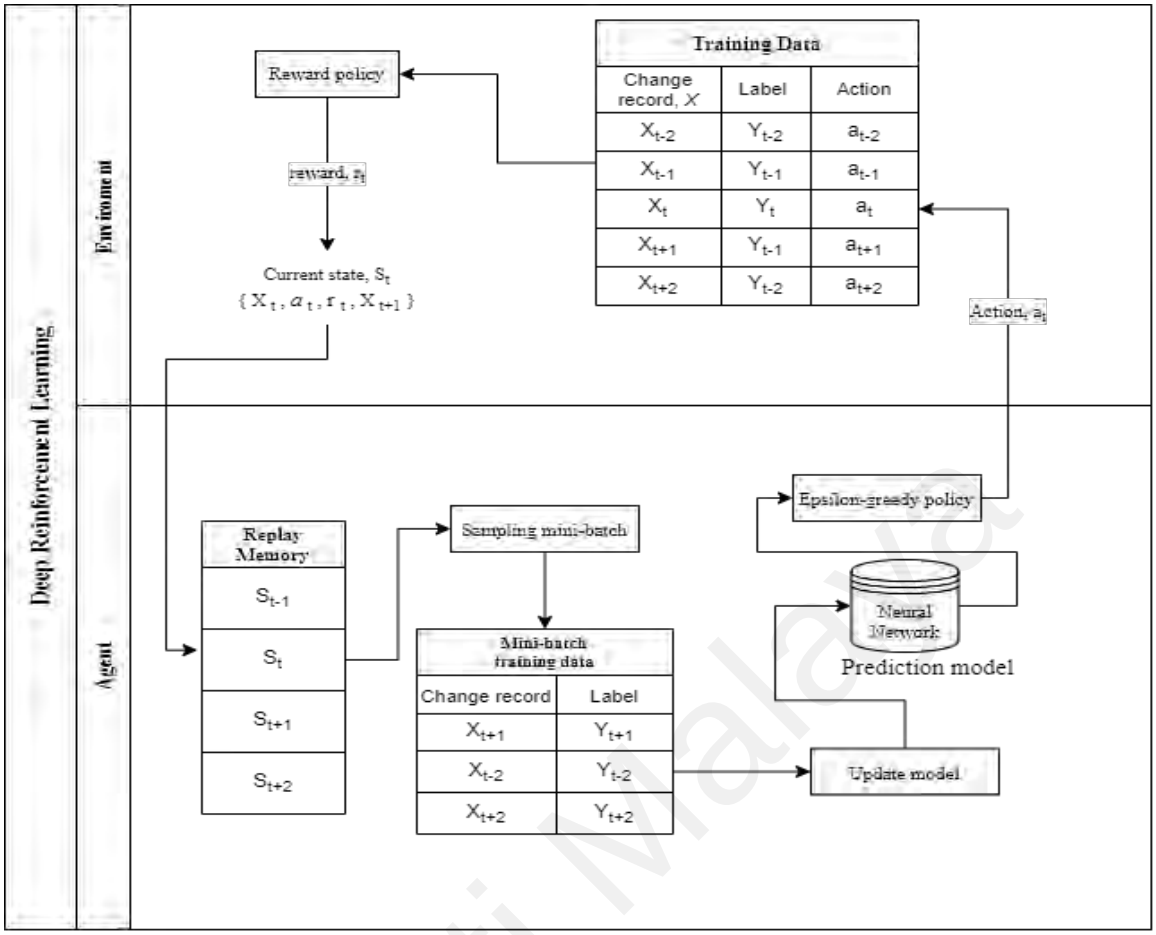


Figure 45: Conceptual diagram of deep reinforcement learning

4.3.2 Agent

In this research, Deep Q-Network (DQN) is chosen as the DRL algorithm. The DQN algorithm integrates Q-learning and neural networks. DQN aims to guide the choice of action given a state by predicting the expected Q-values of all possible actions. DQN training consists of determining the Q-value of a pair of state-action pairs. During neural network training, target actions are determined by the labels of data. During training iterations, the DQN agent is responsible for learning how to decide based on the given batch of data.

DQN agents seek to maximize the cumulative reward at a given time *t*. Cumulative rewards formally are follows:

$$R_t = \sum_{k=0}^{\infty} y^k r_{t+k}$$

Where *k* is the memory pool of the agent where its memory is stored, $k_t = (S_t, a_t, R_t, S_{t+1})$ at timestep *t,* and the discount factor is denoted as y.

To decide the action taken by *Q*-network, *Q*-values of a deep neural network with parameters θ, $Q(s, a, \theta)$ are computed. *Q*-values describe the possibility [0,1] of the given state, S_t to take each of the actions available. The problem definition is either approve or disapprove the code change for further code inspection. The neural network in DQN (Q-network) learns the parameters θ by performing Q-learning updates iteratively. At iteration *i*, the loss function is given:

$$L_i(\theta_i) = E_{(S,a,r,S')}\left[\left(r + y\left(max_{a'} Q(S', a'; \theta_i^-)\right) - Q(S, a; \theta_i)\right)\right]$$

Where θ_i is the parameters of the Q-network at i^{th} iteration and θ_i^- is the parameters of the target network model which is used to calculate target labels. The target model parameters are not trained, but they are periodically synchronized with the parameters of the main Q-network. The idea is that using the target network Q-values to train the main Q-network improves the stability of the training. The target model is updated, and the parameters are set equal to the main Q-network after *K* steps or mini-batch (*K*=64). Figure 46 illustrates the iterative updates of network models. Q-network updates take place based on random mini-batches from the memory pool. This process is considered a replay buffer process. The training process begins with *e* = 1 and uses a decay rate of 0.995 until minimum *e* =0.01.

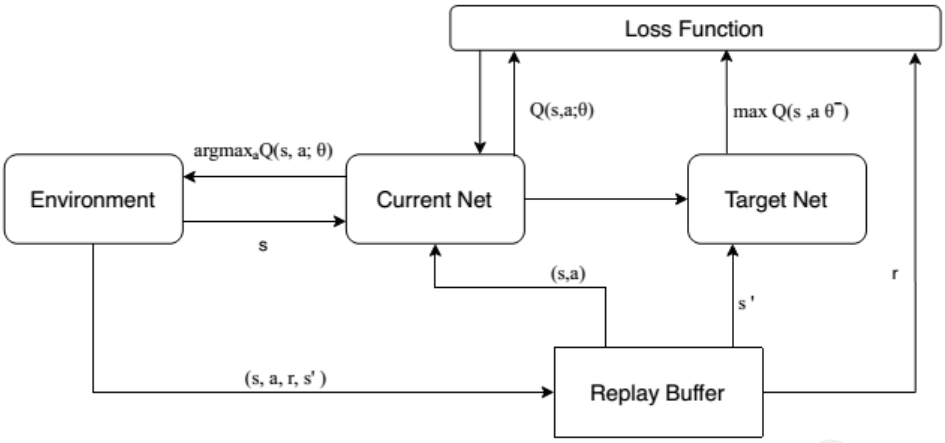


Figure 46: Updates of network models

The agent policy π to approximate the Q-values of the actions for the deep neural network is defined by epsilon-greedy policy. In this policy, the selection of actions can either be randomly selected or using the Q-value of the neural network. The usage of epsilon greedy action enables the agent to take advantage of prior knowledge and exploration to look for new options. Pseudocode of Algorithm 2 describes the process of action selection in π.

Algorithm 2 Pseudo Code of Epsilon-Greedy Action Selection

Input: Q-values generated by neural network, $\boldsymbol{Q_t(a)}$; Current state, *S;* epsilon, *e*
Output: Selected action, *a*
Procedure Begin

1) Select random number, *n* between [0,1]
2) If $n < e$, then
3) a = random action from the action space
4) Else
5) $a = \mathbf{argmax}\, \boldsymbol{Q_t(a)}$
6) Return *a*

4.3.3 Reward

The agent is rewarded with a reward r_t after it takes action a_t in state S_t which guides the agent to maximize the true labels of predicted defect data, y with in mind minimizing the false result. The reward function is defined as:

$$r(s_t, a_t, y_t) = \begin{cases} -1, & \text{if } a = 1 \text{ and } y = 0 \\ +1, & \text{if } a = 1 \text{ and } y = 1 \\ +1, & \text{if } a = 0 \text{ and } y = 0 \\ -1, & \text{if } a = 0 \text{ and } y = 1 \end{cases}$$

The reward function is inspired by the process of code review, where the developers do the peer code inspection. Figure 47 illustrates the peer code review process. Developers tend to inspect the actual defect in a code change during code review. Falsely defective predicted cause frustration and waste of effort to the developers. In contrast, truly predicted defects within a code change ease the process of code inspection. Thus, the aim is to guide the agent in making a decision according to this reward policy, where positive or negative reward are given for the action taken in predicting the label for a code change.

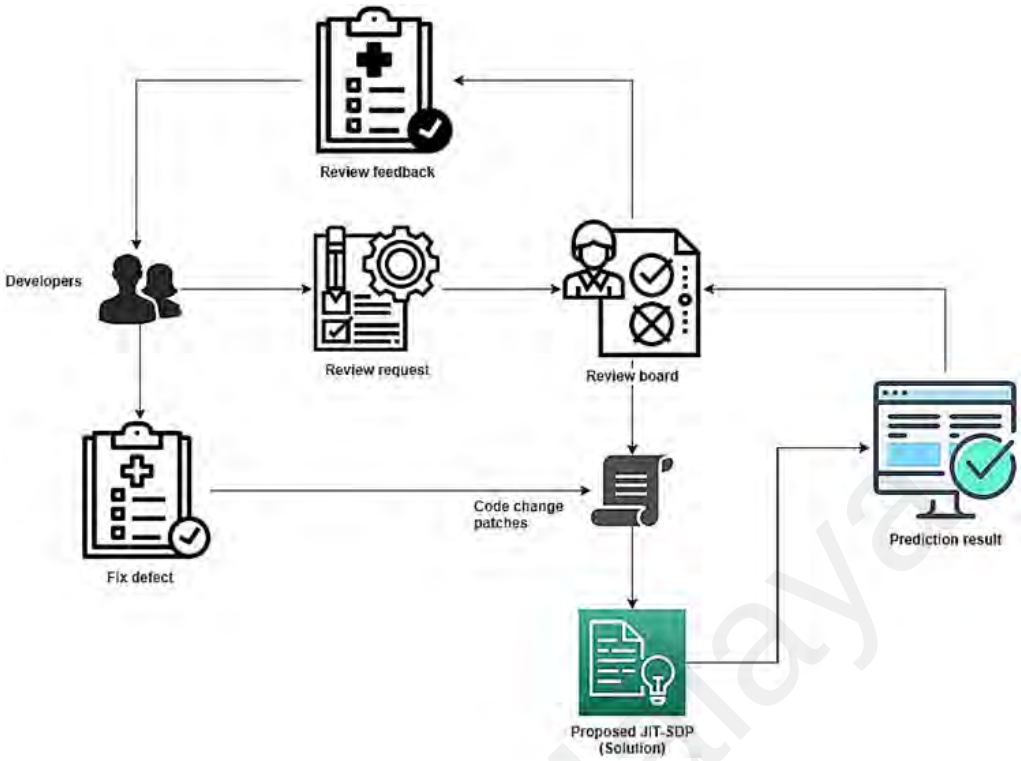


Figure 47: Code review with JIT-SDP model

4.3.4 Q-Network

Deep Q-Network involve combination of Q-learning and deep neural network to acts as nonlinear approximator for actions to be taken by the prediction model. In this scheme, neural networks known as Q-network are used to approximate the Q-function. Prior to training process of DQN, turning hyperparameters for the network model is essential to ensure the DQN able to produce better initial prediction results possible. For turning search, several hyperparameters are considered, which are discussed in the following section. After adjusting the hyperparameters, the DQN agent is now ready to begin training within the allocate episodes

Training of agent in DQN is summarized as shown in Algorithm 3. The training stops when the agent completely took action on the code change transaction in the training until the end of episodes. In this case, the number of episodes for the training is $N = 50$ episodes. For each episode, a random code change is chosen as the initial step,

to avoid overfitting toward a specific sequence of data. The agent is given the budget of 2000 changes, L to take action for each of the episodes. An episode ends when the agent covers target conditions, or it runs out of the budget as given in steps 25 to 26. In this case, two target conditions are defined (i.e., condition in steps 22 to 23). First, if an episode exceeds 200 false results, the episode is done. Second, in case of cumulative efforts for the current step exceed a threshold of 20 percent of the total effort of the training dataset, the episode is terminated. The conditions allow the agent to train under a constraint regime that ensures training is based on reducing false results and predicting defects with minimal amounts of effort. Additionally, these conditions ensure that the training process ends when the agent does not perform satisfactorily or when it exerts excessive effort, preventing the training process from being prolonged or ineffective. At the end of an episode, the target value is solely determined by the immediate reward (steps 14 to 15). In the event that the episode is incomplete, the target value is calculated based on the Q-learning update rule by considering the potential rewards that might be attainable in the next state (steps 16 to 17). By estimating the expected cumulative reward associated with different state-action pairs, the agent learns and guides its learning process.

During the training process, updates of weights for the network model involve the usage of the replay buffer concept. All the actions and observation states by the agent are stored in the buffer memory. Then a batch of samples is randomly selected from the memory for updating weights of the network model. This ensures that the batch is shuffled and contains sufficient diversity from older and newer samples. The idea behind buffer replay is to store the experience by the deque method and use a random subset of these experiences to update the network model instead of using only the most recent experience. This enables breaking potential harmful correlations within the training dataset.

Algorithm 3 Pseudo Code of Training DQN with Experience Replay

Input: Labelled training data, $D = (\boldsymbol{x_1}, \boldsymbol{y_1}), (\boldsymbol{x_2}, \boldsymbol{y_2}), (\boldsymbol{x_3}, \boldsymbol{y_3}), \dots, (\boldsymbol{x_n}, \boldsymbol{y_n})$

Output: Target network parameters, $\boldsymbol{\theta^-}$

1) Initialize replay memory, *M*
2) Initialize size of mini-batch, batch = 32
3) Initialize current network with random parameters, $\boldsymbol{\theta}$
4) For episode *e* = 1 until *N* do
5) Shuffle the training data D
6) Initialize first state $\boldsymbol{S_1} = \boldsymbol{x_1}$
7) For t = 1 until *L* do
8) $\boldsymbol{a_t} = \boldsymbol{\pi_\theta}(\boldsymbol{S_t})$
9) $\boldsymbol{r_t}, \boldsymbol{done_t} = \boldsymbol{Eviroment\_step}(\boldsymbol{a_t})$
10) Set $\boldsymbol{S_{t+1}} = \boldsymbol{x_{t+1}}$
11) Store ($\boldsymbol{S_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{S_{t+1}}, \boldsymbol{done_t}$) into memory *M*
12) If every current step reach size of mini-batch do
13) Randomly sample mini-batch ($\boldsymbol{S_j}, \boldsymbol{a_j}, \boldsymbol{r_j}, \boldsymbol{S_{j+1}}, \boldsymbol{done_j}$) from memory
14) If $\boldsymbol{done_j} = \boldsymbol{True}$ do
15) Set $\boldsymbol{f(q)_j} = \boldsymbol{r_j}$
16) Else
17) Set $\boldsymbol{f(q)_j} = \boldsymbol{r} + \boldsymbol{\gamma}\left(\boldsymbol{max_{a'}Q(S', a'; \theta_i^-)}\right) - \boldsymbol{Q(S, a; \theta_i)}$
18) End
19) Perform gradient descent on loss function, $\boldsymbol{L_i(\theta_i)}$
20) Set target network parameters $\boldsymbol{\theta^-} = \boldsymbol{\theta_i}$
21) End
22) If cumulative reward < 200 or cumulative effort > 20% of effort in *D*
23) $\boldsymbol{done_t} = \boldsymbol{True}$
24) End
25) If $\boldsymbol{done_t} = \boldsymbol{True}$ then
26) Break
27) End
28) End
29) End

4.3.5 Summary

Software defect prediction is formulated as a classification problem with a focus on improving the discovery of software defects. For this reason, many machine learning and data mining approaches are used to detect and predict defect inducing changes. However, whether deep reinforcement learning could be used to improve the performance of JIT-SDP is still unexplored. To bridge this gap, a framework of DQN is designed to address the problem of data drifting occurred in the software project

datasets. In this work, the focus is to use the DQN for the JIT-SDP in achieving results with good accuracy and the lowest numbers of false predictions. Technically, defect prediction problem is formulated as DQN formulation, and a reward function is proposed that aims to maximize the prediction accuracy and keeping a check on the rate of false alarm predictions. Agent of DQN is trained to predict the defect in code changes while under consideration of maintaining a balance between accuracy and false alarm rate. The training of JIT-SDP model using DQN is suitable for sequential or mini batches data which capable to adapt to data drift in a better way. By this solution, the issue of re-training JIT-SDP models is well handled in sequential learning approach which is an inherent problem with most classifiers. DQN for JIT-SDP is expected to help in improving the performance of classification for defects with lower in false positives prediction.

CHAPTER 5: EVALUATION OF IMPROVED JUST-IN-TIME SOFTWARE DEFECT PREDICTION FRAMEWORK

Evaluating prediction accuracy and effort awareness by the proposed framework is the goal of this chapter. In general, two main evaluations are carried out which reflect the comparison of two proposed solutions with the standard existing techniques. For the first section, the proposed solution of oversampling technique, namely Kernel Crossover Oversampling (KCO) is compared with several state-of-art resampling techniques to evaluate the performance when dealing with imbalanced datasets. In the second section, an evaluation of the proposed approach by using the DQN algorithm and KCO technique for JIT-SDP is given. The evaluation involves a comparison of performance with other well-known frameworks in the modeling of JIT-SDP classifier.

5.1 Predictions performance of Kernel Cross-oversampling

Imbalanced class distribution in JIT-SDP datasets is a problem for some conventional learning methods. In addition, spatial class overlap increases the difficulty for the predictors to learn the defective class accurately. The main objective of this experiment is to compare and evaluate the performance of KCO with baseline techniques in resampling data for modelling of JIT-SDP. In the experiment, six imbalanced datasets are selected from a public software repository which consists of overlap between classes residing in the datasets. The performance of the proposed resampling KCO is assessed by comparing it with other baseline techniques. For the comparison, several baseline techniques are considered which are ADASYN (He *et al*., 2008), SMOTE (Chawla *et al*., 2002), Borderline-SMOTE (Han *et al*., 2005), MWMOTE (Barua *et al*., 2014) and MAHAKIL (Bennin *et al*., 2018). The choice of the baseline techniques is due to these techniques not requiring any specific classifier to

work effectively. Thus, more advanced techniques in prior studies (Cabral *et al*., 2019; Tan *et al*., 2015; Zhu *et al*., 2020) are excluded in this analysis. Statistical analysis shows that the prediction model constructed using KCO provides more reasonable defect prediction results and performs best in terms of accuracy and F-score among all tested models.

5.1.1 Baseline Techniques

In verifying the stability of the KCO algorithm, several techniques such as ADASYN (He *et al*., 2008), SMOTE (Chawla *et al*., 2002), Borderline-SMOTE(Han *et al*., 2005), MWMOTE (Barua *et al*., 2014) and MAHAKIL (Bennin *et al*., 2018) are considered in the performance comparison. As some studies (Kamei *et al*., 2013; Li *et al*., 2020; Yang *et al*., 2017) considered random under-sampling (RUS) as the most efficient resampling technique for JIT-SDP, RUS also is included in the comparison.

ADASYN: ADASYN is proposed by He *et al*. (2008) and it assigns weights to the minority classes and dynamically adjusts the weights in a bid to reduce the bias in the imbalanced dataset by considering the characteristics of the data distribution. ADASYN algorithm incorporates a density distribution in automatically deciding the number of synthetic samples needed for each minority class sample. The learning algorithm is induced to focus on the hard-to-learn or classify examples within the minority class samples. Therefore, the samples generated are not equal for all samples.

SMOTE: Proposed by Chawla *et al*. (2002), this technique over-samples the minority class in a dataset by creating synthetic samples. SMOTE oversamples the minority class in a bid to make the dataset as balanced as possible based on the configuration parameter values. To generate these synthetic samples, each minority class sample is

considered, and the new samples are introduced along with the line segments that join any of the *k* minority class nearest neighbors.

Borderline-SMOTE: It is a modification of the SMOTE technique but with the main focus on harder-to-classify minority class data instances, which are referred to as borderline data instances. The algorithm first finds minority class instances that have more majority class instances as nearest neighbors than minority class instances and applies the SMOTE technique to such instances. This approach has the advantage of strengthening the borderline between the majority and minority class data instances.

MWMOTE: It is an oversampling technique proposed by Barua *et al*. (2014), MWMOTE divides positive samples into safety data, boundary data, and potential noise data, and then adopts different sampling strategies for different types of samples. It adaptively assigns the weights to the selected samples according to their importance in learning. The samples closer to the decision boundary are given higher weights than others. Similarly, the samples of the small-sized clusters are given higher weights for reducing within-class imbalance. The synthetic sample generation technique of MWMOTE uses a clustering approach to partition datasets and uses the Euclidean distance similarity measure to find very close class samples and synthetically generate samples based on the weights assigned to the minority class samples.

MAHAKIL: Bennin *et al*. (2018) introduced a synthetic oversampling approach based on the chromosomal theory of inheritance. Each sample of data is regarded as a chromosome. First, positive samples are divided into two initial populations according to the size of Mahalanobis distance, and then new offspring samples are synthesized by using the samples in the initial population to cross continuously. The offspring samples inherit part of the characteristics from the two parent samples, which ensures that the

offspring samples and the parent samples have certain similarities and some new characteristics.

5.1.2 Datasets

A total of six imbalanced datasets are evaluated which comprise Bugzilla, Columba, Eclipse.JDT (JDT), Eclipse.Platform (Platform), Mozilla, and PostgreSQL (Postgres). Note that all the datasets are imbalanced. The most imbalanced dataset, Mozilla, contains only 5% defects, while the most balanced dataset, Bugzilla, contains 36% defects. To ease the analysis of prediction results, these datasets are classified into two severity groups as shown in Table 25. Mild imbalance class is considered as datasets that comprise 25% and above for the percentage of software defects. For the high imbalance class, it is based on datasets that have less than 25% of the defects. The severity of the imbalance class represents the difficulty for data resampling in the imbalance distribution.

Table 25: Imbalanced class datasets

Project	Time	# Instances	Defect %	Severity
Columba	08/1998–12/2006	4455	31	Mild imbalance class
Bugzilla	11/2002–07/2006	4620	36	Mild imbalance class
Postgres	11/2002–07/2006	20431	25	Mild imbalance class
JDT	05/2001–12/2007	35386	14	High imbalance class
Platform	07/1996–05/2010	64250	14	High imbalance class
Mozilla	08/1998–12/2006	98275	5	High imbalance class

5.1.3 Experiment Settings

Artificial neural network algorithm is chosen as the classifier algorithm of the JIT-SDP model in this comparison. The classifier for the prediction is built using the resampled data generated by the resampling techniques. For the convenience of comparison, default hyperparameters are used for all compared techniques. In the experiment, three model performance prediction scenarios are considered. These scenarios are within project prediction, cross-project prediction, and timewise prediction. In particular:

a) Within project validation

The evaluation is conducted based on 10-folds stratified within project validation. Figure 48 presents the F-score values of KCO as compared to those of the baseline techniques respectively. The validation started with the splitting of data into 8:2 ratio, for both training and prediction datasets. Then, the training dataset undergoes 10-fold stratified within project validation. The datasets are divided randomly into 8-folds, 2-folds serve as training data, and the remaining fold serves as test data. In cross-validation, each fold is used as a testing dataset only once. Additionally, the data are folded so that every fold consists of the same proportions as the original dataset. The highest prediction model among these folds is selected for the final prediction. The selected model is used to predict the unseen data which is the prediction dataset. The final prediction result is recorded to show the credibility of the experiment results.

b) Cross project validation

For cross-project validation, the prediction of software defects is evaluated across different software projects. In specific, the models are constructed by one source of

software project and use these models to predict software defects on another target software project.

c) Timewise validation

Within the same project datasets, JIT-SDP takes into account the chronological order of changes in accordance with the commit date. Based on the assumption that the changes are divided into n parts, we first construct the models based upon the changes in part i and $i + 1$. The models will then be employed in predicting the changes in part $i + 4$ and part $i + 5$.

5.1.4 Performance Indicators

The evaluation measure is important to reveal the performance of the classifier, especially for imbalanced datasets. Some conventional measures lead to a wrong conclusion owing to the skewness of the class distribution (Li *et al*., 2018). For example, consider an extremely imbalanced dataset: 99 % of instances are of the majority class, and the remaining 1 % samples belong to the minority class. In case of using the accuracy measure which indicates how many test samples are correctly classified as the evaluation criterion, even if the classifier ignores all of the minority classes, it still reach a very high accuracy rate of 99 %. Therefore, this experiment also considered F1-score, which is a commonly used measure to evaluate classification performance. It combines Precision and Recall which is derived from a confusion matrix. The confusion matrix lists all four possible prediction results. If an instance is correctly classified as a defect, it is a true positive (TP); if an instance is misclassified as a defect, it is a false positive (FP). Similarly, for false negatives (FN) and true negatives (TN). Based on the four numbers, Precision, Recall, and F1-score are calculated. Precision is the ratio of correctly predicted defect instances to all instances predicted as defects ($\text{Precision} = \text{TP}/(\text{TP} + \text{FP})$. Recall is the ratio of the number of correctly

predicted defect instances to the actual number of defect instances ($\text{Recall} = \text{TP}/(\text{TP} + \text{FN})$). Finally, F1-score is a harmonic mean of Precision and Recall, $\text{Fscore} = \frac{1.25 \times \text{Recall} \times \text{Precision}}{(0.25 \times \text{Precision} + \text{Recall})}$. In measuring the diversity of data for resampled datasets (d), sparsity formulation is utilized. $sparsity = 1 - \frac{non\_zero\ (d)}{size(d)}$.

5.1.5 Experimental Results

This section presents the experimental results. The results focus on the performance of JIT-SDP model using different resampling techniques. The details are given in the following four subsections:

a) Analysis of Data Distribution

This section is about analysing the distribution of data after applying a set of baseline resampling techniques on six datasets. Sparsity of data distribution is an important consideration for many machine learning applications especially for high dimensional data. The larger sparsity data contain less information across data classes. Therefore, with a larger sparsity data, accurate predictions more difficult to acquire. Oversampling in training datasets reduces the impact of noise and improve sparsity of data distribution toward dense data. Dense data yields more informative data, which results in more accurate predictions due to more data available for model training.

Table 26 provides data distribution in the different types of resampling techniques. The result indicates KCO achieves the lowest sparsity values across all datasets. Considering the difference in sparsity values, only KCO able to provide significant difference value by 8% to 10% for data sparsity before resampling (original). The only exception is that data sparsity generated by KCO, MWMOTE, and MAHAKIL are similar for Mozilla.

Resampling low sparsity datasets becomes more difficult due to less significant variation among data points within the dataset, making them dense datasets. Among the datasets, Bugzilla exhibits the most dense distribution. As a result, baseline resampling techniques, including SMOTE, Borderline, RUS, ADASYN and MWMOTE fail to significantly improve data sparsity. Indeed, resampling in a dense dataset presents difficulties in generating more data samples in limited empty spaces. On the contrary, KCO provides better data distribution than baseline techniques with more robust performance in identifying empty spaces by using kernel function.

Overall, KCO produces more sparse data than SMOTE, Borderline, RUS, and ADASYN. KCO compares favourably with data generated by MAHAKIL and MWMOTE utilizing Mozilla, Bugzilla, and JDT. Considering that KCO generates more diverse data than other baseline techniques, contributing to data distribution diversity.

Table 26: Sparsity of data distribution

Techniques /Datasets	**Columba**	**Bugzilla**	**Postgres**	**JDT**	**Platform**	**Mozilla**
Original	**34%**	**26%**	**28%**	**33%**	**34%**	**28%**
KCO	23%	18%	20%	22%	22%	18%
Borderline	32%	26%	26%	29%	29%	21%
RUS	32%	25%	25%	31%	29%	20%
ADASYN	33%	26%	26%	31%	30%	23%
SMOTE	32%	25%	25%	30%	29%	22%
MWMOTE	30%	24%	22%	26%	26%	18%
MAHAKIL	28%	20%	22%	24%	25%	18%

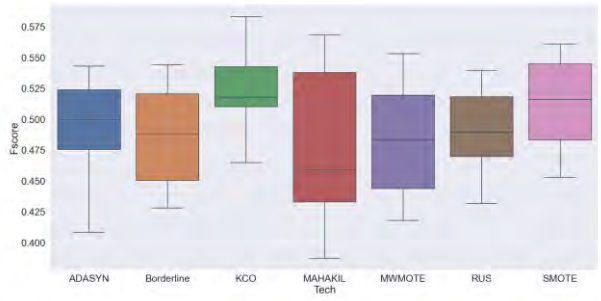
Note: Lower sparsity data indicates more suitable training data for machine learning

b) Analysis for Within Project-Prediction

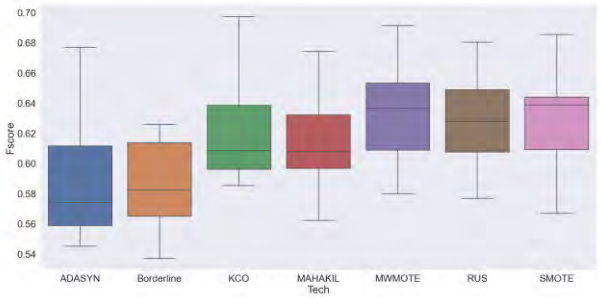
Table 27 shows the performance measures of KCO and baseline techniques based on accuracy and F-score. Before starting with the analysis, several forms of the result are adopted, and the corresponding meaning are as follows:

- The results with bold font represent the best one among the group of contrastive experimentation.
- The results with a red background indicate that the corresponding baseline model is the highest value among the comparison frameworks
- The results with a green background indicate that the proposed approach is the highest value compared to other frameworks model

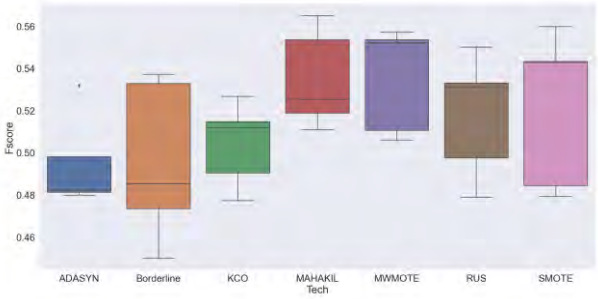
According to results shown in Table 27, KCO, MAHAKIL, and MWMOTE in general are the top performance techniques which outperformed other baseline techniques in terms of F-score measure for almost all datasets. Surprisingly KCO achieved the best performance among them, especially in the severely imbalanced dataset as in Platform and Mozilla. On average, KCO manages to achieve 52.6%, 32%, 35.2%, and 20.7% of the highest average F-score in Columba, JDT, Platform, and Mozilla respectively. Despite a slight improvement of KCO on F-scores compared to MAHAKIL and MWMOTE, the average accuracy across datasets indicates a consistent value between 71% to 80%. Nonetheless, even though RUS is considered the most widely applied in the context of resampling imbalanced datasets, the consistency of its F-score is almost similar to performance with other oversampling techniques such as ADASYN, SMOTE, and Borderline.



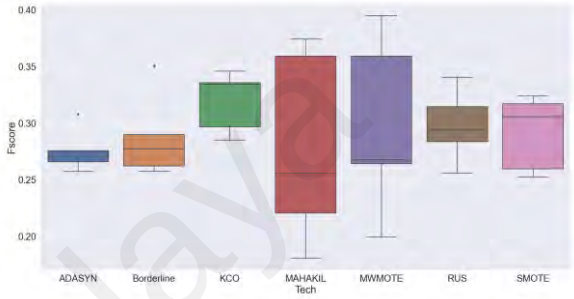
Columba



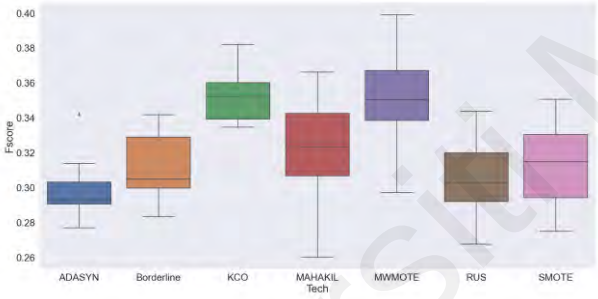
Bugzilla



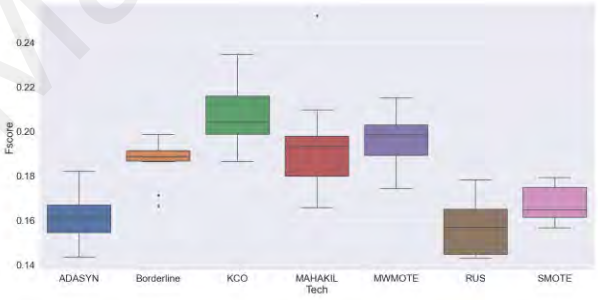
Postgres



JDT



Platform



Mozilla

Figure 48: F-score of six datasets for within project validation

Table 27: Prediction performance in F-score by resampling techniques

Techniques /Datasets	ADASYN	BORDERLINE	MAHAKIL	MWMOTE	RUS	SMOTE	KCO
Columba	49.2	48.6	47.9	48.5	49.1	51.2	**52.6**
Bugzilla	58.9	58.6	61.1	**63.4**	62.9	62.7	62.5
Postgres	49.5	49.6	53.5	**54.6**	51.8	52.2	50.4
JDT	27.6	28.8	27.8	29.7	29.8	29.2	**32.0**
Platform	30.0	31.2	32.0	34.1	30.6	31.2	**35.2**
Mozilla	16.1	18.6	19.4	19.5	15.7	16.8	**20.7**
Average	38.6	39.2	40.3	41.6	40.0	40.6	**42.2**

c) Analysis for cross project-prediction

The analysis further compares proposed KCO to the baseline techniques for cross project prediction as given in Table 28, Table 29, and Figure 49. For example, in Table 29, the case of "COL – BUG" means that Columba datasets is used as training project to construct the prediction model. Then the model predicts the changes in target project Bugzilla. From the result, KCO achieves approximately in range of 33% to 46% across projects prediction for mean of F-score as given in Table 28. KCO outperforms or obtains similar performance to other baselines in almost all datasets, as achieves in the highest average score for JDT, Platform and Mozilla cross prediction. In contrast to other baseline techniques, no single technique attains the highest average of F-score. In exception for ADASYN and Borderline achieving draws in Columba, Bugzilla, and Postgres. Furthermore, MWMOTE, MAHAKIL and RUS unable to produce substantially average in F1-score under cross project prediction setting.

Table 28: Average of F-score for cross project prediction

Techniques /Datasets	ADASYN	BORDERLINE	MAHAKIL	MWMOTE	RUS	SMOTE	KCO
Columba	**33**	**33**	31	31	25	30	**33**
Bugzilla	**27**	**27**	22	24	22	26	26
Postgres	**28**	**28**	26	26	27	27	**28**
JDT	40	39	26	25	26	38	**41**
Platform	40	39	40	25	26	38	**41**
Mozilla	44	43	40	34	33	41	**46**
W/D/L	0/3/3	0/3/3	0/0/6	0/0/6	0/0/6	0/0/6	**3/2/1**

Table 29: F-score of JIT-SDP models for cross project prediction

Source -Target	Baseline Techniques						Proposed Solution
	ADASYN	BORDERLINE	MAHAKIL	MWMOTE	RUS	SMOTE	KCO
COL – BUG	39	**46**	28	27	42	33	38
COL – POS	**53**	50	**53**	52	29	50	52
COL – JDT	24	25	24	25	27	26	**27**
COL – PLA	27	26	**29**	**29**	21	26	27
COL – MOZ	**22**	16	20	21	7	15	21
BUG – COL	32	32	35	30	33	37	32
BUG – POS	40	40	31	33	36	39	39
BUG – JDT	23	21	17	20	17	18	23
BUG – PLA	26	26	18	24	18	27	26
BUG – MOZ	15	14	7	13	8	12	11
POS – COL	36	35	36	36	35	35	36
POS – BUG	56	56	43	45	51	53	56
POS – JDT	17	17	17	18	17	17	17
POS – PLA	18	18	18	18	17	18	18
POS – MOZ	13	12	14	16	13	13	15
JDT – COL	46	45	36	46	36	50	50
JDT – BUG	55	55	42	5	42	48	55
JDT – POS	50	51	29	33	29	48	51
JDT – PLA	33	30	18	23	18	32	33
JDT – MOZ	16	15	6	17	6	14	18
PLA – COL	46	45	52	46	36	50	50
PLA – BUG	55	55	49	5	42	48	55
PLA – POS	50	51	49	33	29	48	51
PLA – JDT	33	30	30	23	18	32	33
PLA – MOZ	16	15	20	17	6	14	18
MOZ – COL	51	50	52	33	35	45	54
MOZ – BUG	51	45	49	26	49	51	53
MOZ – POS	52	54	49	53	43	50	54
MOZ – JDT	34	33	30	23	17	24	34
MOZ – PLA	33	35	20	35	18	33	34

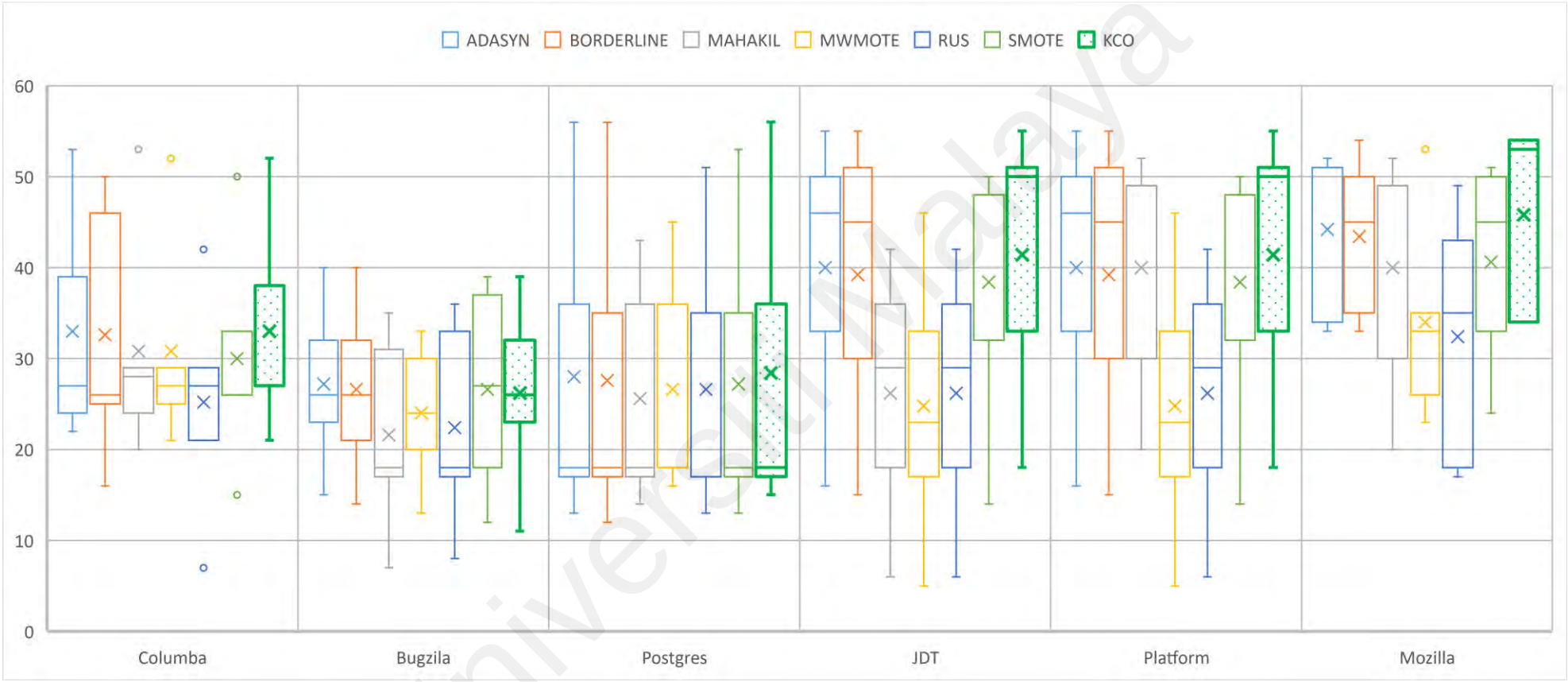


Figure 49: Resampling performance in cross project prediction

d) Analysis for timewise-prediction

Result of prediction performance in timewise validation scenario is further evaluated over the six project datasets shown in Figure 50 and Table 30. The result is evident that proposed technique KCO obtained highest average of F-score only for JDT dataset. MAHAKIL on the other hand surprisingly achieved better performance for Postgres, Platform and Mozilla. That is, MAHAKIL outperformed KCO and other techniques based on timewise prediction. Except for JDT datasets, KCO unable to resampling better performance for JIT-SDP model when compared to other baseline techniques. The result is inconsistence with previous within project predictions, where KCO is found to performs significantly better than all the baseline methods when considering F-score metric.

Table 30: Average of F-score in timewise predictions

Techniques/ Datasets	ADASYN	Borderline	RUS	SMOTE	MAHAKIL	MWMOTE	KCO
Columba	42	37	41	38	39	45	42
Bugzilla	52	53	53	55	52	53	51
Postgres	72	71	74	73	75	73	74
JDT	25	26	26	26	23	23	27
Platform	27	29	30	30	35	33	30
Mozilla	18	21	18	18	24	23	17

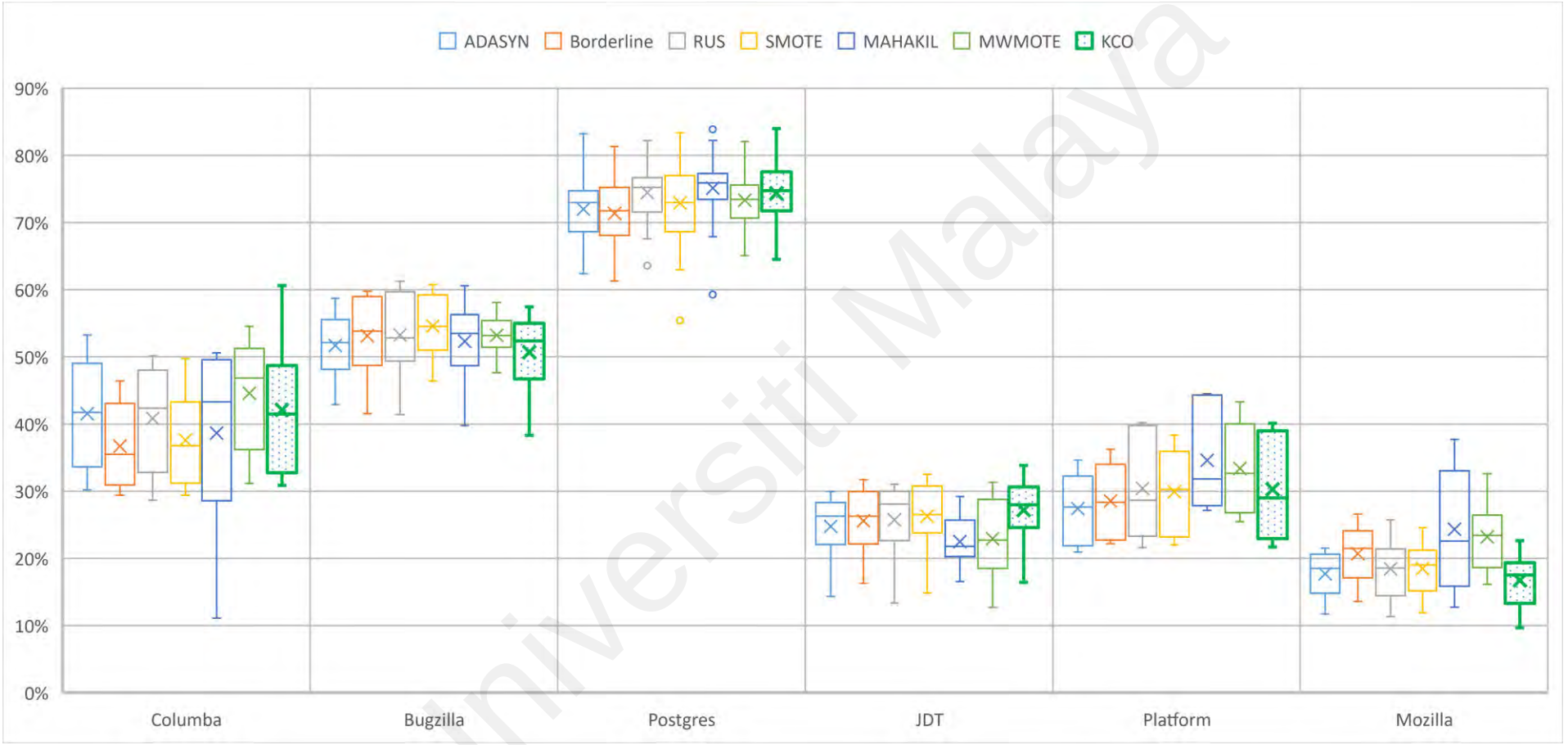


Figure 50: F-score of six datasets 10-fold timewise predictions

5.1.6 Discussion

1) Stability of resampling techniques

For stability, it mainly depends on the robustness of the techniques in facing severely imbalanced datasets. Severity of the imbalanced ratio heavily indeed affects the stability of resampling techniques. Due to this factor, it is observed that the resampling techniques have difficulty to achieve consistency of F-scores when dealing with large datasets as presented in Platform and Mozilla. Oversampling techniques with data partition embedded algorithms such as MAHAKIL, MWMOTE, and KCO exception are competent to attain a better F-score for these datasets, especially KCO which achieves the highest average score for all severely imbalanced datasets. To conclude, the result indicates that the stability of simple techniques such as RUS, SMOTE, and ADASYN is not good enough for imbalanced datasets in JIT-SDP. Contrary to KCO, the results show the most effective resampling technique when dealing with highly imbalanced data. KCO provides a more diverse distribution of data especially for severely imbalance datasets. Specifically, the generation of new synthetic data through multiple levels of inheritance from the original data, improving the diversity of the overall data distribution (Section 4.2.3). The prediction model can learn more discriminative patterns and make better-informed decisions, resulting in improved prediction performance. In contrast to mild imbalanced datasets, KCO fails to provide a reliable result because defect class have dense distributions. Consequently, KCO faces a challenge in conducting diversity analysis through KPCA. For other baseline techniques, mild imbalance datasets prove easier to resample the class distribution considered dense. The main factor is that through diversity measures (Euclidian distance and Mahalobis distance) by baseline techniques can provide meaningful attributes that effectively distinguish classes.

In cross project prediction, KCO demonstrates excellent performance in cross project prediction due to the consideration of the size of the data as an additional attribute for data resampling. Cross project prediction provides more information on the pattern of the defect class, resulting in a more diverse distribution. KCO takes advantage of the large size of defect instances and utilizes similarity analysis provided by KPCA to identify feasible regions for generating new samples (Section 4.2.1). Resulting in a more accurate performance in cross project prediction, even when dealing with varying class distribution imbalance ratios in the original datasets. In essence, KCO leverages the benefits of a larger dataset size and the insights gained from the similarity analysis, which contribute to its superior performance in cross project predictions.

2) Inconsistency in timewise prediction

Despite the fact that KCO underperformed in timewise predictions, this only reflects the specificity rather than the generality of the technique performance. In terms of timewise prediction, KCO is unable to achieve optimal results and actually performed worse than most of baseline techniques. One reason for this is that in KCO, the strength of data partitioning depends on the size of training data, since each training fold consists of various sizes that are not all equal. Identifying suitable regions for interpolation faced difficulty in smaller data sets due to a lack of similarity among data samples. This suggests that KCO is not suitable to resample the smaller data sets. Note that proper hyperparameters for data partitioning in KCO are useful to avoid this shortfall in smaller data sets.

5.1.7 Threat of Validity

A known validity of empirical experiments involves the quality of the data, which is often difficult to obtain and verify. Nevertheless, noise and outliers inherent within

datasets extracted from most open-source projects are likely to have significant effects on the prediction performance (Gray *et al.*, 2012). Therefore, applications of data cleansing techniques for noise detection and elimination remain open for future investigations

The effectiveness of the proposed KCO is dependent on the ability to assess the diversity of data using the KCPA. It is important to note that despite KPCA's benefits, its use entails a high cost. In cases of large data records, it is often difficult to compute the covariance matrix accurately, especially in cases of many features are presence. Thus, it is a requirement to allocate a greater amount of time and memory since these resources increase quadratically rather than linearly with the number of features. However, the issue is not be of significance when a few features are required. The challenge also applies to the approaches based on Euclidean distances. Concerning handling the computation of covariance matrices for large dimensional features, it is advisable necessary to employ a more advanced and time-efficient approach in dealing with covariance aspect of training datasets.

5.1.8 Conclusion

In this section, an experiment is conducted to compare eight resampling techniques for developing JIT-SDP models derived from six state-of-the-art software projects. This study presented an experimental setup aiming at mitigating the likely conclusion instability. Eight resampling techniques is compared for developing JIT-SDP models derived from six state-of-the-art software project. Despite of oversampling helps to improve classification on average, more uneven distribution of data points across different classes or clusters possible generated in case of having extreme dense data distribution. This situation produces a negative impact on the performance of downstream resampling algorithms that rely on accurate predicts of data classes or

cluster membership. Here techniques such as MAHAKIL and MWMOTE which comprise of data partitioning architecture are affected by this drawback. In light of this issue, KCO provides diverse data distribution by using a measure of similarity between data points to avoid the influence of dimension between different attributes of samples. By measure of similarity provided by KPCA, KCO linearly represent multivariate data into lower dimension while retaining its maximum variation. Therefore, covariance among data samples in imbalance distribution is exploited to find feasible spaces for interpolation which in turn reduce the effect of high multivariate data. Furthermore, in reducing near duplicated data after oversampling, KCO ensures nonlinear data distribution in the datasets are handled by cross interpolation approach. Through this approach, the generated data samples are produced by multiple level of pairing inheritance from the original data samples. As a result, KCO produces a more diverse set of data without compromising the origin information of the data distribution. Our work evaluates the performance of KCO on three different prediction settings. Experimental results show KCO consistently achieves higher F-score results for within-project and cross-project predictions. KCO achieves better overall classification performance, proving the feasibility of the approach in this study. Therefore, when dealing with an imbalanced class distribution task, KCO should be used for oversampling to improve JIT-SDP model classification performance. In future work, we plan to explore the impact of the different kernel functions in KPCA and the different activation functions in KCO on the performance of JIT-SDP models..

5.2 Effort Aware Performance of Deep Q-Network and Kernel Cross-Oversampling in Reducing False Positives

Effort awareness is required for an effective framework to produce a reliable JIT-SDP model. The effectiveness of the model is highly dependent on the quality of training data and the mechanism of the classifier in modelling defect patterns. Therefore, to achieve good performance, it is important to have a clear understanding of what factors affect the classification of software defects in code changes. In this experiment, the efficiency of the proposed framework with a combination of DQN and KCO as main modelling components for effort awareness is promising. The experimental results demonstrate that the proposed framework outperforms the state-of-the-art baseline on two different evaluation criteria: 1) accuracy in F-score for the prediction of defect with minimization of false positives and 2) achieving a low density of false positives for effort-aware evaluation.

5.2.1 Baseline Frameworks

The experiments are conducted by comparing JIT-SDP models based on proposed approach and state-of-the-art frameworks. Each framework shares the same data pre-processing in data extraction, features selection and normalization. The choice of resampling techniques and classification algorithm for training JIT-SDP models differs between these frameworks. For each framework, the properties within the framework are given in the following paragraphs.

EALR: Kamei *et al*. (2013) propose the EALR model which uses of logistic regression (LR) classifier and rebalances imbalanced class data with random under-sampling (RUS). EALR is used to predict the risk score of new changes in the testing datasets. For each change in the prediction, they would predict the value of defect

density by $D(c) = \frac{Y(c)}{Effort\,(c)}$ and sort these changes in descending order by the risk scores. Here *Y(c)* is 1 if change *c* is defect and 0 otherwise, and *Effort(c)* is the amount code inspection for the change. Thus, it provides the changes that need to be inspected first according to the testing dataset.

LR + KCO: To provide more options in the comparison, a combination of LR classifier with KCO for oversampling technique is given. The framework is inspired by EALR. Instead of resampling class imbalance data by RUS, we used KCO as the technique for handling the imbalance problem. For the effort awareness, it used a similar approach as EALR in the ranking of effort.

LT: Yang *et al*. (2016) utilized the same metrics as in Kamei *et al*. (2013) works to build a simple unsupervised model. The model uses only one metric among all the available metrics and sorts the changes in descending order according to the given measure. Among all candidate metrics, LT metric is chosen as the unsupervised model due to it is the best performance in most cases. The model predicts a risk score of changes by R(c) = 1/LT(c). Sorting of effort based on LT based on more defect prone need to be inspected first.

CBS: Huang *et al*. (2019) proposed a JIT-SDP approach by the concept of Classify Before Sorting (CBS). The framework in a similar supervised model to EALR using RUS to deal with data imbalance. CBS leverages the advantages of both supervised and unsupervised approaches by combining classification and sorting. For classification, logistic regression is used as the defect classifier. As for sorting, linear regression is based on a scoring list of changes in descending order by the ratio between each defect prone and its size. The intuition is that the smaller changes with high defect proneness must be inspected first.

NN + RUS: The approach uses the basis of EALR by changing its classifier into neural network algorithm. Implementation of the neural network instead of logistic regression in this framework to ensure that the model is more flexible and susceptible to overfitting. The network model also is fed by using the training dataset from the RUS technique. As for the ranking of effort, it follows the intuition of changes with high risk need to be inspected first.

NN + KCO: A combination of the supervised model of neural network algorithm with resampled data from the KCO technique. The framework provides the analysis of how the KCO affects the advanced classifier such in the deep learning approach. The framework also works by sorting the predicted changes based on the ranking of risk of defect prone.

DQN + RUS: A framework of combination DQN as a classifier with RUS as the resampling technique. The choice of DQN as a classifier for code change in the prediction model ensures that less false positive instances are predicted. For effort awareness, DQN with RUS also utilized the idea of changes with high defect proneness must be highly ranked for further code review.

5.2.2 Datasets

To verify the effectiveness of the proposed framework, experiments of effort awareness for JIT-SDP on six large open-source projects are conducted. The datasets include Bugzilla (BUG), Columba (COL), Eclipse JDT (JDT), Eclipse Platform (PLA), Mozilla (MOZ), and PostgreSQL (POS). The baseline six projects are large long-lived projects that cover a wide range of domains and sizes. Each instance corresponds to a change committed when the code is submitted to the version control system. Table 31 summarizes the statistics of the studied projects, including the time period of each

project, the total number of changes, and the percentage of defective changes. All datasets in this experiment are imbalanced with the percentage of defects ranging from 5 to 36%. Thus, the training datasets need to perform resampling class imbalance first.

Table 31: Statistics of datasets

Project	Time	# Instances	Defect %	Average of effort per change
Columba	08/1998–12/2006	4455	31	149
Bugzilla	11/2002–07/2006	4620	36	38
Postgres	11/2002–07/2006	20431	25	1001
JDT	05/2001–12/2007	35386	14	74
Platform	07/1996–05/2010	64250	14	72
Mozilla	08/1998–12/2006	98275	5	107

5.2.3 Performance Indicators

For prediction of software defect performance, accuracy and F-score are used as the measurement indicators for the comparison. Accuracy metric is important for measuring the results in which true positives and true negatives are more important. In this respect, the accuracy metric provides the performance in terms of code change classification whether the predicted changes are true defects or true clean. As for F-score, it is intended to capture the prediction performance in imbalanced class distribution as per existed in the prediction datasets. Furthermore, F-score provides a harmonic mean of precision and recall which gives a better measure of incorrectly classified cases than the accuracy metric.

For evaluating the predictive effectiveness of a JIT-SDP model, the effort required to inspect those changes predicted as defect-prone is considered to find whether they are defective changes. Consistent with Kamei *et al*. (2013), the code churn which describes the total number of lines added and deleted by a change is used as a proxy for the effort required to inspect the change. Similar to Mendes *et al*. (2010) works, *ACC* and *Popt* are used to evaluate the effort-aware performance of the JIT-SDP models. *ACC* denotes the recall of defect-inducing changes when using 20% of the entire effort required to inspect all changes. However, the *ACC* metric is unreliable if too many false positives are predicted, as this can mislead the nature of the prediction with limited effort. Therefore, in this experiment, we improvised the *ACC* metric so that we only consider recall of true positives under 20% of the total effort, which we refer to as *Benefit*. *Popt* is the normalized version of the effort-aware performance indicator originally introduced by Mende and Koschke (2010). Note that both *Benefit* and *Popt* with a higher value are preferable.

5.2.4 Prediction Settings

In this experiment, the involved prediction settings include within project prediction, cross-project prediction and timewise prediction.

- Within project prediction is performed within same project. In this setting, the dataset is divided into ten folds, nine of which are used as training datasets and one as testing datasets. Cross validation implies that each fold is only used as a testing dataset once. Futhermore, each fold consist of the same class proportion as the original dataset.
- Timewise prediction follow a certain time order which based on timesensitive validation strategy. For each project datasets, the changes are grouped into the same month in chronological order according to the commit date. For training

datasets, 24 months commits are grouped as training instances. The testing datasets is not immediately following the training datasets, but there is a gap of 2 months between training and testing datasets.

- Cross project predictions are performed across different project. In this setting, one project serves as a training dataset and another project acts as testing dataset. This experiment evaluate six subject project, therefore a total of 30 prediction value are produced ($n \times (n\text{-}1)$).

5.2.5 Hyperparameter Tuning

Choosing the right configuration of hyperparameters for neural network model within DQN is essential before the actual training process. Estimations for learning rates, epochs, the number of hidden layers and the size of hidden units in each hidden layer used to train a neural network vary according to software project datasets. In this respect, Neural network architecture in DQN is based on four layers which comprises of an input layer, two hidden layers and an output layer. Input layer consists of 12 nodes with reflected the number of software metrics considered by the JIT-SDP datasets. For each of hidden layer, the number of nodes is tuned with different values. The number of neurons must be tuned to the solution complexity. Lastly, the output layer contains two nodes for Q-values that are responsible for DQN's actions. The summary of the architecture of network model in DQN are given in Figure 51.

Aside from the number of nodes in each hidden layer, the learning rate of the network model and the number of epochs are also tuned for optimal network updates. For each of hyperparameter combinations, a Hyperband tuner (Li *et al*., 2018) is used to search for the optimal combination configuration given these hyperparameters. Table *32* and Table 33 contain a list of hyperparameters tuned to the current optimal configuration for the datasets used in this framework.

Figure 52 provides accuracy results from hyperparameter tuning for each trial in software project datasets. Each row represents a trial, and the percentage values indicate the accuracy achieved by the tuned neural network configurations for the corresponding trial and dataset. Results indicate that accuracy improves over time, suggesting that the Hyperband algorithm effectively explores and refines the hyperparameter space to lead to improved model accuracy. The accuracy for each dataset shows an irregular rise from trial to trial, indicating the stochastic nature of the optimization process. Accuracy values exhibit variability across trials for all datasets. This observation proves the importance of conducting multiple trials to mitigate randomness during hyperparameter tuning. Mozilla exhibits the fastest improvement in accuracy across trials, followed closely by JDT and Platform datasets, while Columba, Bugzilla, and Postgres datasets demonstrate a slower convergence rate. The observed trends and analysis provide valuable insights into the effectiveness of tuning processes to optimize neural network configurations in DQN framework.

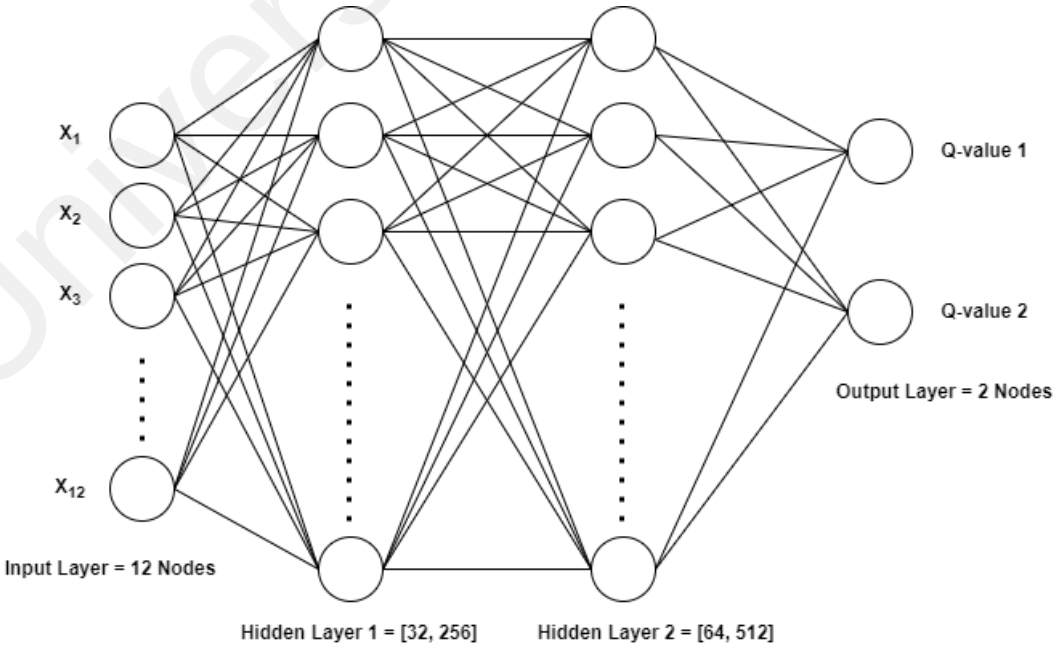


Figure 51: Architecture of Network model in DQN

Table 32: Hyperparameters considered in Q-network of DQN

Hyperparameters	Description	Rational	Range (min-max)
Hidden layer 1	Number of neurons in the hidden layer 1	Assist the neural network in selecting the best combination of features based on predefined hidden nodes	[32 -256]
Hidden layer 2	Number of neurons in the hidden layer 2	Assist the neural network in selecting the best combination of features based on predefined hidden nodes	[64-512]
Learning rate	Determine how fast the model learns and generalizes from data to reach its desired accuracy level quickly	A high learning rate speeds up training but also lead to overfitting if set too low	[0.001-0.1]
Epoch	Epochs are the repetition of learning process until the network system calculates an optimal solution based on the given data inputs	As the number of epochs increases, the weights in the neural network are changed more frequently, and the curve shifts from underfitting to optimal to overfitting.	[0-50]

Table 33: Configuration of optimized hyperparameters in network model of DQN

Hyperparameters /Datasets	Nodes in hidden layer 1	Nodes in hidden layer 2	Learning rate	Epoch
Columba	512	16	0.01	50
Bugzilla	64	128	0.001	50
Postgres	64	128	0.01	50
JDT	320	16	0.01	50
Platform	160	80	0.001	50
Mozilla	480	240	0.01	50

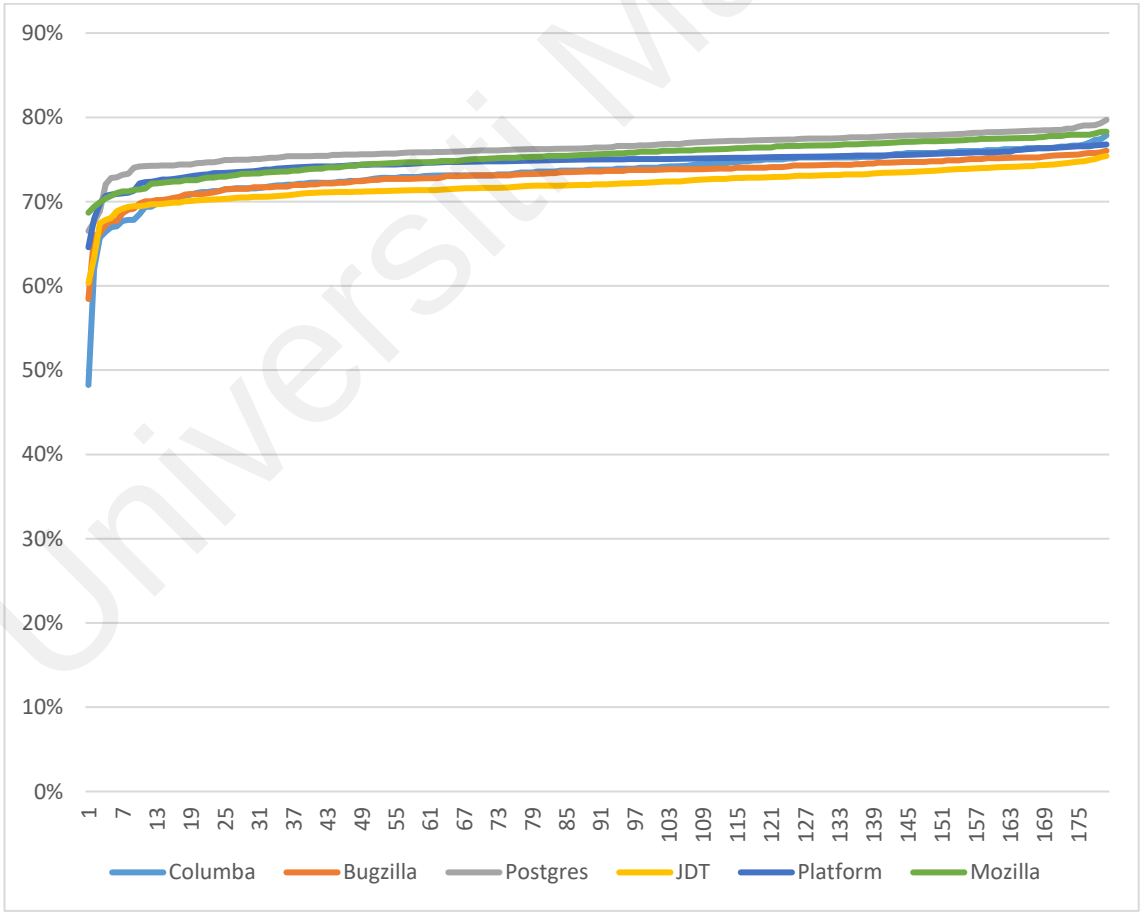


Figure 52: Accuracy of network model for each tuning trial

5.2.6 Experimental Results

In this section, the experimental results of DQN in JIT-SDP when compared to baseline frameworks. The results given are based on the average performance for each prediction settings. Several forms of the result are presented in Table 35 until Table 43. The results in bold font represent the best values among the group of contrastive experimentation. A red background value indicates the corresponding baseline framework, which is better than the proposed framework and other frameworks. In contrast to a green background, it represents the result of the proposed framework better than other frameworks.

1) Within project prediction

Table 34 Table 35 and Table 36 respectively, summarize the proposed frameworks and baseline models with the results of F-score, Benefit, and Popt over the six projects. From Table 34, following results are recorded. First, the proposed framework of DQNKCO outperformed baseline frameworks in three datasets of Postgres, JDT and Mozilla with highest F-score of 58%, 33%, and 22% respectively. Secondly, in the remaining datasets, NNKCO and CBS produce better F-score in Columba, Bugzilla and Platform datasets. The resulting F-score is consistent with the initial expectation. KCO assists both NN and DQN classifiers in providing reasonable training sets, which results in better performance than those without KCO frameworks. According to Table 35, effort awareness based on *Benefit* demonstrates CBS, NNRUS, and DQNRUS perform better than DQNKCO in Bugzilla, Platform, and Mozilla, respectively. Despite that, DQNKCO still remains the best framework for the JIT-SDP model, outperforming all other frameworks by achieving the highest *Benefit* with 27%, 23%, and 14% of defect predicted in Columba, Postgres and JDT respectively. In term of *Popt* metric as shown in Table 36, DQNKCO achieves the best performance in two out of the six projects,

which is comparable to LT's performance. From the result, JIT-SDP model by DQNKCO is unable to produce an optimal model performance based on *Popt* when considering the current datasets. On average, the prediction models by DQNKCO only achieve around 60% for highest area under curve of *Popt*.

Table 34: Prediction accuracy in within project prediction

Datasets/ Frameworks	**Columba**	**Bugzilla**	**Postgres**	**JDT**	**Platform**	**Mozilla**
EALR	51.4	57.8	51.9	28.7	27.6	14.2
LR + KCO	53.7	55.9	49.6	28.6	32.3	13.9
LT	34.7	40.6	30	17.2	19.6	7
CBS	54.6	**60.2**	55.6	30	28.2	14.4
NN + RUS	52.1	55.7	52.3	29.2	33.9	15.5
NN + KCO	**58.3**	56.9	51	32.1	**38.2**	15.4
DQN + RUS	56.9	52.6	56.2	29.6	36	18
DQN + KCO	52.8	55.6	**58.2**	**32.8**	36.8	**21.8**

Table 35 : Benefit of effort awareness in within project prediction

Frameworks/ Datasets	**Columba**	**Bugzilla**	**Postgres**	**JDT**	**Platform**	**Mozilla**	**Wins**
EALR	23.9	32.8	22.2	12.8	14.3	4.6	0
LR + KCO	24.0	21.3	19.3	10.3	13.9	4.6	0
LT	16.6	19.2	12.9	8.4	9.7	2.2	0
CBS	24.3	**33.7**	17.7	9.1	7.1	5.5	1
NN + RUS	25.3	30.5	22.3	13.9	**15.2**	5.3	1
NN + KCO	25.2	21.6	20.1	12.1	14.4	5.1	0
DQN + RUS	23.4	22.7	23.1	12.1	14.7	**7.8**	1
DQN + KCO	**26.6**	30.6	**23.2**	**14.3**	14.7	6.5	**3**

Table 36 : *Popt* performance in within project prediction

Frameworks/ Datasets	Columba	Bugzilla	Postgres	JDT	Platform	Mozilla	Wins
EALR	40.5	61.1	46.6	50.6	**51.0**	46.0	1
LR + KCO	35.7	47.4	36.7	40.0	45.8	44.3	0
LT	**48.1**	48.6	**55.3**	50.2	48.0	52.3	2
CBS	45.4	53.3	49.0	49.3	47.5	52.7	0
NN + RUS	42.0	61.6	45.1	48.6	48.0	49.3	0
NN + KCO	30.0	50.2	35.5	50.6	42.3	46.3	0
DQN + RUS	28.3	50.2	34.2	**54.7**	45.2	54.2	1
DQN + KCO	40.0	**62.6**	43.5	50.8	42.3	**55.1**	**2**

2) Cross project prediction

Further comparison of proposed framework of DQNKCO with baseline frameworks for cross predictions are shown in Table 37, Table 38 and Table 39 for F-score, Benefit and *Popt* respectively. For F-score based on average value in Table 37, DQNKCO significantly perform better than all baseline frameworks in majority of datasets by three out of six projects. The average F-score of DQNKCO ranges from 26% to 45%. DQNRUS is also able to achieve the highest F-scores in Postgres and Platform thanks to the efficacy of DQN in classifying code change in reducing false alarm results. Even though DQNRUS and DQNKCO perform better in almost all projects, under across prediction, EALR did better for the Bugzilla project. According to Benefit in Table 38, DQNKCO followed by DQNRUS and EALR outperform other baseline frameworks. Overall, the observation indicates DQNKCO provides best performance, identifying 12% to 24% defects based on a 20% effort to inspect all changes. Moreover, effort

awareness under *Popt* in Table 39 shows that DQNKCO unable to perform better than other baseline frameworks except for JDT project. Simple unsupervised model by LT on the other hand, dominates the highest *Popt* metric for three out six projects. Surprisingly, the prediction model under the current baseline frameworks and proposed framework are incapable to generate near optimal solutions with only less than 60% of *Popt*.

Table 37: F-scores in cross prediction of baseline frameworks

Datasets/ Frameworks	Columba	Bugzilla	Postgres	JDT	Platform	Mozilla
EALR	28	**29**	28	35	33	39
LR + KCO	27	28	28	34	31	38
LT	23	22	24	26	26	28
CBS	27	28	28	34	32	37
NN + RUS	31	27	28	31	37	42
NN + KCO	30	26	27	38	35	41
DQN + RUS	30	22	**31**	33	**40**	38
DQN + KCO	**38**	26	27	**41**	35	**45**

Table 38: Benefit of effort awareness in cross project prediction

Datasets/ Frameworks	Columba	Bugzilla	Postgres	JDT	Platform	Mozilla	Wins
EALR	8	8	**14**	11	14	15	1
LR + KCO	7	9	13	12	14	15	0
LT	11	10	12	13	12	14	0
CBS	7	8	13	11	14	14	0
NN + RUS	9	8	13	14	16	18	0
NN + KCO	7	11	13	14	14	17	0
DQN + RUS	7	12	13	15	**18**	20	1
DQN + KCO	**12**	**22**	13	**18**	13	**24**	**4**

Table 39: *Popt* performance in cross project prediction

Frameworks/ Datasets	Columba	Bugzilla	Postgres	JDT	Platform	Mozilla	Wins
EALR	35	45	52	42	47	39	0
LR + KCO	33	46	51	43	41	38	0
LT	**53**	51	50	49	**50**	**50**	**3**
CBS	37	45	51	46	46	40	0
NN + RUS	36	51	**55**	43	46	38	1
NN + KCO	33	**56**	47	44	42	39	1
DQN + RUS	35	53	47	40	47	46	0
DQN + KCO	35	52	47	**55**	39	41	1

3) Timewise prediction

The empirical results presented in Table 40 demonstrate that compared to the baseline framework with random undersampling (RUS), the classification performance of DQNKCO is superior. Additionally, CBS, NNRUS, and EALR frameworks are comparable based on a "win/draw/loss" analysis. These frameworks achieve comparable or drawn in F-score performance for Postgres and JDT datasets. In conclusion, the experimental result in timewise prediction indicates that DQNKCO is superior to other baseline frameworks in achieving a good F-score by achieving two wins. Results for effort awareness for each timewise prediction are given in Table 41 and Table 42. For *Benefit* metric, based on average value, DQNKCO predicts the highest scores with 20%, 15%, and 6% of defects when using 20% effort for Postgres, Platform and Mozilla. Nonetheless, the performance of DQNKCO and NNRUS in the Mozilla project is comparable. Meanwhile, *Popt* for DQNKCO performs worse than baseline frameworks, particularly in lower imbalance ratio datasets. Overall, DQNKCO's performance for *Popt* is unable to outperform other frameworks

in terms of producing superior results. DQNKCO generates an average score of between 34% and 51% for achieving the best models for effort awareness.

Table 40: F-scores in timewise prediction of baseline frameworks

Datasets/ Frameworks	Columba	Bugzilla	Postgres	JDT	Platform	Mozilla	Win/Draw/Loss
EALR	39	52	51	25	28	16	1/1/4
LR + KCO	38	51	50	25	27	15	0/0/6
LT	28	42	30	16	20	8	0/0/6
CBS	38	51	51	26	28	16	0/2/4
NN + RUS	40	51	51	26	30	17	0/2/4
NN + KCO	40	50	50	25	30	15	0/0/6
DQN + RUS	41	50	50	26	29	15	1/1/4
DQN +KCO	37	49	48	23	32	18	2/0/3

Table 41: Benefit of effort awareness in timewise prediction

Frameworks /Datasets	Columba	Bugzilla	Postgres	JDT	Platform	Mozilla	Win/Draw /Loss
EALR	12	25	19	10	14	5	1/1/4
LR + KCO	13	24	19	9	13	5	0/0/6
LT	11	21	12	6	10	3	0/0/6
CBS	12	24	19	10	13	5	0/1/5
NN + RUS	13	23	19	10	14	6	0/2/4
NN + KCO	12	23	19	9	14	5	0/0/6
DQN + RUS	14	21	18	9	14	5	1/0/5
DQN + KCO	10	23	20	8	15	6	2/1/3

Table 42: *Popt* performance in timewise prediction

Frameworks/ Datasets	Columba	Bugzilla	Postgres	JDT	Platform	Mozilla	Win/Draw/Loss
EALR	31	**55**	41	**51**	48	**54**	1/2/3
LR + KCO	28	54	42	50	47	49	0/0/6
LT	**39**	53	**47**	49	**49**	50	2/1/3
CBS	37	**55**	40	**51**	**49**	53	0/3/3
NN + RUS	32	52	41	49	48	51	0/0/6
NN + KCO	38	53	44	48	46	49	0/0/6
DQN + RUS	31	**55**	42	50	48	52	0/1/5
DQN + KCO	34	51	45	48	**49**	51	0/1/5

5.2.7 Discussion

1) Performance comparisons under different prediction scenarios

Recently, many studies used various prediction scenarios to perform JIT defect prediction (Zhao *et al*., 2023). To compare with these works, this experiment analyses different prediction settings for the baseline frameworks and the proposed framework. Table 34, Table 37, and Table 40 present the performance of DQNKCO and other baseline frameworks in term of F-scores. According to these results, DQNKCO consistently outperforms baseline frameworks for the majority of project datasets. DQNKCO generates good classification results, particularly in within and cross project predictions, because the training datasets provided by KCO are sufficient and diverse to construct a good prediction model. As with timewise predictions, each training fold has a unique imbalance distribution that reflects the proportion of defect data collected during software development. Since the data is split, most training folds have a limited amount of data. This may negatively affect the model's performance. The model may

not be able to learn complex patterns and generalize well to unseen data due to the limited number of instances available for training during certain periods. Considering the different imbalance proportions, DQNKCO is unable to consistently make better timewise predictions in most projects. In order for DQNKCO to function effectively, it requires a diverse defect data to be included in the model training process.

When the effort awareness of the prediction model constructed by the baseline and proposed framework is evaluated, DQNKCO obtains the best cases, particularly for *Benefit*, with 11/18 when making predictions across three prediction settings. However, when the *Popt* metric is considered, unsupervised learning by LT achieves the highest average score. This observation suggests that while LT provides more near optimal models across test cases, it performs less effectively in predicting true defects within 20% effort when the *Benefit* is considered. DQNKCO, on the other hand, by utilising supervised reinforcement learning, more defects are predicted within limited effort and still have competitive performance to the baseline framework for *Popt*. Even though different predictions are involved, the above observation demonstrate the proposed framework capable to predicting more defects in limited inspection effort despite not reaching near-optimal models (i,e. referring to *Popt*) for effort aware JIT-SDP.

2) The impact of different combination of resampling and classifiers approaches

For prediction accuracy observation, the proposed framework is remarkable to produce a better prediction model with F-score on average when compared to the baseline frameworks such as EALR, LT, and CBS. The results are not significantly improved by the proposed framework. The reason is due to the capability of KCO only benefits from high-class imbalanced datasets as presented in JDT, Platform and Mozilla. The implementation of KCO in resampling produce better training sets for DQN to perform better in those datasets but poor in other datasets with smaller imbalanced

datasets. Furthermore, the performance of the model constructed through KCO oversampling is observed to be superior to that of the RUS technique. Oversampling via KCO provides diverse training sets for NN and DQN, allowing them to perform significantly better in F-score predictions. With the assistance of KCO in rebalancing the class distribution of project datasets, NN and DQN capable to effectively deal with highly imbalanced data, particularly for within and across project predictions.

Ghotra *et al*. (2015) pointed out that the choice of different classification techniques produces a significant impact on the performance of defect prediction models. Thus, the combination of resampling techniques and strong classifiers such as DQN and NN typically outperforms simpler classifiers in this case LR, SVM and NB. In particular, DQNKCO and DQNRUS outperform in terms of consistently improving F-score performance across multiple prediction cases. Meanwhile, baseline frameworks such as NNKCO and NNRUS outperform those frameworks that use a simpler classifier, such as EALR and CBS, which both use a logistic regression classifier. This implies that it is vital to train the prediction model using strong classifiers such as DQN and NN to ensure consistency and reliability of prediction results. The above observations indicate that different base classifiers usually help the frameworks to obtain preferable performance for JIT-SDP model. Particularly for DQN, we can observe that the improvement of DQN in reducing false positives by achieving reliable performances in Benefit compared to the baseline classifiers. The main reason is DQN capable to avoid of producing high number of false positives with help of reward policy during the model training (Section 4.3.3). The reward policy enables the model to learnt based on reward and punishment mechanism which in turn help in minimizing the false defects and maximize the true defects predicted. Another important factor that affects performance is the buffer memory of agent in DQN that helps to supply the model with sufficient training data in replay batch despite of having small defect data. Software

project datasets in this experiment have different imbalance class properties. In respect to this observation, the limited defect data is handled by the capability of replay memory in DQN to provides more diverse training data for the trained neural network. In overall, above analysis indicates that the proposed framework DQNKCO capable to obtain stable and promising performance no matter weather predicting defective changes for within project, cross project or timewise predictions.

5.2.8 Threat of validity

How is the performance of the proposed framework under different settings?

The widely used open-source software projects considered by the proposed framework are large enough to allow drawing statistically meaningful conclusions. The proposed framework uses the same datasets that are used in previous effort-aware JIT-SDP studies (Kamei *et al*., 2013; Kondo *et al*., 2019; Li *et al*., 2020; Pascarella *et al*., 2019). The results are not generalizable to other software projects that have features different from those of the studied datasets. The defect-inducing changes are discovered by the commonly used SZZ algorithm and incomplete. Furthermore, the measure of the effort required to review a change is considered as the total number of lines modified by that change, which reflect inconsistent with that in the real world. However, the threats represent problems inherent to most studies and need to be further explored by further research.

How generic is the proposed framework DQNKCO?

The proposed framework has capability of producing a good trade-off of accuracy and lower false positives prediction. With this capability, the proposed framework is applicable to other research domains that related to the problem of high false positives prediction in imbalanced class datasets. However, the effect of classifier selection for

the framework is still unknown. Here, the proposed framework considered ANN as a base classifier within DQN modelling. The effectiveness of other deep learning classifiers remains unverified and needs to be studied in the future. Furthermore, the independent variables used by the proposed framework are commonly used change metrics (Kamei *et al*., 2013). The degree to which the metrics accurately measure the concepts that intend to measure is already investigated. The distribution of effort value is not the same as that reported in prior results, which results in reaching conclusions different by this framework. Nonetheless, the construct validity of the independent variable is considered acceptable in this research

5.2.9 Conclusion

The feasibility of the proposed framework, as well as baseline frameworks, is investigated in this experiment. To demonstrate the effectiveness of DQNKCO, an extensive comparison experiment is carried out. The results of six software projects show that DQNKCO produces a considerable advantage over the baseline frameworks. DQNKCO, on the one hand, improves the ability to predict defective changes within imbalanced datasets while ensuring high prediction performance with the goal of reducing false positives. In the prediction scenarios, it outperforms almost all compared frameworks. DQNKCO, on the other hand, retains the advantage of deep reinforcement learning models to emphasise predicting more defects with limited effort while taking into account of producing small false positives. Despite DQNKCO is very effective in effort aware JIT-SDP, it is overlooked in existing studies. With high prediction effectiveness and potential of reaching near-optimal models, it is a good choice for practitioners to implement DQNKCO in practice.

CHAPTER 6: CONCLUSION

This chapter summarizes the study by elaborating on the achievements throughout the research. It highlights the most critical findings over the course of this research and the limitations that accompany them. Finally, the chapter wraps up with suggestions for future works in the domain to enhance the proposed framework in the future.

6.1 Contributions

The research started by exploring the software defect prediction domain in general. The research dug deeper into the domain and proposed an updated classification taxonomy of JIT-SDP with respect to inaccurate prediction issue besides critically reviewing the latest works and other issues related to the domain. Based on the review, this research proposed the improved approach of JIT-SDP as an alternative solution to address the gaps in rebalancing imbalance class distribution and reducing false positives prediction. The proposed framework utilized deep reinforcement learning with improved oversampling strategy and demonstrated significant performance in predicting software defect at code change level prediction. The study also evaluated the proposed framework using several prediction settings, including within project prediction, cross project prediction and timewise prediction to provide reliable validation and performance benchmarks. Throughout the process of developing the entire framework, the study successfully produced several achievements as follows.

1. *An updated classification taxonomy for accurate prediction in JIT-SDP*. Throughout the review, this research discovered the lack of a standard classification taxonomy in the domain of JIT-SDP. Specifically, Chapter 2 reviewed the existing approaches in data pre-processing and modelling of JIT-SDP regarding machine learning aspects. Thus, a new updated taxonomy of JIT-

SDP is presented, emphasizing the machine learning and data pre-processing factors contributed toward achieving an accurate JIT-SDP model.

2. *A reliable enhanced oversampling technique for imbalance datasets*. Most of existing studies utilized random based under-sampling technique for addressing imbalance datasets. However, this research proposing a new oversampling technique based on kernel analysis and spectral clustering to provides a better-balanced class dataset. During identification of feasible spaces for interpolate, KPCA is utilized to select top candidate for data template. With selected candidate data for interpolate, cross interpolation across multiple generations is then conducted to achieve the desired class distributions. The outcome of this oversampling, rebalancing of class imbalance data in modelling of JIT-SDP became more reliable and quality of training datasets.
3. *A robust deep reinforcement learning architecture for effort aware defect prediction.* Previous classifiers for JIT-SDP adopted supervised, unsupervised and semi-supervised learning approaches to building the prediction model. This study however utilized deep reinforcement learning which exploiting the usefulness of Deep Q-Network to reinforce the learning process under a restricted policy. The policy is inspired by acts during code review process. For every correct predicted result, the model is given reward and vise verse for falsely predicted results. The reward policy learning encourages the prediction model to minimise the possibility of producing false positives while maximising prediction results. The training of the JIT-SDP model with DQN is appropriate for sequential or mini batches data, allowing it to adapt to data drift more effectively. The issue of re-training JIT-SDP models is well handled in the sequential learning approach by this solution, which is an inherent problem with most classifiers.

4. *A framework for modelling JIT-SDP*. The study develops a framework for developing a comprehensive JIT-SDP that reduces the possibility of false alarms as well as predicting defects. This framework identifies potential risky changes before they are incorporated into the codebase, thereby mitigating their risk. A novel oversampling and classifier are also proposed in this framework, which considers the most influential factors (rebalancing data and choice of classifier) that influence the prediction of software defects. This framework produces better classification results for code changes during software development as a result.

6.2 Research limitations

Despite of having all significant achievements in previous section, several limitations faced during throughout this research. The identified limitations provide room for improvements and future research opportunities.

1. Input of software metrics. The JIT-SDP model is affected by the uncertainty of input data. In different releases, metrics are distributed differently. As a result, the training and testing data sets do not have similar distributions, i.e., the training data no longer matches the current project data. Concept drift is not fully addressed by the current proposed approach in the context of online learning.
2. Software project datasets. The datasets used in this study are from an open-source repository (GitHub). Data from private/commercial company repositories is not included in this study. It is possible that the results are biased towards open-source systems due to defects reported on them. Depending on the design of a program, results are not applicable to industry context yet.
3. Optimization of hyperparameters coverage. JIT-SDP models encode the regularities in a set of model parameters. An important concern is whether we

can uniquely determine the model parameters from the input data, either theoretically or numerically. This is called the parameter optimization problem. In this study however, parameters optimization only limited to hyperparameters for neural network within DQN. Due to limited resources and time constraint, several other hyperparameters for DQN such as choice of reward policy, epsilon value and decay rate are based on default settings. The performance of DQN in different hyperparameters settings is unknown and untestable.

4. The usability of the proposed approach. The proposed of JIT-SDP consist of three development phases. However, for each phase, specific processes unsuitable or generalized well with other OSS project other than those tested in this research. For example, the proposed KCO is not ideal to be implemented in small datasets and overly diverse distribution data due to orthogonal transformation unable to be done effectively. Besides, DQN proposed here will not perform well in case of noisy training data sets is not well treated because more duplicated instances lead to overfitting model. DQN underperform toward overfitted model when the training data have more duplicated instances which hindered from learning the whole data pattern.

6.3 Future Works

Apart from the current works discussed in previous sections, several interesting and emerging topics still relevant in JIT-SDP study including data privacy, software metrics and defect severity priority.

Data privacy. Data privacy requires publicly available defect datasets to accelerate cross-project defect prediction studies. Due to sensitive attribute values, software companies are reluctant to share their defect datasets. Consequently, cross-project defect prediction studies typically involve open-source software or only a few proprietary

systems. Cross-project validation revealed significant performance degradation for most of the datasets studied for JIT-SDP approaches in this research. In this case, JIT-SDP could modify instances at a random distance while maintaining the class boundary. So, JIT-SDP could preserve original datasets and still achieve superior prediction performance as in models trained from original defect datasets. Investigating privacy issues in cross-project JIT-SDP is required because evaluation of prediction models will stronger if we have more available private datasets. Simply increasing the size of the training data does not improve the prediction accuracy of the investigated approaches. Given the wealth of data available from code repositories, we believe that a finer-grained JIT is possible.

Software metrics. Software metrics are the most prevalent type of measurement in software artefacts. As demonstrated in the preceding chapter, the potential of current software metrics has nearly been realised, as there is no significant improvement in prediction performance regardless of the classifiers chosen. Thus, future attempts will necessitate a wider adoption of other measures, particularly by utilization of ITS-based data. In ITS, developers describe and discuss change requests, provide feedback on the code for code review, and suggest future improvements. Although uncommon, ITS data such as issue reports, discussions, and change requests could be useful to JIT-SDP, especially for predicting future changes to address reported issues. By using this data in addition to software change metrics will perform better than those using only software change metrics. These data sets will facilitate a wider adoption and further investigation of the ITS data in JIT-SDP. New metrics and models must be continuously investigated. As a result, it is necessary to continue investigating project context factors and software metrics, as well as their relationship with defect prediction results, given that a variety of software metrics derived from a variety of data sources aid in enhancing defect predictive performance. With the ability of the proposed approach DQN and KCO to

deal with large combinations of metrics (multidimensional data), the use of ITS data with current metrics is required in the future improve prediction performance.

Defect severity. In effort-aware JIT-SDP, the current quality assurance effort is measured by the reviewed lines of code. Future JIT-SDP implementations utilising the proposed framework need to consider alternative quality assurance effort measures, such as differentiating the efforts to address various types of failure. In fact, software defects produce result in a variety of failure types and occurrences at various stages of software development. Consequently, the severity of software defects in various locations varies. The potential further works for JIT-SDP exist for predicting and prioritizing defect severity as a multiclass classification problem, with the defect severity classes serving as the dependent variable. Moreover, incorporating the proposed JIT-SDP approach with static defect localisation is beneficial for defect management. Using semantic abstractions of the source code, localisation defects are detected. Consequently, combining JIT-SDP and defect localisation could be mutually compensatory, as they can detect various types of defects (severity). Moreover, defect localisation with a warning system prioritised by JIT-SDP could result in superior performance compared to native defect localisation.

6.4 Research Impact

The proposed framework driven toward automated code review, which is the process of reviewing source code automatically using a predefined set of rules to identify inefficient code or defects. Automated code review is essential for standardising and scaling software development efforts within an organisation. Since the review process handles the majority of common source code defects, human reviewers can concentrate on higher level code issues. It is a cost-effective strategy because it automates portions of a code review that are not susceptible to human error, avoid

incorporate personal bias, and quickly identify defects. Thus, a robust review process allows developer teams to spend less time and money on bad code. Consequently, organization can achieve a valuable and sustainable competitive advantage.

The pandemic heightened the need to reduce software development costs, and many organisations made this a top priority. The development phase is a natural starting point for cost reductions. According to Gartner analyst Robert Snow (2021), new application development accounts for 17% of total IT expenditures, making it a fruitful area to search for opportunities to reduce costs, optimise expenditures, or increase value. Through JIT-SDP of the proposed framework, agile application design and development teams employ tactics that combine short-term savings via static analysis of JIT-SDP with strategic long-term savings via software testing. As Malaysian organisations accelerate their transformation plans, it is imperative that they incorporate more advanced static analysis technology, such given by the proposed JIT-SDP into their software development strategies.

Encourage the application and use of artificial intelligence (AI) and software innovation as a locally made service to empower future technology in alignment with the Twelfth Malaysia Plan (12MP) under the Malaysia Digital Economy Blueprint. This will help create more highly skilled and experienced talents. It is an initiative towards the achievement of ICT excellence in supporting the development of digital government, with the advancement of the proposed JIT-SDP being accomplished through automated code review. The initiative conducts quality assurance checks on various software and information technology (IT) projects carried out by the government. In addition, the possibility of software delays and security breaches is reduced so as to maximise the likelihood of producing high-quality software.

REFERENCE

Akmel, F., Birihanu, E., & Siraj, B. (2018). A Literature Review Study of Software Defect Prediction using Machine Learning Techniques. *International Journal of Emerging Research in Management and Technology*, *6*(6), 300. https://doi.org/10.23956/ijermt.v6i6.286

Al Mamun, M. A., Berger, C., & Hansson, J. (2017). Correlations of software code metrics: An empirical study. *ACM International Conference Proceeding Series*, *Part F1319*(May), 255–266. https://doi.org/10.1145/3143434.3143445

Alami, A. (2016). Why Do Information Technology Projects Fail? *Procedia Computer Science*, *100*, 62–71. https://doi.org/10.1016/j.procs.2016.09.124

Albahli, S. (2019). A Deep Ensemble Learning Method for Effort-Aware Just-In-Time Defect Prediction. *Future Internet*, *11*(12), 246. https://doi.org/10.3390/fi11120246

Amasaki, S., Aman, H., & Yokogawa, T. (2021). A Preliminary Evaluation of CPDP Approaches on Just-in-Time Software Defect Prediction. *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 279–286. https://doi.org/10.1109/SEAA53835.2021.00042

Arisholm, E., Briand, L. C., & Johannessen, E. B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, *83*(1), 2–17. https://doi.org/10.1016/j.jss.2009.06.055

Azzeh, M., Elsheikh, Y., Nassif, A. B., & Angelis, L. (2023). Examining the performance of kernel methods for software defect prediction based on support vector machine. *Science of Computer Programming*, *226*, 102916. https://doi.org/10.1016/j.scico.2022.102916

Barua, S., Islam, M. M., Yao, X., & Murase, K. (2014). MWMOTE - Majority weighted minority oversampling technique for imbalanced data set learning. *IEEE Transactions on Knowledge and Data Engineering*, *26*(2), 405–425. https://doi.org/10.1109/TKDE.2012.232

Baum, T., & Schneider, K. (2016). On the Need for a New Generation of Code Review Tools. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Vol. 10027 LNCS* (pp. 301–308). https://doi.org/10.1007/978-3-319-49094-6_19

Beller, M., Bacchelli, A., Zaidman, A., & Juergens, E. (2014). Modern code reviews in open-source projects: Which problems do they fix?. *Proceedings - 11th Working Conference on Mining Software Repositories, MSR 2014*, 202–211. https://doi.org/10.1145/2597073.2597082

Bellinger, C., Drummond, C., & Japkowicz, N. (2016). *Beyond the Boundaries of SMOTE* (pp. 248–263) . In: Frasconi, P., Landwehr, N., Manco, G., Vreeken, J. (eds) Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2016. Lecture Notes in Computer Science(), vol 9851. Springer, Cham..

https://doi.org/10.1007/978-3-319-46128-1_16

Bennin, K. E., Keung, J., Phannachitta, P., Monden, A., & Mensah, S. (2018). MAHAKIL: Diversity Based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction. *IEEE Transactions on Software Engineering*, *44*(6), 534–550. https://doi.org/10.1109/TSE.2017.2731766

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. (2009). Fair and balanced? Bias in bug-fix datasets. *ESEC-FSE'09 - Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, *June 2014*, 121–130. https://doi.org/10.1145/1595696.1595716

Bowes, D., Hall, T., & Petrić, J. (2018). Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, *26*(2), 525–552. https://doi.org/10.1007/s11219-016-9353-3

Branco, P., Torgo, L., & Ribeiro, R. P. (2016). A Survey of Predictive Modeling on Imbalanced Domains. *ACM Computing Surveys*, *49*(2), 1–50. https://doi.org/10.1145/2907070

Cabral, G. G., Minku, L. L., Shihab, E., & Mujahid, S. (2019). Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 666–676. https://doi.org/10.1109/ICSE.2019.00076

Çarka, J., Esposito, M., & Falessi, D. (2022). On effort-aware metrics for defect prediction. *Empirical Software Engineering*, *27*(6). https://doi.org/10.1007/s10664-022-10186-7

Chawla, N. v., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, *16*, 321–357. https://doi.org/10.1613/jair.953

Chen, L., Fang, B., Shang, Z., & Tang, Y. (2016). Tackling class overlap and imbalance problems in software defect prediction. *Software Quality Journal*. https://doi.org/10.1007/s11219-016-9342-6

Chen, X., Mu, Y., Liu, K., Cui, Z., & Ni, C. (2021). Revisiting heterogeneous defect prediction methods: How far are we? *Information and Software Technology*, *130*(September 2020), 106441. https://doi.org/10.1016/j.infsof.2020.106441

Chen, X., Zhao, Y., Wang, Q., & Yuan, Z. (2018). MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology*, *93*, 1–13. https://doi.org/10.1016/j.infsof.2017.08.004

Chen, Y., Qian, H., Wang, X., Wang, D., & Han, L. (2022). A GloVe Model for Urban Functional Area Identification Considering Nonlinear Spatial Relationships between Points of Interest. *ISPRS International Journal of Geo-Information*, *11*(10), 498. https://doi.org/10.3390/ijgi11100498

Choirunnisa, S., Meidyani, B., & Rochimah, S. (2018). Software Defect Prediction using Oversampling Algorithm: A-SUWO. *2018 Electrical Power, Electronics, Communications, Controls and Informatics Seminar (EECCIS)*, 337–341. https://doi.org/10.1109/EECCIS.2018.8692874

Ebert, C. (2007). The impacts of software product management. *Journal of Systems and Software*, *80*(6), 850–861. https://doi.org/10.1016/j.jss.2006.09.017

Eskonen, J., Kahles, J., & Reijonen, J. (2020). Automating GUI Testing with Image-Based Deep Reinforcement Learning. *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 160–167. https://doi.org/10.1109/ACSOS49614.2020.00038

Fan, Y., Xia, X., Alencar da Costa, D., Lo, D., Hassan, A. E., & Li, S. (2019). The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering*, *PP*(c), 1. https://doi.org/10.1109/TSE.2019.2929761

Fan, Y., Xia, X., da Costa, D. A., Lo, D., Hassan, A. E., & Li, S. (2021). The Impact of Mislabeled Changes by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering*, *47*(8), 1559–1586. https://doi.org/10.1109/TSE.2019.2929761

Feng, S., Keung, J., Yu, X., Xiao, Y., Bennin, K. E., Kabir, M. A., & Zhang, M. (2021). COSTE: Complexity-based OverSampling TEchnique to alleviate the class imbalance problem in software defect prediction. *Information and Software Technology*, *129*, 106432. https://doi.org/10.1016/j.infsof.2020.106432

Fu, W., Menzies, T., & Shen, X. (2016). Tuning for software analytics: Is it really necessary? *Information and Software Technology*, *76*, 135–146. https://doi.org/10.1016/j.infsof.2016.04.017

Fernandez, A., Garcia, S., Herrera, F., & Chawla, N. V. (2018). SMOTE for Learning from Imbalanced Data: Progress and Challenges. *Journal of Artificial Intelligence Research*, 61, 863–905. https://www.jair.org/index.php/jair/article/view/11192

Gazzah, S., Hechkel, A., & Ben Amara, N. E. (2015). A hybrid sampling method for imbalanced data. *12th International Multi-Conference on Systems, Signals and Devices, SSD 2015*, 1–6. https://doi.org/10.1109/SSD.2015.7348093

Ghotra, B., McIntosh, S., & Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. *Proceedings - International Conference on Software Engineering*, *1*, 789–800. https://doi.org/10.1109/ICSE.2015.91

Gong, L., Jiang, S., & Jiang, L. (2019). Tackling Class Imbalance Problem in Software Defect Prediction Through Cluster-Based Over-Sampling With Filtering. *IEEE Access*, *7*, 145725–145737. https://doi.org/10.1109/ACCESS.2019.2945858

Gray, D., Bowes, D., Davey, N., Sun, Y., & Christianson, B. (2011). The misuse of the NASA Metrics Data Program data sets for automated software defect prediction. *IET Seminar Digest*, *2011*(1), 96–103. https://doi.org/10.1049/ic.2011.0012

Han, H., Wang, WY., Mao, BH. (2005). Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning. In: Huang, DS., Zhang, XP., Huang, GB. (eds) Advances in Intelligent Computing. ICIC 2005. *Lecture Notes in Computer Science, vol 3644. Springer, Berlin, Heidelberg*. https://doi.org/10.1007/11538059_91

Han, M., Guo, H., Li, J., & Wang, W. (2023). Global-local information based oversampling for multi-class imbalanced data. *International Journal of Machine Learning and Cybernetics*, *14*(6), 2071–2086. https://doi.org/10.1007/s13042-022-01746-w

Harries, L., Clarke, R. S., Chapman, T., Nallamalli, S. V. P. L. N., Ozgur, L., Jain, S., Leung, A., Lim, S., Dietrich, A., Hernández-Lobato, J. M., Ellis, T., Zhang, C., & Ciosek, K. (2020). *DRIFT: Deep Reinforcement Learning for Functional Software Testing*. *NeurIPS*.

Hawkins, D. M. (2004). The Problem of Overfitting. *Journal of Chemical Information and Computer Sciences*, *44*(1), 1–12. https://doi.org/10.1021/ci0342472

He, H., Bai, Y., Garcia, E. A., & Li, S. (2008). ADASYN: Adaptive synthetic sampling approach for imbalanced learning. *In 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)* (pp. 1322–1328). https://doi.org/10.1109/IJCNN.2008.4633969

Hedberg, H. (2004). Introducing the Next Generation of Software Inspection Tools. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *3009*(May), V. https://doi.org/10.1007/978-3-540-24659-6

Herzig, K., Just, S., & Zeller, A. (2013). It's not a bug, it's a feature: How misclassification impacts bug prediction. *Proceedings - International Conference on Software Engineering*, 392–401. https://doi.org/10.1109/ICSE.2013.6606585

Herzig, K., Just, S., & Zeller, A. (2016). The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, *21*(2), 303–336. https://doi.org/10.1007/s10664-015-9376-6

Ho, A., Nhat Hai, N., & Thi-Mai-Anh, B. (2022). Combining Deep Learning and Kernel PCA for Software Defect Prediction. *ACM International Conference Proceeding Series*, 360–367. https://doi.org/10.1145/3568562.3568587

Hosseini, S., Turhan, B., & Mäntylä, M. (2018). A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. *Information and Software Technology*, *95*, 296–312. https://doi.org/10.1016/j.infsof.2017.06.004

Hu, Z., Beuran, R., & Tan, Y. (2020). Automated Penetration Testing Using Deep Reinforcement Learning. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (pp. 2-10). Genoa, Italy. https://doi.org/10.1109/EuroSPW51379.2020.00010.

Huang, Q., Xia, X., & Lo, D. (2019). Revisiting supervised and unsupervised models

for effort-aware just-in-time defect prediction. *Empirical Software Engineering*, *24*(5), 2823–2862. https://doi.org/10.1007/s10664-018-9661-2

Huda, S., Alyahya, S., Mohsin Ali, M., Ahmad, S., Abawajy, J., Al-Dossari, H., & Yearwood, J. (2017). A Framework for Software Defect Prediction and Metric Selection. *IEEE Access*, *6*, 2844–2858. https://doi.org/10.1109/ACCESS.2017.2785445

Huda, S., Liu, K., Abdelrazek, M., Ibrahim, A., Alyahya, S., Al-Dossari, H., & Ahmad, S. (2018). An Ensemble Oversampling Model for Class Imbalance Problem in Software Defect Prediction. *IEEE Access*, *6*(c), 24184–24195. https://doi.org/10.1109/ACCESS.2018.2817572

Jahanshahi, H., Jothimani, D., Başar, A., & Cevik, M. (2019). Does chronology matter in JIT defect prediction? *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, 90–99. https://doi.org/10.1145/3345629.3351449

Jiang, T., Tan, L., & Kim, S. (2013). Personalized defect prediction. *Proceedings - 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, 279–289. https://doi.org/10.1109/ASE.2013.6693087

Jiarpakdee, J., Tantithamthavorn, C., & Hassan, A. E. (2021). The Impact of Correlated Metrics on the Interpretation of Defect Models. *IEEE Transactions on Software Engineering*, 47(2), 320-331. https://doi.org/10.1109/TSE.2019.2891758.

Jiarpakdee, J., Tantithamthavorn, C., & Treude, C. (2018). Autospearman: Automatically mitigating correlated software metrics for interpreting defect models. *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, 92–103. https://doi.org/10.1109/ICSME.2018.00018

Jing, X. Y., Wu, F., Dong, X., & Xu, B. (2017). An Improved SDA Based Defect Prediction Framework for Both Within-Project and Cross-Project Class-Imbalance Problems. *IEEE Transactions on Software Engineering*, *43*(4), 321–339. https://doi.org/10.1109/TSE.2016.2597849

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, *39*(6), 757–773. https://doi.org/10.1109/TSE.2012.70

Kim, J., Kwon, M., & Yoo, S. (2018). Generating test input with deep reinforcement learning. *Proceedings - International Conference on Software Engineering*, 51–58. https://doi.org/10.1145/3194718.3194720

Kim, S., Zhang, H., Wu, R., & Gong, L. (2011). Dealing with noise in defect prediction. *Proceedings - International Conference on Software Engineering*, 481–490. https://doi.org/10.1145/1985793.1985859

Kondo, M., German, D. M., Mizuno, O., & Choi, E. H. (2019). The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering*.

https://doi.org/10.1007/s10664-019-09736-3

Kononenko, O., Baysal, O., & Godfrey, M. W. (2016). Code review quality: How developers see it. *Proceedings - International Conference on Software Engineering*, *14-22-May-*, 1028–1038. https://doi.org/10.1145/2884781.2884840

Laradji, I. H., Alshayeb, M., & Ghouti, L. (2014). Software defect prediction using ensemble learning on selected features. *Information and Software Technology*. https://doi.org/10.1016/j.infsof.2014.07.005

Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., & Whitehead, E. J. (2013). Does bug prediction support human developers? Findings from a Google case study. *Proceedings - International Conference on Software Engineering*, 372–381. https://doi.org/10.1109/ICSE.2013.6606583

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, *18*, 1–52.

Li, W., Zhang, W., Jia, X., & Huang, Z. (2020). Effort-Aware semi-Supervised just-in-Time defect prediction. *Information and Software Technology*, *126*(April), 106364. https://doi.org/10.1016/j.infsof.2020.106364

Li, Z., Jing, X. Y., & Zhu, X. (2018). Progress on approaches to software defect prediction. *IET Software*, *12*(3), 161–175. https://doi.org/10.1049/iet-sen.2017.0148

Li, Z., Zhang, X., Guo, J., & Shang, Y. (2019). Class Imbalance Data-Generation for Software Defect Prediction. *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, 276–283. https://doi.org/10.1109/APSEC48747.2019.00045

Liu, M., Miao, L., & Zhang, D. (2014). Two-stage cost-sensitive learning for software defect prediction. *IEEE Transactions on Reliability*, *63*(2), 676–686. https://doi.org/10.1109/TR.2014.2316951

Liu, X. Y., Wu, J., & Zhou, Z. H. (2008). Exploratory under-sampling for class-imbalance learning. *Proceedings - IEEE International Conference on Data Mining, ICDM*, 965–969. https://doi.org/10.1109/TSMCB.2008.2007853

Liu, Y., Han, W., Wang, X., & Li, Q. (2020). Oversampling Algorithm Based on Spatial Distribution of Data Sets for Imbalance Learning. *2020 5th International Conference on Computer and Communication Systems (ICCCS)*, 45–49. https://doi.org/10.1109/ICCCS49078.2020.9118573

Lorena, A. C., Garcia, L. P. F., Lehmann, J., Souto, M. C. P., & Ho, T. K. (2020). How Complex Is Your Classification Problem? *ACM Computing Surveys*, *52*(5), 1–34. https://doi.org/10.1145/3347711

Lunardon, N., Menardi, G., & Torelli, N. (2014). ROSE: a Package for Binary Imbalanced Learning. *The R Journal*, *6*(1), 79. https://doi.org/10.32614/RJ-2014-008

Mahmood, Z., Bowes, D., Hall, T., Lane, P. C. R., & Petrić, J. (2018). Reproducibility and replicability of software defect prediction studies. *Information and Software Technology*, *99*, 148–163. https://doi.org/10.1016/j.infsof.2018.02.003

Malhotra, R., & Khan, K. (2020). A Study on Software Defect Prediction using Feature Extraction Techniques. *ICRITO 2020 - IEEE 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)*, 1139–1144. https://doi.org/10.1109/ICRITO48877.2020.9197999

Mantyla, M. V., & Lassenius, C. (2009). What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, *35*(3), 430–448. https://doi.org/10.1109/TSE.2008.71

Meiliana, Karim, S., Warnars, H. L. H. S., Gaol, F. L., Abdurachman, E., & Soewito, B. (2017). Software metrics for fault prediction using machine learning approaches: A literature review with PROMISE repository dataset. *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, *2*(6), 19–23. https://doi.org/10.1109/CYBERNETICSCOM.2017.8311708

Mende, T., & Koschke, R. (2010). Effort-aware defect prediction models. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 107–116. https://doi.org/10.1109/CSMR.2010.18

Menzies, T., Dekhtyar, A., Distefano, J., & Greenwald, J. (2007). Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors.'" *IEEE Transactions on Software Engineering*, *33*(9), 637–640. https://doi.org/10.1109/TSE.2007.70721

Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., & Jiang, Y. (2008). Implications of ceiling effects in defect predictors. *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering - PROMISE '08*, 47. https://doi.org/10.1145/1370788.1370801

Misirli, A. T., Shihab, E., & Kamei, Y. (2016). Studying high impact fix-inducing changes. *Empirical Software Engineering*, *21*(2), 605–641. https://doi.org/10.1007/s10664-015-9370-z

Mockus, A. (2016). Operational data are missing, incorrect, and decontextualized. In *Perspectives on Data Science for Software Engineering*. Elsevier Inc. https://doi.org/10.1016/b978-0-12-804206-9.00057-x

Mockus, A., & Weiss, D. M. (2002). Predicting risk of software changes. *Bell Labs Technical Journal*, *5*(2), 169–180. https://doi.org/10.1002/bltj.2229

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., & Murphy, B. (2010). Change bursts as defect predictors. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 309–318. https://doi.org/10.1109/ISSRE.2010.25

Nayrolles, M., & Hamou-Lhadj, A. (2018). CLEVER: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. *Proceedings - International Conference on Software Engineering*, 153–164. https://doi.org/10.1145/3196398.3196438

Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2021). Machine learning based methods for software fault prediction: A survey. *Expert Systems with Applications*, *172*, 114595. https://doi.org/10.1016/j.eswa.2021.114595

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

Piotrowski, P., & Madeyski, L. (2020). Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review. In *Data-Centric Business and Applications. Lecture Notes on Data Engineering and Communications Technologies* (pp. 77–99). Springer. https://doi.org/10.1007/978-3-030-34706-2_5

Punitha, K., & Chitra, S. (2013). Software defect prediction using software metrics - A survey. *2013 International Conference on Information Communication and Embedded Systems, ICICES 2013*, 555–558. https://doi.org/10.1109/ICICES.2013.6508369

Qiao, L., & Wang, Y. (2019). Effort-aware and just-in-time defect prediction with neural network. *PLoS ONE*, *14*(2), 1–19. https://doi.org/10.1371/journal.pone.0211359

Quach, S., Lamothe, M., Adams, B., Kamei, Y., & Shang, W. (2021). Evaluating the impact of falsely detected performance bug-inducing changes in JIT models. *Empirical Software Engineering*, *26*(5). https://doi.org/10.1007/s10664-021-10004-6

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, *55*(8), 1397–1418. https://doi.org/10.1016/j.infsof.2013.02.009

Rahman, F., & Devanbu, P. (2013). How, and why, process metrics are better. *Proceedings - International Conference on Software Engineering*, 432–441. https://doi.org/10.1109/ICSE.2013.6606589

Ramler, R., Buchgeher, G., Klammer, C., Pfeiffer, M., Salomon, C., Thaller, H., & Linsbauer, L. (2019). *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud* (D. Winkler, S. Biffl, & J. Bergsmann (eds.); Vol. 338). Springer International Publishing. https://doi.org/10.1007/978-3-030-05767-1

Rodríguez-Torres, F., Martínez-Trinidad, J. F., & Carrasco-Ochoa, J. A. (2022). An Oversampling Method for Class Imbalance Problems on Large Datasets. *Applied Sciences*, *12*(7), 3424. https://doi.org/10.3390/app12073424

Rosen, C., Grawi, B., & Shihab, E. (2015). Commit guru: Analytics and risk prediction of software commits. *Proceedings - 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015* , 966–969. https://doi.org/10.1145/2786805.2803183

Ryu, D., Choi, O., & Baik, J. (2016). Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering*, *21*(1), 43–71. https://doi.org/10.1007/s10664-014-9346-4

Sadowski, C., Lewis, C., Lin, Z., Zhu, X., & Whitehead, E. J. (2011). An empirical analysis of the FixCache algorithm. *Proceeding of the 8th Working Conference on Mining Software Repositories - MSR '11*, 219. https://doi.org/10.1145/1985441.1985475

Sharma, S., Gosain, A., & Jain, S. (2022). *A Review of the Oversampling Techniques in Class Imbalance Problem* (pp. 459–472). https://doi.org/10.1007/978-981-16-2594-7_38

Shihab, E., Hassan, A. E., Adams, B., & Jiang, Z. M. (2012). An industrial study on the risk of software changes. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*, 1–11. https://doi.org/10.1145/2393596.2393670

Shivaji, S., James Whitehead, E., Akella, R., & Kim, S. (2013). Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, *39*(4), 552–569. https://doi.org/10.1109/TSE.2012.43

Siers, M. J., & Islam, M. Z. (2015). Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. *Information Systems*, *51*, 62–71. https://doi.org/10.1016/j.is.2015.02.006

Sikic, L., Afric, P., Kurdija, A. S., & Silic, M. (2021). Improving Software Defect Prediction by Aggregated Change Metrics. *IEEE Access*, *9*, 19391–19411. https://doi.org/10.1109/ACCESS.2021.3054948

Singh, A., Bhatia, R., & Singhrova, A. (2018). Taxonomy of machine learning algorithms in software fault prediction using object oriented metrics. *Procedia Computer Science*, *132*, 993–1001. https://doi.org/10.1016/j.procs.2018.05.115

Singh, D., Sekar, V. R., Stolee, K. T., & Johnson, B. (2017). Evaluating how static analysis tools can reduce code review effort. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, *2017-Octob*, 101–105. https://doi.org/10.1109/VLHCC.2017.8103456

Singh, V. B., & Chaturvedi, K. K. (2013). Improving the quality of software by quantifying the code change metric and predicting the bugs. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7972 LNCS*(PART 2), 408–426. https://doi.org/10.1007/978-3-642-39643-4_30

Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, *30*(4), 1. https://doi.org/10.1145/1082983.1083147

Son, L., Pritam, N., Khari, M., Kumar, R., Phuong, P., & Thong, P. (2019). Empirical Study of Software Defect Prediction: A Systematic Mapping. *Symmetry*, *11*(2), 212. https://doi.org/10.3390/sym11020212

Song, Q., Guo, Y., & Shepperd, M. (2018). A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering*, *5589*(APRIL). https://doi.org/10.1109/TSE.2018.2836442

Taba, S. E. S., Khomh, F., Zou, Y., Hassan, A. E., & Nagappan, M. (2013). Predicting bugs using antipatterns. *IEEE International Conference on Software Maintenance, ICSM*, 270–279. https://doi.org/10.1109/ICSM.2013.38

Tabassum, S., Minku, L. L., & Feng, D. (2023). Cross-Project Online Just-In-Time Software Defect Prediction. *IEEE Transactions on Software Engineering*, *49*(1), 268–287. https://doi.org/10.1109/TSE.2022.3150153

Tan, M., Tan, L., Dara, S., & Mayeux, C. (2015). Online Defect Prediction for Imbalanced Data. *Proceedings - International Conference on Software Engineering*, *2*, 99–108. https://doi.org/10.1109/ICSE.2015.139

Tang, W., & Khoshgoftaar, T. M. (2004). Noise identification with the k-means algorithm. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, 373–378. https://doi.org/10.1109/ictai.2004.93

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2019). The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering*, *45*(7), 683–711. https://doi.org/10.1109/TSE.2018.2794977

Tosun, A., Bener, A., Turhan, B., & Menzies, T. (2010). Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. *Information and Software Technology*, *52*(11), 1242–1257. https://doi.org/10.1016/j.infsof.2010.06.006

Trautsch, A., Herbold, S., & Grabowski, J. (2020). Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, 127–138. https://doi.org/10.1109/ICSME46990.2020.00022

Xia, S., Xiong, Z., Luo, Y., WeiXu, & Zhang, G. (2015). Effectiveness of the Euclidean Wan, Z., Xia, X., Hassan, A. E., Lo, D., Yin, J., & Yang, X. (2018). Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering*, *5589*(c). https://doi.org/10.1109/TSE.2018.2877678

Wang, S., Liu, T., Nam, J., & Tan, L. (2018). Deep Semantic Feature Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering*, *5589*(c), 1–26. https://doi.org/10.1109/TSE.2018.2877612

Wang, T., Zhang, Z., Jing, X., & Zhang, L. (2016). Multiple kernel ensemble learning for software defect prediction. *Automated Software Engineering*, *23*(4), 569–590. https://doi.org/10.1007/s10515-015-0179-1

Wu, F., Jing, X. Y., Dong, X., Cao, J., Xu, B., & Ying, S. (2016). Cost-sensitive local collaborative representation for software defect prediction. *Proceedings - 2016 International Conference on Software Analysis, Testing and Evolution, SATE 2016*, *Cddl*, 102–107. https://doi.org/10.1109/SATE.2016.24

Xia, Y., Yan, G., & Zhang, H. (2014). Analyzing the significance of process metrics for TT&C software defect prediction. *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, 77–81. https://doi.org/10.1109/ICSESS.2014.6933517

distance in high dimensional spaces. *Optik*, *126*(24), 5614–5619. https://doi.org/10.1016/j.ijleo.2015.09.093

Xu, Z., Liu, J., Luo, X., Yang, Z., Zhang, Y., Yuan, P., Tang, Y., & Zhang, T. (2019). Software defect prediction based on kernel PCA and weighted extreme learning machine. *Information and Software Technology*, *106*, 182–200. https://doi.org/10.1016/j.infsof.2018.10.004

Yan, M., Xia, X., Fan, Y., Lo, D., Hassan, A. E., & Zhang, X. (2020). Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)* (pp. 1308-1319). https://doi.org/10.1145/3368089.3417048

Yang, L., Li, X., & Yu, Y. (2017). VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes. *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 1–7. https://doi.org/10.1109/GLOCOM.2017.8254428

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

Yang, X., Lo, D., Xia, X., Zhang, Y., & Sun, J. (2015). Deep Learning for Just-in-Time Defect Prediction. *Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015*, *1*, 17–26. https://doi.org/10.1109/QRS.2015.14

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., & Leung, H. (2016). Effort-Aware just-in-Time defect prediction: Simple unsupervised models could be better than supervised models. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, *13-18-Nove*, 157–168. https://doi.org/10.1145/2950290.295035

Zha, Q., Yan, X., & Zhou, Y. (2018). Adaptive Centre-Weighted Oversampling for Class Imbalance in Software Defect Prediction. *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, 223–230. https://doi.org/10.1109/BDCloud.2018.00044

Zhang, Y., Yan, X., & Khan, A. A. (2020). A Kernel Density Estimation-Based Variation Sampling for Class Imbalance in Defect Prediction. *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 1058–1065.

https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00159

Zhang, Y., Zuo, T., Fang, L., Li, J., & Xing, Z. (2021). An Improved MAHAKIL Oversampling Method for Imbalanced Dataset Classification. *IEEE Access*, *9*, 16030–16040. https://doi.org/10.1109/ACCESS.2020.3047741

Zhao, K., Xu, Z., Zhang, T. Z., Tang, Y., & Yan, M. (2021). Simplified deep forest model based just-in-time defect prediction for android mobile apps. *IEEE Transactions on Reliability*, *70*(2), 848–859. https://doi.org/10.1109/TR.2021.3060937

Zhao, Y., Damevski, K., & Chen, H. (2023). A Systematic Survey of Just-in-Time Software Defect Prediction. *ACM Computing Surveys*, *55*(10), 1–35. https://doi.org/10.1145/3567550

Zheng, W., Shen, T., & Chen, X. (2021). Just-in-Time Defect Prediction Technology based on Interpretability Technology. *Proceedings - 2021 8th International Conference on Dependable Systems and Their Applications, DSA 2021*, 78–89. https://doi.org/10.1109/DSA52907.2021.00017

Zimmermann, T., & Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. *Proceedings - International Conference on Software Engineering*, 531–540. https://doi.org/10.1145/1368088.1368161

Zhu, K., Zhang, N., Ying, S., & Zhu, D. (2020). Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. *IET Software*, *14*(3), 185–195. https://doi.org/10.1049/iet-sen.2019.0278