UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Dinaldo Andrade Pessoa

Batch Algorithms and Fixed Prediction Rates for Online Just-In-Time Software Defect Prediction

Recife

2021

Dinaldo Andrade Pessoa

Batch Algorithms and Fixed Prediction Rates for Online Just-In-Time Software Defect Prediction

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de Concentração: inteligência computacional

Orientador: Adriano Lorena Inácio de Oliveira

Coorientador: George Gomes Cabral

Recife

2021

Catalogação na fonte
Bibliotecário Cristiano Cosme S. dos Anjos, CRB4-2290

P475b Pessoa, Dinaldo Andrade

Batch algorithms and fixed prediction rates for online just-in-time software defect prediction / Dinaldo Andrade Pessoa. – 2021.

93 f.: il., fig., tab.

Orientador: Adriano Lorena Inácio de Oliveira.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2021.

Inclui referências e apêndices.

1. Inteligência Computacional. 2. Predição de defeito de software. 3. Latência de verificação. 4. Desbalanceamento de classes. I. Oliveira, Adriano Lorena Inácio de (orientador). II. Título.

006.31 CDD (23. ed.) UFPE - CCEN 2021 – 92

Dinaldo Andrade Pessoa

"**Batch Algorithms and Fixed Prediction Rates for Online Just-In-Time Software Defect Prediction**"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 26/03/2021.

BANCA EXAMINADORA

Prof. Dr. George Darmiton da Cunha Cavalcanti

Centro de Informática / UFPE

Prof. Dr. Luiz Eduardo Soares de Oliveira

Departamento de Informática / UFPR

Prof. Dr. Adriano Lorena Inácio de Oliveira

Centro de Informática / UFPE

(**Orientador**)

Dedico este trabalho a minha família, que é o meu porto seguro perante as dificuldades da vida.

ACKNOWLEDGEMENTS

Agradeço ao meu orientador Adriano Lorena Inácio de Oliveira e ao meu co-orientador George Gomes Cabral pela disponibilização de infraestrutura para a execução dos experimentos e por guiarem o desenvolvimento deste trabalho.

Agradeço ao professor Leandro L. Minku pela colaboração na definição das questões de pesquisa.

Agradeço ao colega Gustavo Henrique Ferreira de Miranda Oliveira pelos conselhos práticos sobre como avançar na pesquisa e pela ajuda na execução dos experimentos.

Agradeço ao Centro de Informática da UFPE por dispor de disciplinas e professores que viabilizaram a construção do capital intelectual necessário para o desenvolvimento deste trabalho.

ABSTRACT

Just-In-Time Software Defect Prediction (JIT-SDP) is aimed at predicting the presence of defects in code changes at the commit time instead of inspecting modules (i.e., files or packages) in offline mode, as performed in traditional Software Defect Prediction (SDP). In a real-world application of JIT-SDP, predictions must be done in an online fashion so that the developer is informed about the presence of defect as soon as the code change is submitted, providing to the developer the opportunity to further inspect the change while it is still fresh in one's mind. On the other hand, the model training can be done in an online or a batch fashion, since this problem domain does not have real-time requirements. Regardless the type of training, it is important to note that the code change is not labeled immediately after its submission to the source code repository. The labelling time may take days or months, depending on the time spent by the software development team to find and fix each defect. So, the model must wait some time to trust in a label of a code change. And this amount of time is known as verification latency. Another challenge faced by a JIT-SDP model is the fluctuation of the class imbalance rate through time. This kind of concept drift is known as class imbalance evolution. This work investigates the use of batch algorithms for dealing with JIT-SDP in the context of verification latency and class imbalance evolution. In comparison to the state-of-the-art, which is based on online algorithms, our approach (BORB) achieved improvements between $+2\%$ and $+11\%$ on 9 of the 10 investigated datasets, in terms of g-mean. In only one dataset, BORB achieved a result inferior to the state-of-the-art approach, a decrease of -2% in terms of g-mean. Besides that, this work investigates the predictive performance in a context in which the model is constrained to output a fixed defect prediction rate. More specifically, the defect prediction rate is an online rate that corresponds to the number of predictions which return the defect class divided by the total of predictions in a time interval. And a fixed defect prediction rate means to constraint the model to maintain the specified rate over time. That said, the results of the experiments show that, under this constraint, methods with higher capability to maintain the defect prediction rate close to the fixed defect prediction set by the hyperparameter tuning also obtain a higher predictive performance in the testing data, i.e., there is a meaningful correlation between this capability and the predictive performance. The correlation coefficient between them is 0.44. This result, added to the simplicity of the approach, suggests that a fixed defect prediction rate may be used as a standard baseline to the problem of class imbalance evolution.

Keywords: software defect prediction; verification latency; class imbalance; concept drift.

RESUMO

Just-In-Time Software Defect Prediction (JIT-SDP) tem o objetivo de identificar a presença de defeitos em mudanças de código no momento do commit ao invés de inspecionar módulos (i.e., arquivos e pacotes) de maneira offline, como é realizado em Software Defect Prediction (SDP) tradicional. Em uma aplicação real de JIT-SDP, as predições devem ser feitas de forma online para que o desenvolvedor seja informado sobre a presença de defeito logo após a mudança de código ser submetida, provendo ao desenvolvedor a oportunidade de inspecionar a mudança enquanto ela ainda está fresca em sua mente. Por outro lado, o treinamento do modelo pode ser feito de forma online ou em lote, uma vez que este domínio de problema não possui requisitos de tempo real. Independente do tipo de treinamento, é importante notar que a mudança de código não é rotulada imediatamente após a sua submissão para o repositório de código fonte. O tempo de rotulagem pode levar dias ou meses, dependendo do tempo gasto pela equipe de desenvolvimento de software para descobrir e corrigir cada erro. Então, o modelo deve esperar um tempo para confiar no rótulo de uma mudança de código. E este período de tempo é conhecido como latência de verificação. Outro desafio enfrentado por um modelo de JIT-SDP é a flutuação da taxa de desbalanceamento das classes ao longo do tempo. Este tipo de mudança de conceito é conhecido como evolução no desbalanceamento das classes. Este trabalho investiga o uso de algoritmos em lote para lidar com JIT-SDP no contexto de latência de verificação e evolução no desbalanceamento das classes. Em comparação com o estado da arte, que é baseado em algoritmos online, nossa abordagem (BORB) alcançou melhorias entre $+2\%$ e $+11\%$ em 9 das 10 bases de dados investigadas, em termos de g-mean. Em apenas uma base de dados, BORB obteve um resultado inferior ao da abordagem estado da arte, uma baixa de -2% em termos de g-mean. Além disso, este trabalho investiga a performance preditiva em um contexto no qual o modelo é restrito a retornar uma taxa de predição de defeito fixa. Mais especificamente, a taxa de predição de defeito é uma taxa online que corresponde ao número de predições que retornam a classe de defeito dividido pelo total de predições em um intervalo de tempo. E a taxa de predição de defeito fixa significa restringir o modelo para manter a taxa especificada ao longo do tempo. Dito isso, os resultados dos experimentos mostraram que, submetido a esta restrição, métodos com mais capacidade de manter a taxa de predição de defeito próximo à taxa de predição fixa definida pela otimização de hiperparâmetros também obtém uma melhor performance preditiva no dados de teste, i.e., há uma correlação significativa entre esta capacidade e a performance preditiva. O coeficiente

de correlação entre elas é 0.44. Este resultado, adicionado à simplicidade da abordagem, sugere que a taxa de predição de defeito fixa pode ser usada como uma linha de base padrão para o problema de evolução no desbalanceamento das classes.

Palavras-chaves: predição de defeito de software; latência de verificação; desbalanceamento de classes; mudança de conceito.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

BORB	Batch Oversampling Rate Boosting
BORB-NB	Batch Oversampling Rate Boosting associated with Naive Bayes
BORB-IRF	Batch Oversampling Rate Boosting associated with Iterative Random Forest
BORB-IHF	Batch Oversampling Rate Boosting associated with Iterative Hoeffding Forest
BORB-LR	Batch Oversampling Rate Boosting associated with Logistic Regression
BORB-MLP	Batch Oversampling Rate Boosting associated with Multilayer Perceptron
CPSQ	Cost of Poor Software Quality
DT	Decision Tree
HATT	Hoeffding Anytime Tree
HT	Hoeffding Tree
IHF	Iterative Hoeffding Forest
IRF	Iterative Random Forest
JIT-SDP	Just-In-Time Software Defect Prediction
LOC	lines of code
LR	Logistic Regression
MLP	Multilayer Perceptron
N	Negative
NB	Naive Bayes
OHT	Online Bagging of Hoeffding Trees
OO	object-oriented
ORB	Oversampling Rate Boosting
ORB-OHT	Oversampling Rate Boosting associated with Online Bagging of Hoeffding Trees
P	Positive

RF	Random Forest
SDP	Software Defect Prediction
TN	True Negative
TP	True Positive
USA	United States of America

LIST OF SYMBOLS

α	Alpha
θ	Theta
$\in$	Belongs to
$\|A\|$	Cardinality of the collection
$\rightarrow$	Imply
$\vec{a}$	Vector
∞	Infinity
$<$	Less than
$\leq$	Less than or equals
$>$	Greater than
$\geq$	Greater than or equals
$=$	Equals
$+$	Addition
$-$	Subtraction
$\times$	Multiplication
$\frac{a}{b}$	Division
$\%$	Percentual
$\sqrt{}$	Square root
$\sum$	Summation
$\|a\|$	Absolute value
mod	Module
$f(x)$	Function

$\underset{x}{arg\,max}\ f(x)$ Argument x that maximizes $f(x)$

$\leftarrow$ Assignment

CONTENTS

1 INTRODUCTION

Nowadays, software is a ubiquitous technology and is considered important to all sort of industries. Software can be seen as a way to scale up business processes and a defect introduced during the software development is scaled up too. Despite the advances in software development tools and processes, it is virtually impossible to conduct an entire software development process without introducing defects. Apparently, this is a problem that can be mitigated, but not totally solved.

1.1 MOTIVATION

According to Krasner (2020), the total Cost of Poor Software Quality (CPSQ) in the United States of America (USA) for the year 2020 was 2.08 trillion (T), without considering the future cost of the technical debt residing in defects that need to be corrected. He also said that:

- The largest contributor to CPSQ is operational software failures. For 2020, we estimated that it is $1.56 T, a 22% growth over 2 years — but that could be underestimated given the meteoric rise in cybersecurity failures, and that many failures go unreported. The underlying cause is primarily unmitigated flaws in the software.
- The next largest contributor to CPSQ is unsuccessful development projects totaling $260 billion (B), which rose by 46% since 2018. The project failure rate has been steady at 19% for over a decade. The underlying causes are varied, but one consistent theme has been the lack of attention to quality.
- Legacy system problems contributed $520 B to CPSQ (down from $635 B in 2018).

Krasner (2020) general recommendations emphasize prevention. However, when the prevention fails, he says that "the next best approach is to address weaknesses and vulnerabilities in software by isolating, mitigating, and correcting them as closely as possible to where they were injected to limit the damage". More specifically, he gives two recommendations that can be benefited from the research area of Software Defect Prediction (SDP): "(1) Recognize the

inherent difficulties of developing software and use effective tools to help deal with those difficulties. (2) Ensure early and regular analysis of source code to detect violations, weaknesses, and vulnerabilities". More specifically, SDP applies machine learning techniques to create effective tools (i.e., models) that analyze the source code to detect defect-prone modules (LI; JING; ZHU, 2018; NAM, 2014) and, in the case of Just-In-Time Software Defect Prediction (JIT-SDP), analyze code changes in order to identify defect-inducing changes (KAMEI et al., 2013).

In addition, Jones (2011) showed that the average cost to build, maintain and support software applications for five years in the USA drops from $2,000 to $1,200 per function point when the development team uses effective combinations of defect prevention and defect removal activities to achieve high quality levels. Thus, the identification of software defects is a relevant problem in the current software development environment. And, this work tackles it by applying machine learning techniques.

1.2 SOFTWARE DEVELOPMENT CONTEXT

In a software development process, a source code repository can be seen as a stream of code changes in which each committed code change may be intended to add a new feature, modify an existing one, or to fix a defect introduced by a previous code change. In this context, it would be helpful for a development team to know as soon as possible whether a code change induces a defect or not. With this information, the developer who touched the code may review it and, if necessary, fix it. Or some colleague may give more attention to that specific change by conducting a peer-review session. One advantage of the focus on code changes is that the development team reviews only specific code snippets, while approaches based on the inspection of modules (i.e., files and packages) are time-consuming and impractical for a large software system (KAMEI et al., 2013).

To train and evaluate a model to classify code changes, one must consider the following four characteristics of a source code repository: 1) There are many more clean code changes than defect-inducing ones (i.e., this is a *class imbalanced* problem). 2) The code changes are created in a certain *time order* and a code change can only be classified with a model trained with the previous code changes. 3) There is a time period to confirm whether a code change induces a defect or not, as the development team may take days or months to find and fix the defect after the defect-inducing code change has been created. The delay for receiving

the true labels is known as *verification latency*. 4) The data used to train the model changes over time. These modifications are called *concept drifts*. And the concept drifts on the class imbalance is also known as *class imbalance evolution*.

Most of the works in the literature of Software Defect Prediction (SDP) considers the presence of class imbalance in the source code repository, and some of them study this topic in depth (WANG; YAO, 2013). However, to the best of our knowledge, the time order, verification latency and concept drifts have been neglected in the literature of SDP up until Tan et al. (2015)'s work, when they introduced verification latency and a time sensitive approach to train and evaluate the model. Since then, more works moved towards a realistic scenario by considering time order, verification latency and concept drifts. McIntosh (2018) investigated how the concept drifts on the properties of the code changes impact the performance and interpretation of the model. And Cabral et al. (2019) investigated the verification latency in depth, and how the class imbalance evolution impacts the performance of the model.

This work focuses on the creation of a model to predict whether a code change induces a defect or not in a realistic scenario, in which we assume the presence of class imbalance, time order, verification latency and concept drifts. Similarly to the state-of-the-art (CABRAL et al., 2019), our approach (1) balances the training data so that the model learns both classes properly, (2) respects the time order of the code changes, (3) and waits some time period before using the code changes for training the model.

On the other hand, our approach tackles concept drifts differently. It periodically restarts and retrains the model using random samples of the historical data while the state-of-the-art model learns new concepts incrementally over the old ones. As a consequence, we expect that our approach be able to take advantage of the historical data to improve the predictive power when compared to the state-of-the-art. In addition, we expect that our approach maintains the predictive performance between periodic retrainings since McIntosh (2018) showed that models trained using more data tends to retain the performance for a longer time. Additionally, more modifications were introduced in order to allow the investigation of the predictive performance in a context in which the model is constrained to output a fixed defect prediction rate. More details about these modifications are given in Chap. 4.

Before moving forward, we must specify that the defect prediction rate is the number of predictions which return the defect-inducing class divided by the total number of predictions, regardless of the defect predictions being true positives or false positives. More formally, assuming the clean class as the negative class and the defect-inducing class as the positive one,

$defect\ prediction\ rate = (true\ positives + false\ positives)/predictions$. Furthermore, a fixed defect prediction rate means to constraint the model to maintain the specified rate over time. This work focuses on the defect prediction rate since the prediction rate of the clean class is the complement of the former. The same relationship of complementary applies to the fixed defect prediction rate.

In practice, the fixed defect prediction rate may be set in two ways: 1) Discretionarily, in which the project manager sets the expected rate of defect predictions according to the resources available (i.e., time and staff) to review the code changes. 2) Or automatically, in which some evaluation metric should be optimized according to the fixed defect prediction rate. In this work, we use the second option. More specifically, we choose the fixed defect prediction rate that maximizes the g-mean during the hyperparameter tuning procedure.

1.3 PROCESS AND MODEL INTEGRATION

Given the advent of distributed version control systems, particularly Git (CHACON; STRAUB, 2014), a new paradigm for distributed software development emerged: pull-based software development. In this paradigm, developers make the code changes in their own repositories and when they are finished, they submit the code changes in a *pull-request* to the main repository and wait for a peer revision. Then, the peer can ask for changes, accept or reject the pull-request. This process may iterate over and over until acceptance or rejection of the pull-request. After some time, pull-based software development became so popular in both open- and closed-source projects that new platforms were created to support it (GOUSIOS; PINZGER; DEURSEN, 2014), notably GitHub is the most popular.

In online Just-In-Time Software Defect Prediction (JIT-SDP) literature, the state-of-the-art solution (CABRAL et al., 2019) achieved advances in predictive performance considering a realistic scenario with respect to the order of the code changes and the verification latency, but they failed in obtaining a performance level that could allow practical applications on some investigated datasets. For example, when evaluated on the entire data stream (i.e., entire software project), the worst g-mean achieved by our reimplementation of the state-of-the-art was 51.03%. The predictive performance was especially poor on the beginning of the software project. With this level of performance, the software practitioners definitely would not rely on this classifier.

The poor performance on some investigated datasets may have happened, in part, due

to the limitations imposed by the online setting applied for training the model, which is constrained to use only the most recent labeled code change per training iteration. In present work, we assume that the code changes are created in a relatively low frequency so that the machine learning model does not have to learn them in real-time. Besides that, we assume that the pull-based software development is the paradigm chosen by the development team, similarly to Tan et al. (TAN et al., 2015). So, we can expect that the code changes arrive in batches appended to the pull-requests, and that each pull-request takes a substantial amount of time to be accepted, particularly in open-source projects (GOUSIOS; PINZGER; DEURSEN, 2014).

In this context, to incorporate the most recent data, the model can be rebuilt based on the historical data, which includes recent and old instances, after one or more pull-requests being accepted so that the use of the model in practice is not affected. This is feasible because the repository compression makes the data volume small. In fact, the repository is small enough to be kept in a single hard disk, as the developers do in their workstations. Additionally, over the lifecycle of the software, the impact of the increasing number of code changes in the training phase can be addressed by subsampling techniques, for example. The subsampling allow us to set an upper bound on the computational resources spend for training the model using a batch algorithm.

Batch algorithms have been successfully applied in online JIT-SDP literature (TAN et al., 2015; YANG et al., 2016; CHEN et al., 2018; MCINTOSH; KAMEI, 2018). In this work, we expect that they achieve better predictive performance when compared to the online ones since they can revisit the training data many times and, when associated to random sampling, they are expected to not emphasize recent data. Notice that we avoid emphasizing recent data because we do not have prior information about the concept drifts on the investigated datasets. Since our model does not know whether the next code change comes from an old concept or a new one, it relies on the accumulated training data to learn the most prevalent concepts while accepting the risk of a slower adaptation to new concepts.

Finally, despite the training phase being based on batch algorithms, our approach assumes that the model has online requirements for the testing phase so that it can be used as an on-demand service by any developer of the team.

1.4 RESEARCH QUESTIONS

This work has the objective of investigating the predictive performance of batch algorithms compared to online algorithms and investigating how the capability to output a fixed defect prediction rate is correlated to the predictive performance.

Our study considers Oversampling Rate Boosting (ORB) (CABRAL et al., 2019) as the baseline. To the best of our knowledge, ORB is the state-of-the-art approach for online JIT-SDP, but it has shown to achieve poor results on some datasets. The major weaknesses that we observed in ORB are related to its association with an online base learner that overemphasizes recent data and the susceptibility to noise of the instance-based oversampling. The base learner in question is the Hoeffding Tree (HT) (DOMINGOS; HULTEN, 2000; GAMA et al., 2014).

Our approach can be seen as a ORB modified to, regardless of the choice of base learner, avoid overemphasizing recent data and decrease the susceptibility to noise by resampling historical data in a batch mode. Despite the use of batch resampling, our approach can be associated with both batch and online base learners. In fact, the introduction of batch algorithms into ORB inspired the name of our approach: Batch Oversampling Rate Boosting (BORB). Besides that, BORB is designed to output a fixed defect prediction rate.

That said, we can now list the research questions that we answered in this work:

RQ1 Why does ORB perform poorly in terms of predictive performance on some datasets?

RQ2 How to modify ORB to use historical data for resampling and output a fixed defect prediction rate? How well BORB performs compared to ORB in terms of predictive performance?

RQ3 How does BORB perform when using different base learners? In particular, do the use of different base learners further improves the predictive performance of BORB?

RQ4 Is BORB able to output a fixed defect prediction rate? How correlated are the capability to output a fixed defect prediction rate and the predictive performance? And how adequate is each base learner to output a fixed defect prediction rate?

Regarding RQ1, RQ2 and RQ3, we formalize the methodology in which the predictive performance is improved in Subsection 3.1.4. And, regarding RQ4, we formalize the hypotheses about the fixed defect prediction rate in Subsection 3.1.5.

1.5 CONTRIBUTIONS

After finished the study of the research questions, this work ended up with the following novel contributions to the field of JIT-SDP:

1. Proposing a new approach able to considerably improve the predictive performance in comparison to the state-of-the-art approach.
2. Evaluating the predictive performance of the novel approach with different base learners, including online and batch algorithms.
3. Investigating the predictive performance in a context in which the model is constrained to output a fixed defect prediction rate.
4. Finding a meaningful correlation between the capability to output a fixed defect prediction and the predictive performance.
5. Evaluating which base learners are most capable to output a fixed defect prediction rate.

1.6 DOCUMENT STRUCTURE

The remainder of this document is structured as follows. Chap. 2 reviews the main works related to SDP, JIT-SDP, online JIT-SDP and class imbalance. Chap. 3 introduces key concepts for a proper understanding of this work. Chap. 4 discusses the limitations in the state-of-the-art approach and presents the proposed approach. Chap. 5 describes the base learners, the evaluation metrics, the datasets and the hyperparameter tuning used in the experiments. Then, it also describes the experimental setups defined to answer the research questions, and discusses the results. Finally, Chap. 6 summarizes our conclusions and presents open questions that may be investigated in future work.

2 RELATED-WORK

In this chapter, we first discuss Software Defect Prediction (SDP) literature and its subfields more related to the present work: JIT-SDP and online JIT-SDP. Then, we discuss class imbalance evolution in the context of JIT-SDP. In the following discussion, we try to consider previous works also connecting them with ours.

2.1 SOFTWARE DEFECT PREDICTION

To the best of our knowledge, the first study of SDP estimated the number of defects in a software project by using the lines of code (LOC) as a complexity measure (AKIYAMA, 1971). However, according to Nam (2014), this code metric is too simple to properly represent the complexity of a software. So, the cyclomatic complexity (MCCABE, 1976) and the Halstead complexity (HALSTEAD, 1977) were introduced, and became very popular code metrics for estimating the number of defects in a software. Shen et al. (1985) used that code metrics to build a linear regression model to predict the number of defects in modified modules and in new modules. Munson (1992) added more code metrics to the set of the previous work, and built a discriminant model to identify the modules most prone to defects. He also applied principal component analysis to reduce the multicollinearity problem in the code metrics. After the increase of popularity of object-oriented (OO) design, Chidamber (1994) proposed a suite of code metrics for that design paradigm. And Basili et al. (1996) used them to predict defect-prone classes.

Few years later, given the vast adoption of version control systems making code changes being accumulated in these repositories, many new process metrics were proposed (NAGAPPAN; BALL, 2005; KIM et al., 2007; HASSAN, 2009; D'AMBROS; LANZA; ROBBES, 2010; BIRD et al., 2011). They were used in a new subfield of SDP that would be later called Just-In-Time Software Defect Prediction (JIT-SDP). In this context, process metrics are statistics of code changes computed between two versions of a module. For instance, time spent since the last change of the module. These statistics can also be related to the changes itself instead of the touched module. For instance, experience of the developer who created the change. On the other hand, code metrics come from the source code of one specific version of a module. For instance, number of parameters of a function and number of lines of a function.

Table 1 – Aspects of SDP research.

	Aspect	Value	Description
SDP	Training data	**Within-project**	Model is built and test on data from the same project
		Cross-project	Model is built on data from multiple projects and tested on one or many projects
	Granularity	Module	A module version is a problem instance
		Code change	A code change is a problem instance
	Metrics	Code metrics	Metrics from a source code version
		Process metrics	Metrics from code changes
	Time Order	Static	No time order
		Dynamic	Time order
	Algorithm	**Batch**	Batch resampling and learning
		Online	Online resampling and learning

Source: Created by the author (2021)

Moser et al. (2008) and Rahman (2013) conducted a comparative analysis of code metrics and process metrics. Their works suggest the use of process metrics instead of code metrics. Rahman (2013) shows that code metrics may not evolve with the changing distribution of defects over the files. In that case, the model is led to focus on recurrently defective modules instead of defect-dense modules. A more detailed review of SDP literature can be found in Nam (2014) and Li et al. (2018)'s works.

To better segregate the subfields of SDP, Table 1 shows key characteristics of the different branches of SDP. In this table, names in bold represent characteristics addressed in the present work. According to them, our work uses data from a single project to build and test the model; the instances of the datasets are code changes; the features of the problem come from statistics related to the code changes; the chronology of the changes is respected so that the model is trained only with instances labeled before current test time; and the algorithms proposed to improve the predictive performance are batch.

2.2 JUST-IN-TIME SOFTWARE DEFECT PREDICTION

Traditional SDP inspects entire modules to classify which ones are defective in offline mode. Differently, JIT-SDP is aimed at predicting the presence of defects in code changes at the commit time. To the best of our knowledge, Mockus (2000) was the first work to handle code changes. He identified the causes of the code changes by processing the textual description field of these changes. The causes addressed by that work were the addition of new features, the correction of defects, and code refactoring. However, that work did not identify which code changes induce defects. Śliwerski et al. (2005) were the first ones to

label code changes as defect-inducing or clean by linking the version control system to a bug tracking database. Today, their algorithm is known as the SZZ, after the three authors: Śliwerski, Zimmermann, and Zeller. Since then, several studies introduced new features to code changes for different purposes. Kim et al. (2008) introduced the change classification focused on file changes. They added features from change metadata, complexity metrics at the moment of the change, change log messages, source code and file names. Eyolfson et al. (2011) investigated the correlation between social characteristics of the commits (code changes) and the probability of being defect-inducing. The investigated characteristics were the time of the day of the commit, the day of the week of the commit and the developer's experience. Shihab et al. (2012) found the 5 best indicators to identify risky changes (changes likely to be defect-inducing) from a total of 23 indicators. The number of lines added, the number of locations that the added lines are spread, the worst ratio between fix changes and total changes by file being changed, the number of bug reports linked to the change, and the developer's experience were considered the best indicators of a change to be defect-inducing. Misirli et al. (2016) specialized the model to identify high impact defect-inducing changes. The high impact is defined with respect to the developer's perspective. So, it is considered the amount of churn (i.e., number of lines of code modified), number of modified files and the number of subsystems affected by the change.

Kamei et al. (2013) conducted a large-scale study on six open-source and five commercial projects based on 13 characteristics of the code changes. They built an effort-aware model to maximize the number of identified defect-inducing changes while minimizing the effort necessary to review these changes. Obtaining these 13 features from open projects became easier when Rosen (2015) released Commit Guru, a tool to extract data from GitHub repositories and label the code changes as clean or defect-inducing.

None of the mentioned JIT-SDP studies investigated the predictive performance in a context in which the model is constrained to output a fixed defect prediction rate.

2.3 ONLINE JUST-IN-TIME SOFTWARE DEFECT PREDICTION

Online JIT-SDP can be considered as a subfield of JIT-SDP in which time plays a relevant role in how the model is trained and tested. The code changes are tested following the order that they are committed, and the training phase can use only code changes whose label is available before the code change being tested. Tan et al. (2015) were the first ones to study

online JIT-SDP. They also introduced the requirement of waiting a fixed time period after the code change commit so that the labels assigned to the training instances are more likely to be the true labels. The delay for receiving the true labels is known as verification latency (CABRAL et al., 2019) and, in JIT-SDP domain, it takes place because the software development team may take days or months to find out that a code change is defect-inducing after its commit. McIntosh (2018) investigated how the properties of the code changes fluctuates over time. He found out that those fluctuations (concept drifts) causes a gradual loss of predictive performance for the model. So, the model needs to be updated periodically to incorporate the most recent data. He also found out that the models trained with long historical data retain predictive performance for longer than when trained with short historical data. Cabral et al. (2019) investigated how the class imbalance evolution hinders the predictive performance of the model in the presence of verification latency. Differently from Tan et al. (TAN et al., 2015), Cabral et al. (2019) proposed to use a defect-inducing change as soon as it is labeled by the fix instead of waiting for a fixed verification latency time interval for both classes. In this context, they proposed Oversampling Rate Boosting (ORB) to tackle the class imbalance evolution. ORB performs an online resampling based on the prequential ratio between the classes in the training data multiplied by an adjustable boosting factor intended to emphasize the class that have been underemphasized on the most recent predictions. Due to the online resampling mechanism, as limitation, ORB can be associated only with online base learners.

Online JIT-SDP was also studied in correlated subfields of JIT-SDP. Tabassum et al. (2020) used ORB to investigate how data from different software projects (cross-project data) can improve the model predictive performance on one target software project in an online scenario. Cross-project data improved the overall predictive performance and the predictive performance in the initial phase of the project. Besides that, cross-project data prevented sudden drops in the predictive performance due to concept drifts. Yang et al. (2016) and Chen et al. (2018) showed, with different approaches, that effort-aware models can better maximize the number of defect-inducing changes identified while minimizing the effort required to review these changes. They used timewise-cross-validation (online scenario) among other types of validation.

In this work, we modified ORB to support batch algorithms and output a fixed defect prediction rate. The former modification has the objective of investigating the predictive performance of batch algorithms compared to online algorithms, and the latter one is aimed at investigating how the capability to output the fixed defect prediction is correlated to the predictive performance in an online scenario. To the best of our knowledge, this is the first work

to do such investigations.

2.4 CLASS IMBALANCE IN JUST-IN-TIME SOFTWARE DEFECT PREDICTION

Class imbalance is a usual challenge in the JIT-SDP domain. Most of the studies apply resampling techniques as preprocessing (KAMEI et al., 2013; TAN et al., 2015; MISIRLI; SHIHAB; KAMEI, 2016), but some of them conduct a more in depth evaluation of these techniques (KAMEI et al., 2016; MALHOTRA; KHANNA, 2017; CABRAL et al., 2019). Kamei et al. (2013) and Misirli et al. (2016) applied undersampling to tackle class imbalance in an offline scenario. Tan et al. (2015) experimented four types of resampling techniques on the hyperparameter tuning procedure in order to choose the best option to apply on the testing data in online JIT-SDP. Using cross-project data in an offline scenario, Kamei et al. (2016) found out that undersampling tends to provide better predictive performance in terms of F-measure and recall. Malhotra (2017) experimented three resampling methods and one cost-sensitive method to deal with class imbalance. By doing this, they significantly improved the predictive performance of the models in an offline scenario. Cabral et al. (2019) experimented six resampling techniques to show that ORB, which is designed to tackle the class imbalance evolution in the presence of verification latency, can improve the predictive performance in online JIT-SDP. In their work, they identified the oversampling as the most reliable resampling technique in the investigated scenario.

2.5 SUMMARY

In this chapter, we have provided a literature review of SDP and the subfields related to this work: JIT-SDP, online JIT-SDP and class imbalance in JIT-SDP. We have also seen that online JIT-SDP considers a more realistic scenario when compared to offline JIT-SDP. In online JIT-SDP, the incoming code changes are classified respecting the chronology of its creation. Besides that, the classifier must wait a realistic time interval, verification latency, to simulate the true delay for receiving the training instances labels. Furthermore, when the verification latency is associated to the class imbalance problem, an even more challenging context is defined. In this work, we assume this realistic scenario to investigate our hypotheses about the fixed defect prediction rate and the performance's impact when using batch algorithms in the specified scenario.

3 FUNDAMENTALS

In this chapter, we introduce key concepts for a proper understanding of this work: data stream, model and classifier, the main evaluation metric, the methodology, and our hypotheses about the fixed defect prediction rate. In addition, we give an intuition on how to maximize the g-mean when constrained by a fixed defect prediction rate over the data stream. Finally, we define the verification latency and show the distribution of the delay to fix a defect on each dataset.

3.1 PROBLEM DEFINITION

3.1.1 Data Stream

As mentioned in Section 1.2, when we assume a pull-based software development, a source code repository can be seen as a stream of code changes that arrive in batches appended to pull-requests. However, for the sake of our problem formulation, we can ignore the pull-requests to work on the code change level. And define $d_t = (\vec{x}_t, y_t)$ as the stream of code changes, where $t \, \epsilon \, [0, \infty)$ is the time, $y_t \, \epsilon \, \{0, 1\}$ is the class of the code change — 0 for clean and 1 for defect-inducing — and $\vec{x}_t$ is the feature vector of the incoming code change.

3.1.2 Model and Classifier

The true posterior probability distribution of classes can be defined as

$$P_t(y|\vec{x}) = \frac{P_t(\vec{x}|y)P_t(y)}{P_t(\vec{x})}, \tag{3.1}$$

where y is the class, $\vec{x}$ is the feature vector, and t is the time. It is important to note that we can have a different distribution at each time t. The change on the probability distribution is called concept drift. And we can define the concept drift between the time t_0 and t_1 as $\exists \vec{x} : P_{t_0}(y, \vec{x}) \neq P_{t_1}(y, \vec{x})$, where $P_t(y, \vec{x}) = P_t(y|\vec{x})P_t(\vec{x}) = P_t(\vec{x}|y)P_t(y)$ is the joint probability at time t between y and $\vec{x}$ (GAMA et al., 2014).

As we do not have access to the true posterior probability distribution defined by Eq. 3.1, we need to create a model that estimates the distribution, defined as

$$P'_t(y|\vec{x}, D_t) = \frac{P_t(\vec{x}|y, D_t)P_t(y|D_t)}{P_t(\vec{x}|D_t)}, \tag{3.2}$$

where $t = 0, 1, ...$ is now the *timestep*, a discretization of the time, and $D_t = \{d_0, ..., d_t\}$ is the historical data at timestep t.

Furthermore, a step of thresholding is necessary to transform the posterior probability into the prediction. The threshold is the value used to decide if the posterior probability yielded by the model means a defect-inducing change or a clean one (HERNÁNDEZ-ORALLO; FLACH; FERRI, 2012). And the threshold choice method is the mechanism used to choose a threshold (HERNÁNDEZ-ORALLO; FLACH; FERRI, 2012). Our approach dynamically chooses thresholds to maintain the fixed defect prediction rate over the data stream. In this case, our classifier can be defined as

$$C'_t(\vec{x}, D_t) = \begin{cases} 1, & \text{if } P'_t(y = 1|\vec{x}, D_t) \geq th_t \\ 0, & \text{otherwise} \end{cases}, \tag{3.3}$$

where th_t is the threshold at the timestep t. So, when a code change has a posterior probability greater than or equal to th_t, it is classified as defect-inducing; otherwise, it is classified as clean.

Due to the non-stationary nature of the problem, the model needs to be updated periodically. In this work, we assume that there is a fixed number of code changes to wait before updating the model. That number is called the pull-request size (ps). After waiting for ps code changes, the model is restarted and trained with the historical data D_t. To implement that, Eq. 3.4 incorporates this hyperparameter to the classifier as defined by

$$P'_t(y|\vec{x}, D_t, \vec{\theta}) = \frac{P_t(\vec{x}|y, D_t, \vec{\theta})P_t(y|D_t)}{P_t(\vec{x}|D_t)}, \tag{3.4}$$

where $\vec{\theta}$ is the hyperparameters vector that includes ps. Furthermore, it is important to note that $\vec{\theta}$ is constant over the data stream. More details about ps and other hyperparameters are given in Section 4.2.

In view of the classifier's hyperparameters, Eq. 3.3 can be now rewritten as

$$C'_t(\vec{x}, D_t, \vec{\theta}) = \begin{cases} 1, & \text{if } P'_t(y = 1|\vec{x}, D_t, \vec{\theta}) \geq th_t \\ 0, & \text{otherwise} \end{cases}. \tag{3.5}$$

3.1.3 Main Evaluation Metric

The g-mean is our main evaluation metric. And, to formulate it, we first need to define the indicator function, the prequential recalls and the prequential g-means. The indicator function helps to segregate the code changes by class and the predictions by correctness. And it is

defined as

$$I(y', y) = \begin{cases} 1, & y' = y \\ 0, & \text{otherwise} \end{cases}, \tag{3.6}$$

where y' is the expected class and y is the true class of the code change or its prediction, depending on the metric being calculated. It is important to say that we calculate the prequential metrics as described in Gama et al. (2014)'s work. To do that, we assume some decay factor $\alpha \in [0, 1]$. Then, we make counts using the decay effect and calculate one prequential recall per class defined as

$$n_{c,t} = \begin{cases} 1 + \alpha n_{c,t-1}, & I(c, y_t) = 1 \\ n_{c,t-1}, & \text{otherwise} \end{cases}, \tag{3.7}$$

$$h_{c,t}(D_t, \vec{\theta}) = \begin{cases} I(y_t, C'_t(\vec{x}_t, D_t, \vec{\theta})) + \alpha h_{c,t-1}, & I(c, y_t) = 1 \\ h_{c,t-1}, & \text{otherwise} \end{cases}, \tag{3.8}$$

$$r_{c,t}(D_t, \vec{\theta}) = \frac{h_{c,t}(D_t, \vec{\theta})}{n_{c,t}}, \tag{3.9}$$

where c is the class that the recall is related to, t is the timestep, D_t is the historical data, $\vec{\theta}$ is the hyperparameters vector, and $n_{c,0} = 0$ and $h_{c,0} = 0$ are the initial values. Eq. 3.7 counts the prequential number of code changes of the class c, Eq. 3.8 counts the prequential number of correct classifications of the class c, and Eq. 3.9 calculates the prequential recall of the class c. Now, we can aggregate the prequential recalls of both classes into the prequential g-mean defined as

$$g\text{-}mean_t(D_t, \vec{\theta}) = \sqrt{r_{1,t}(D_t, \vec{\theta}) \times r_{0,t}(D_t, \vec{\theta})}. \tag{3.10}$$

Then, we average the prequential g-mean to calculate the g-mean defined as

$$g\text{-}mean(D, \vec{\theta}) = \frac{1}{|D|} \sum_{i=0}^{|D|} g\text{-}mean_i(D_i, \vec{\theta}), \tag{3.11}$$

where D is a segment of the data stream, $\vec{\theta}$ is the hyperparameters vector and $g\text{-}mean_i(D_i, \vec{\theta})$ is the prequential g-mean, defined by Eq. 3.10.

3.1.4 Methodology

The methodology used to evaluate the approaches follows the recommendations described by Salzberg (1997). First, we split the data stream into two non-overlapping sequential seg-

ments defined as

$$V = D_v,$$

$$T = D_t - D_v,$$

where $D_i = \{d_0, ..., d_i\}$ is the historical data at timestep i, $v \; \epsilon \; (0, t)$ is the timestep used for splitting D_t, V is the validation segment, and T is the testing segment. Then, we configure the model in the hyperparameter tuning procedure using V, and evaluate the configured model using T, as defined by

$$\begin{aligned} &\text{evaluate} \quad g\text{-}mean(T, \vec{\theta}_v) \\ &\text{subject to} \quad \vec{\theta}_v = \underset{\vec{\theta}}{arg\,max}\; g\text{-}mean(V, \vec{\theta}), \end{aligned} \tag{3.12}$$

where $\vec{\theta}_v$ is the hyperparameters vector that maximizes the g-mean on the segment V of the data stream, and $g\text{-}mean$ is defined by Eq. 3.11. More details about hyperparameter tuning are given in Section 5.4.

3.1.5 Hypotheses

The idea to investigate the fixed defect prediction rate comes from the fact that the mean is the point estimate that minimizes the mean squared error, as it is demonstrated in the following proof.

Let $E[X]$ be the expected value of the random variable X, and p be a point estimate of X. The mean squared error (MSE) of the point estimate p can be calculated as follows:

$$\begin{aligned} MSE(p) &= E[(X - p)^2] \\ &= E[X^2 - 2Xp + p^2] \\ &= E[X^2] - 2pE[X] + p^2. \end{aligned}$$

Then, to minimize $MSE(p)$, we find the value of p in which the derivative of $MSE(p)$ is equal to zero:

$$\begin{aligned} \frac{\partial MSE(p)}{\partial p} &= 0 \\ -2E[X] + 2p &= 0 \\ p &= E[X]. \end{aligned}$$

Thus, the expected value of X ($E[X]$) is the point estimate that minimizes the mean squared error. And $E[X]$ can be empirically estimated as the mean of the observations of X.

In an optimistic scenario, the model would know the true probability distributions of classes to produce a perfect ranking of the instances, and the classifier would know the true prequential defect rate over the data stream in order to choose optimal thresholds for the predictions. On that conditions, the predictive performance of the classifier would be perfect (HERNÁNDEZ-ORALLO; FLACH; FERRI, 2012). However, if we constrained the classifier to output a fixed defect prediction rate, the mean defect rate calculated on the whole data stream could be used as the best point estimate to the prequential defect rate in terms of mean squared error. In fact, the mean defect rate would give better information to support the classifier on the predictions over the data stream than a complete absence of information about the prequential defect rate (HERNÁNDEZ-ORALLO; FLACH; FERRI, 2012). To take advantage of that information, the classifier would have to maintain the prequential defect prediction rate equals to the mean defect rate over the data stream. In other words, the fixed defect prediction rate should be equals to the mean defect rate.

On the other hand, in practice, the model does not yield posterior probabilities that form a perfect ranking and the classifier does not know the true defect rate. Thus, there is no mean defect rate calculated on the whole data stream to be used as the fixed defect prediction rate. And, even if the true mean defect rate was available, its association with imperfect posterior probabilities would lead the classifier to suboptimal results. Only by including both uncertainties (i.e., unknown defect rate and imperfect posterior probabilities) into the decision-making process of the fixed defect prediction rate we can expect optimal results (FERRI; HERNÁNDEZ-ORALLO; FLACH, 2019). So, in this work, the fixed defect prediction rate that maximizes the g-mean is found empirically in the hyperparameter tuning procedure.

Notice that the fixed defect prediction rate is a simplistic approach to deal with class imbalance evolution since it does not adapt to fluctuations on the prequential defect rate. However, we show that it may be considered as a standard baseline to this problem since, after choosing the fixed defect prediction rate that maximizes the g-mean in the hyperparameter tuning, the capability to maintain this rate over the testing segment is related to higher g-means. More details about g-mean maximization using a fixed defect prediction rate are given in Section 3.2.

That said, we can now formulate the hypotheses. The first hypothesis is that our approach is able to output a fixed defect prediction rate over the data stream. To evaluate that, we

need to calculate the prequential defect prediction rate as defined by

$$n_t = 1 + \alpha n_{t-1}, \tag{3.13}$$

$$h_{1,t}(D_t, \vec{\theta}) = I(1, C'_t(\vec{x}_t, D_t, \vec{\theta})) + \alpha h_{1,t-1}, \tag{3.14}$$

$$pr_{1,t}(D_t, \vec{\theta}) = \frac{h_{1,t}(D_t, \vec{\theta})}{n_t}, \tag{3.15}$$

where t is the timestep, D_t is the historical data, $\vec{\theta}$ is the hyperparameters vector, $n_0 = 0$ and $h_{1,0} = 0$ are the initial values. Eq. 3.13 counts the prequential number of predictions, Eq. 3.14 counts the prequential number of defect predictions, and Eq. 3.15 calculates the prequential defect prediction rate. Then, we need to average the prequential distance between the fixed defect prediction rate and the prequential defect prediction rate as defined by

$$|fr_1 - pr_1|(D, \vec{\theta}) = \frac{1}{|D|} \sum_{i=0}^{|D|} |fr_1 - pr_{1,i}(D_i, \vec{\theta})|, \tag{3.16}$$

where D is a segment of the data stream, $\vec{\theta}$ is the hyperparameters vector, fr_1 is the fixed defect prediction rate and $pr_{1,i}$ is the prequential defect prediction rate, defined by Eq. 3.15. Then, we can test if the distance between the fixed defect prediction rate and the defect prediction rate (i.e., the average of the distance) is below a specific value with statistical significance as defined by

$$\begin{aligned} H_0 &: |fr_1 - pr_1| = d, \\ H_a &: |fr_1 - pr_1| < d, \end{aligned} \tag{3.17}$$

where $|fr_1 - pr_1|$ is the distance between the fixed defect prediction rate fr_1 and the defect prediction rate pr_1, defined by Eq. 3.16, and d is a value that corresponds to the maximum acceptable distance, which is defined in Subsection 5.6.

The second hypothesis is that the capability to output the fixed defect prediction is correlated to the g-mean. A high capability to output a fixed defect prediction rate corresponds to a low distance between the fixed defect prediction rate and the induced defect prediction rate. And the induced defect prediction rate is the defect prediction rate achieved by the classifier during the training phase. More details about the induced defect prediction rate are given in Section 4.2. For now, we just need to assume that that distance can measure how capable the classifier is to output a fixed defect prediction rate. So, the second hypothesis can be formulated as

$$corr(|fr_1 - ir_1|, g\text{-}mean) = -c, \tag{3.18}$$

where $|fr_1 - ir_1|$ is the distance between the fixed defect prediction rate fr_1 and the induced defect prediction rate ir_1, $g\text{-}mean$ is defined by the Eq. 3.11 and c is a value that corresponds to a strong or moderate negative correlation, which is defined in Subsection 5.6.

3.2 G-MEAN MAXIMIZATION

Suppose one step of a cross-validation where the model outputs perfect ranked scores, and the related classifier has perfect knowledge about the class imbalance (defect rate) in the test data: the number of clean changes N and the number of defect-inducing changes P. In that case, our classifier can take all the first N code changes ranked by ascending score and classify them as clean. Then, take all the remaining P code changes and classify them as defect-inducing. That would result in a number of true negatives equals to the number of clean changes, $TN = N$; and a number of true positives equals to the number of defect-inducing changes, $TP = P$. So, the recalls and the g-mean would be 100% as defined by

$$\begin{aligned} r_0 &= \frac{TN}{N} = 100\%, \\ r_1 &= \frac{TP}{P} = 100\%, \\ g\text{-}mean &= \sqrt{r_0 \times r_1} = 100\%. \end{aligned} \tag{3.19}$$

Now, suppose another step of a cross-validation where the model outputs almost perfect ranked scores and the classifier still has perfect knowledge about the class imbalance in the test data. If the classifier outputs the same defect rate on the predictions, most of the code changes would be correctly classified, but $e > 0$ clean changes would be misclassified as defect-inducing and vice-versa. So, since JIT-SDP is an imbalanced problem where the clean class is the majority class, i.e., $N > P$, the recalls and the g-mean would be hindered as defined by

$$\begin{aligned} (\frac{TN}{N} = \frac{TP}{P} = 100\% \wedge N > P) \rightarrow \frac{TN - e}{N} > \frac{TP - e}{P} \rightarrow r'_0 > r'_1, \\ r'_0 = \frac{TN - e}{N} < 100\%, \\ r'_1 = \frac{TP - e}{P} < 100\%, \\ g\text{-}mean' = \sqrt{r'_0 \times r'_1} < 100\%. \end{aligned} \tag{3.20}$$

The new recalls would not be balanced and would not be 100%. As a consequence, the new g-mean would not be 100% too. To maximize the g-mean in this scenario, the classifier must ignore the perfect knowledge about the class imbalance and tries different defect rates to

accept more misclassifications of the clean class, dividing r'_0 by e_0, in exchange for more correct classifications of the defect-inducing class, multiplying r'_1 by e_1. In an optimistic scenario, the e_1 would be greater than e_0 and the new g-mean would be maximized as defined by

$$\begin{aligned} r''_0 &= \frac{r'_0}{e_0} < 100\%, \\ r''_1 &= r'_1 e_1 < 100\%, \\ 1 < e_0 < e_1 \rightarrow \frac{r'_0}{r''_0} < \frac{r''_1}{r'_1} &\rightarrow r'_0 r'_1 < r''_0 r''_1 \rightarrow g\text{-}mean' < g\text{-}mean'', \\ g\text{-}mean'' = \sqrt{r''_0 \times r''_1} = \sqrt{\frac{e_1}{e_0}}\sqrt{r'_0 \times r'_1} &= \sqrt{\frac{e_1}{e_0}} g\text{-}mean' < 100\%. \end{aligned} \tag{3.21}$$

With perfect ranked scores, the knowledge about the operating condition (defect rate) on the test data would be effective to set the best threshold (HERNÁNDEZ-ORALLO; FLACH; FERRI, 2012). However, in practice, the model outputs imperfect ranked scores and the classifier does not have perfect knowledge about the class imbalance in the test data. So, the classifier has to find the defect prediction rate that maximizes the g-mean empirically.

Optimal threshold choice methods such as ROC Analysis (FAWCETT, 2006) chooses the threshold that optimizes the evaluation metric using validation data, and assuming that that data comes from the same distribution as the testing data. In a step of a cross-validation, the optimal threshold choice method would be applied only once to choose the threshold that maximizes the g-mean. In an online evaluation, a distinct threshold would have to be chosen for each tested instance of the data stream in order to maximize the prequential g-mean in each timestep.

However, in this work, we want to investigate the fixed defect prediction rate that maximizes the g-mean (Eq. 3.11) instead of the many thresholds that maximize the prequential g-mean (Eq. 3.10). So, the classifier has to choose the thresholds in order to maintain the fixed defect prediction rate over the data stream, while the choice of the fixed defect prediction rate that maximizes the g-mean is done by the hyperparameter tuning.

That said, our approach applies two mechanisms to achieve the fixed defect prediction rate. They are the rate-driven resampling, that induces the model to output the fixed defect prediction rate during the training phase. And the rate-driven threshold choice method, that changes the threshold to achieve the fixed defect prediction rate during the testing phase. The calculation of the defect prediction rate over the testing data is based on a sliding window of the predictions.

The overall procedure of evaluation is the following: First, the fixed defect prediction is tuned in the hyperparameter tuning over the validation segment of the data stream. Then, the same value is applied over the testing segment. It is important to add that the validation segment should be big enough to include some cycles of the class imbalance evolution. That increases the chances of the hyperparameter tuning to find a fixed defect prediction rate that better generalizes over the testing segment.

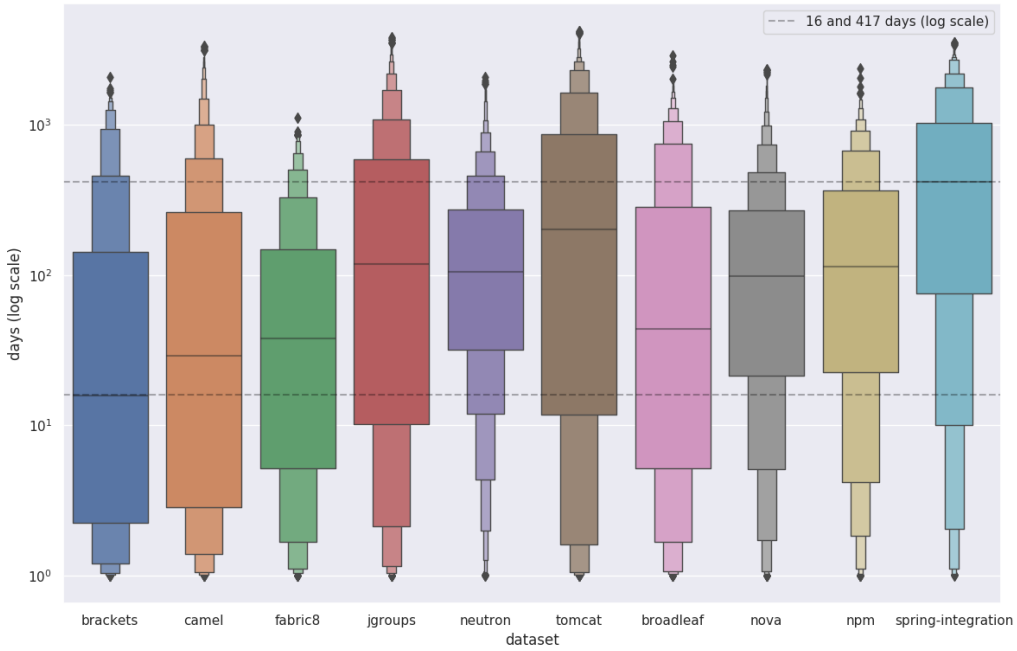
3.3 VERIFICATION LATENCY

As mentioned in Section 1.2, we need to wait a time period to confirm that a code change induces a defect or not. That comes from the fact that, after introducing a defect in the software, the development team takes some time to discover it and fix it. The delay for receiving the true labels is known as verification latency.

One way to implement the verification latency is to wait before labeling the training instances of both clean and defect-inducing changes. After that waiting time, the instance can be labeled using the SZZ algorithm (TAN et al., 2015). Another option is to wait a time period before labeling the clean changes, and labeling the defect-inducing changes as soon as they are fixed (CABRAL et al., 2019). In this work, we used the latter approach. More details about it are given in the Section 4.2.

Regardless the approach used for labeling, there is a trade-off between noise and concept drift when setting the waiting time. The greater the time period is, the lesser is the noise in the labels, and the larger are the chances of a concept drift impacting $P_t(\vec{x}|y)$ (CABRAL et al., 2019). Fig. 1 shows the distribution of the delay to fix a defect by dataset. In that figure, we can see that the medians range from 16 to 417 days. That suggests using a different waiting time for each dataset. In this work, we set up the hyperparameter tuning to find the best value for each dataset.

Figure 1 – Delay to fix a defect by dataset.



Source: Created by the author (2021)

3.4 SUMMARY

In this chapter, we have seen the fundamentals of our study: data stream, model and classifier, the main evaluation metric, the methodology and the hypotheses. Then, we have seen an intuition about how to maximize the g-mean when constrained by a fixed defect prediction rate over the data stream. Finally, we have described the verification latency and shown the distribution of the delay to fix a defect on the datasets used in this work.

4 PROPOSED APPROACH

In this chapter, we discuss the limitations in the state-of-the-art approach, ORB, and we present BORB as a solution to that problems. BORB is described in detail, including the framework to label the instances in the context of verification latency.

4.1 LIMITATIONS IN THE STATE-OF-THE-ART

To the best of our knowledge, ORB (CABRAL et al., 2019) is the state-of-art in online JIT-SDP, but it has achieved poor predictive performance on some investigated datasets. On the evaluation of present work, the performance was relatively poor on 7 of the 10 datasets in terms of rank based on the g-mean, as explained in Section 5.7. The following problems may be the cause of poor predictive performance on that datasets:

1. ORB is based on instance-incremental learning, so its oversampling is focused on the most recent training instance, even if that instance is noisy. The instance-based oversampling has another bad interaction with the base learner. An HT leaf uses the Hoeffding bound to determine the number of instances that must be observed before a split with confidence. However, in each timestep, the instance-based oversampling resamples only the current training instance. That increases the number of instances observed by the HT's leaf without the increase in the variance of the observed sample. In other words, there is a chance of the split being done with statistics from many repeated instances, which may result in overfitting the most recent instances.

2. The base learner associated with ORB, which is an online bagging (OZA, 2005) of Hoeffding Trees (HT) (DOMINGOS; HULTEN, 2000; GAMA; FERNANDES; ROCHA, 2006), adapts to concept drifts at the expense of old concepts. In fact, the only way that an HT can learn a new concept is to increase its depth by splitting the leaves using continuous attributes. The new concept is put over the previous one at every leaf of an increasingly complex tree (MANAPRAGADA; WEBB; SALEHI, 2018). Additionally, it is important to say that the majority of the attributes present in datasets used by ORB's paper are continuous. So, there is an infinite number of possible splits. This problem contributes to overemphasizing the recent data.

Hoeffding Anytime Tree (HATT) (MANAPRAGADA; WEBB; SALEHI, 2018) is an HT adaptation that is resilient to concept drifts and does not put new concepts over the old ones. In fact, it revisits previous node splits to check if the statics accumulated since the beginning of the data stream implicates in a different split in comparison to the current split. Thus, HATT gives the same emphasis to old and new concepts since information of the whole historical data is used to revisit the splits. However, its association with ORB would not remove the risk of oversampling noisy instances as this is an intrinsic problem of ORB. That said, it was necessary to create a new approach to avoid the risk of oversampling noisy instances and the risk of overfitting the most recent training by resampling historical data in a batch mode. Besides that, the new approach also ensure the same emphasis to old and new concepts by restarting the base learner periodically so that it revisits the whole historical data in the training phase.

As a consequence, we expect that, when associated with a batch base learner, our approach be able to take advantage of the historical data to improve the predictive power when compared to the state-of-the-art. In addition, we expect that the use of historical data in the training phase maintains the predictive performance between periodic retrainings, since McIntosh (2018) showed that models trained using more data tends to retain the performance for a longer time.

4.2 BATCH OVERSAMPLING RATE BOOSTING

Batch Oversampling Rate Boosting (BORB) consists of a ORB modified to support batch algorithms and output a fixed defect prediction rate over the data stream. Its pseudo-code is presented in Algorithm 1.

To be able to classify a code change any time, BORB contains an online test functionality (lines 3 to 5). However, in the beginning of the data stream, the base learner B predicts that all code changes are clean until at least one instance of each class is appended to the historical data D (line 6). After that, B is trained (lines 8 to 16) and becomes able to output effective scores and predictions.

Since then, all the steps of the algorithm starts to work as the following description. First, the base learner B yields the scores C on the most recent feature vectors $\vec{x_j}$ such as $j = t-w+1, ..., t$ (line 3). It is important to add that the base learner B is a model as defined by Eq. 3.4. So, $Score(B, \vec{x})$ can be implemented as the probability $P(y = 1|\vec{x})$ yielded by B. Then, in order to maintain the fixed defect prediction rate fr_1, the threshold th is calculated

Algorithm 1: BORB's testing and training procedure

1 **Input**: Incoming code changes $d_t = (\vec{x_t}, ?)$, base learner B, sample size s, parameters of the adjustment function (fr_1, l_0, l_1, m), window size w, pull request size ps, number of iterations n, waiting time wt
2 **for** $t \leftarrow 0$ ***to*** ∞ **do**
3 $C \leftarrow \{Score(B, \vec{x_j}) \mid j = t - w + 1, ..., t\}$;
4 $th \leftarrow Quantile(C, fr_1)$;
5 $\hat{y}_t \leftarrow Predict(B, \vec{x_t}, th)$;
6 $D \leftarrow Append(D, d_t)$;
7 **if** $t \mod ps = 0$ **then**
8 $T \leftarrow Labeled(D, wt)$;
9 $B \leftarrow Restart(B)$;
10 $(obf_0, obf_1) \leftarrow (1, 1)$;
11 **for** $i \leftarrow 1$ ***to*** n **do**
12 $S \leftarrow Weighted.Sample(T, obf_0, obf_1, s)$;
13 $B \leftarrow Update(B, S)$;
14 $ir_1 \leftarrow \frac{1}{w} \sum_{j=t-w+1}^{t} Predict(B, \vec{x_j}, 0.5)$;
15 $obf_0 \leftarrow OBF_0(ir_1, fr_1, l_0, m)$;
16 $obf_1 \leftarrow OBF_1(ir_1, fr_1, l_1, m)$;

as the fr_1-quantile of the scores C (line 4). And B and th are used to make the prediction on the current feature vector $\vec{x_t}$, and the result is recorded in $\hat{y}_t$ (line 5). It is important to add that the base learner B is also a classifier as defined by Eq. 3.5. So, $Predict(B, \vec{x}, th)$ can be implemented as the class $C(\vec{x})$ returned by B. Finally, the current instance d_t is appended to the historical data D (line 6), closing this block of steps.

The next block starts with the conditional statement (line 7) that waits the arrival of ps code changes between each training of the base learner B. When that happens, the training set T is created as the labeled instances in the historical data D (line 8). And the instances in D are labeled according to the following rules:

1. If the code change d_i fix the code change d_j, then d_j is labeled as defect-inducing: $d_j = (x_j, 1)$;
2. If the code change d_j is older than the waiting time wt, then d_j is labeled as clean: $d_j = (x_j, 0)$;
3. If the code change d_j is younger than the waiting time wt, then d_j is kept unlabeled and is not used for training.

After the training set has been created, the base learner B is restarted (line 9) and trained from scratch (lines 10 to 16). It is important to note that, even when associated with an online base learner, BORB does not allow incremental learning over the data stream due to that restart.

The oversampling boosting factors obf_0 and obf_1 are the variables that controls the emphasis given to each class (line 10). As they are initialized with the same value, no class is emphasized in the first iteration of the training. From the second iteration, obf_0 and obf_1 may be different, and the algorithm enters the general case, in which there is an emphasis on one of the classes.

The emphasis affects the generation of the sample S (line 12). S is generated from a weighted selection, with replacement, of the code changes in the training set T. The size of the sample S is constrained to be less than or equals to sample size s and less than or equals to the size of the training set T. In addition, the weighted selection considers that code changes from different classes have different probabilities of being selected according to the oversampling boosting factors obf_0 and obf_1. However, inside one specific class, the probability is the same for all code changes as defined by

$$P(T_c) = \frac{1}{|T_c|} \times \frac{obf_c}{obf_0 + obf_1}, \tag{4.1}$$

where T_c is the set of code changes of the class c, $c \, \epsilon \, \{0, 1\}$, and $T = T_0 \cup T_1$.

After the update of the base learner B (line 13), B yields the predictions on the most recent feature vectors x_j such as $j = t - w + 1, ..., t$. And the induced defect prediction rate ir_1 is calculated as the average of that predictions (line 14). It is important to note that, at this point, a threshold of 0.5 is used to make the predictions.

Then, the functions OBF_0 and OBF_1 are used to update, respectively, obf_0 and obf_1 (lines 15 to 16) in order to emphasize the class that is underrepresented in the predictions on the most recent feature vectors. The values of obf_0 and obf_1 are expected to change until ir_1 converges to fr_1. Regardless of that convergence, the training continues until iteration n.

The functions OBF_0 and OBF_1 are aimed at adjusting the emphasis given to each class during the training phase, and they are defined in Eq. 4.2 and Eq. 4.3. To illustrate their behavior, Fig. 2 shows how they change according to different values of the induced defect prediction rate ir_1 and m, while fr_1, l_0 and l_1 are constants. ir_1 is the induced defect prediction rate, m determines the growth of the exponential function, fr_1 stands for the fixed defect prediction rate, and l_0 and l_1 control the maximum boosting factor values (i.e., the boosting

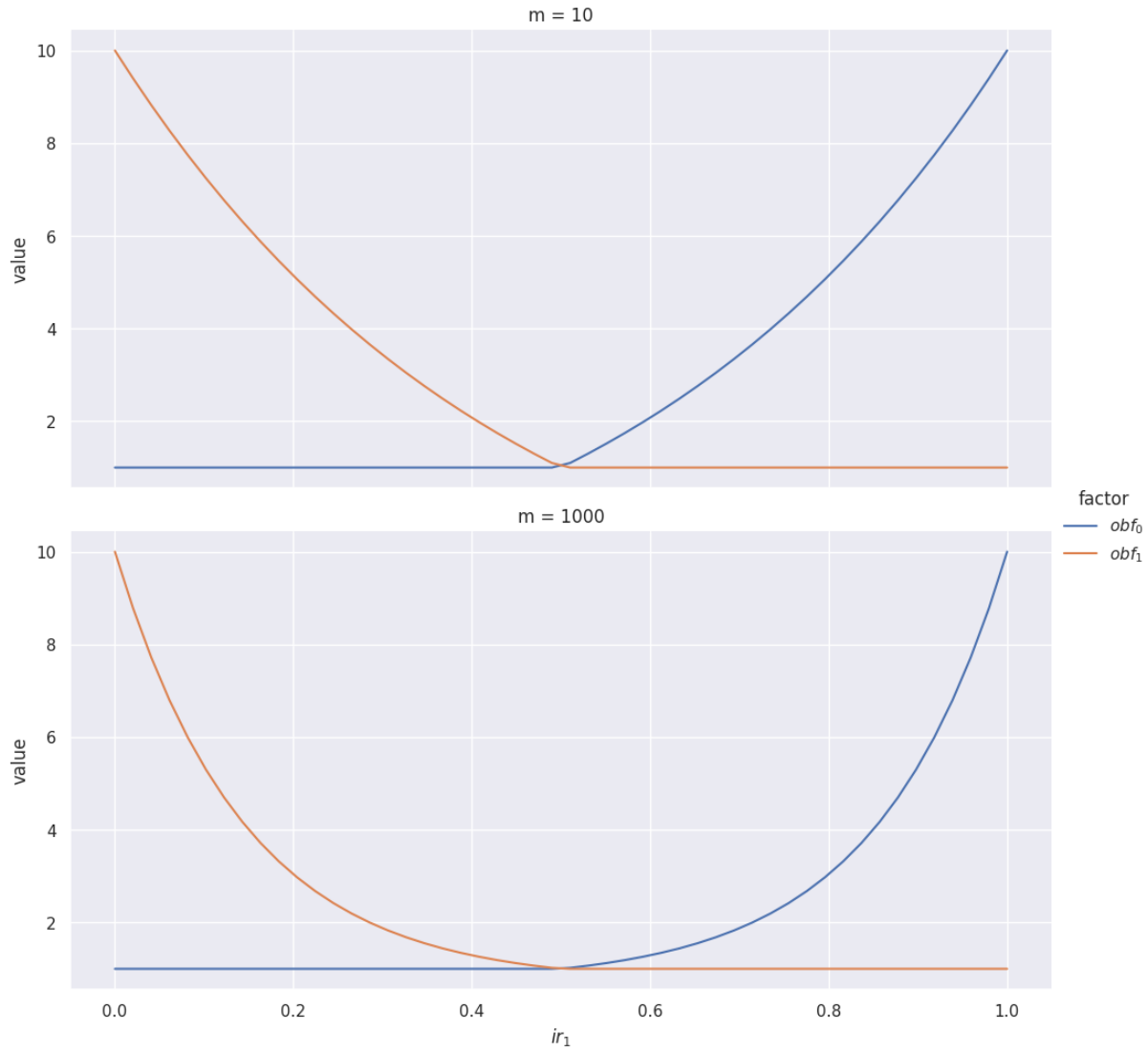
factors varies from 1 to $1 + l_0$ and $1 + l_1$). The class to emphasize is chosen according to the relationship between the induced defect prediction rate ir_1 and the fixed defect prediction rate fr_1. When $ir_1 > fr_1$, the emphasis is given to the clean class; when $ir_1 < fr_1$, the emphasis is given to the defect-inducing class; otherwise, there is no emphasis. It is important to note that as the induced defect prediction rate ir_1 approaches to 0 or 1, the boosting factors growth gets steeper, depending on the value of the parameter m. Cabral et al. (CABRAL et al., 2019) created the functions OBF_0 and OBF_1. In present work, the names of the parameters were changed to better fit in our context.

$$OBF_0(ir_1, fr_1, l_0, m) = \begin{cases} \left(\frac{m^{ir_1} - m^{fr_1}}{m - m^{fr_1}} \times l_0\right) + 1, & \text{if } ir_1 > fr_1 \\ 1, & \text{otherwise} \end{cases} \tag{4.2}$$

$$OBF_1(ir_1, fr_1, l_1, m) = \begin{cases} \left(\frac{m^{(fr_1 - ir_1)} - 1}{m^{fr_1} - 1} \times l_1\right) + 1, & \text{if } ir_1 \leq fr_1 \\ 1, & \text{otherwise} \end{cases} \tag{4.3}$$

Finally, it is important to say that the base learner associated with BORB have to conduct its training phase over many iterations (or epochs, in case of neural networks). Thus, after each iteration, BORB has the chance to adjust the oversampling boosting factors and take a rebalanced sample for the next iteration in order to make the induced defect prediction rate converges to the fixed defect prediction rate. This procedure is not supported by a base learner that does a single iteration training (e.g., Decision Tree and Random Forest). In this work, we used some off-the-shelf iterative base learners, and customized others to become iterative. More details about them are given in the Section 5.1.

Figure 2 – Oversampling boosting factors with $fr_1 = 0.5$, $l_0 = 9$, $l_1 = 9$ and different values of ir_1 and m.



Source: Created by the author (2021)

4.3 SUMMARY

In this chapter, we have seen the limitations in ORB that are related to its association with an online base learner that overemphasizes recent data and the susceptibility to noise of the instance-based oversampling. Then, BORB was presented as a solution to solve these problems while maintaining a fixed defect prediction rate over the data stream.

5 EXPERIMENTS

This chapter presents the base learners experimented in the proposed approach (BORB), the metrics used to evaluate the classifiers' predictive performances, the investigated datasets, the hyperparameter tuning procedure, the experimental setups used to answer the research questions and the related discussions.

5.1 BASE LEARNERS

As mentioned in Section 4.2, the base learner associated with BORB is required to perform the training over many iterations. So, algorithms such as Naive Bayes (NB), Logistic Regression (LR) and Multilayer Perceptron (MLP) are suitable options to associate with BORB. Besides that, we created two customized tree-based ensembles to match this training characteristic: Iterative Random Forest (IRF) and Iterative Hoeffding Forest (IHF). Three libraries were used to implement the base learners: Scikit-learn (PEDREGOSA et al., 2011) for NB, LR and IRF; Scikit-multiflow (MONTIEL et al., 2018) for IHF; and PyTorch (PASZKE et al., 2019) for MLP.

IRF consists of an ensemble of Decision Trees (DT) which implements the CART algorithm (BREIMAN et al., 1984) trained sequentially. It is similar to a Random Forest (RF) (BREIMAN, 2001). The main difference is that, instead of training all DTs at once, IRF grows one single tree when it receives the first training sample. Then, it grows the second tree when it receives the second sample, and so on. As a consequence, each DT may be grown using data with a different class imbalance since BORB changes the oversampling boosting factors (obf_0 and obf_1) across the training iterations, as explained in Section 4.2. This procedure results in an ensemble of DTs with different prior probabilities (DÍEZ-PASTOR et al., 2015). Therefore, IRF becomes a diversified ensemble while the model, as a whole, is induced to output a specific defect prediction rate. Finally, it is important to mention that this customization was necessary because, as we discussed in Section 4.2, RF fails to meet the iterative training requirement for being associated with BORB.

IHF is a bagging of Hoeffding Trees (HT) (DOMINGOS; HULTEN, 2000) trained sequentially. It is similar to IRF. The only difference is that we replace the DTs with HTs. In fact, IRF and IHF share the same pseudo-code for the update and the scoring procedures presented in, respectively, Algorithm 2 and Algorithm 3. As we can see in Algorithm 2, the tree T is trained

with the sample S_i and is recorded as E_i (line 2). Then, E_i is appended to the ensemble E as its i-th tree. And, as we can see in Algorithm 3, the score of the ensemble s is the average prediction of the trees in the ensemble E (line 2). Despite HTs already having an iterative training, IHF's customization was necessary to apply HTs in the same context as DTs and, as a consequence, to make a fair comparison between them.

Algorithm 2: Iterative Forest's update procedure

1 **Input**: Sample S_i of i-th update, ensemble of trees E, tree T
2 $E_i \leftarrow Train(T, S_i)$;
3 $E \leftarrow Append(E, E_i)$;

Algorithm 3: Iterative Forest's scoring procedure

1 **Input**: Ensemble of trees E, feature vector $\vec{x}$, threshold th
2 $s \leftarrow \frac{1}{|E|} \sum_{i=1}^{|E|} Predict(E_i, \vec{x}, th)$;

With respect to the baseline, we reimplemented ORB in our source code repository, and associated it with an online bagging (OZA, 2005) of Hoeffding Trees (OHT). We implemented OHT as a composite of components using the Scikit-multiflow (MONTIEL et al., 2018) similar to Cabral et al. (2019), which implemented using the framework MOA (BIFET et al., 2010). It is important to note that by reimplementing ORB in our source code repository, we ensure a fair comparison between ORB and BORB as they are associated with the same implementation of HT, and they are submitted to the same hyperparameter tuning procedure.

The complete list of classifiers (i.e., BORB and ORB's associations) evaluated in our experiments is as follows: (1) ORB-OHT, (2) BORB-IHF, (3) BORB-IRF, (4) BORB-LR, (5) BORB-MLP and (6) BORB-NB. And the source code used in this work is available at GitHub[1].

5.2 EVALUATION METRICS

As vastly discussed before, JIT-SDP is a class imbalanced problem. Therefore, the metrics used to compute the classifiers' performances must be sensitive to the class imbalance issue. As in Cabral et al. (CABRAL et al., 2019)'s work, we use the recalls of the clean and defect-inducing classes, the g-mean between these recalls and the distance between recalls. In addition, given that this work investigates the defect prediction rate, two more metrics were added to

[1] https://github.com/dinaldoap/jit-sdp-nn/tree/dissertation

evaluate BORB's capability to achieve the desired rate throughout the data stream and the base learners' capability to achieve the same rate in the training phase.

The complete list of evaluation metrics used in our experiments is as follows:

1. The g-mean, which is already defined by Eq. 3.11.

2. The recalls on the clean and defect-inducing classes, which are defined as

$$r_c(D, \vec{\theta}) = \frac{1}{|D|} \sum_{i=0}^{|D|} r_{c,i}(D_i, \vec{\theta}), \tag{5.1}$$

   where D is a segment of the data stream, $\vec{\theta}$ is the hyperparameters vector and $r_{c,i}$ is the prequential recall, defined by Equation 3.9.

3. The distance between recalls defined as

$$|r_0 - r_1|(D, \vec{\theta}) = \frac{1}{|D|} \sum_{i=0}^{|D|} |r_{0,i}(D_i, \vec{\theta}) - r_{1,i}(D_i, \vec{\theta})|, \tag{5.2}$$

   where D is a segment of the data stream, $\vec{\theta}$ is the hyperparameters vector, and $r_{0,i}$ and $r_{1,i}$ are the prequential recalls, defined by Equation 3.9.

4. The distance between the fixed defect prediction rate and the defect prediction rate, which is defined by Eq. 3.16.

5. And the distance between the fixed defect prediction rate and the induced defect prediction rate defined as

$$|fr_1 - ir_1|(D, \vec{\theta}) = \frac{1}{|D|} \sum_{i=0}^{|D|} |fr_1 - ir_{1,i}(D_i, \vec{\theta})|, \tag{5.3}$$

   where D is a segment of the data stream, $\vec{\theta}$ is the hyperparameters vector, fr_1 is the fixed defect prediction rate, and $ir_{1,i}$ corresponds to the induced defect prediction rate, defined in Algorithm 1, at the end of the training phase.

5.3 DATASETS

The datasets used in this work are the same used by Cabral et al. (CABRAL et al., 2019). They comprise characteristics extracted from commits by the tool Commit Guru (ROSEN; GRAWI; SHIHAB, 2015). Each instance includes 14 characteristics grouped in 5 dimensions: diffusion, size, purpose, history and experience. However, some of them are correlated (KAMEI et al.,

Table 2 – Information and statistics about the datasets.

Dataset	Code changes	Defect rate			Language
		Entire dataset	Validation segment	Testing segment	
brackets	17572	24%	30%	21%	JavaScript
broadleaf	15010	17%	23%	14%	Java
camel	30739	21%	42%	17%	Java
fabric8	13483	20%	21%	20%	Java
jgroups	18434	17%	23%	15%	Java
neutron	19689	24%	37%	19%	Python
nova	48989	25%	34%	24%	Python
npm	7920	18%	23%	09%	JavaScript
spring-integration	8750	27%	25%	29%	Java
tomcat	18960	28%	36%	24%	Java

Source: Created by the author (2021)

2016). So, we set up the base learners susceptible to multicollinearity with proper regularization: LR is regularized with elastic net (ZOU; HASTIE, 2005), and MLP is regularized with dropout (SRIVASTAVA et al., 2014). More details about the characteristics can be found in Kamei et al. (2016)'s work. The datasets used in this work are available at GitHub[2].

Table 2 shows some information and statistics about the datasets. As we can see, three programming languages were used to develop those software projects. The number of code changes ranges from 7920 to 48989. The defect rate on the entire dataset ranges from 17% to 28%. And, in some datasets, the difference between the defect rate in the validation segment and the defect rate in the testing segment is relatively large. This brings an additional challenge to, on the validation segment, tune a classifier that generalizes to the testing segment. Finally, the diversified characteristics of the datasets assure some level of generalization to the results obtained in this study, but they may not generalize to other software projects or to different problems than JIT-SDP.

5.4 HYPERPARAMETER TUNING

As mentioned in Subsection 3.1.4, in this work, the data streams are first split into two non-overlapping segments: V, that is used for hyperparameter tuning, and T, that is used for testing. Similarly to Cabral et al. (2019), we used 5000 code changes for hyperparameter tuning, but this amount could also be defined as a percentage multiplied by the dataset size.

[2] https://github.com/dinaldoap/jit-sdp-data/tree/dissertation

After splitting the data stream, the hyperparameter tuning procedure finds, within the specified configuration space, the hyperparameters vector $\vec{\theta}_v$ that maximizes the classifiers' performance on the segment V. The configuration space corresponds to the classifier's hyperparameters and their possible values. As our approach and the baseline approach have a high dimensional configuration space (i.e., many hyperparameters), we applied the random search (BERGSTRA; BENGIO, 2012) as hyperparameter tuning procedure. The random search implementation from Hyperopt (BERGSTRA et al., 2015) was adopted.

As recommended by Bergstra (2012), each combination of classifier and dataset was tuned independently. Specifically, we defined one configuration space for each classifier and that same configuration space is used by the random search over all datasets. However, since g-mean on the validation segment, defined in Subsection 3.1.4, is maximized separately for each dataset, we expect that the random search finds a different best configuration for each combination of classifier and dataset.

The configuration space of ORB-OHT was set based on the values used by Cabral et al. (2019), and the configuration space of the remaining classifiers were set based on preliminary experiments. It is important to add that BORB and ORB share some hyperparameters, and the configuration space related to these hyperparameters were set identical in order to produce a fair comparison. This concern was also applied to the configuration spaces of OHT and IHF since they share the hyperparameters related to the HTs.

The hyperparameter tuning procedure conducted in this work can be described as the following steps: First, 128 random configurations are evaluated with 3 repetitions for each combination of classifier and dataset (a distinct random seed is used for each repetition). Then, for each combination of classifier, dataset and configuration, the average g-mean of the 3 repetitions is calculated and the configuration with the maximum g-mean is chosen as the best. Finally, the best configuration can be used in the testing phase. The configuration space and the best configuration for each combination of classifier and dataset is described in Appendix B.

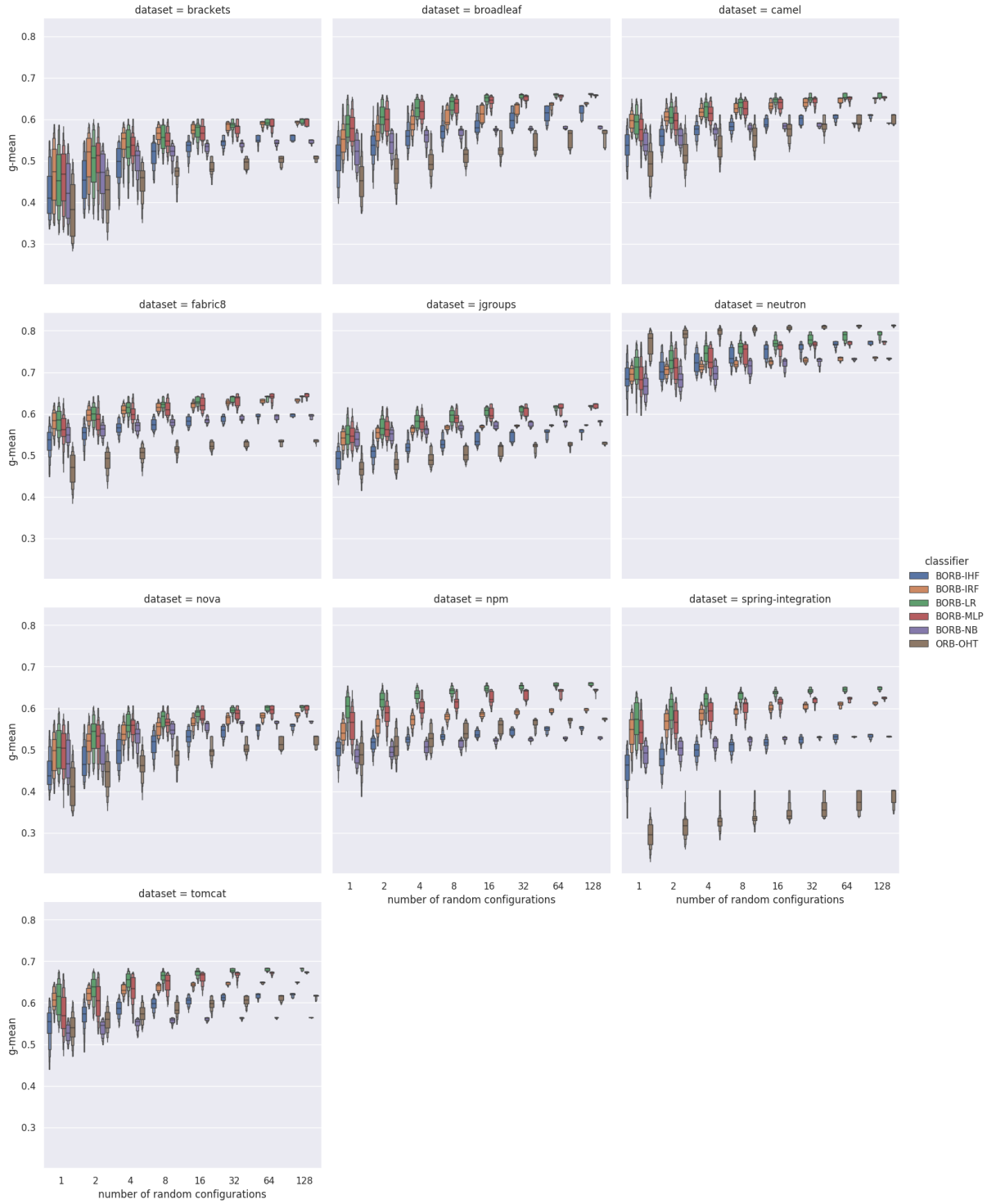
Besides tuning the classifiers, we assessed the effectivity of the procedure by estimating the empirical distribution of the maximum g-mean for each combination of classifier, dataset and number of random configurations evaluated. For example, assume one combination of classifier, dataset and 128 random configurations. So, from the 128 g-means of that combination, we take a sample, with replacement, of 128 g-means and record the maximum g-mean. We repeat that resampling 1000 times in total. After that, we have the data of the empirical distribution

of the maximum g-mean for 128 random configurations. Then, we do the same procedure for the remaining combinations of classifier, dataset and number of random configurations.

After estimating the empirical distributions, we created a visualization with letter-values plots (HOFMANN; WICKHAM; KAFADAR, 2017) to analyze how the distributions change when the number of random configurations is increased. Fig. 3 shows them grouped by dataset. With that visualization, we can compare how well each classifier performed in terms of g-mean convergence on the validation segment of each dataset. The more skewed is the distribution towards the top g-mean the better is the performance of the classifier for the specified number of random configurations. As can be observed, ORB-OHT presents relatively bad convergence on broadleaf, nova and spring-integration datasets. BORB-NB and BORB-LR also presents relatively bad convergence on, respectively, broadleaf and neutron. This suggests that the configuration space may not be well set for those combinations of classifier and dataset. In other words, there is a large region in the configuration space that results in poor g-means. So, the probability of the classifier achieving poor g-means with all random configurations is high. That implicates on a broader distribution of the maximum g-mean even when using 128 random configurations. The remaining combinations of classifier and dataset presented relatively good convergence.

It is important to mention that, despite Fig. 3 being similar to Bergstra (2012)'s efficiency curves, we applied a different approach in present work. As defined in Eq. 3.12, we assume a certainty about the best configuration to use on testing data being the one with the maximum g-mean on the validation data, while Bergstra (2012) takes into account any uncertainty in the choice of which configuration is actually the best-performing one.

Figure 3 – G-mean convergence on the validation segment of the data stream.



Source: Created by the author (2021)

5.5 TRAINING DATA

As mentioned in Section 3.1.4, our procedure finds the classifiers' configuration that maximizes the g-mean on the validation segment of the data stream and uses the same configuration on the testing segment. And, regardless of the segment, when the classifier is requested to make a prediction for a code change, it can use all previous code changes for training, including both segments. For example, the classifier can make a prediction for a code change in the testing segment after learning from the previous code changes of the testing segment and from the whole validation segment, since the validation segment comes before the testing one.

Notice that since ORB and BORB are based on, respectively, online and batch algorithms, there are some differences between them with respect to consuming training data. A ORB's classifier learns from training data incrementally, starting from the beginning of the validation segment until right before the code change being tested. On the other hand, a BORB's classifier accumulates training data to use them as a whole on each periodic training, as described in Section 4.2. So, for a BORB's classifier, the available training data tends to increase indefinitely over the lifecycle of the software.

The impacts of the increasing number of code changes in the training data are addressed by subsampling. More specifically, during the training phase, a BORB's classifier learns from random samples of the training data. The sample size is a hyperparameter of BORB set within the range from 1000 to 4000. The maximum value of this range was chosen according to the dataset with the greatest number of code changes available for training during hyperparameter tuning. As defined in Section 5.4, the total number of code changes in the validation segment is 5000, but, in the presence of verification latency, the amount available for training is reduced to approximately 4000.

Another aspect of setting the sample size according to the training data available in the validation segment is that the hyperparameter tuning procedure tends to configure the model to have lower capacity than it is appropriate for the amount of training data available in the testing phase. This decreases the risk of overfitting during the testing phase, but it increases the risk of underfitting. We accepted the latter risk and, as a consequence, no additional mechanism to avoid overfitting during the testing phase was required for any BORB's classifier. For example, early stopping (PRECHELT, 2012; BENGIO, 2012) was not applied to MLP, and pruning (DUROUX; SCORNET, 2016; SCORNET, 2017) was not applied to IRF.

5.6 EXPERIMENTAL SETUPS

Research Question 1 (RQ1)

This setup requires tuning and testing ORB-OHT over all datasets to analyze how robust are the results achieved by that classifier in terms of rank based on the g-mean and rank based on the distance between recalls. As mentioned in Section 5.4, the hyperparameter tuning finds the best configuration for each combination of classifier and dataset. Then, the testing executes each best configuration on its respective dataset, 30 times, using distinct seeds. The last step is to average the metrics by dataset. It is important to say that the data recorded by this experimental setup is also used by some following setups.

Research Question 2 (RQ2)

This setup requires data from the setup for RQ1, and additional tuning and testing for BORB-IHF over all datasets. In this setup, we isolate the comparison between BORB and ORB as both BORB-IHF and ORB-OHT are HT-based. In fact, among the proposed classifiers, BORB-IHF is the most similar to ORB-OHT. The statistical test of the comparison of this research question is part of a multiple comparison that is described in the setup for RQ3.

Research Question 3 (RQ3)

This setup requires data from the setup for RQ1, data from the setup for RQ2, and additional tuning and testing of BORB-LR, BORB-MLP, BORB-NB and BORB-IRF over all datasets. In this setup, we compare the mentioned classifiers against the baseline. It is important to note that the setups from RQ1 to RQ3 have the same procedure and metrics. So, we show the results of all the classifiers in the same table, visualizations and statistical tests in the beginning of Section 5.7. On the other hand, the discussions of the results are segregated following the research questions structure. The statistical tests used for multiple comparisons against one baseline over multiple datasets are the Friedman test and the Bonferroni-Dunn post-hoc test as described by Demsar (2006).

Research Question 4 (RQ4)

The setup for RQ2 and the setup for RQ3 also record data related to the distance between the fixed defect prediction rate and the defect prediction rate, and the distance between the fixed defect prediction rate and the induced defect prediction rate for all combinations of BORB's classifier and dataset, but not for ORB as it does not use a fixed defect prediction rate. With that data, we evaluate the first hypothesis (Eq. 3.17) and the second hypothesis (Eq. 3.18). But, first, we set the maximum acceptable distance between the fixed defect prediction rate and the defect prediction rate as $d = 0.05$, and a moderate negative correlation as $c \in [0.4, 0.6]$. The first hypothesis is evaluated using Wilcoxon signed-rank test (WILCOXON, 1945), and the second hypothesis is evaluated using Spearman's rank correlation coefficient (SPEARMAN, 1987).

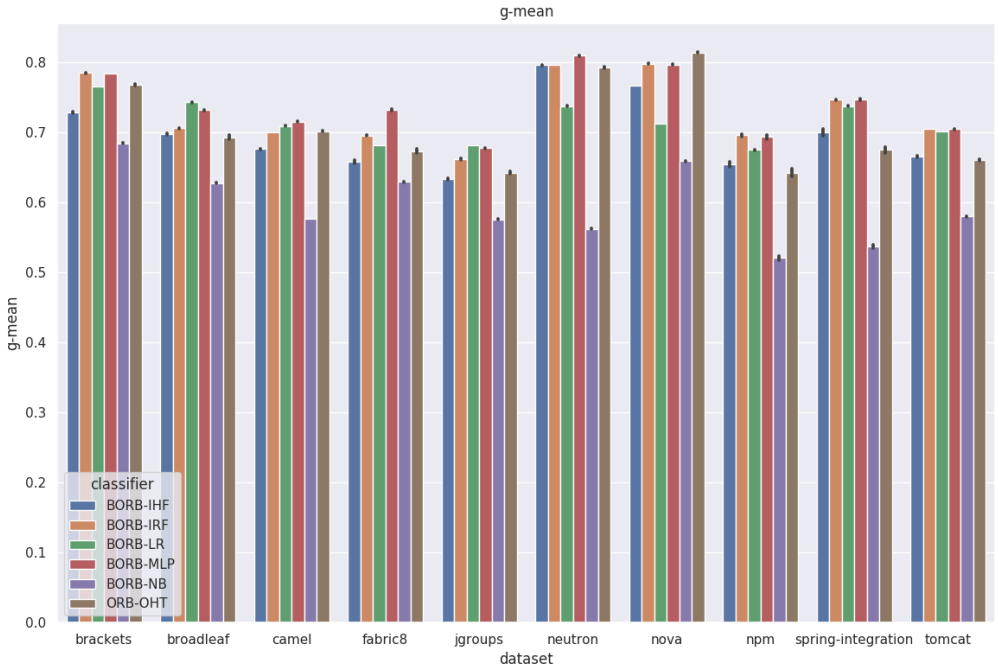
After the hypotheses, we evaluate which base learner is best suited to output a fixed defect prediction rate doing multiple comparisons over multiple datasets using Friedman test and Nemenyi post-hoc test as described by Demsar (2006), based on the distance between the fixed defect prediction rate and the induced defect prediction, defined by Eq. 5.3. In contrast to the g-mean, lower distances indicate better results.

5.7 RESULTS AND DISCUSSIONS

Before discussing the research questions, we present the results of the experiments of the first three research questions. Fig. 4 shows the average and the standard deviation of the g-mean by dataset and classifier, in which higher g-means indicate better performances. Fig. 5 shows the average ranks based on the g-mean, in which smaller average ranks indicate better performances, and Bonferroni-Dunn critical distance. It is important to mention that we rank the classifiers for each dataset separately, where the best classifier gets the rank of 1, the second best rank 2, and so on. And the classifier's average rank corresponds to the mean of ranks of the specified classifier over all datasets (DEMSAR, 2006). Fig. 6 shows the heatmap of ranks based on the g-mean.

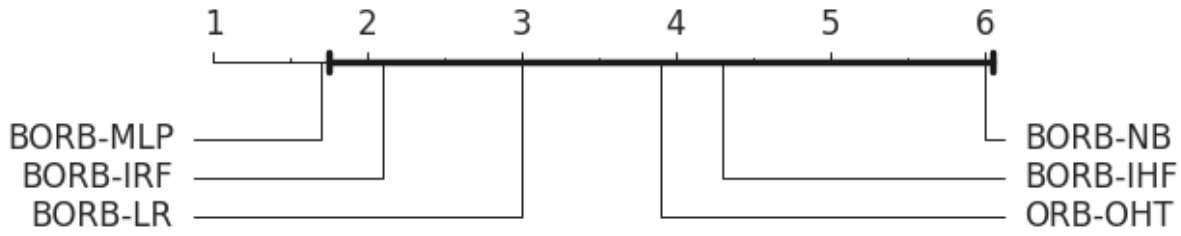
Fig. 7 shows the average and the standard deviation of the distance between recalls by dataset and classifier, in which smaller distances indicate more effective class imbalance treatments. Fig. 8 shows the average ranks based on the distance between recalls, in which smaller ranks indicates better performances, Bonferroni-Dunn critical distance. And Fig. 9 shows the

Figure 4 – Average and standard deviation (black vertical bar) of the g-mean by dataset and classifier.



Source: Created by the author (2021)

Figure 5 – Average ranks based on the g-mean and Bonferroni-Dunn critical distance (bold horizontal bar).



Source: Created by the author (2021)

heatmap of ranks based on the distance between recalls.

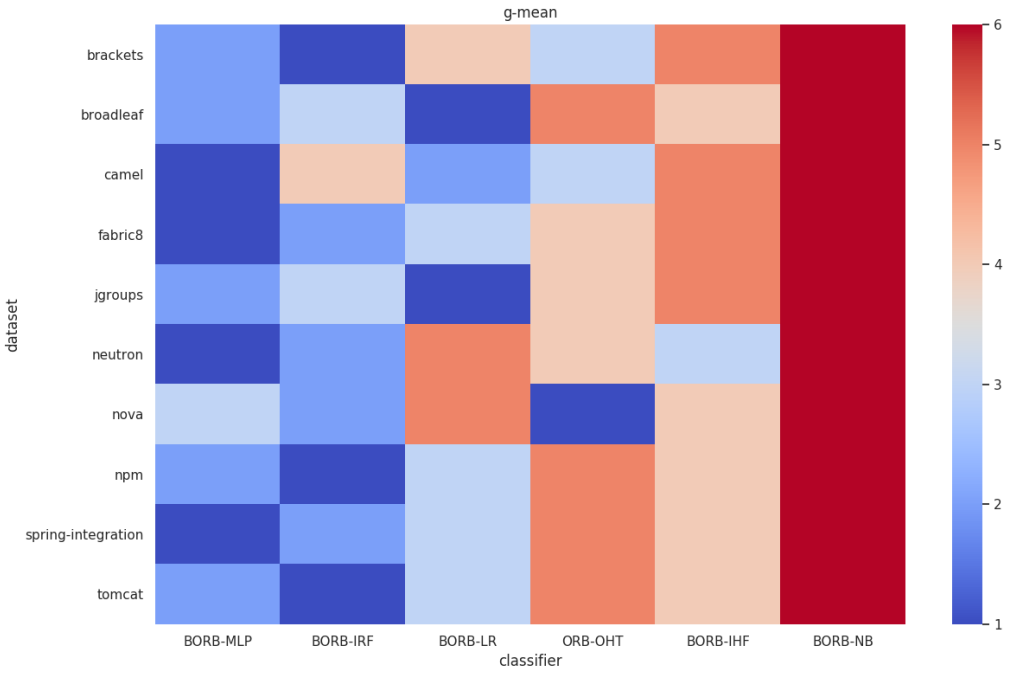
Table 3 shows the results of all combinations of dataset, classifier and metric. Besides that, we have also added, in Appendix A, the results of the experiments when using the same methodology as Cabral et al. (2019). This enabled us to validate our reimplementation of ORB-OHT, and to confirm that our hyperparameter tuning procedure is not hindering the performance of ORB-OHT when compared to the procedure used by Cabral et al. (2019). More details about that methodology are also given in Appendix A.

Table 3 – Results of all combinations of classifier and dataset for all metrics.

Dataset	Classifier	r_0 (std)	r_1 (std)	$\|r_0 - r_1\|$ (std)	g-mean (std)	$\|fr_1 - ir_1\|$ (std)	$\|fr_1 - pr_1\|$ (std)
brackets	BORB-IHF	60.52% (00.61%)	88.20% (00.52%)	27.68% (00.89%)	72.86% (00.39%)	05.72% (00.55%)	04.52% (00.43%)
	BORB-IRF	77.72% (00.08%)	79.55% (00.19%)	06.87% (00.21%)	78.51% (00.08%)	02.18% (00.05%)	03.18% (00.05%)
	BORB-LR	65.77% (00.06%)	89.48% (00.10%)	23.72% (00.12%)	76.61% (00.05%)	03.67% (00.04%)	02.90% (00.02%)
	BORB-MLP	74.41% (00.12%)	82.84% (00.25%)	09.70% (00.22%)	78.41% (00.14%)	04.63% (00.23%)	02.86% (00.05%)
	BORB-NB	60.14% (00.14%)	78.14% (00.20%)	18.20% (00.23%)	68.46% (00.12%)	14.38% (00.16%)	03.10% (00.06%)
	ORB-OHT	75.91% (00.59%)	78.04% (00.68%)	05.97% (00.71%)	76.87% (00.38%)	—	—
broadleaf	BORB-IHF	73.90% (00.23%)	66.21% (00.46%)	09.16% (00.38%)	69.77% (00.30%)	06.63% (00.33%)	02.33% (00.08%)
	BORB-IRF	58.19% (00.19%)	86.22% (00.35%)	28.32% (00.35%)	70.62% (00.20%)	13.61% (00.20%)	03.56% (00.12%)
	BORB-LR	70.33% (00.09%)	78.96% (00.22%)	11.88% (00.24%)	74.32% (00.11%)	06.87% (00.08%)	02.81% (00.03%)
	BORB-MLP	65.01% (00.21%)	82.91% (00.28%)	19.32% (00.34%)	73.20% (00.19%)	06.45% (00.13%)	03.61% (00.12%)
	BORB-NB	66.16% (00.13%)	59.79% (00.61%)	07.53% (00.34%)	62.80% (00.34%)	30.35% (01.16%)	02.20% (00.05%)
	ORB-OHT	74.63% (01.11%)	64.76% (01.20%)	11.86% (01.18%)	69.33% (00.78%)	—	—
camel	BORB-IHF	74.20% (00.12%)	61.87% (00.36%)	12.68% (00.27%)	67.66% (00.22%)	07.09% (00.21%)	02.36% (00.04%)
	BORB-IRF	59.13% (00.10%)	83.26% (00.20%)	24.21% (00.20%)	70.03% (00.11%)	10.24% (00.16%)	03.47% (00.05%)
	BORB-LR	61.47% (00.05%)	82.19% (00.08%)	20.79% (00.09%)	70.94% (00.05%)	09.22% (00.02%)	03.46% (00.02%)
	BORB-MLP	65.44% (00.09%)	78.63% (00.21%)	13.90% (00.20%)	71.57% (00.10%)	05.58% (00.08%)	03.73% (00.06%)
	BORB-NB	64.39% (00.06%)	52.06% (00.15%)	13.63% (00.14%)	57.69% (00.09%)	19.49% (00.14%)	04.07% (00.04%)
	ORB-OHT	66.77% (00.28%)	74.44% (00.46%)	11.37% (00.45%)	70.20% (00.26%)	—	—
fabric8	BORB-IHF	67.73% (00.59%)	64.30% (01.11%)	07.40% (00.58%)	65.82% (00.63%)	09.66% (01.10%)	04.25% (00.20%)
	BORB-IRF	73.25% (00.16%)	66.38% (00.29%)	08.87% (00.24%)	69.54% (00.20%)	02.83% (00.10%)	03.89% (00.10%)
	BORB-LR	61.28% (00.07%)	76.11% (00.11%)	15.21% (00.10%)	68.15% (00.07%)	04.26% (00.05%)	03.64% (00.04%)
	BORB-MLP	71.95% (00.22%)	74.79% (00.53%)	07.01% (00.35%)	73.23% (00.30%)	06.81% (00.34%)	03.73% (00.11%)
	BORB-NB	65.21% (00.12%)	61.26% (00.18%)	09.35% (00.17%)	62.96% (00.12%)	20.01% (00.42%)	05.12% (00.06%)
	ORB-OHT	76.61% (00.77%)	59.99% (01.11%)	18.03% (01.02%)	67.32% (00.87%)	—	—
jgroups	BORB-IHF	71.94% (00.24%)	56.01% (00.57%)	16.58% (00.61%)	63.33% (00.33%)	04.19% (00.31%)	02.97% (00.10%)
	BORB-IRF	70.44% (00.14%)	62.48% (00.44%)	09.60% (00.36%)	66.19% (00.26%)	01.45% (00.08%)	04.26% (00.04%)
	BORB-LR	68.75% (00.12%)	67.83% (00.22%)	05.90% (00.15%)	68.16% (00.11%)	03.27% (00.11%)	03.58% (00.04%)
	BORB-MLP	67.58% (00.15%)	68.19% (00.44%)	05.85% (00.21%)	67.75% (00.23%)	04.95% (00.24%)	03.83% (00.07%)
	BORB-NB	70.47% (00.16%)	47.33% (00.30%)	23.29% (00.32%)	57.60% (00.20%)	34.23% (01.70%)	04.00% (00.09%)
	ORB-OHT	70.27% (00.46%)	59.32% (00.92%)	13.67% (00.90%)	64.27% (00.46%)	—	—
neutron	BORB-IHF	70.60% (00.16%)	90.55% (00.33%)	20.00% (00.39%)	79.64% (00.15%)	03.65% (00.14%)	04.41% (00.07%)
	BORB-IRF	67.74% (00.07%)	94.84% (00.12%)	27.11% (00.15%)	79.69% (00.06%)	03.03% (00.05%)	05.33% (00.03%)
	BORB-LR	57.68% (00.12%)	96.74% (00.06%)	39.06% (00.15%)	73.76% (00.10%)	05.49% (00.02%)	09.05% (00.11%)
	BORB-MLP	69.37% (00.19%)	95.40% (00.22%)	26.03% (00.24%)	81.00% (00.18%)	02.89% (00.08%)	04.47% (00.14%)
	BORB-NB	40.88% (00.21%)	81.19% (00.12%)	40.33% (00.21%)	56.21% (00.17%)	49.25% (00.20%)	21.96% (00.15%)
	ORB-OHT	78.46% (00.98%)	80.55% (01.50%)	08.96% (01.31%)	79.31% (00.50%)	—	—
nova	BORB-IHF	79.02% (00.09%)	75.82% (00.20%)	14.37% (00.20%)	76.68% (00.13%)	10.08% (00.13%)	04.59% (00.04%)
	BORB-IRF	83.40% (00.05%)	77.61% (00.12%)	13.96% (00.12%)	79.84% (00.06%)	09.24% (00.08%)	03.73% (00.02%)
	BORB-LR	57.46% (00.07%)	93.60% (00.04%)	36.40% (00.08%)	71.24% (00.06%)	05.64% (00.01%)	11.80% (00.06%)
	BORB-MLP	74.75% (00.17%)	86.91% (00.08%)	17.34% (00.17%)	79.72% (00.15%)	05.72% (00.08%)	05.45% (00.13%)
	BORB-NB	53.11% (00.15%)	86.39% (00.11%)	33.96% (00.13%)	65.91% (00.14%)	18.01% (00.48%)	13.51% (00.10%)
	ORB-OHT	79.50% (00.30%)	83.81% (00.51%)	10.13% (00.47%)	81.42% (00.25%)	—	—
npm	BORB-IHF	58.41% (00.52%)	73.68% (01.93%)	15.80% (01.78%)	65.43% (00.98%)	05.63% (00.79%)	03.34% (00.23%)
	BORB-IRF	63.15% (00.31%)	77.01% (01.24%)	14.03% (01.34%)	69.63% (00.53%)	03.40% (00.54%)	03.25% (00.14%)
	BORB-LR	57.59% (00.18%)	79.64% (00.12%)	22.54% (00.18%)	67.54% (00.12%)	04.03% (00.09%)	03.79% (00.05%)
	BORB-MLP	66.00% (00.46%)	73.31% (01.55%)	09.25% (01.03%)	69.38% (00.80%)	05.75% (00.64%)	03.46% (00.26%)
	BORB-NB	67.43% (00.45%)	40.78% (01.32%)	26.79% (01.48%)	52.09% (00.79%)	33.78% (01.22%)	05.89% (00.34%)
	ORB-OHT	69.44% (01.42%)	60.54% (02.86%)	13.53% (02.58%)	64.28% (01.57%)	—	—
spring-integration	BORB-IHF	66.12% (02.18%)	75.71% (00.94%)	14.90% (02.37%)	70.03% (01.42%)	08.02% (00.62%)	06.22% (01.69%)
	BORB-IRF	72.23% (00.20%)	77.98% (00.31%)	11.12% (00.22%)	74.72% (00.17%)	09.50% (00.21%)	03.03% (00.09%)
	BORB-LR	67.32% (00.11%)	81.59% (00.14%)	15.84% (00.13%)	73.79% (00.09%)	09.34% (00.07%)	04.15% (00.07%)
	BORB-MLP	77.07% (00.31%)	73.09% (00.58%)	10.96% (00.43%)	74.75% (00.36%)	06.51% (00.18%)	04.14% (00.14%)
	BORB-NB	51.97% (00.60%)	60.77% (00.58%)	25.27% (00.48%)	53.72% (00.55%)	36.15% (00.79%)	16.06% (00.31%)
	ORB-OHT	66.77% (00.94%)	71.01% (02.44%)	21.43% (02.08%)	67.53% (01.38%)	—	—
tomcat	BORB-IHF	63.83% (00.36%)	69.58% (00.47%)	07.38% (00.48%)	66.54% (00.27%)	07.39% (00.21%)	03.23% (00.15%)
	BORB-IRF	70.54% (00.12%)	70.67% (00.21%)	05.94% (00.16%)	70.51% (00.12%)	02.89% (00.06%)	03.21% (00.04%)
	BORB-LR	67.56% (00.06%)	73.09% (00.10%)	07.32% (00.07%)	70.17% (00.05%)	00.61% (00.04%)	02.65% (00.02%)
	BORB-MLP	70.98% (00.19%)	70.14% (00.32%)	05.89% (00.19%)	70.46% (00.21%)	07.14% (00.45%)	03.50% (00.08%)
	BORB-NB	64.68% (00.15%)	52.34% (00.34%)	13.22% (00.29%)	58.00% (00.20%)	55.38% (00.46%)	03.74% (00.05%)
	ORB-OHT	64.87% (00.58%)	67.88% (00.53%)	08.66% (00.54%)	66.05% (00.40%)	—	—

Source: Created by the author (2021)

Figure 6 – Heatmap of ranks based on the g-mean, horizontally sorted by average rank.



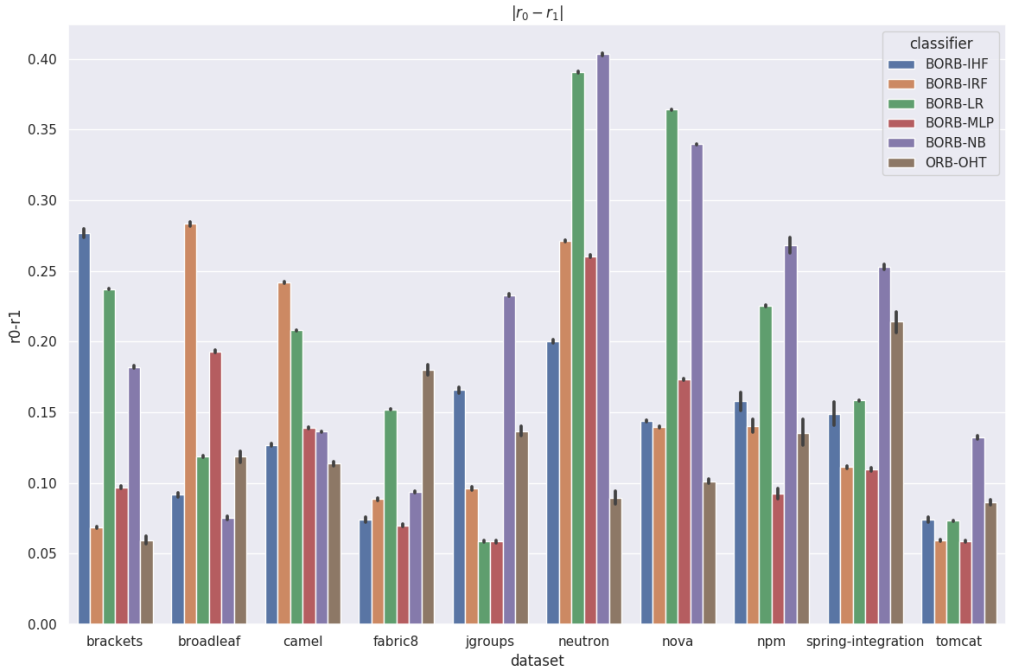
Source: Created by the author (2021)

RQ1's discussion

As we can see in Fig. 6, in terms of rank based on the g-mean, ORB-OHT performs below the median on the datasets broadleaf, fabric8, jgroups, neutron, npm, spring-integration and tomcat (i.e., 7 out of 10 datasets). On the other hand, as we can see in Fig. 9, in terms of rank based on the distance between recalls, ORB-OHT performs below the median on fabric8, jgroups, spring-integration and tomcat (i.e., 4 out of 10 datasets).

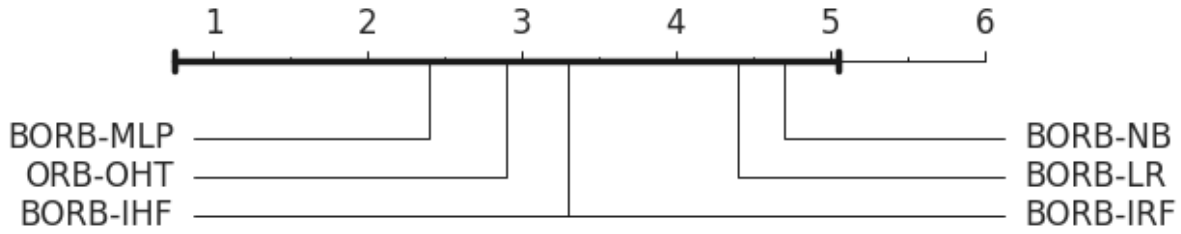
It was mentioned in Section 4.1 that ORB-OHT's predictive performance may be hindered by its association with an online base learner that overemphasizes recent data and the susceptibility to noise of the instance-based oversampling. To answer that, we need to use data from RQ2's experiments to compare the results of ORB-OHT and BORB-IHF since BORB-IHF was designed to solve the mentioned problems of ORB-OHT while maintains an ensemble of HTs as base learner. BORB-IHF avoid overemphasizing recent data and decrease the susceptibility to noise by resampling historical data in a batch mode. However, the results shows that the replacement of ORB-OHT with BORB-IHF is not enough to improve the g-mean, as can be seen in . So, we can not assign the poor g-mean of ORB-OHT on some datasets

Figure 7 – Average and standard deviation (black vertical bar) of the distance between recalls by dataset and classifier.



Source: Created by the author (2021)

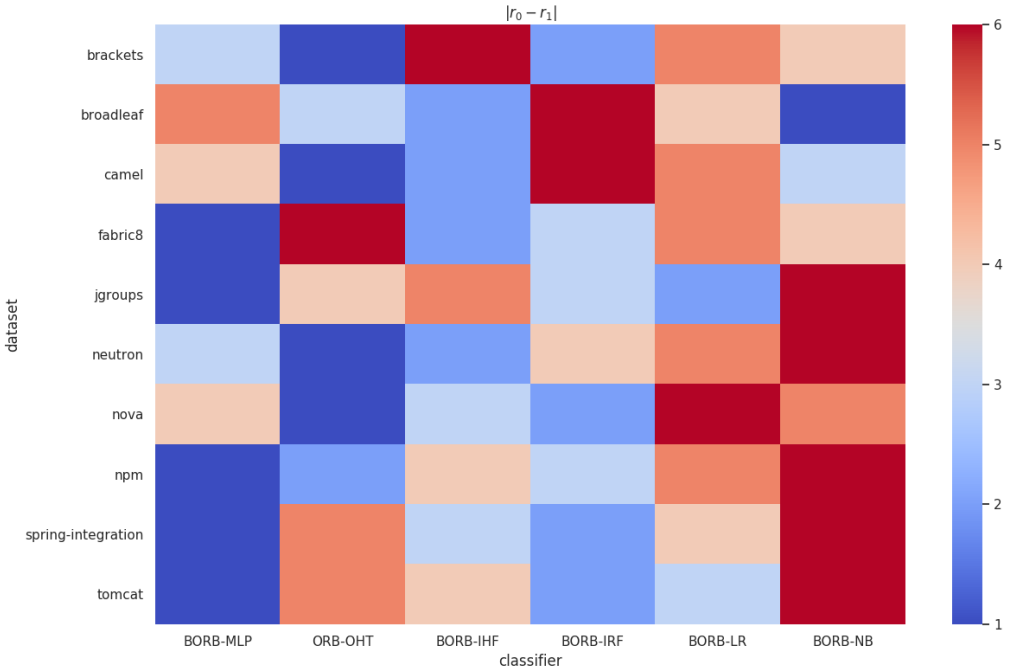
Figure 8 – Average ranks based on the distance between recalls and Bonferroni-Dunn critical distance (bold horizontal bar).



Source: Created by the author (2021)

to the mentioned problems. In fact, the results shows that the base learner is a key factor which determines the g-mean of BORB since only BORB-MLP achieved an improvement with statistical significance, as can be seen in . Associating a larger amount of similar base learners with both BORB and ORB would allow a better comparison between them. However, this subject remained for future work.

Figure 9 – Heatmap of ranks based on the distance between recalls, horizontally sorted by average rank.



Source: Created by the author (2021)

RQ2's discussion

As shown in Fig. 5, BORB-IHF and ORB-OHT perform similarly in terms of average rank based on the g-mean. However, as shown in Fig. 6, in terms of rank by dataset based on the g-mean, they are placed in different sides of the median rank on brackets, camel, neutron and nova (i.e., 4 out of 10 datasets). BORB-IHF changes the g-mean in comparison to the baseline in the following descending order: $+4\%$ on spring-integration, $+2\%$ on npm, $+1\%$ on tomcat, $+1\%$ on broadleaf, $+0\%$ on neutron, -1% on jgroups, -2% on fabric8, -4% on camel, -5% on brackets and -6% on nova. The Friedman test applied to the multiple comparison of BORB's classifiers against ORB-OHT rejects the null hypothesis with $p\text{-}value = 1.08 \times 10^{-6}$. However, as can be seen in Fig. 5, the Bonferroni-Dunn test does not present significant difference between the average rank of BORB-IHF and the average rank of ORB-OHT based on the g-mean. In fact, both are close to the 4^{th} place in Fig. 5.

Despite BORB being designed to use historical data for resampling, it takes samples from the training set that are constrained by the sample size s, as defined in Algorithm 1, to alleviate the computational burden. In the case of BORB-IHF, it means s instances for each

HT training. On the other hand, ORB-OHT does not rely on this constraint. It accumulates the statistics from all instances on its HTs since the beginning of the data stream. In other words, the HTs of ORB-OHT have access to an unlimited amount of data while the HTs of BORB-IHF are constrained by the sample size s, which is set in the hyperparameter tuning procedure within a range from 1000 to 4000.

The fact that BORB-IHF reaches a similar g-mean to ORB-OHT using fewer data suggests that, when properly sampled, smaller portions of the historical data can provide proper information to the classifier. So, a more powerful classifier should be able to improve the g-mean when associated with BORB. In fact, we can use data from RQ3's experiments to compare BORB-IHF and BORB-IRF to observe the evolution of the g-mean when we maintain BORB and replace the ensemble of HTs with an ensemble of DTs. So, in our experiments, the DTs are able to capture more signal from the sample than the HTs.

As shown in Fig. 8, in terms of average rank based on the distance between recalls, BORB-IHF and ORB-OHT perform similarly. However, as shown in Fig. 9, in terms of rank by dataset based on the distance between recalls, BORB-IHF and ORB-OHT are placed in different sides of the median rank on brackets, fabric8, neutron and spring-integration (i.e., 4 out of 10 datasets). Fig. 8 shows that the Bonferroni-Dunn test does not present significant difference between the average rank of BORB-IHF and the average rank of ORB-OHT based on the distance between recalls. In fact, both are close to the 3^{th} place in Fig. 8.

Furthermore, according to Table 3, it is possible to note that neither BORB-IHF nor ORB-OHT are skewed towards a specific class. In fact, BORB-IHF and ORB-OHT achieved better recall for the defect-inducing class r_1 when compared to the recall for the clean class r_0 on, respectively, 5 and 6 of the 10 datasets.

RQ3's discussion

As we can see in Fig. 4 and in Fig. 6, BORB-IRF and BORB-MLP are able to improve the g-mean in comparison to ORB-OHT on most of the datasets. These improvements correspond to increases that range from $+2\%$ to $+11\%$. Both classifiers have a decrease in the g-mean of -2% in only one dataset. BORB-MLP changes the g-mean in comparison to the baseline in the following descending order: $+11\%$ on spring-integration, $+9\%$ on fabric8, $+8\%$ on npm, $+7\%$ on tomcat, $+6\%$ on broadleaf, $+5\%$ on jgroups, $+2\%$ on neutron, $+2\%$ on brackets, $+2\%$ on camel and -2% on nova. BORB-IRF changes the g-mean in comparison to the baseline in

the following descending order: $+11\%$ on spring-integration, $+8\%$ on npm, $+7\%$ on tomcat, $+3\%$ on fabric8, $+3\%$ on jgroups, $+2\%$ on brackets, $+2\%$ on broadleaf, $+0\%$ on neutron, $+0\%$ on camel and -2% on nova.

Despite the improvements on the g-mean achieved by BORB-IRF, the Bonferroni-Dunn test does not present significant difference between the average ranks of BORB-IRF and ORB-OHT, as can be seen in Fig. 5. The reason is that Bonferroni-Dunn has a low power (DEMSAR, 2006). So, the addition of more datasets in future work is recommended to avoid the risk of type 2 error on that statistical test. Only BORB-MLP presents an improvement with statistical significance (i.e., $p\text{-}value < 0.05$) when compared to the baseline, as can be seen in Fig. 5. So, to better understand the results of our best classifier, Fig. 10 shows the prequential g-mean, r_0, r_1 and $|r_0 - r_1|$ of BORB-MLP paired with ORB-OHT over the data streams.

BORB-LR is also able to improve the g-mean on most of the datasets when compared to ORB-OHT, as we can see in Fig. 4 and in Fig. 6. However, it performs highly worse than ORB-OHT on neutron and nova, datasets on which ORB-OHT achieves its best results in terms of g-mean. BORB-LR changes the g-mean in comparison to the baseline in the following descending order: $+9\%$ on spring-integration, $+7\%$ on broadleaf, $+6\%$ on tomcat, $+6\%$ on jgroups, $+5\%$ on npm, $+1\%$ on fabric8, $+1\%$ on camel, $+0\%$ on brackets, -7% on neutron and -13% on nova.

Finally, BORB-NB performs much worse than ORB-OHT over all datasets. This suggests that BORB-NB is not adequate to the datasets used in this work. According to Rish (2001), NB is expected to better perform with completely independent features and functionally dependent features (i.e., deterministic dependencies), while it reaches the worst performance between these extremes. Thus, non-deterministic dependencies between the features may be hindering the predictive performance of BORB-NB since the investigated datasets present correlated features, as we mentioned in Section 5.3.

Two evidences can be analyzed to understand the cause of the poor g-mean for BORB-LR on neutron and nova. First, as we can see in Fig. 7 and in Table 3, BORB-LR overemphasized the minority class on that datasets, as r_1 is typically much larger than r_0. Second, as we can observe in Table 2 and in Fig. 11, there is a decreasing trend in the defect rate between the validation segment and the testing segment of that data streams. So, poor ranked scores in the beginning of the data stream added to a higher defect rate on the validation segment may have led BORB-LR to be tuned with a fixed defect prediction rate that overemphasized the minority class on the testing segment. However, it is important to note that BORB-IRF and

BORB-MLP were not affected by that and achieved more robust results.

Another thing to keep in mind while analyzing the results is that BORB's classifiers output a fixed defect prediction rate over the data stream. If a dataset presents periods where the defect rate abruptly changes to a value distant from the fixed defect prediction rate, the classifier has more chances of performing poorly on those periods. For instance, during a refactoring period, fewer defects are introduced in the software. So, when consecutive code changes with low scores are tested, the classifier decreases its threshold so that it outputs the fixed defect prediction rate on that period. Then, due to the low threshold, the following code changes with middle scores are classified as defect-inducing until the classifier increases its threshold. The threshold is increased after the classifier updates the sliding window of scores, as described in Algorithm 1. The opposite happens when consecutive code changes with high scores are tested. To illustrate that, Fig. 11 shows that BORB-LR was tuned with higher fixed defect prediction rates fr_1 than BORB-MLP on neutron and nova. And, on that datasets, BORB-LR has more sudden drops in the g-mean and in the recall of the clean class r_0 than BORB-MLP over the oscillations and the decreasing trend of the defect rate.

As shown in Fig. 7, in terms of distance between recalls, at least one of the classifiers performs relatively different on each dataset. Besides that, the ranks by dataset based on the distance between recalls (Fig. 9) looks more random when compared to the ranks by dataset based on the g-mean (Fig. 6). The greater level of randomness in the former ranks fits the result of the respective statistical test. Fig. 8 shows that the Bonferroni-Dunn test does not present significant difference between the average ranks of all pairs of classifiers based on the distance between recalls. That suggests that both BORB and ORB share a similar capability to balance the recalls, regardless the base learner that BORB is associated with.

In addition, Table 3 shows some emphasis from most of the batch classifiers on the recall of the defect-inducing class r_1. In fact, BORB-IRF achieved better results on the recall of the defect-inducing class r_1 when compared to recall of the clean class r_0 on 7 of the 10 datasets, BORB-LR achieved the same on 9 of the 10, BORB-MLP achieved the same on 8 of the 10, and BORB-NB achieved the same on 4 of the 10. So, BORB-NB was the only batch classifier to emphasize to r_0.

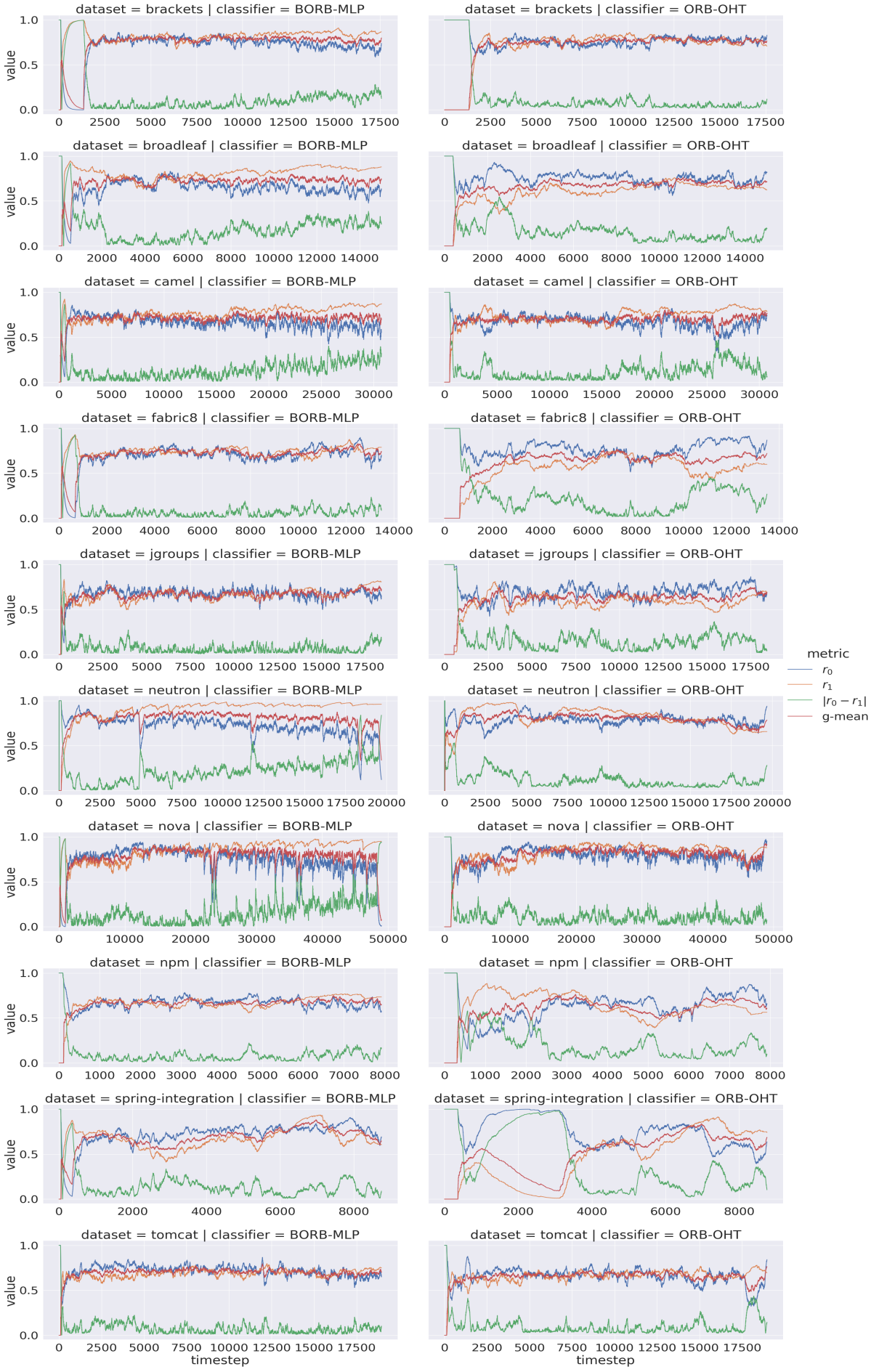
RQ4's discussion

The Wilcoxon signed rank test rejects the null hypothesis defined by Eq. 3.17 with $p\text{-}value = 2.5 \times 10^{-4}$. So, our conclusion is that BORB is able to output a fixed defect prediction rate. More specifically, BORB is able to maintain the distance between the fixed defect prediction rate and the defect prediction rate below 0.05 with statistical significance (i.e., $p\text{-}value < 0.05$).

The Spearman's rank correlation coefficient defined by Eq. 3.18 is -0.44 with $p\text{-}value = 1.3 \times 10^{-3}$. That means that the correlation between the g-mean and the distance between the fixed defect prediction rate and the induced defect prediction rate is moderate. A moderate correlation shows that we can maximize the g-mean assuming a fixed defect prediction rate, but a better capability to output the fixed defect prediction is not always rewarded with an improvement in the g-mean, and vice versa. Besides that, we know in advance that some drawbacks of the fixed defect prediction rate may compromise further improvements in other parts of our approach. More details about this and future work are given in Section 6.1.

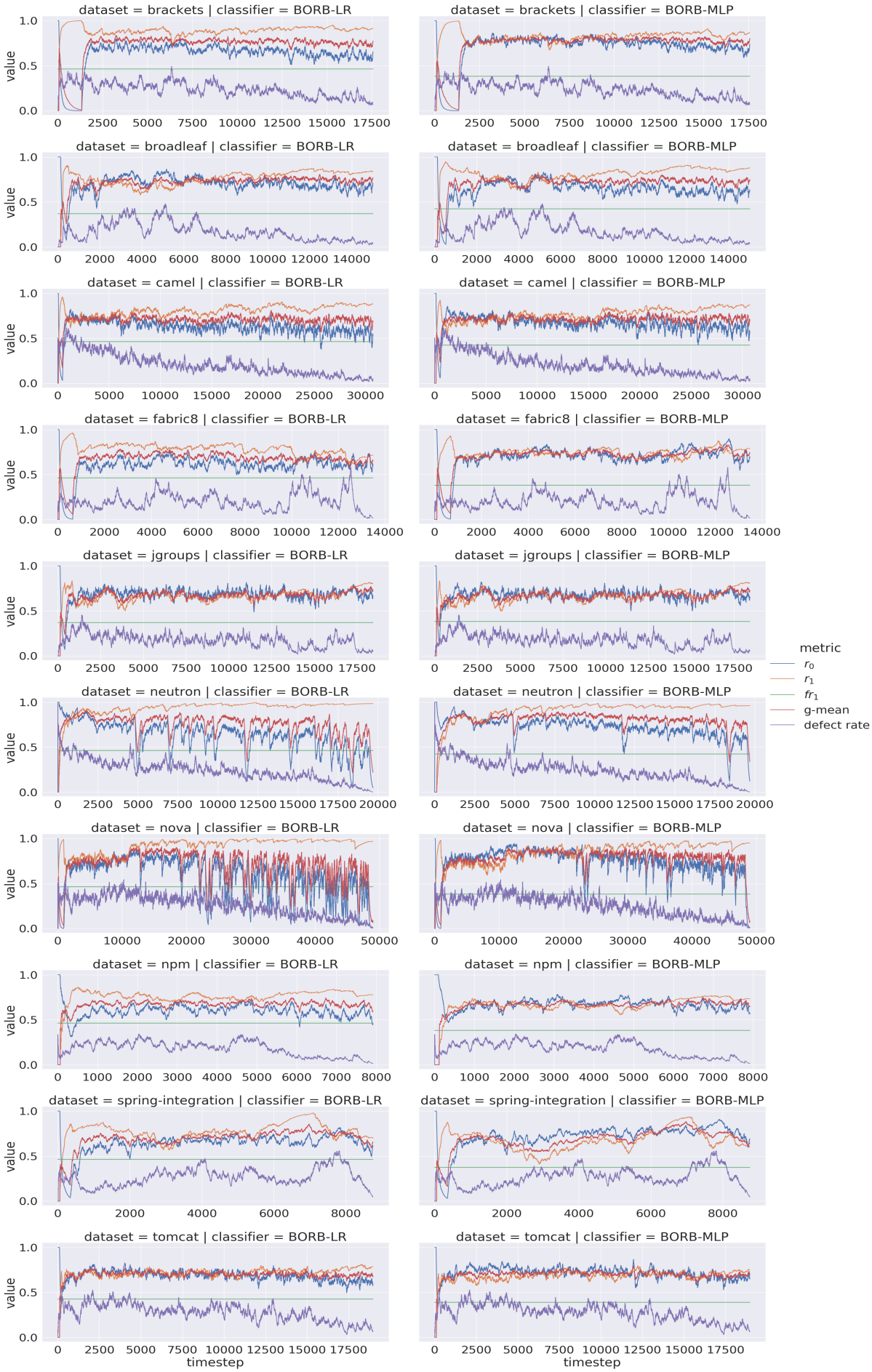
Analyzing the multiple pairwise comparison between the base learners in terms of the distance between the fixed defect prediction rate and the induced defect prediction rate, we can find out which base learner is best suited to output a fixed defect prediction rate. Fig. 12 and Fig. 14 shows that BORB-NB achieves the worst results for all datasets, and the other classifiers achieve relatively good results. The Friedman test rejects the null hypothesis with $p\text{-}value = 2.1 \times 10^{-21}$, confirming that there is a difference with statistical significance (i.e., $p\text{-}value < 0.05$) between some pair of classifiers. As we can observe in Fig. 13, the Nemenyi post-hoc test confirms that BORB-IRF, BORB-LR and BORB-MLP have a difference with statistical significance (i.e., $p\text{-}value < 0.05$) from BORB-NB, and BORB-IHF does not have a difference with statistical significance from the other two groups.

Figure 10 – G-mean, r_0, r_1 and $|r_0 - r_1|$ of BORB-MLP and ORB-OHT over the data streams.



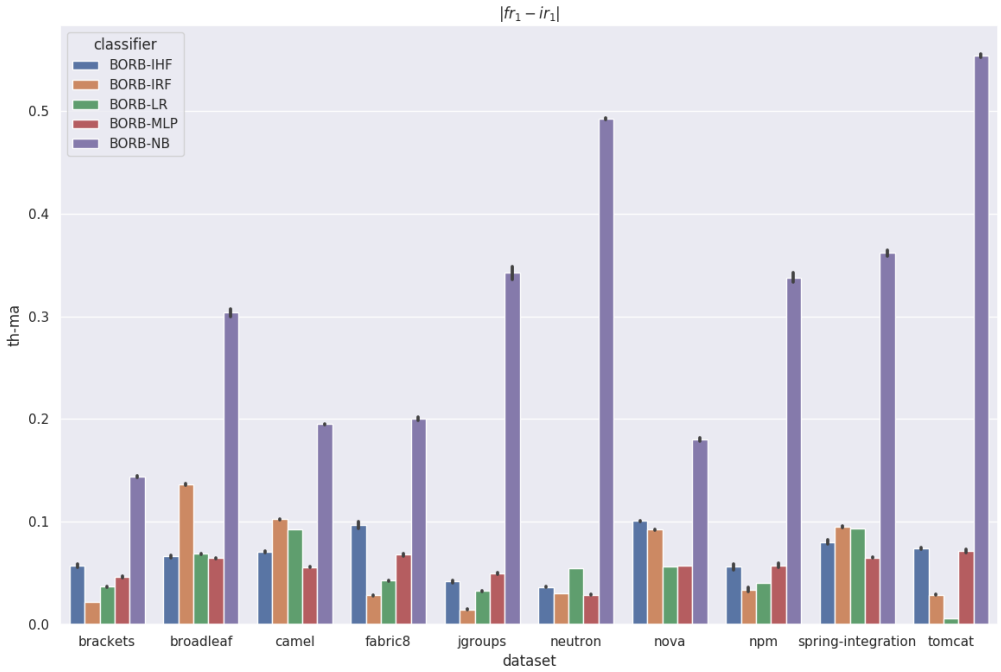
Source: Created by the author (2021)

Figure 11 – Relationship between the fixed defect prediction rate (fr_1), the defect rate and sudden drops in g-mean and r_0 of BORB-LR and BORB-MLP over the data streams.



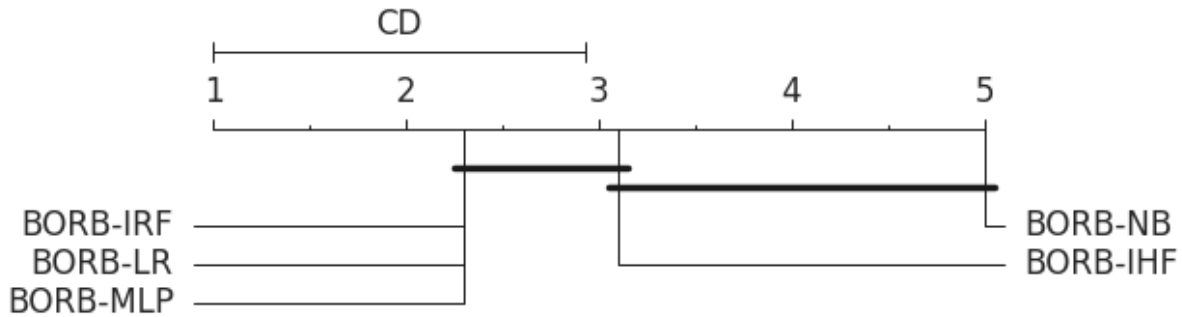
Source: Created by the author (2021)

Figure 12 – Average and standard deviation (black vertical bar) of the distance between the fixed defect prediction rate and the induced defect prediction rate.



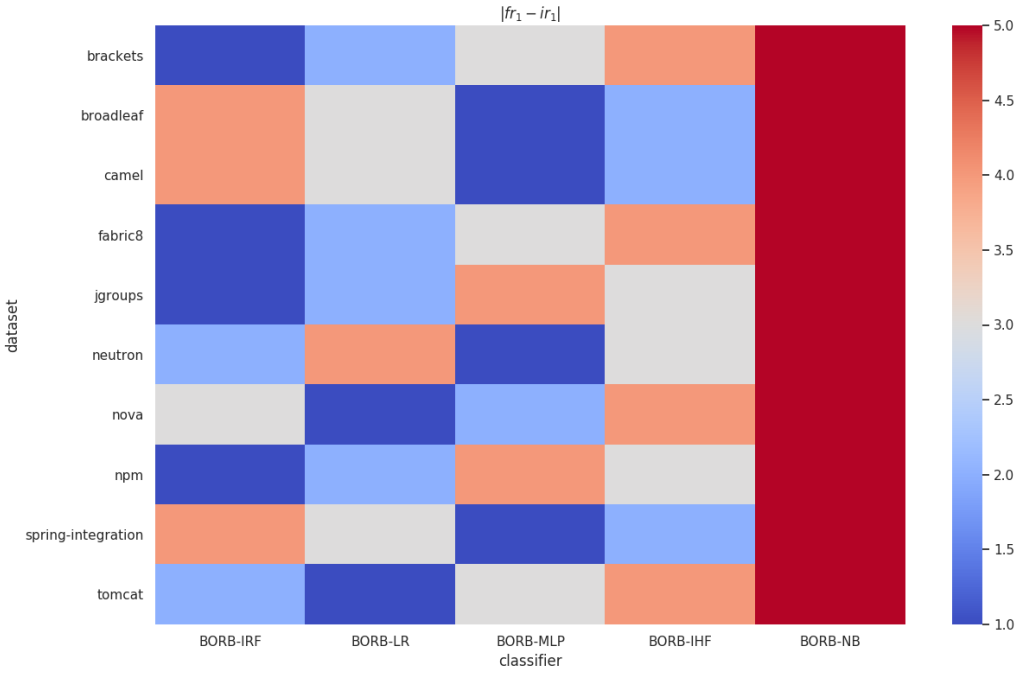
Source: Created by the author (2021)

Figure 13 – Average rank based on the distance between the fixed defect prediction rate and the induced defect prediction rate and Nemenyi critical distance (CD).



Source: Created by the author (2021)

Figure 14 – Heatmap of ranks based on the distance between the fixed defect prediction rate and the induced defect prediction rate, horizontally sorted by average rank.



Source: Created by the author (2021)

5.8 SUMMARY

In this chapter, we have presented the experiments carried out to answer the research questions defined in Section 1.4. The results showed that BORB, when associated with robust batch base learners, manages to consistently improve the g-mean. For instance, BORB-MLP was the best classifier and achieved improvements between $+2\%$ and $+11\%$ on 9 of the 10 investigated datasets, in terms of g-mean, and a decrease of -2% in only one dataset. Besides that, we have confirmed our two hypotheses defined by Eq. 3.17 and by Eq. 3.18. That means, respectively, that BORB is able to output a fixed defect prediction rate; and the better the model is to output the fixed defect prediction rate, the better is its predictive performance in terms of g-mean. Additionally, we have seen that, of the evaluated classifiers, BORB-IRF, BORB-LR and BORB-MLP are the best suited to output a fixed defect prediction rate.

6 CONCLUSIONS

In this work, we have investigated batch algorithms using a fixed defect prediction rate that maximizes the g-mean in the presence of verification latency and class imbalance evolution in online JIT-SDP. Our objective was to investigate the predictive performance of batch algorithms compared to online algorithms while testing two hypotheses about the fixed defect prediction rate. The first hypothesis (3.17) tests whether our approach (BORB) is able to output a fixed defect prediction rate, and the second one (3.18) evaluates the correlation between the capability to output the fixed defect prediction and the predictive performance. To do that investigations, four research questions were defined in Section 1.4. Then, four batch classifiers (BORB-IRF, BORB-LR, BORB-MLP and BORB-NB), one online classifier (ORB-OHT), and one hybrid classifier (BORB-IHF) were set up in our experiments to answer the research questions. And the results led us to set the following answers:

RQ1 Why does ORB perform poorly in terms of predictive performance on some datasets? **Answer**: ORB-OHT performs below the median on 7 of the 10 datasets in terms of rank based on the g-mean. However, the results do not support assigning the poor performance on some datasets to the known limitations in ORB related to its association with an online base learner that overemphasizes recent data and the susceptibility to noise of the instance-based oversampling. In fact, BORB-IHF was designed to solve these limitations, but it did not achieve an improvement with statistical significance (i.e., $p\text{-}value < 0.05$) in terms of g-mean. A larger amount of similar base learners needs to be associated with both BORB and ORB to improve the comparison between them.

RQ2 How to modify ORB to use historical data for resampling and output a fixed defect prediction rate? How well BORB performs compared to ORB in terms of predictive performance? **Answer**: BORB can be seen as a ORB modified to avoid overemphasizing recent data and decrease the susceptibility to noise by resampling historical data in a batch mode. When both BORB and ORB are associated with HTs to create, respectively, BORB-IHF and ORB-OHT, their performances in terms of g-mean are similar. BORB-IHF wins on half of the datasets and ORB-OHT wins on the other half. However, we must note that BORB-IHF achieves a similar predictive performance using fewer data than ORB-OHT. This suggests that, when properly sampled, even small fractions of

the historical data can provide enough information so that the base learner captures the most relevant concepts.

RQ3 How does BORB perform when using different base learners? In particular, do the use of different base learners further improves the predictive performance of BORB? **Answer**: There is an improvement in the predictive performance from BORB-IHF to BORB-IRF. This improvement suggests that DTs are better suited than HTs to capture the most relevant concepts from small portions of the historical data on the investigated datasets. BORB-MLP and BORB-LR also improves the predictive performance in terms of the average rank based on the g-mean, but only BORB-MLP achieved an improvement with statistically significance (i.e., $p\text{-}value < 0.05$). On the other hand, BORB-NB achieved poor relative results on all datasets, which suggests that NB is not adequate to the characteristics of the datasets used in this work. BORB-MLP wins on 9 of the 10 datasets, BORB-IRF wins on 8 of the 10, BORB-LR wins on 7 of the 10, and BORB-NB loses on 10 of the 10 when compared to ORB-OHT

RQ4 Is BORB able to output a fixed defect prediction rate? How correlated are the capability to output a fixed defect prediction rate and the predictive performance? And how adequate is each base learner to output a fixed defect prediction rate? **Answer**: BORB was capable of maintaining the distance between the fixed defect prediction rate and the defect prediction rate below 0.05 with statistical significance (i.e., $p\text{-}value < 0.05$). So, we can conclude that BORB is able to output a fixed defect prediction rate. Additionally, the correlation between the g-mean and the distance between the fixed defect prediction rate and the induced defect prediction rate is -0.44. So, we can conclude that there is a moderate negative correlation between them. In other words, assuming that the classifier is constrained to output a fixed defect prediction rate, it achieves a better performance in testing when it is more capable to output the fixed defect prediction set in the hyperparameter tuning. Finally, the average rank based on the distance between the fixed defect prediction rate and the induced defect prediction rate shows us that BORB-IRF, BORB-LR and BORB-MLP are equally adequate to output a fixed defect prediction rate as they share the first position in the ranking, BORB-IHF comes next in the fourth position, and BORB-NB comes in the last position, reflecting its poor relative results on all datasets.

6.1 FUTURE WORK

There are three main issues to be addressed by future work. They are related to the experimental setup and the creation of new approaches to tackle class imbalance evolution, as follows:

- A larger amount of similar base learners needs to be associated with both BORB and ORB to improve the comparison between them and better evaluate the known limitations in ORB.
- The addition of more datasets is necessary so that low power statistical tests for multiple pairwise comparison, such as Nemenyi and Bonferroni-Dunn tests, do not result on type 2 error. For instance, BORB-IRF presented great improvements when compared to ORB-OHT, but that was not reflected on the statistical test.
- The fixed defect prediction rate is not suited to tackle trends and high volatility in the class imbalance evolution, but that concept drifts happen in some datasets used in this work. So, there is room for improvements in the predictive performance by replacing the fixed defect prediction rate with a more flexible mechanism. For example, the mechanism could start with a defect prediction rate in the beginning of the software project and change this rate over time, according to the expected trend for the defect rate.

Due to its simplicity, the fixed defect prediction can be considered as a standard baseline to the problem of class imbalance evolution. Thus, we recommend implementing the fixed defect prediction rate in the baseline of future work related to class imbalance evolution.

REFERENCES

AKIYAMA, F. An Example of Software System Debugging. In: FREIMAN, C. V.; GRIFFITH, J. E.; ROSENFELD, J. L. (Ed.). *IFIP Congress*. North-Holland, 1971. v. 1, p. 353–359. ISBN 0-7204-2063-6. Available at: <http://dblp.uni-trier.de/db/conf/ifip/ifip71-1.html{#}Akiyama71>.

BASILI, V. R.; BRIAND, L. C.; MELO, W. L. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, v. 22, n. 10, p. 751–761, 1996. ISSN 00985589.

BENGIO, Y. Practical Recommendations for Gradient-Based Training of Deep Architectures. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2nd. ed. Springer-Verlag, 2012. v. 7700, p. 437–478. ISBN 9783642352881. Available at: <http://deeplearning.net/software/pylearn2http://link.springer.com/10.1007/978-3-642-35289-8{_}26>.

BERGSTRA, J.; BENGIO, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, v. 13, n. 10, p. 281–305, 2012. ISSN 15324435. Available at: <http://jmlr.org/papers/v13/bergstra12a.html>.

BERGSTRA, J.; KOMER, B.; ELIASMITH, C.; YAMINS, D.; COX, D. D. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, IOP Publishing, v. 8, n. 1, p. 14008, 2015. Available at: <https://doi.org/10.1088/1749-4699/8/1/014008>.

BIFET, A.; HOLMES, G.; PFAHRINGER, B.; KRANEN, P.; KREMER, H.; JANSEN, T.; SEIDL, T. MOA: Massive Online Analysis, a Framework for Stream Classification and Clustering. *Journal of Machine Learning Research (JMLR) Workshop and Conference Proceedings*, PMLR, v. 11, p. 44–50, 2010. ISSN 1938-7228. Available at: <http://proceedings.mlr.press/v11/bifet10a.html>.

BIRD, C.; NAGAPPAN, N.; MURPHY, B.; GALL, H.; DEVANBU, P. Don't touch my code! Examining the effects of ownership on software quality. In: *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, New York, USA: ACM Press, 2011. p. 4–14. ISBN 9781450304436.

BREIMAN, L. Random forests. *Machine Learning*, Springer, v. 45, n. 1, p. 5–32, 2001. ISSN 08856125. Available at: <https://link.springer.com/article/10.1023/A:1010933404324>.

BREIMAN, L.; FRIEDMAN, J. H.; OLSHEN, R. A.; STONE, C. J. *Classification and regression trees*. 1st. ed. [S.l.]: CRC Press, 1984. 358 p. ISBN 9781351460491.

CABRAL, G. G.; MINKU, L. L.; SHIHAB, E.; MUJAHID, S. Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction. In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 2019. p. 666–676. ISBN 9781728108698. ISSN 02705257. Available at: <https://research.birmingham.ac.uk/portal/en/publications/class-imbalance-evolution-and-verification-latency-in-justintime-software-defect-prediction(9ea03c93-276
.html>.

CHACON, S.; STRAUB, B. *Pro Git*. 2nd. ed. [S.l.]: Apress, 2014. 458 p.

CHEN, X.; ZHAO, Y.; WANG, Q.; YUAN, Z. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology*, Elsevier B.V., v. 93, p. 1–13, 2018. ISSN 09505849.

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, v. 20, n. 6, p. 476–493, 1994. ISSN 00985589.

D'AMBROS, M.; LANZA, M.; ROBBES, R. An extensive comparison of bug prediction approaches. In: *Proceedings - International Conference on Software Engineering*. [S.l.: s.n.], 2010. p. 31–41. ISBN 9781424468034. ISSN 02705257.

DEMSAR, J. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, v. 7, p. 1–30, 2006. ISSN 15337928. Available at: <http://www.jmlr.org/papers/v7/demsar06a.html>.

DÍEZ-PASTOR, J. F.; RODRÍGUEZ, J. J.; GARCÍA-OSORIO, C.; KUNCHEVA, L. I. Random Balance: Ensembles of variable priors classifiers for imbalanced data. *Knowledge-Based Systems*, Elsevier, v. 85, p. 96–111, 2015. ISSN 09507051.

DOMINGOS, P.; HULTEN, G. Mining high-speed data streams. In: *Proceeding of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, New York, USA: ACM Press, 2000. p. 71–80. ISBN 1581132336. Available at: <http://portal.acm.org/citation.cfm?doid=347090.347107>.

DUROUX, R.; SCORNET, E. *Impact of subsampling and pruning on random forests*. 2016. Available at: <http://arxiv.org/abs/1603.04261>.

EYOLFSON, J.; TAN, L.; LAM, P. Do time of day and developer experience affect commit bugginess. In: *Proceedings - International Conference on Software Engineering*. New York, New York, USA: ACM Press, 2011. p. 153–162. ISBN 9781450305747. ISSN 02705257. Available at: <http://portal.acm.org/citation.cfm?doid=1985441.1985464>.

FAWCETT, T. An introduction to ROC analysis. *Pattern Recognition Letters*, North-Holland, v. 27, n. 8, p. 861–874, 2006. ISSN 01678655.

FERRI, C.; HERNÁNDEZ-ORALLO, J.; FLACH, P. Setting decision thresholds when operating conditions are uncertain. *Data Mining and Knowledge Discovery*, Springer New York LLC, v. 33, n. 4, p. 805–847, 2019. ISSN 1573756X.

GAMA, J.; FERNANDES, R.; ROCHA, R. Decision trees for mining data streams. *Intelligent Data Analysis*, IOS Press, v. 10, n. 1, p. 23–45, 2006. ISSN 1088467X. Available at: <https://dl.acm.org/citation.cfm?id=1239079>.

GAMA, J.; ZLIOBAITE, I.; BIFET, A.; PECHENIZKIY, M.; BOUCHACHIA, A. A survey on concept drift adaptation. *ACM Computing Surveys*, Association for Computing Machinery, v. 46, n. 4, p. 1–37, 2014. ISSN 15577341.

GOUSIOS, G.; PINZGER, M.; DEURSEN, A. V. An exploratory study of the pull-based software development model. In: *Proceedings - International Conference on Software Engineering*. New York, New York, USA: IEEE Computer Society, 2014. p. 345–355. ISSN 02705257. Available at: <http://dl.acm.org/citation.cfm?doid=2568225.2568260>.

HALSTEAD, M. H. *Elements of software science library*. [S.l.]: Elsevier, 1977. 127 p. ISBN 0-444-00205-7, 0-444-00215-4.

HASSAN, A. E. Predicting faults using the complexity of code changes. In: *Proceedings - International Conference on Software Engineering*. IEEE, 2009. p. 78–88. ISBN 9781424434527. ISSN 02705257. Available at: <http://ieeexplore.ieee.org/document/5070510/>.

HERNÁNDEZ-ORALLO, J.; FLACH, P.; FERRI, C. A unified view of performance metrics: Translating threshold choice into expected classification loss. *Journal of Machine Learning Research*, v. 13, p. 2813–2869, 2012. ISSN 15324435.

HOFMANN, H.; WICKHAM, H.; KAFADAR, K. Letter-Value Plots: Boxplots for Large Data. *Journal of Computational and Graphical Statistics*, American Statistical Association, v. 26, n. 3, p. 469–477, 2017. ISSN 15372715.

JONES, C.; BONSIGNOUR, O. *The Economics of Software Quality*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2011. 624 p.

KAMEI, Y.; FUKUSHIMA, T.; MCINTOSH, S.; YAMASHITA, K.; UBAYASHI, N.; HASSAN, A. E. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, Springer US, v. 21, n. 5, p. 2072–2106, 2016. ISSN 15737616. Available at: <http://link.springer.com/10.1007/s10664-015-9400-x>.

KAMEI, Y.; SHIHAB, E.; ADAMS, B.; HASSAN, A. E.; MOCKUS, A.; SINHA, A.; UBAYASHI, N. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, v. 39, n. 6, p. 757–773, 2013. ISSN 00985589.

KIM, S.; WHITEHEAD, E. J.; ZHANG, Y. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, v. 34, n. 2, p. 181–196, 2008. ISSN 00985589.

KIM, S.; ZIMMERMANN, T.; WHITEHEAD, E. J.; ZELLER, A. Predicting faults from cached history. In: *Proceedings - International Conference on Software Engineering*. IEEE, 2007. p. 489–498. ISBN 0769528287. ISSN 02705257. Available at: <http://ieeexplore.ieee.org/document/4222610/>.

KRASNER, H. *The Cost of Poor Software Quality in the US: A 2020 Report*. [S.l.], 2020. 44 p. Available at: <https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report.htm>.

LI, Z.; JING, X. Y.; ZHU, X. Progress on approaches to software defect prediction. *IET Software*, Institution of Engineering and Technology, v. 12, n. 3, p. 161–175, 2018. ISSN 17518806. Available at: <https://onlinelibrary.wiley.com/doi/10.1049/iet-sen.2017.0148>.

MALHOTRA, R.; KHANNA, M. An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering*, Springer New York LLC, v. 22, n. 6, p. 2806–2851, 2017. ISSN 15737616. Available at: <https://link.springer.com/article/10.1007/s10664-016-9488-7>.

MANAPRAGADA, C.; WEBB, G. I.; SALEHI, M. Extremely fast decision tree. In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, New York, USA: ACM Press, 2018. p. 1953–1962. ISBN 9781450355520. Available at: <http://dl.acm.org/citation.cfm?doid=3219819.3220005>.

MCCABE, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2, n. 4, p. 308–320, 1976. ISSN 00985589.

MCINTOSH, S.; KAMEI, Y. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering*, v. 44, n. 5, p. 412–428, 2018. ISSN 00985589. Available at: <https://ieeexplore.ieee.org/document/7898457/>.

MISIRLI, A. T.; SHIHAB, E.; KAMEI, Y. Studying high impact fix-inducing changes. *Empirical Software Engineering*, Springer New York LLC, v. 21, n. 2, p. 605–641, 2016. ISSN 15737616. Available at: <https://link.springer.com/article/10.1007/s10664-015-9370-z>.

MOCKUS, A.; VOTTA, L. G. Identifying reasons for software changes using historic databases. In: *Conference on Software Maintenance*. [S.l.]: IEEE, 2000. p. 120–130.

MONTIEL, J.; READ, J.; BIFET, A.; ABDESSALEM, T. Scikit-multiflow: A multi-output streaming framework. *Journal of Machine Learning Research*, v. 19, n. 72, p. 1–5, 2018. ISSN 1533-7928. Available at: <http://jmlr.org/papers/v19/18-251.html.>

MOSER, R.; PEDRYCZ, W.; SUCCI, G. A Comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: *Proceedings - International Conference on Software Engineering*. [S.l.: s.n.], 2008. p. 181–190. ISBN 9781605580791. ISSN 02705257.

MUNSON, J. C.; KHOSHGOFTAAR, T. M. The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering*, v. 18, n. 5, p. 423–433, 1992. ISSN 00985589.

NAGAPPAN, N.; BALL, T. Use of relative code churn measures to predict system defect density. In: *Proceedings - 27th International Conference on Software Engineering, ICSE05*. New York, New York, USA: ACM Press, 2005. p. 284–292. ISBN 1595939632. Available at: <http://support.microsoft.com/>.

NAM, J. *Survey on Software Defect Prediction*. 34 p. Phd Thesis (PhD Thesis), 2014. Available at: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.722.3147>.

OZA, N. C. Online bagging and boosting. In: *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*. [S.l.: s.n.], 2005. v. 3, p. 2340–2345. ISSN 1062922X.

PASZKE, A.; GROSS, S.; MASSA, F.; LERER, A.; BRADBURY, J.; CHANAN, G.; KILLEEN, T.; LIN, Z.; GIMELSHEIN, N.; ANTIGA, L.; DESMAISON, A.; KÖPF, A.; YANG, E.; DEVITO, Z.; RAISON, M.; TEJANI, A.; CHILAMKURTHY, S.; STEINER, B.; FANG, L.; BAI, J.; CHINTALA, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2019. ISSN 23318422. Available at: <https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf>.

PEDREGOSA, F.; MICHEL, V.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; VANDERPLAS, J.; COURNAPEAU, D.; PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; THIRION, B.; GRISEL, O.; DUBOURG, V.; PASSOS, A.; BRUCHER, M.; ANDÉDOUARDAND, M. P.; DUCHESNAY, A.; Duchesnay EDOUARDDUCHESNAY, F. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, v. 12, n. 85, p. 2825–2830, 2011. ISSN 1533-7928. Available at: <http://scikit-learn.sourceforge.net.>

PRECHELT, L. Early stopping - But when? In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2nd. ed. Springer Verlag, 2012. v. 7700, p. 53–67. ISBN 9783642352881. Available at: <https://link.springer.com/chapter/10.1007/978-3-642-35289-8{_}5>.

RAHMAN, F.; DEVANBU, P. How, and why, process metrics are better. In: *Proceedings - International Conference on Software Engineering*. [s.n.], 2013. p. 432–441. ISBN 9781467330763. ISSN 02705257. Available at: <http://git.apache.org>.

RISH, I. An empirical study of the naive Bayes classifier. In: *IJCAI 2001 workshop on empirical methods in artificial intelligence*. [S.l.: s.n.], 2001. v. 3, p. 41–46.

ROSEN, C.; GRAWI, B.; SHIHAB, E. Commit guru: Analytics and risk prediction of software commits. In: *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*. New York, New York, USA: Association for Computing Machinery, Inc, 2015. p. 966–969. ISBN 9781450336758. Available at: <http://dl.acm.org/citation.cfm?doid=2786805.2803183>.

SALZBERG, S. L. On comparing classifiers: Pitfalls to avoid and a recommended approach. *Data Mining and Knowledge Discovery*, Kluwer Academic Publishers, v. 1, n. 3, p. 317–328, 1997. ISSN 13845810. Available at: <https://link.springer.com/article/10.1023/A:1009752403260>.

SCORNET, E. Tuning parameters in random forests. *ESAIM: Proceedings and Surveys*, EDP Sciences, v. 60, p. 144–162, 2017. ISSN 2267-3059. Available at: <https://doi.org/10.1051/proc/201760144>.

SHEN, V. Y.; YU, T. J.; THEBAUT, S. M.; PAULSEN, L. R. Identifying Error-Prone Software—An Empirical Study. *IEEE Transactions on Software Engineering*, SE-11, n. 4, p. 317–324, 1985. ISSN 00985589.

SHIHAB, E.; HASSAN, A. E.; ADAMS, B.; JIANG, Z. M. An industrial study on the risk of software changes. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*. New York, New York, USA: ACM Press, 2012. p. 1–11. ISBN 9781450316149. Available at: <http://dl.acm.org/citation.cfm?doid=2393596.2393670>.

ŚLIWERSKI, J.; ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, Association for Computing Machinery (ACM), v. 30, n. 4, p. 1–5, 2005. ISSN 0163-5948. Available at: <https://dl.acm.org/doi/10.1145/1082983.1083147>.

SPEARMAN, C. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology*, University of Illinois Press, v. 100, n. 3/4, p. 441–471, 1987. ISSN 00029556. Available at: <http://www.jstor.org/stable/1422689>.

SRIVASTAVA, N.; HINTON, G.; KRIZHEVSKY, A.; SUTSKEVER, I.; SALAKHUTDINOV, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, v. 15, n. 56, p. 1929–1958, 2014. ISSN 15337928. Available at: <http://jmlr.org/papers/v15/srivastava14a.html>.

TABASSUM, S.; MINKU, L. L.; FENG, D.; CABRAL, G. G. An Investigation of Cross-Project Learning in Online Just-In-Time Software Defect Prediction. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. [s.n.], 2020. p. 554–565. Available at: <https://conf.researchr.org/details/icse-2020/icse-2020-papers/27/An-Investigation-of-Cross-Project-Learning-in-Online-Just-In-Time-Software-Defect-Pre>.

TAN, M.; TAN, L.; DARA, S.; MAYEUX, C. Online Defect Prediction for Imbalanced Data. In: *Proceedings - International Conference on Software Engineering*. IEEE, 2015. v. 2, p. 99–108. ISBN 9781479919345. ISSN 02705257. Available at: <http://ieeexplore.ieee.org/document/7202954/>.

WANG, S.; YAO, X. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, v. 62, n. 2, p. 434–443, 2013. ISSN 00189529.

WILCOXON, F. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, JSTOR, v. 1, n. 6, p. 80–83, 1945. ISSN 00994987.

YANG, Y.; ZHOU, Y.; LIU, J.; ZHAO, Y.; LU, H.; XU, L.; XU, B.; LEUNG, H. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: . [S.l.]: Association for Computing Machinery (ACM), 2016. p. 157–168.

ZOU, H.; HASTIE, T. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, v. 67, n. 2, p. 301–320, 2005. ISSN 13697412.

APPENDIX A – TESTING ON ENTIRE DATASETS

In this appendix, we present our results when the first 5000 code changes of the data stream are used for validation, and the entire data stream is used for testing. In other words, the hyperparameter tuning is done as described in Section 5.4, but the testing is done with data that overlaps the validation segment. As that might create an advantage for the classifiers able to overfit the validation segment, we preferred not to include that segment into our evaluation. Despite that, Table 4 shows our results using the entire data streams for testing so that they can be compared to the results presented by Cabral et al. (2019), that used the methodology described in this appendix.

Table 4 – Results using the entire data stream for testing.

Dataset	Classifier	r_0 (std)	r_1 (std)	$\|r_0 - r_1\|$ (std)	g-mean (std)	$\|fr_1 - ir_1\|$ (std)	$\|fr_1 - pr_1\|$ (std)
brackets	BORB-IHF	58.04% (00.41%)	85.65% (00.31%)	31.26% (00.59%)	67.86% (00.25%)	09.01% (00.57%)	08.33% (00.29%)
	BORB-IRF	73.19% (00.10%)	78.53% (00.20%)	13.53% (00.19%)	73.05% (00.12%)	05.77% (00.05%)	07.54% (00.05%)
	BORB-LR	62.06% (00.04%)	89.12% (00.07%)	27.85% (00.07%)	71.86% (00.04%)	06.87% (00.03%)	06.67% (00.01%)
	BORB-MLP	70.02% (00.10%)	82.16% (00.19%)	14.95% (00.15%)	73.15% (00.10%)	08.10% (00.15%)	07.15% (00.04%)
	BORB-NB	57.06% (00.09%)	78.86% (00.13%)	23.68% (00.15%)	64.60% (00.08%)	15.74% (00.08%)	06.81% (00.04%)
	ORB-OHT	77.21% (00.48%)	70.34% (00.63%)	14.23% (00.58%)	69.67% (00.39%)	-	-
broadleaf	BORB-IHF	73.26% (00.14%)	64.71% (00.44%)	15.00% (00.37%)	67.46% (00.25%)	08.39% (00.42%)	04.42% (00.08%)
	BORB-IRF	58.30% (00.18%)	83.45% (00.30%)	28.23% (00.32%)	68.35% (00.19%)	13.04% (00.24%)	05.52% (00.10%)
	BORB-LR	69.73% (00.06%)	75.77% (00.21%)	13.71% (00.17%)	71.72% (00.11%)	07.13% (00.09%)	04.48% (00.03%)
	BORB-MLP	64.27% (00.15%)	80.43% (00.30%)	19.44% (00.29%)	70.73% (00.17%)	06.47% (00.16%)	05.45% (00.10%)
	BORB-NB	64.41% (00.11%)	60.45% (00.37%)	12.88% (00.27%)	61.10% (00.20%)	32.69% (00.91%)	04.84% (00.06%)
	ORB-OHT	75.90% (00.88%)	59.06% (01.22%)	18.20% (01.07%)	65.29% (00.91%)	-	-
camel	BORB-IHF	74.54% (00.12%)	60.55% (00.23%)	15.72% (00.23%)	66.58% (00.15%)	08.45% (00.15%)	03.31% (00.04%)
	BORB-IRF	60.11% (00.09%)	81.13% (00.21%)	22.99% (00.22%)	69.11% (00.10%)	10.02% (00.14%)	04.29% (00.05%)
	BORB-LR	62.02% (00.04%)	80.52% (00.07%)	19.55% (00.07%)	70.14% (00.04%)	08.88% (00.02%)	04.08% (00.02%)
	BORB-MLP	66.04% (00.08%)	76.61% (00.19%)	14.05% (00.21%)	70.50% (00.09%)	05.85% (00.09%)	04.47% (00.04%)
	BORB-NB	64.59% (00.06%)	53.12% (00.15%)	14.92% (00.11%)	57.87% (00.08%)	23.64% (00.12%)	05.09% (00.03%)
	ORB-OHT	67.28% (00.40%)	72.31% (00.42%)	13.25% (00.55%)	68.52% (00.28%)	-	-
fabric8	BORB-IHF	66.05% (00.49%)	64.21% (00.77%)	10.41% (00.54%)	63.75% (00.50%)	10.88% (00.75%)	06.51% (00.25%)
	BORB-IRF	70.71% (00.09%)	67.62% (00.24%)	11.78% (00.15%)	67.40% (00.14%)	04.44% (00.13%)	06.60% (00.06%)
	BORB-LR	59.61% (00.07%)	77.93% (00.10%)	19.66% (00.13%)	66.78% (00.04%)	06.92% (00.05%)	05.70% (00.03%)
	BORB-MLP	69.09% (00.17%)	73.96% (00.34%)	10.05% (00.26%)	70.04% (00.19%)	07.83% (00.25%)	06.02% (00.07%)
	BORB-NB	62.90% (00.11%)	64.45% (00.13%)	12.14% (00.10%)	61.99% (00.08%)	22.64% (00.25%)	07.56% (00.04%)
	ORB-OHT	77.68% (00.55%)	54.37% (00.95%)	24.18% (00.89%)	62.37% (00.68%)	-	-
jgroups	BORB-IHF	71.69% (00.20%)	53.04% (00.49%)	19.56% (00.52%)	60.98% (00.29%)	08.08% (00.30%)	03.84% (00.09%)
	BORB-IRF	68.74% (00.14%)	61.82% (00.30%)	13.95% (00.32%)	63.70% (00.17%)	03.92% (00.22%)	06.52% (00.05%)
	BORB-LR	68.18% (00.07%)	66.08% (00.21%)	08.10% (00.12%)	66.45% (00.12%)	04.47% (00.12%)	04.72% (00.03%)
	BORB-MLP	67.55% (00.18%)	66.01% (00.35%)	07.45% (00.28%)	66.29% (00.21%)	07.14% (00.25%)	04.53% (00.07%)
	BORB-NB	68.95% (00.14%)	49.62% (00.26%)	21.99% (00.26%)	57.80% (00.17%)	32.60% (00.80%)	05.89% (00.08%)
	ORB-OHT	70.43% (00.40%)	56.39% (00.80%)	17.26% (00.73%)	61.01% (00.43%)	-	-
neutron	BORB-IHF	72.82% (00.18%)	87.12% (00.32%)	18.08% (00.33%)	78.99% (00.18%)	04.94% (00.13%)	04.66% (00.05%)
	BORB-IRF	68.45% (00.07%)	91.71% (00.11%)	25.04% (00.10%)	77.96% (00.07%)	04.82% (00.09%)	06.72% (00.02%)
	BORB-LR	62.58% (00.09%)	93.47% (00.06%)	33.46% (00.12%)	75.10% (00.07%)	05.00% (00.02%)	08.20% (00.07%)
	BORB-MLP	72.07% (00.34%)	90.50% (00.27%)	22.56% (00.39%)	79.77% (00.28%)	02.98% (00.15%)	04.79% (00.14%)
	BORB-NB	49.78% (00.19%)	79.17% (00.11%)	31.97% (00.19%)	60.67% (00.14%)	40.06% (00.09%)	17.68% (00.13%)
	ORB-OHT	78.23% (00.82%)	81.52% (01.15%)	12.11% (01.01%)	79.40% (00.42%)	-	-
nova	BORB-IHF	77.25% (00.10%)	74.77% (00.20%)	16.44% (00.16%)	74.52% (00.12%)	12.02% (00.16%)	06.08% (00.07%)
	BORB-IRF	82.21% (00.05%)	75.75% (00.11%)	15.85% (00.10%)	77.64% (00.06%)	10.48% (00.08%)	04.79% (00.03%)
	BORB-LR	57.37% (00.06%)	92.00% (00.03%)	35.22% (00.06%)	70.18% (00.05%)	06.45% (00.01%)	11.82% (00.04%)
	BORB-MLP	73.50% (00.17%)	85.17% (00.13%)	18.01% (00.16%)	77.69% (00.14%)	06.74% (00.07%)	06.27% (00.12%)
	BORB-NB	53.47% (00.12%)	84.76% (00.10%)	33.10% (00.14%)	64.99% (00.11%)	21.75% (00.43%)	13.54% (00.08%)
	ORB-OHT	78.94% (00.29%)	80.74% (00.49%)	12.25% (00.48%)	78.51% (00.28%)	-	-
npm	BORB-IHF	58.60% (00.43%)	63.50% (00.92%)	13.61% (00.90%)	58.81% (00.48%)	11.23% (00.64%)	06.40% (00.32%)
	BORB-IRF	62.26% (00.39%)	67.52% (00.72%)	13.05% (00.62%)	62.75% (00.43%)	07.80% (00.77%)	06.64% (00.30%)
	BORB-LR	59.79% (00.15%)	76.01% (00.25%)	18.83% (00.31%)	66.70% (00.12%)	07.94% (00.10%)	04.98% (00.10%)
	BORB-MLP	67.23% (00.36%)	66.25% (00.69%)	09.05% (00.56%)	65.78% (00.40%)	12.39% (00.42%)	04.85% (00.18%)
	BORB-NB	61.53% (00.23%)	50.44% (00.51%)	32.86% (00.58%)	51.99% (00.35%)	39.40% (00.46%)	13.73% (00.15%)
	ORB-OHT	62.98% (00.58%)	62.41% (01.10%)	22.71% (01.13%)	59.47% (00.56%)	-	-
spring-integration	BORB-IHF	58.34% (01.12%)	66.60% (00.66%)	18.73% (01.22%)	59.77% (00.74%)	15.14% (00.81%)	10.27% (00.89%)
	BORB-IRF	66.80% (00.13%)	71.44% (00.23%)	14.17% (00.24%)	67.32% (00.13%)	13.89% (00.45%)	05.81% (00.08%)
	BORB-LR	63.76% (00.08%)	77.06% (00.12%)	16.43% (00.15%)	69.05% (00.07%)	15.93% (00.05%)	05.90% (00.04%)
	BORB-MLP	71.56% (00.27%)	66.82% (00.67%)	14.80% (00.54%)	67.91% (00.39%)	09.01% (00.23%)	06.35% (00.15%)
	BORB-NB	58.74% (00.43%)	54.68% (00.22%)	24.42% (00.40%)	53.94% (00.35%)	35.64% (00.40%)	13.32% (00.26%)
	ORB-OHT	75.56% (01.35%)	46.07% (01.56%)	40.54% (01.85%)	51.03% (01.31%)	-	-
tomcat	BORB-IHF	63.92% (00.32%)	67.75% (00.38%)	07.82% (00.36%)	65.39% (00.22%)	09.11% (00.31%)	03.81% (00.11%)
	BORB-IRF	69.77% (00.13%)	70.30% (00.20%)	09.43% (00.14%)	69.04% (00.11%)	04.24% (00.10%)	04.84% (00.06%)
	BORB-LR	68.52% (00.05%)	71.60% (00.08%)	07.21% (00.05%)	69.62% (00.04%)	01.03% (00.04%)	03.08% (00.02%)
	BORB-MLP	71.76% (00.14%)	68.31% (00.30%)	07.68% (00.27%)	69.61% (00.16%)	07.17% (00.38%)	03.91% (00.10%)
	BORB-NB	65.03% (00.12%)	52.19% (00.20%)	14.24% (00.18%)	57.68% (00.14%)	55.08% (00.28%)	04.47% (00.05%)
	ORB-OHT	65.91% (00.50%)	65.47% (00.52%)	10.26% (00.62%)	64.91% (00.40%)	-	-

Source: Created by the author (2021)

APPENDIX B – CONFIGURATION SPACES AND BEST CONFIGURATIONS

This appendix describes the configuration space and the best configuration for each combination of classifier and dataset. And each dimension of the configuration space (i.e., each hyperparameter) is structured as a triple $(range\ start\ value,$ **chosen value**, $range\ end\ value)$ or as a set $\{$**chosen value**, $other\ categorical\ values...\}$ in the following list:

1. Classifier = **BORB-IHF**; dataset = **broadleaf**; borb-l_0 = (1, **1.38**, 20); borb-l_1 = (1, **1.73**, 20); borb-m = (1.1, **1.94**, e); borb-window-size = (50, **69**, 200); borb-pull-request-size = (50, **191**, 500); borb-sample-size = (1000, **1859**, 4000); borb-fr_1 = (3, **0.31**, 5); borb-waiting-time = (90, **97**, 180); ihf-grace-period = (100, **271**, 500); ihf-leaf-prediction = {mc, nb, **nba**}; ihf-n-estimators = (10, **30**, 30); ihf-no-preprune = {**true**, false}; ihf-split-confidence = (1×10^{-7}, **1.33×10^{-5}**, 0.5); ihf-split-criterion = {gini, information gain, **hellinger**}; ihf-tie-threshold = (0.05, **0.35**, 0.5);

2. Classifier = **BORB-IHF**; dataset = **camel**; borb-l_0 = (1, **1.38**, 20); borb-l_1 = (1, **1.73**, 20); borb-m = (1.1, **1.94**, e); borb-window-size = (50, **69**, 200); borb-pull-request-size = (50, **191**, 500); borb-sample-size = (1000, **1859**, 4000); borb-fr_1 = (3, **0.31**, 5); borb-waiting-time = (90, **97**, 180); ihf-grace-period = (100, **271**, 500); ihf-leaf-prediction = {mc, nb, **nba**}; ihf-n-estimators = (10, **30**, 30); ihf-no-preprune = {**true**, false}; ihf-split-confidence = (1×10^{-7}, **1.33×10^{-5}**, 0.5); ihf-split-criterion = {gini, information gain, **hellinger**}; ihf-tie-threshold = (0.05, **0.35**, 0.5);

3. Classifier = **BORB-IHF**; dataset = **jgroups**; borb-l_0 = (1, **1.38**, 20); borb-l_1 = (1, **1.73**, 20); borb-m = (1.1, **1.94**, e); borb-window-size = (50, **69**, 200); borb-pull-request-size = (50, **191**, 500); borb-sample-size = (1000, **1859**, 4000); borb-fr_1 = (3, **0.31**, 5); borb-waiting-time = (90, **97**, 180); ihf-grace-period = (100, **271**, 500); ihf-leaf-prediction = {mc, nb, **nba**}; ihf-n-estimators = (10, **30**, 30); ihf-no-preprune = {**true**, false}; ihf-split-confidence = (1×10^{-7}, **1.33×10^{-5}**, 0.5); ihf-split-criterion = {gini, information gain, **hellinger**}; ihf-tie-threshold = (0.05, **0.35**, 0.5);

4. Classifier = **BORB-IHF**; dataset = **nova**; borb-l_0 = (1, **1.38**, 20); borb-l_1 = (1, **1.73**, 20); borb-m = (1.1, **1.94**, e); borb-window-size = (50, **69**, 200); borb-pull-request-size = (50, **191**, 500); borb-sample-size = (1000, **1859**, 4000); borb-fr_1 = (3, **0.31**, 5); borb-waiting-time = (90, **97**, 180); ihf-grace-period = (100, **271**, 500); ihf-leaf-prediction

= {mc, nb, **nba**}; ihf-n-estimators = (10, **30**, 30); ihf-no-preprune = {**true**, false}; ihf-split-confidence = (1×10^{-7}, **1.33×10^{-5}**, 0.5); ihf-split-criterion = {gini, information gain, **hellinger**}; ihf-tie-threshold = (0.05, **0.35**, 0.5);

5. Classifier = **BORB-IHF**; dataset = **tomcat**; borb-l_0 = (1, **1.45**, 20); borb-l_1 = (1, **1.67**, 20); borb-m = (1.1, **2.54**, e); borb-window-size = (50, **119**, 200); borb-pull-request-size = (50, **73**, 500); borb-sample-size = (1000, **1246**, 4000); borb-fr_1 = (3, **0.44**, 5); borb-waiting-time = (90, **155**, 180); ihf-grace-period = (100, **133**, 500); ihf-leaf-prediction = {mc, nb, **nba**}; ihf-n-estimators = (10, **12**, 30); ihf-no-preprune = {**true**, false}; ihf-split-confidence = (1×10^{-7}, **6.8×10^{-5}**, **5×10^{-5}**, 0.5); ihf-split-criterion = {**gini**, information gain, hellinger}; ihf-tie-threshold = (0.05, **0.33**, 0.5);

6. Classifier = **BORB-IHF**; dataset = **brackets**; borb-l_0 = (1, **1.49**, 20); borb-l_1 = (1, **19.48**, 20); borb-m = (1.1, **1.77**, e); borb-window-size = (50, **121**, 200); borb-pull-request-size = (50, **272**, 500); borb-sample-size = (1000, **2463**, 4000); borb-fr_1 = (3, **0.47**, 5); borb-waiting-time = (90, **94**, 180); ihf-grace-period = (100, **332**, 500); ihf-leaf-prediction = {**mc**, nb, nba}; ihf-n-estimators = (10, **18**, 30); ihf-no-preprune = {true, **false**}; ihf-split-confidence = (1×10^{-7}, **0.02**, 0.5); ihf-split-criterion = {gini, **information gain**, hellinger}; ihf-tie-threshold = (0.05, **0.24**, 0.5);

7. Classifier = **BORB-IHF**; dataset = **fabric8**; borb-l_0 = (1, **19.74**, 20); borb-l_1 = (1, **3.54**, 20); borb-m = (1.1, **1.23**, e); borb-window-size = (50, **167**, 200); borb-pull-request-size = (50, **129**, 500); borb-sample-size = (1000, **2857**, 4000); borb-fr_1 = (3, **0.39**, 5); borb-waiting-time = (90, **93**, 180); ihf-grace-period = (100, **443**, 500); ihf-leaf-prediction = {mc, nb, **nba**}; ihf-n-estimators = (10, **21**, 30); ihf-no-preprune = {**true**, false}; ihf-split-confidence = (1×10^{-7}, **0.006**, 0.5); ihf-split-criterion = {**gini**, information gain, hellinger}; ihf-tie-threshold = (0.05, **0.16**, 0.5);

8. Classifier = **BORB-IHF**; dataset = **neutron**; borb-l_0 = (1, **3.95**, 20); borb-l_1 = (1, **15.06**, 20); borb-m = (1.1, **2.27**, e); borb-window-size = (50, **138**, 200); borb-pull-request-size = (50, **71**, 500); borb-sample-size = (1000, **1215**, 4000); borb-fr_1 = (3, **0.4**, 5); borb-waiting-time = (90, **161**, 180); ihf-grace-period = (100, **122**, 500); ihf-leaf-prediction = {mc, **nb**, nba}; ihf-n-estimators = (10, **15**, 30); ihf-no-preprune = {true, **false**}; ihf-split-confidence = (1×10^{-7}, **7.791×10^{5}**, 0.5); ihf-split-criterion = {gini, **information gain**, hellinger}; ihf-tie-threshold = (0.05, **0.21**, 0.5);

9. Classifier = **BORB-IHF**; dataset = **spring-integration**; borb-l_0 = (1, **4.54**, 20); borb-l_1 = (1, **2.43**, 20); borb-m = (1.1, **1.69**, e); borb-window-size = (50, **87**, 200); borb-pull-request-size = (50, **199**, 500); borb-sample-size = (1000, **2052**, 4000); borb-fr_1 = (3, **0.42**, 5); borb-waiting-time = (90, **137**, 180); ihf-grace-period = (100, **206**, 500); ihf-leaf-prediction = {**mc**, nb, nba}; ihf-n-estimators = (10, **19**, 30); ihf-no-preprune = {true, **false**}; ihf-split-confidence = (1×10^{-7}, **0.02**, 0.5); ihf-split-criterion = {gini, information gain, **hellinger**}; ihf-tie-threshold = (0.05, **0.34**, 0.5);

10. Classifier = **BORB-IHF**; dataset = **npm**; borb-l_0 = (1, **9.25**, 20); borb-l_1 = (1, **3.4**, 20); borb-m = (1.1, **1.41**, e); borb-window-size = (50, **90**, 200); borb-pull-request-size = (50, **221**, 500); borb-sample-size = (1000, **2066**, 4000); borb-fr_1 = (3, **0.44**, 5); borb-waiting-time = (90, **108**, 180); ihf-grace-period = (100, **150**, 500); ihf-leaf-prediction = {mc, nb, **nba**}; ihf-n-estimators = (10, **14**, 30); ihf-no-preprune = {**true**, false}; ihf-split-confidence = (1×10^{-7}, **0.13**, 0.5); ihf-split-criterion = {gini, information gain, **hellinger**}; ihf-tie-threshold = (0.05, **0.1**, 0.5);

11. Classifier = **BORB-IRF**; dataset = **spring-integration**; borb-l_0 = (1, **1.12**, 20); borb-l_1 = (1, **2.88**, 20); borb-m = (1.1, **1.68**, e); borb-window-size = (50, **74**, 200); borb-pull-request-size = (50, **140**, 500); borb-sample-size = (1000, **2809**, 4000); borb-fr_1 = (3, **0.42**, 5); borb-waiting-time = (90, **103**, 180); irf-criterion = {**gini**, information gain, hellinger}; irf-max-features = (3, **6**, 7); irf-min-samples-leaf = (100, **181**, 300); irf-n-estimators = (20, **86**, 100);

12. Classifier = **BORB-IRF**; dataset = **nova**; borb-l_0 = (1, **1.8**, 20); borb-l_1 = (1, **12.18**, 20); borb-m = (1.1, **1.42**, e); borb-window-size = (50, **133**, 200); borb-pull-request-size = (50, **369**, 500); borb-sample-size = (1000, **3088**, 4000); borb-fr_1 = (3, **0.31**, 5); borb-waiting-time = (90, **97**, 180); irf-criterion = {gini, **entropy**}; irf-max-features = (3, **7**, 7); irf-min-samples-leaf = (100, **104**, 300); irf-n-estimators = (20, **91**, 100);

13. Classifier = **BORB-IRF**; dataset = **npm**; borb-l_0 = (1, **12.72**, 20); borb-l_1 = (1, **9.11**, 20); borb-m = (1.1, **1.64**, e); borb-window-size = (50, **59**, 200); borb-pull-request-size = (50, **191**, 500); borb-sample-size = (1000, **3510**, 4000); borb-fr_1 = (3, **0.41**, 5); borb-waiting-time = (90, **102**, 180); irf-criterion = {gini, **entropy**}; irf-max-features = (3, **7**, 7); irf-min-samples-leaf = (100, **124**, 300); irf-n-estimators = (20, **30**, 100);

14. Classifier = **BORB-IRF**; dataset = **neutron**; borb-l_0 = (1, **2.23**, 20); borb-l_1 = (1, **2.88**, 20); borb-m = (1.1, **2.16**, e); borb-window-size = (50, **147**, 200); borb-pull-request-size = (50, **109**, 500); borb-sample-size = (1000, **3251**, 4000); borb-fr_1 = (3, **0.42**, 5); borb-waiting-time = (90, **119**, 180); irf-criterion = {**gini**, information gain, hellinger}; irf-max-features = (3, **5**, 7); irf-min-samples-leaf = (100, **167**, 300); irf-n-estimators = (20, **86**, 100);

15. Classifier = **BORB-IRF**; dataset = **broadleaf**; borb-l_0 = (1, **2.26**, 20); borb-l_1 = (1, **1.66**, 20); borb-m = (1.1, **1.22**, e); borb-window-size = (50, **184**, 200); borb-pull-request-size = (50, **205**, 500); borb-sample-size = (1000, **1935**, 4000); borb-fr_1 = (3, **0.48**, 5); borb-waiting-time = (90, **101**, 180); irf-criterion = {gini, **entropy**}; irf-max-features = (3, **5**, 7); irf-min-samples-leaf = (100, **110**, 300); irf-n-estimators = (20, **27**, 100);

16. Classifier = **BORB-IRF**; dataset = **camel**; borb-l_0 = (1, **2.26**, 20); borb-l_1 = (1, **1.66**, 20); borb-m = (1.1, **1.22**, e); borb-window-size = (50, **184**, 200); borb-pull-request-size = (50, **205**, 500); borb-sample-size = (1000, **1935**, 4000); borb-fr_1 = (3, **0.48**, 5); borb-waiting-time = (90, **101**, 180); irf-criterion = {gini, **entropy**}; irf-max-features = (3, **5**, 7); irf-min-samples-leaf = (100, **110**, 300); irf-n-estimators = (20, **27**, 100);

17. Classifier = **BORB-IRF**; dataset = **brackets**; borb-l_0 = (1, **5.65**, 20); borb-l_1 = (1, **7.45**, 20); borb-m = (1.1, **1.20**, e); borb-window-size = (50, **193**, 200); borb-pull-request-size = (50, **111**, 500); borb-sample-size = (1000, **3050**, 4000); borb-fr_1 = (3, **0.35**, 5); borb-waiting-time = (90, **90**, 180); irf-criterion = {**gini**, information gain, hellinger}; irf-max-features = (3, **4**, 7); irf-min-samples-leaf = (100, **165**, 300); irf-n-estimators = (20, **77**, 100);

18. Classifier = **BORB-IRF**; dataset = **fabric8**; borb-l_0 = (1, **5.65**, 20); borb-l_1 = (1, **7.45**, 20); borb-m = (1.1, **1.20**, e); borb-window-size = (50, **193**, 200); borb-pull-request-size = (50, **111**, 500); borb-sample-size = (1000, **3050**, 4000); borb-fr_1 = (3, **0.35**, 5); borb-waiting-time = (90, **90**, 180); irf-criterion = {**gini**, information gain, hellinger}; irf-max-features = (3, **4**, 7); irf-min-samples-leaf = (100, **165**, 300); irf-n-estimators = (20, **77**, 100);

19. Classifier = **BORB-IRF**; dataset = **jgroups**; borb-l_0 = (1, **5.65**, 20); borb-l_1 = (1, **7.45**, 20); borb-m = (1.1, **1.20**, e); borb-window-size = (50, **193**, 200); borb-pull-request-

size = (50, **111**, 500); borb-sample-size = (1000, **3050**, 4000); borb-fr$_1$ = (3, **0.35**, 5); borb-waiting-time = (90, **90**, 180); irf-criterion = {**gini**, information gain, hellinger}; irf-max-features = (3, **4**, 7); irf-min-samples-leaf = (100, **165**, 300); irf-n-estimators = (20, **77**, 100);

20. Classifier = **BORB-IRF**; dataset = **tomcat**; borb-l$_0$ = (1, **8.47**, 20); borb-l$_1$ = (1, **2.1**, 20); borb-m = (1.1, **2.43**, e); borb-window-size = (50, **127**, 200); borb-pull-request-size = (50, **78**, 500); borb-sample-size = (1000, **2627**, 4000); borb-fr$_1$ = (3, **0.4**, 5); borb-waiting-time = (90, **122**, 180); irf-criterion = {**gini**, information gain, hellinger}; irf-max-features = (3, **6**, 7); irf-min-samples-leaf = (100, **170**, 300); irf-n-estimators = (20, **56**, 100);

21. Classifier = **BORB-LR**; dataset = **broadleaf**; borb-l$_0$ = (1, **19.54**, 20); borb-l$_1$ = (1, **1.28**, 20); borb-m = (1.1, **1.41**, e); borb-window-size = (50, **130**, 200); borb-pull-request-size = (50, **138**, 500); borb-sample-size = (1000, **1997**, 4000); borb-fr$_1$ = (3, **0.37**, 5); borb-waiting-time = (90, **101**, 180); lr-alpha = (0.01, **0.03**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **64**, 80);

22. Classifier = **BORB-LR**; dataset = **jgroups**; borb-l$_0$ = (1, **19.54**, 20); borb-l$_1$ = (1, **1.28**, 20); borb-m = (1.1, **1.41**, e); borb-window-size = (50, **130**, 200); borb-pull-request-size = (50, **138**, 500); borb-sample-size = (1000, **1997**, 4000); borb-fr$_1$ = (3, **0.37**, 5); borb-waiting-time = (90, **101**, 180); lr-alpha = (0.01, **0.03**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **64**, 80);

23. Classifier = **BORB-LR**; dataset = **brackets**; borb-l$_0$ = (1, **2.24**, 20); borb-l$_1$ = (1, **1.45**, 20); borb-m = (1.1, **2.02**, e); borb-window-size = (50, **158**, 200); borb-pull-request-size = (50, **58**, 500); borb-sample-size = (1000, **3120**, 4000); borb-fr$_1$ = (3, **0.46**, 5); borb-waiting-time = (90, **90**, 180); lr-alpha = (0.01, **0.08**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **53**, 80);

24. Classifier = **BORB-LR**; dataset = **camel**; borb-l$_0$ = (1, **2.24**, 20); borb-l$_1$ = (1, **1.45**, 20); borb-m = (1.1, **2.02**, e); borb-window-size = (50, **158**, 200); borb-pull-request-size = (50, **58**, 500); borb-sample-size = (1000, **3120**, 4000); borb-fr$_1$ = (3, **0.46**, 5); borb-waiting-time = (90, **90**, 180); lr-alpha = (0.01, **0.08**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **53**, 80);

25. Classifier = **BORB-LR**; dataset = **fabric8**; borb-l_0 = (1, **2.24**, 20); borb-l_1 = (1, **1.45**, 20); borb-m = (1.1, **2.02**, e); borb-window-size = (50, **158**, 200); borb-pull-request-size = (50, **58**, 500); borb-sample-size = (1000, **3120**, 4000); borb-fr_1 = (3, **0.46**, 5); borb-waiting-time = (90, **90**, 180); lr-alpha = (0.01, **0.08**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **53**, 80);

26. Classifier = **BORB-LR**; dataset = **neutron**; borb-l_0 = (1, **2.24**, 20); borb-l_1 = (1, **1.45**, 20); borb-m = (1.1, **2.02**, e); borb-window-size = (50, **158**, 200); borb-pull-request-size = (50, **58**, 500); borb-sample-size = (1000, **3120**, 4000); borb-fr_1 = (3, **0.46**, 5); borb-waiting-time = (90, **90**, 180); lr-alpha = (0.01, **0.08**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **53**, 80);

27. Classifier = **BORB-LR**; dataset = **nova**; borb-l_0 = (1, **2.24**, 20); borb-l_1 = (1, **1.45**, 20); borb-m = (1.1, **2.02**, e); borb-window-size = (50, **158**, 200); borb-pull-request-size = (50, **58**, 500); borb-sample-size = (1000, **3120**, 4000); borb-fr_1 = (3, **0.46**, 5); borb-waiting-time = (90, **90**, 180); lr-alpha = (0.01, **0.08**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **53**, 80);

28. Classifier = **BORB-LR**; dataset = **npm**; borb-l_0 = (1, **2.24**, 20); borb-l_1 = (1, **1.45**, 20); borb-m = (1.1, **2.02**, e); borb-window-size = (50, **158**, 200); borb-pull-request-size = (50, **58**, 500); borb-sample-size = (1000, **3120**, 4000); borb-fr_1 = (3, **0.46**, 5); borb-waiting-time = (90, **90**, 180); lr-alpha = (0.01, **0.08**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **53**, 80);

29. Classifier = **BORB-LR**; dataset = **spring-integration**; borb-l_0 = (1, **2.24**, 20); borb-l_1 = (1, **1.45**, 20); borb-m = (1.1, **2.02**, e); borb-window-size = (50, **158**, 200); borb-pull-request-size = (50, **58**, 500); borb-sample-size = (1000, **3120**, 4000); borb-fr_1 = (3, **0.46**, 5); borb-waiting-time = (90, **90**, 180); lr-alpha = (0.01, **0.08**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **53**, 80);

30. Classifier = **BORB-LR**; dataset = **tomcat**; borb-l_0 = (1, **5.36**, 20); borb-l_1 = (1, **10.16**, 20); borb-m = (1.1, **1.24**, e); borb-window-size = (50, **89**, 200); borb-pull-request-size = (50, **107**, 500); borb-sample-size = (1000, **3292**, 4000); borb-fr_1 = (3, **0.42**, 5); borb-waiting-time = (90, **100**, 180); lr-alpha = (0.01, **0.33**, 1); lr-batch-size = (128, **129**, 512); lr-log-transformation = {**true**, false}; lr-n-epochs = (10, **69**, 80);

31. Classifier = **BORB-MLP**; dataset = **broadleaf**; borb-l_0 = (1, **13.28**, 20); borb-l_1 = (1, **3.50**, 20); borb-m = (1.1, **2.71**, e); borb-window-size = (50, **194**, 200); borb-pull-request-size = (50, **134**, 500); borb-sample-size = (1000, **3317**, 4000); borb-fr_1 = (3, **0.42**, 5); borb-waiting-time = (90, **102**, 180); mlp-batch-size = (128, **132**, 512); mlp-dropout-hidden-layer = (0.3, **0.32**, 0.5); mlp-dropout-input-layer = (0.1, **0.15**, 0.3); mlp-hidden-layers-size = (5, **9**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **79**, 80); mlp-n-hidden-layers = (1, **2**, 3);

32. Classifier = **BORB-MLP**; dataset = **camel**; borb-l_0 = (1, **13.28**, 20); borb-l_1 = (1, **3.50**, 20); borb-m = (1.1, **2.71**, e); borb-window-size = (50, **194**, 200); borb-pull-request-size = (50, **134**, 500); borb-sample-size = (1000, **3317**, 4000); borb-fr_1 = (3, **0.42**, 5); borb-waiting-time = (90, **102**, 180); mlp-batch-size = (128, **132**, 512); mlp-dropout-hidden-layer = (0.3, **0.32**, 0.5); mlp-dropout-input-layer = (0.1, **0.15**, 0.3); mlp-hidden-layers-size = (5, **9**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **79**, 80); mlp-n-hidden-layers = (1, **2**, 3);

33. Classifier = **BORB-MLP**; dataset = **neutron**; borb-l_0 = (1, **13.28**, 20); borb-l_1 = (1, **3.50**, 20); borb-m = (1.1, **2.71**, e); borb-window-size = (50, **194**, 200); borb-pull-request-size = (50, **134**, 500); borb-sample-size = (1000, **3317**, 4000); borb-fr_1 = (3, **0.42**, 5); borb-waiting-time = (90, **102**, 180); mlp-batch-size = (128, **132**, 512); mlp-dropout-hidden-layer = (0.3, **0.32**, 0.5); mlp-dropout-input-layer = (0.1, **0.15**, 0.3); mlp-hidden-layers-size = (5, **9**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **79**, 80); mlp-n-hidden-layers = (1, **2**, 3);

34. Classifier = **BORB-MLP**; dataset = **brackets**; borb-l_0 = (1, **2.02**, 20); borb-l_1 = (1, **1.24**, 20); borb-m = (1.1, **2.03**, e); borb-window-size = (50, **142**, 200); borb-pull-request-size = (50, **96**, 500); borb-sample-size = (1000, **1207**, 4000); borb-fr_1 = (3, **0.38**, 5); borb-waiting-time = (90, **91**, 180); mlp-batch-size = (128, **131**, 512); mlp-dropout-hidden-layer = (0.3, **0.42**, 0.5); mlp-dropout-input-layer = (0.1, **0.14**, 0.3); mlp-hidden-layers-size = (5, **14**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **51**, 80); mlp-n-hidden-layers

= (1, **3**, 3);

35. Classifier = **BORB-MLP**; dataset = **fabric8**; borb-l_0 = (1, **2.02**, 20); borb-l_1 = (1, **1.24**, 20); borb-m = (1.1, **2.03**, e); borb-window-size = (50, **142**, 200); borb-pull-request-size = (50, **96**, 500); borb-sample-size = (1000, **1207**, 4000); borb-fr_1 = (3, **0.38**, 5); borb-waiting-time = (90, **91**, 180); mlp-batch-size = (128, **131**, 512); mlp-dropout-hidden-layer = (0.3, **0.42**, 0.5); mlp-dropout-input-layer = (0.1, **0.14**, 0.3); mlp-hidden-layers-size = (5, **14**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **51**, 80); mlp-n-hidden-layers = (1, **3**, 3);

36. Classifier = **BORB-MLP**; dataset = **jgroups**; borb-l_0 = (1, **2.02**, 20); borb-l_1 = (1, **1.24**, 20); borb-m = (1.1, **2.03**, e); borb-window-size = (50, **142**, 200); borb-pull-request-size = (50, **96**, 500); borb-sample-size = (1000, **1207**, 4000); borb-fr_1 = (3, **0.38**, 5); borb-waiting-time = (90, **91**, 180); mlp-batch-size = (128, **131**, 512); mlp-dropout-hidden-layer = (0.3, **0.42**, 0.5); mlp-dropout-input-layer = (0.1, **0.14**, 0.3); mlp-hidden-layers-size = (5, **14**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **51**, 80); mlp-n-hidden-layers = (1, **3**, 3);

37. Classifier = **BORB-MLP**; dataset = **nova**; borb-l_0 = (1, **2.02**, 20); borb-l_1 = (1, **1.24**, 20); borb-m = (1.1, **2.03**, e); borb-window-size = (50, **142**, 200); borb-pull-request-size = (50, **96**, 500); borb-sample-size = (1000, **1207**, 4000); borb-fr_1 = (3, **0.38**, 5); borb-waiting-time = (90, **91**, 180); mlp-batch-size = (128, **131**, 512); mlp-dropout-hidden-layer = (0.3, **0.42**, 0.5); mlp-dropout-input-layer = (0.1, **0.14**, 0.3); mlp-hidden-layers-size = (5, **14**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **51**, 80); mlp-n-hidden-layers = (1, **3**, 3);

38. Classifier = **BORB-MLP**; dataset = **npm**; borb-l_0 = (1, **2.02**, 20); borb-l_1 = (1, **1.24**, 20); borb-m = (1.1, **2.03**, e); borb-window-size = (50, **142**, 200); borb-pull-request-size = (50, **96**, 500); borb-sample-size = (1000, **1207**, 4000); borb-fr_1 = (3, **0.38**, 5); borb-waiting-time = (90, **91**, 180); mlp-batch-size = (128, **131**, 512); mlp-dropout-hidden-layer = (0.3, **0.42**, 0.5); mlp-dropout-input-layer = (0.1, **0.14**, 0.3); mlp-hidden-layers-size = (5, **14**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-

log-transformation = {**true**, false}; mlp-n-epochs = (10, **51**, 80); mlp-n-hidden-layers = (1, **3**, 3);

39. Classifier = **BORB-MLP**; dataset = **spring-integration**; borb-l_0 = (1, **2.95**, 20); borb-l_1 = (1, **12.14**, 20); borb-m = (1.1, **1.98**, e); borb-window-size = (50, **193**, 200); borb-pull-request-size = (50, **51**, 500); borb-sample-size = (1000, **1828**, 4000); borb-fr_1 = (3, **0.37**, 5); borb-waiting-time = (90, **99**, 180); mlp-batch-size = (128, **129**, 512); mlp-dropout-hidden-layer = (0.3, **0.48**, 0.5); mlp-dropout-input-layer = (0.1, **0.18**, 0.3); mlp-hidden-layers-size = (5, **8**, 15); mlp-learning-rate = (0.0001, **0.0008**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **77**, 80); mlp-n-hidden-layers = (1, **2**, 3);

40. Classifier = **BORB-MLP**; dataset = **tomcat**; borb-l_0 = (1, **4.76**, 20); borb-l_1 = (1, **11.06**, 20); borb-m = (1.1, **2.50**, e); borb-window-size = (50, **140**, 200); borb-pull-request-size = (50, **92**, 500); borb-sample-size = (1000, **2515**, 4000); borb-fr_1 = (3, **0.39**, 5); borb-waiting-time = (90, **105**, 180); mlp-batch-size = (128, **129**, 512); mlp-dropout-hidden-layer = (0.3, **0.34**, 0.5); mlp-dropout-input-layer = (0.1, **0.17**, 0.3); mlp-hidden-layers-size = (5, **9**, 15); mlp-learning-rate = (0.0001, **0.01**, 0.01); mlp-log-transformation = {**true**, false}; mlp-n-epochs = (10, **25**, 80); mlp-n-hidden-layers = (1, **1**, 3);

41. Classifier = **BORB-NB**; dataset = **tomcat**; borb-l_0 = (1, **1.12**, 20); borb-l_1 = (1, **16.81**, 20); borb-m = (1.1, **1.83**, e); borb-window-size = (50, **104**, 200); borb-pull-request-size = (50, **140**, 500); borb-sample-size = (1000, **1527**, 4000); borb-fr_1 = (3, **0.39**, 5); borb-waiting-time = (90, **119**, 180); nb-n-updates = (10, **74**, 80);

42. Classifier = **BORB-NB**; dataset = **broadleaf**; borb-l_0 = (1, **1.15**, 20); borb-l_1 = (1, **1.23**, 20); borb-m = (1.1, **2.34**, e); borb-window-size = (50, **53**, 200); borb-pull-request-size = (50, **154**, 500); borb-sample-size = (1000, **3501**, 4000); borb-fr_1 = (3, **0.37**, 5); borb-waiting-time = (90, **91**, 180); nb-n-updates = (10, **17**, 80);

43. Classifier = **BORB-NB**; dataset = **neutron**; borb-l_0 = (1, **1.42**, 20); borb-l_1 = (1, **9.69**, 20); borb-m = (1.1, **1.61**, e); borb-window-size = (50, **161**, 200); borb-pull-request-size = (50, **88**, 500); borb-sample-size = (1000, **3968**, 4000); borb-fr_1 = (3, **0.41**, 5); borb-waiting-time = (90, **166**, 180); nb-n-updates = (10, **34**, 80);

44. Classifier = **BORB-NB**; dataset = **brackets**; borb-l_0 = (1, **1.43**, 20); borb-l_1 = (1, **11.12**, 20); borb-m = (1.1, **2.26**, e); borb-window-size = (50, **140**, 200); borb-pull-request-size = (50, **134**, 500); borb-sample-size = (1000, **1438**, 4000); borb-fr_1 = (3, **0.48**, 5); borb-waiting-time = (90, **90**, 180); nb-n-updates = (10, **80**, 80);

45. Classifier = **BORB-NB**; dataset = **camel**; borb-l_0 = (1, **2.18**, 20); borb-l_1 = (1, **11.19**, 20); borb-m = (1.1, **1.14**, e); borb-window-size = (50, **150**, 200); borb-pull-request-size = (50, **162**, 500); borb-sample-size = (1000, **3533**, 4000); borb-fr_1 = (3, **0.38**, 5); borb-waiting-time = (90, **91**, 180); nb-n-updates = (10, **32**, 80);

46. Classifier = **BORB-NB**; dataset = **jgroups**; borb-l_0 = (1, **3.16**, 20); borb-l_1 = (1, **6.12**, 20); borb-m = (1.1, **2.14**, e); borb-window-size = (50, **183**, 200); borb-pull-request-size = (50, **83**, 500); borb-sample-size = (1000, **1314**, 4000); borb-fr_1 = (3, **0.32**, 5); borb-waiting-time = (90, **96**, 180); nb-n-updates = (10, **15**, 80);

47. Classifier = **BORB-NB**; dataset = **npm**; borb-l_0 = (1, **3.16**, 20); borb-l_1 = (1, **6.12**, 20); borb-m = (1.1, **2.14**, e); borb-window-size = (50, **183**, 200); borb-pull-request-size = (50, **83**, 500); borb-sample-size = (1000, **1314**, 4000); borb-fr_1 = (3, **0.32**, 5); borb-waiting-time = (90, **96**, 180); nb-n-updates = (10, **15**, 80);

48. Classifier = **BORB-NB**; dataset = **fabric8**; borb-l_0 = (1, **4.72**, 20); borb-l_1 = (1, **7.28**, 20); borb-m = (1.1, **2.66**, e); borb-window-size = (50, **193**, 200); borb-pull-request-size = (50, **101**, 500); borb-sample-size = (1000, **2608**, 4000); borb-fr_1 = (3, **0.41**, 5); borb-waiting-time = (90, **92**, 180); nb-n-updates = (10, **51**, 80);

49. Classifier = **BORB-NB**; dataset = **spring-integration**; borb-l_0 = (1, **4.72**, 20); borb-l_1 = (1, **7.28**, 20); borb-m = (1.1, **2.66**, e); borb-window-size = (50, **193**, 200); borb-pull-request-size = (50, **101**, 500); borb-sample-size = (1000, **2608**, 4000); borb-fr_1 = (3, **0.41**, 5); borb-waiting-time = (90, **92**, 180); nb-n-updates = (10, **51**, 80);

50. Classifier = **BORB-NB**; dataset = **nova**; borb-l_0 = (1, **7.56**, 20); borb-l_1 = (1, **1.33**, 20); borb-m = (1.1, **1.82**, e); borb-window-size = (50, **100**, 200); borb-pull-request-size = (50, **259**, 500); borb-sample-size = (1000, **2444**, 4000); borb-fr_1 = (3, **0.45**, 5); borb-waiting-time = (90, **91**, 180); nb-n-updates = (10, **18**, 80);

51. Classifier = **ORB-OHT**; dataset = **brackets**; oht-grace-period = (100, **104**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **15**, 40); oht-no-preprune =

{true, **false**}; oht-split-confidence = (1×10^{-7}, **0.0002**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.43**, 0.5); orb-decay-factor = (0.9, **0.96**, 0.999); orb-l_0 = (1, **1.89**, 20); orb-l_1 = (1, **2.63**, 20); orb-m = (1.1, **1.52**, e); orb-window-size = (50, **58**, 200); orb-n = (3, **4**, 7); orb-th = (3, **0.39**, 5); orb-waiting-time = (90, **95**, 180);

52. Classifier = **ORB-OHT**; dataset = **broadleaf**; oht-grace-period = (100, **104**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **15**, 40); oht-no-preprune = {true, **false**}; oht-split-confidence = (1×10^{-7}, **0.0002**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.43**, 0.5); orb-decay-factor = (0.9, **0.96**, 0.999); orb-l_0 = (1, **1.89**, 20); orb-l_1 = (1, **2.63**, 20); orb-m = (1.1, **1.52**, e); orb-window-size = (50, **58**, 200); orb-n = (3, **4**, 7); orb-th = (3, **0.39**, 5); orb-waiting-time = (90, **95**, 180);

53. Classifier = **ORB-OHT**; dataset = **fabric8**; oht-grace-period = (100, **104**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **15**, 40); oht-no-preprune = {true, **false**}; oht-split-confidence = (1×10^{-7}, **0.0002**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.43**, 0.5); orb-decay-factor = (0.9, **0.96**, 0.999); orb-l_0 = (1, **1.89**, 20); orb-l_1 = (1, **2.63**, 20); orb-m = (1.1, **1.52**, e); orb-window-size = (50, **58**, 200); orb-n = (3, **4**, 7); orb-th = (3, **0.39**, 5); orb-waiting-time = (90, **95**, 180);

54. Classifier = **ORB-OHT**; dataset = **nova**; oht-grace-period = (100, **104**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **15**, 40); oht-no-preprune = {true, **false**}; oht-split-confidence = (1×10^{-7}, **0.0002**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.43**, 0.5); orb-decay-factor = (0.9, **0.96**, 0.999); orb-l_0 = (1, **1.89**, 20); orb-l_1 = (1, **2.63**, 20); orb-m = (1.1, **1.52**, e); orb-window-size = (50, **58**, 200); orb-n = (3, **4**, 7); orb-th = (3, **0.39**, 5); orb-waiting-time = (90, **95**, 180);

55. Classifier = **ORB-OHT**; dataset = **npm**; oht-grace-period = (100, **111**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **33**, 40); oht-no-preprune = {**true**, false}; oht-split-confidence = (1×10^{-7}, **4.27**$\times$**10^{7}**, 0.5); oht-split-criterion = {**gini**, information gain, hellinger}; oht-tie-threshold = (0.05, **0.38**, 0.5); orb-decay-factor = (0.9, **0.95**, 0.999); orb-l_0 = (1, **1.74**, 20); orb-l_1 = (1, **14.07**, 20); orb-m =

(1.1, **2.28**, e); orb-window-size = (50, **122**, 200); orb-n = (3, **3**, 7); orb-th = (3, **0.43**, 5); orb-waiting-time = (90, **107**, 180);

56. Classifier = **ORB-OHT**; dataset = **camel**; oht-grace-period = (100, **112**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **26**, 40); oht-no-preprune = {true, **false**}; oht-split-confidence = (1×10^{-7}, **0.02**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.13**, 0.5); orb-decay-factor = (0.9, **0.98**, 0.999); orb-l_0 = (1, **1.73**, 20); orb-l_1 = (1, **10.09**, 20); orb-m = (1.1, **1.10**, e); orb-window-size = (50, **84**, 200); orb-n = (3, **5**, 7); orb-th = (3, **0.39**, 5); orb-waiting-time = (90, **102**, 180);

57. Classifier = **ORB-OHT**; dataset = **spring-integration**; oht-grace-period = (100, **112**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **26**, 40); oht-no-preprune = {true, **false**}; oht-split-confidence = (1×10^{-7}, **0.02**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.13**, 0.5); orb-decay-factor = (0.9, **0.98**, 0.999); orb-l_0 = (1, **1.73**, 20); orb-l_1 = (1, **10.09**, 20); orb-m = (1.1, **1.10**, e); orb-window-size = (50, **84**, 200); orb-n = (3, **5**, 7); orb-th = (3, **0.39**, 5); orb-waiting-time = (90, **102**, 180);

58. Classifier = **ORB-OHT**; dataset = **tomcat**; oht-grace-period = (100, **112**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **26**, 40); oht-no-preprune = {true, **false**}; oht-split-confidence = (1×10^{-7}, **0.02**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.13**, 0.5); orb-decay-factor = (0.9, **0.98**, 0.999); orb-l_0 = (1, **1.73**, 20); orb-l_1 = (1, **10.09**, 20); orb-m = (1.1, **1.10**, e); orb-window-size = (50, **84**, 200); orb-n = (3, **5**, 7); orb-th = (3, **0.39**, 5); orb-waiting-time = (90, **102**, 180);

59. Classifier = **ORB-OHT**; dataset = **jgroups**; oht-grace-period = (100, **140**, 500); oht-leaf-prediction = {**mc**, nb, nba}; oht-n-estimators = (10, **37**, 40); oht-no-preprune = {**true**, false}; oht-split-confidence = (1×10^{-7}, **4.151**$\times$**10^{7}**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.26**, 0.5); orb-decay-factor = (0.9, **0.97**, 0.999); orb-l_0 = (1, **9.59**, 20); orb-l_1 = (1, **5.23**, 20); orb-m = (1.1, **1.64**, e); orb-window-size = (50, **109**, 200); orb-n = (3, **4**, 7); orb-th = (3, **0.48**, 5); orb-waiting-time = (90, **125**, 180);

60. Classifier = **ORB-OHT**; dataset = **neutron**; oht-grace-period = (100, **383**, 500); oht-leaf-prediction = {mc, **nb**, nba}; oht-n-estimators = (10, **20**, 40); oht-no-preprune = {true, **false**}; oht-split-confidence = (1×10^{-7}, **0.07**, 0.5); oht-split-criterion = {gini, information gain, **hellinger**}; oht-tie-threshold = (0.05, **0.34**, 0.5); orb-decay-factor = (0.9, **0.92**, 0.999); orb-l_0 = (1, **3.09**, 20); orb-l_1 = (1, **4.13**, 20); orb-m = (1.1, **1.55**, e); orb-window-size = (50, **97**, 200); orb-n = (3, **4**, 7); orb-th = (3, **0.47**, 5); orb-waiting-time = (90, **124**, 180);