

Online Cross-Project Prediction of Defect-Inducing Software Changes

By

Sadia Tabassum

ORCID: 0000-0002-5096-7100

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham, UK
July 2022

UNIVERSITY OF
BIRMINGHAM

University of Birmingham Research Archive

e-theses repository

This unpublished thesis/dissertation is copyright of the author and/or third parties. The intellectual property rights of the author or third parties in respect of this work are as defined by The Copyright Designs and Patents Act 1988 or as modified by any successor legislation.

Any use made of information contained in this thesis/dissertation must be in accordance with that legislation and must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the permission of the copyright holder.

© Copyright by SADIA TABASSUM, 2022

All Rights Reserved

ABSTRACT

Just-In-Time Software Defect Prediction (JIT-SDP) is concerned with predicting whether software changes are defect-inducing or clean, based on machine learning classifiers. JIT-SDP operates in an online scenario where training examples are continuously received over time, necessitating the updating of the JIT-SDP models with the new data. Building JIT-SDP models require a sufficient amount of training data that is not available at the beginning of a software project. Cross-Project (CP) JIT-SDP can address the problem of a lack of training data required for JIT-SDP models to perform effectively at the beginning of the projects. However, such approaches have never been investigated in realistic online learning scenario where the JIT-SDP models need to update with both incoming Cross-Project (CP) and Within-Project (WP) software changes over time. Since there are no online CP JIT-SDP approach available, it is unknown how useful CP data can be in this circumstance. In particular, it's unclear whether CP data are just valuable in the project's early stages, when there aren't many WP data, or if they might be beneficial for an extended period of time. Besides, based on the model's learning process, many existing JIT-SDP studies used ensemble learners that consisted of either offline or online base models. It is unknown whether adapting offline models to operate in an online scenario can provide advantages over online models and whether such advantages would reflect on CP learning. Most machine learning models are also sensitive to hyper-parameter choice and may underperform if hyper-parameters are not configured properly. Tuning methods used for offline learning are not suitable for online learning because in online learning, the best hyper-parameter choice may

change over time. It is unknown whether the best hyper-parameter configuration changes over time for online JIT-SDP. If it does, then an online hyper-parameter tuning approach for JIT-SDP would be essential to improve its predictive performance.

This thesis aims to address the aforementioned issues in JIT-SDP. We conduct the first investigation of when and to what extent CP data is helpful for JIT-SDP in a realistic online scenario. For that, we propose three novel online CP JIT-SDP approaches, which can be updated to take into account incoming CP and WP training examples. Additionally, we provide a comprehensive evaluation of the predictive performance and computational costs of online and offline learning models in online CP JIT-SDP scenario. Furthermore, we provide an online hyper-parameter tuning method for online JIT-SDP that can recommend the best hyper-parameter choices over time.

DEDICATION

To my family

ACKNOWLEDGMENTS

The completion of this doctoral thesis has been an incredibly enriching and transformative journey. This work reflects the valuable experiences that have shaped both my academic growth and personal development.

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Leandro Minku. His constant support and guidance have enabled me to refine my research skills, enhance my attention to detail, and cultivate resilience in the face of challenges. Moreover, he has generously imparted his invaluable advice, steering me towards achieving highest standards in my research endeavors. He has established a benchmark of excellence and dedication that I aspire to mirror in my own pursuits.

I would also like to extend my appreciation to the members of my thesis group, Dr. Rami Bahsoon and Prof. Xin Yao, for their thoughtful insights and constructive feedback. Their encouragement and support during the thesis group meetings have been sources of motivation. Also, thanks to Dr. George Cabral for the valuable support and discussions during our collaboration on the SPDISC project.

Thanks to the administrative team at the School of Computer Science, especially Sarah Brookes. Also, thanks to Dr. Peter Hancox for introducing me to research skills.

I would like to express my deepest gratitude to my wonderful family for being there for me. Special thanks to Saiful Ahmed Chowdhury, Shaheda Edib, Fahim Masud Choudhury, Manha Mahveen Ahmed and Kafil Ahmed for their love, encouragement and support. My family has consistently been my pillars of strength, providing me with the support I required

during this PhD journey. I've always aspired to be a motivating force for my daughter, encouraging her to dream big and nurture her ambition. Remarkably, I've also received an immense amount of love and encouragement from her, which has provided me with the strength to persevere through out this journey. She has always been my biggest cheerleader.

Furthermore, I'd like to honour the memory of my late Nana (grandfather), an educator and scholar, who greatly inspired me from childhood, as well as my late Nanu (grandmother).

I'd also like to acknowledge my fellow PhD friends from office 144 of the Computer Science building; our casual conversations often served as a source of stress relief. I'm grateful for my friends and close ones who encouraged me and lent a listening ear when I needed.

Finally, I'd like to give credit to my stubborn and determined younger self, who was never afraid to dream big, and persisted through multitude of challenges. I am profoundly grateful to the Almighty for bestowing upon me abundant blessings.

Contents

Page

1 Introduction **1**

1.1 Definitions 5

1.1.1 Data Stream 5

1.1.2 Test Example 6

1.1.3 Training Example 6

1.1.4 Online Scenarios with Verification Latency 6

1.1.5 Concept Drift 7

1.1.6 Online learning 8

1.1.7 Offline learning 9

1.1.8 Within-Project (WP) Scenario 10

1.1.9 Cross-Project (CP) Scenario 10

1.1.10 Other-Project (OP) Scenario 11

1.2 Aims and Objectives 11

1.3 Thesis Contribution 13

1.4 Publications 15

1.5 Thesis Roadmap 15

2 Background **18**

2.1 Base Learners 18

2.1.1 Offline Base Learners 19

2.1.2 Online Base Learners . . . 19
2.1.3 Hyperparameters of the Base Learners . . . 21
2.2 Ensemble Learning . . . 24
2.2.1 Oversampling-based Online Bagging (OOB) . . . 25
2.2.2 Oversampling Rate Boosting (ORB) . . . 26
2.3 Performance Evaluation . . . 29
2.3.1 Predictive Performance . . . 29
2.3.2 Statistical Tests and Effect Size . . . 34

3 Literature Review **35**
3.1 Software Defect Prediction . . . 36
3.1.1 Component-Based SDP . . . 36
3.1.1.1 WP Component-Based SDP . . . 36
3.1.1.2 CP Component-Based SDP . . . 38
3.1.2 Offline JIT-SDP . . . 41
3.1.2.1 WP JIT-SDP . . . 41
3.1.2.2 CP JIT-SDP . . . 45
3.1.3 Online JIT-SDP . . . 48
3.2 Hyper-Parameter Tuning . . . 50
3.2.1 AutoML . . . 51
3.2.2 Offline Hyper-Parameter Tuning Approaches . . . 53
3.2.2.1 Grid search . . . 53
3.2.2.2 Random search . . . 53
3.2.2.3 Gradient-based Optimisation . . . 54
3.2.2.4 Bayesian optimisation . . . 54
3.2.2.5 Evolutionary Algorithms . . . 55
3.2.3 Studies Analysing Hyper-Parameter Tuning Approaches . . . 56

3.2.3.1 Offline Hyper-Parameter Tuning in SDP 59
3.2.3.2 Offline Hyper-Parameter Tuning in JIT-SDP 62
3.2.3.3 Chunk-based Hyper-Parameter Tuning Approaches 63
3.2.3.4 Online Hyper-Parameter Tuning Approaches 64
3.3 Summary 65

4 Datasets **67**
4.1 SZZ Algorithm 67
4.2 Open source projects 68
4.3 Proprietary projects 75
4.4 Reasons for concept drift and performance drops in JIT-SDP models 76

5 Online Cross-project JIT-SDP **79**
5.1 Introduction 79
5.2 Online CP JIT-SDP Approaches 83
5.2.1 All-in-One (AIO) Approach 83
5.2.2 Ensemble Approach 85
5.2.3 Filtering Approach 87
5.3 Experimental Setup 89
5.3.1 Compared Approaches 90
5.3.2 Predictive Performance 92
5.3.3 Identifying Initial Periods and Periods of Sudden Drops in Predictive Performance 93
5.3.4 Statistical Tests and Effect Size 95
5.3.5 Hyper-Parameter Tuning 96
5.4 Experimental Results and Discussion 97
5.4.1 Preliminary Analysis 97
5.4.2 RQ1.1: Initial Phase of the Project 101

5.4.3 RQ1.2: Periods with Sudden Drops in WP Classifier's Predictive Performance . . . 106
5.4.4 RQ1.3: Effects of CP Data on the Predictive Performance During Stable Periods of The Projects . . . 112
5.4.5 RQ1.4: Overall effect of using CP and WP data together on the predictive performance . . . 115
5.4.6 RQ1.5: The Need for Updating CP Approaches Over Time . . . 121
5.4.7 Characteristics of Dropped CP Instances in Filtering approach . . . 124
5.5 Sensitivity Analysis . . . 129
5.6 Analysis of Computational Cost . . . 134
5.7 Implications to Practice . . . 136
5.8 Threats to Validity . . . 138
5.9 Summary . . . 139

6 Impact of Online and Offline Base Learners on Online Cross-project JIT-SDP Scenarios 141

6.1 Introduction . . . 141
6.2 BORB Algorithm . . . 146
6.3 Experimental Setup . . . 152
6.3.1 Predictive Performance . . . 153
6.3.2 Statistical Tests and Effect Size . . . 153
6.3.3 Hyper-Parameter Tuning . . . 154
6.4 Experimental Results and Discussion . . . 154
6.4.1 RQ2.1: Can offline learning help to improve predictive performance in online WP JIT-SDP scenarios? Which base learners usually perform best? . . . 155

6.4.2 RQ2.2: How beneficial is CP data to improve predictive performance of offline models compared to online models in online CP JIT-SDP scenarios? . . . 162
6.4.3 RQ2.3: How high is the computational cost of offline learning in online scenarios compared to that of online learning models? . . . 164
6.4.3.1 Comparison of computational cost between offline and online models . . . 165
6.4.3.2 Comparison of computational cost between best offline and online models in terms of predictive performance . . . 168
6.5 Threats to Validity . . . 171
6.6 Summary . . . 172

7 Online Hyper-Parameter Tuning in JIT-SDP 174
7.1 Introduction . . . 174
7.2 Proposed Approach . . . 177
7.2.1 Hyperon . . . 177
7.2.2 ADE-ACM . . . 181
7.3 Experimental Setup . . . 183
7.3.1 Compared Approaches . . . 184
7.3.2 Predictive Performance . . . 185
7.3.3 ORB, Grid Search and Hyperon Hyper-parameters . . . 186
7.3.4 Statistical Tests and Effect Size . . . 187
7.3.5 Labelling Procedure of the Training Examples . . . 187
7.4 Experimental Results and Discussion . . . 188
7.4.1 RQ3.1: What is the impact of hyper-parameters on the performance of online JIT-SDP? . . . 188

7.4.2 RQ3.2: How to automatically tune hyper-parameters for JITSDP in an online manner? How well does this approach perform in terms of suggesting appropriate hyper-parameters to improve JIT-SDP's predictive performance? . . . 190
7.4.3 RQ3.3: How high is the computational cost of the online hyper-parameter tuning approach? In particular, how feasible is the new approach considering the trade-off between predictive performance improvement and runtime? . . . 196
7.5 Threats to Validity . . . 197
7.6 Summary . . . 198

8 Conclusions and Future Work 199
8.1 Conclusions . . . 199
8.1.1 Online Cross-project JIT-SDP . . . 199
8.1.2 Impact of Online and Offline Base Learners on Online Cross-project JIT-SDP Scenarios . . . 202
8.1.3 Online Hyper-parameter Tuning for JIT-SDP . . . 204
8.2 Future Work . . . 205

A Supplementary materials for Chapter Two 208
A.1 Computing four quadrants of confusion matrix . . . 208

B Supplementary materials for Chapter Four 210
B.1 Summary statistics and input features plots for open source datasets . . . 210
B.1.1 Summary statistics tables for opensource datasets . . . 212
B.1.2 Plots for input features for opensource datasets . . . 216

C Supplementary Materials for Chapter Five 236
C.1 Tables reporting G-Mean, Recall0 and Recall1 for Chapter Five . . . 236

C.2 Tables reporting FPR, FNR, Balanced Accuracy and Informedness for open source and proprietary data for Chapter Five including overall, drop, stable and initial periods 249
C.3 Plots of Predictive Performance Through Time 282
C.4 Hyper-parameter Sensitivity Analysis 284

D Supplementary materials for Chapter Six **287**
D.1 Hyper-parameter values investigated for hyper-parameter tuning in Chapter Six 287
D.2 Online approach and offline approach comparison 289
D.3 Recall0, Recall1, FPR and FNR for Chapter Six 292

E Supplementary materials for Chapter Seven **298**
E.1 Recall0, Recall1, FPR and FNR for Chapter Seven 298

References **300**

List of Figures

5.1 G-Mean for open source datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods. 103
5.2 G-Mean for proprietary datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods. 104
5.3 G-Mean for all datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods. 108
5.4 G-Mean for all datasets through time using ORB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods. 117
5.5 Plots of Marginal Means for Filtering-OOB's Factors K, $maxDist$ and $dataset$ (Opensource data). 132
5.6 Plots of Marginal Means for Filtering-OOB's Factors K, $maxDist$ and $dataset$ (Proprietary data). 133

6.1 OBF 151
6.2 G-Mean for all datasets through time for best ranked BORB and ORB approaches with WP data. 160

6.3 G-Mean for all datasets through time for best ranked BORB and ORB approaches with WP and CP data. 161
6.4 Computational Cost Analysis for BORB and ORB 169

7.1 G-Mean for all datasets through time for best hyper-parameter combination with other hyper-parameter combinations based on initial 3000 commits. . . 189
7.2 New hyper-parameter combinations discovered by Hyperon over time that obtained best g-mean. The point corresponding to a given hyperparameter combination is plotted when this combination is selected to be used for predictions and is omitted in the consecutive time steps where it is used, to improve readability of the plots. 191
7.3 G-Mean for all datasets through time using Hyperon, Grid search and other 242 combinations. 192

B.1 Plots for numerical and categorical input features for Tomcat 217

C.1 G-Mean for all datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods. 282
C.2 G-Mean for all datasets through time using ORB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods. 283
C.3 Plots of Marginal Means for Filtering-ORB's Factors K, $maxDist$ and $dataset$ (Open source data) 284
C.4 Plots of Marginal Means for Filtering-ORB's Factors K, $maxDist$ and $dataset$ (Proprietary data) 284
C.5 Plots of Marginal Means for Filtering-ORB's Factors $cpqSize$, $maxDist$ and $dataset$ (Proprietary data) 284

D.1 G-Mean for Online learning (ORB-OHT) and Offline learning (BORB-IHF) approach 291

List of Tables

4.1 Kappa scores 70
4.2 An overview of the projects (Open source and Proprietary) 72
4.3 Change Metrics (adapted from [6]) 74

5.1 Number of initial time steps of the initial phase, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare OOB-based approaches on this initial phase for open source data 102
5.2 Number of initial time steps of the initial phase, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare OOB-based approaches on this initial phase for proprietary data 102
5.3 Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB for open source data during the drop periods . . 107
5.4 Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB for proprietary data during the drop periods . . 107
5.5 Number of time steps of the stable periods, and average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare OOB-based approaches on stable periods for open source data 111

5.6 Number of time steps of the stable periods, and average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare OOB-based approaches on stable periods for open source data 113
5.7 Number of time steps of the stable periods, and average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare OOB-based approaches on stable periods for proprietary data 113
5.8 Overall predictive performance, A12 effect sizes and Scott-Knott.BA12 statistical tests to compare learning approaches for open source data 116
5.9 Number of dropped, reused and completely dropped CP instances for open source datasets 125
5.10 Summary statistics for dropped CP instances for open source datasets compared to wp instances (in percentage) 126
5.11 Summary statistics for dropped CP instances for open source datasets compared to wp instances (in total count) 127
5.12 Min, max, median percentage for dropped CP instances for open source datasets compared to wp instances 128
5.13 ANOVA and Effect Size Results for Open Source Data 130
5.14 ANOVA and Effect Size Results for Proprietary Data 131
5.15 Runtime Analysis of CP Approaches 134

6.1 Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare BORB with WP data using different base learners 156
6.2 Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare ORB with WP data using different base learners 156

6.3 Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare BORB with CP data using different base learners 157
6.4 Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare ORB with CP data using different base learners 157
6.5 Average run time in seconds and Scott-Knott.BA12 ranking for BORB with WP data 166
6.6 Average run time in seconds and Scott-Knott.BA12 ranking for ORB with WP data 166
6.7 Average run time in seconds and Scott-Knott.BA12 ranking for BORB with CP data 167
6.8 Average run time in seconds and Scott-Knott.BA12 ranking for ORB with CP data 167
6.9 Computational Cost Analysis for ORB-OHT-CP 170

7.1 Average G-Means of Hyperon, Grid search and Dummy classifier, A12 effect sizes, Scott-Knott.BA12 to compare Hyperon, Grid search and Dummy classifier. 194
7.2 Computational Cost Analysis for Hyperon and Grid search 196

B.1 Percentage of negative values for features lt, age and rexp 211
B.2 Summary statistics for dataset 'Tomcat' 212
B.3 Data Statistics for JGroups 212
B.4 Data Statistics for Camel 213
B.5 Data Statistics for Brackets 213
B.6 Data Statistics for Nova 213
B.7 Data Statistics for Fabric8 214
B.8 Data Statistics for Spring-Integration 214

B.9 Data Statistics for Npm 214
B.10 Data Statistics for Neutron 215
B.11 Data Statistics for Broadleaf 215

C.1 Number of initial time steps of the initial phase, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data 237
C.2 Number of initial time steps of the initial phase, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB 238
C.3 Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB and ORB for open source data during the drop periods 239
C.4 Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB and ORB for proprietary data during the drop periods 239
C.5 Number of time steps of the stable periods, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this stable periods for OOB and ORB for open source data 240
C.6 Number of time steps of the stable periods, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this stable periods for OOB and ORB for proprietary data 241
C.7 Overall predictive performance, A12 effect sizes and Scott-Knott.BA12 statistical tests to compare learning approaches for proprietary data 242

C.8 Number of initial time steps of the initial phase, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data 243
C.9 Number of initial time steps of the initial phase, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data 243
C.10 Number of time steps of the drop periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data 243
C.11 Number of time steps of the drop periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data 244
C.12 Number of time steps of the stable periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data 244
C.13 Number of time steps of the stable periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data 245
C.14 Number of initial time steps of the initial phase, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data 245
C.15 Number of initial time steps of the initial phase, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data 246
C.16 Number of time steps of the drop periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data 246

C.17 Number of time steps of the drop periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data 247
C.18 Number of time steps of the stable periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data 247
C.19 Number of time steps of the stable periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data 248
C.20 FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for opensource data 249
C.21 FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for opensource data 251
C.22 FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Opensource data 253
C.23 FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Opensource data 255
C.24 FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Opensource data 257
C.25 FPR, FNR, BalancedAccuracy, Informedness for Drift period using ORB for Opensource data 259
C.26 FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using OOB for Opensource data 261
C.27 FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using ORB for Opensource data 263
C.28 FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for Proprietary data 265

C.29 FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for Proprietary data 267
C.30 FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Proprietary data 269
C.31 FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Proprietary data 271
C.32 FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Proprietary data 273
C.33 FPR, FNR, BalancedAccuracy, Informedness for Drift period using ORB for Proprietary data 275
C.34 FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using OOB for Proprietary data 276
C.35 FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using ORB for Proprietary data 279
C.36 ANOVA and Effect Size Results for Filtering-ORB (Opensource Data) . . . 285
C.37 ANOVA Analysis of Hyperparameters for Filtering-ORB (Proprietary Data) 286

D.1 List of hyper-parameter values explored in hyper-parameter tuning using random search 288
D.2 Data points required by ORB-OHT to reach average g-mean of BORB-IHF for WP data 289
D.3 Average Recall0 for ORB with WP data using different base learners 292
D.4 Average Recall1 for ORB with WP data using different base learners 292
D.5 Average Recall0 for BORB with WP data using different base learners 293
D.6 Average Recall1 for BORB with WP data using different base learners 293
D.7 Average Recall0 for ORB with CP data using different base learners 293
D.8 Average Recall1 for ORB with CP data using different base learners 294

D.9 Average Recall0 for BORB with CP data using different base learners 294
D.10 Average Recall1 for BORB with CP data using different base learners 294
D.11 FPR for BORB-WP 295
D.12 FPR for BORB-CP 295
D.13 FPR for ORB-WP 295
D.14 FPR for ORB-CP 296
D.15 FNR for BORB-WP 296
D.16 FNR for BORB-CP 296
D.17 FNR for ORB-WP 297
D.18 FNR for ORB-CP 297

E.1 Recall0, Recall1, FPR and FNR for Hyperon and Grid search 299

Acronyms

ADE Asynchronous Differential Evolution.

$ADE - ACM$ Asynchronous Differential Evolution With Adaptive Correlation Matrix.

AIO All-in-One.

$BORB$ Batch Oversampling Rate Boosting.

$CMA - ES$ Covariance Matrix Adaption Evolutionary Strategy.

CP Cross-Project.

DE Differential Evolution.

$G - Mean$ Geometric Mean.

GA Genetic Algorithm.

IHF Iterative Hoeffding Forest.

IRF Iterative Random Forest.

$JIT - SDP$ Just-in-Time Software Defect Prediction.

LR Logistic Regression.

MLP Multilayer Perceptron.

NB Naïve Bayes.

OHT Oza Bagging of Hoeffding Trees.

OOB Oversampling-based Online Bagging.

OP Other-Project.

ORB Oversampling Rate Boosting.

RQ Research Question.

SDP Software Defect Prediction.

SVM Support Vector Machine.

WP Within-Project.

Chapter One

Introduction

Software systems are nowadays an essential part of our everyday lives. The structures of these systems have been growing larger and ever more complex to fulfill the increasing demands from uncountable different sectors. Hence, ensuring quality of software systems has become a very critical task and improving the software quality highly depends on reducing the amount of software defects [1]. This requires allocating sufficient and appropriate quality assurance activities. As software companies often have limited budget and time constraints [2], accommodating enough resources for finding and fixing defects is challenging.

Software Defect Prediction (SDP) is a popular research area in the Software Engineering domain [3, 4, 5]. The main objective of software defect prediction is to predict which parts of the software are most likely to contain defects so that resources such as time and budget can be more effectively allocated to support software quality assurance activities. Most of the early work on SDP focused on predicting defect-inducing files or modules, which we refer to as component-based SDP. In recent years, another branch of SDP has emerged that focuses on predicting defect-inducing software changes. This is known as Just-in-Time Software Defect Prediction (JIT-SDP) [6]. JIT-SDP predicts whether a change in the code

will induce defects or not as soon as it is committed to a software repository. The main advantages of JIT-SDP over traditional file level prediction are: (i) the prediction is performed at fine granularity as changes usually impact small code block which require less effort to inspect than files or modules; (ii) the defect (or bug) fixing task can be assigned to the right developer as changes can be easily mapped to the person who committed that change; and (iii) defect prediction is made immediately after committing the change so that the code is still fresh in the developer's mind, i.e., there's no need to wait until the project is finished [7].

From a Machine Learning perspective, JIT-SDP can be considered as a binary classification problem that classifies software change as either defect-inducing or clean based on software change data. Software change data are usually consist of a number of input features such as *number of modified directories*, *number of modified files*, and *number of developers*. McIntosh and Kamei [8] showed that there can be fluctuations in the importance of the characteristics of defect-inducing software changes over time, which may be a result of concept drift during the software development process. Concept drift can be described as a change in the data generating process, affecting the underlying probability distribution of the data. Due to concept drift, the relationship between the input features and the output variable (defect-inducing or clean class) may change over time. It may negatively impact predictive performance of the models, if they are predominantly built on old data. To deal with this issue, it is important for models to be able to learn and adapt to new data over time.

A major factor that hinders the use of JIT-SDP from its real-world use is that most of the existing JIT-SDP studies implicitly assume an offline scenario, where a pre-existing training set is available beforehand and additional training examples are never received anymore [7, 9]. However, in practice, JIT-SDP operates in an online scenario, where software changes become labelled as either clean or defect-inducing and become available as training data over time. Very few studies before this research started that considered JIT-SDP as an

online learning problem. This thesis thus focuses on online JIT-SDP.

Most of the JIT-SDP studies are investigated in Within-Project (WP) settings [6] which means models are trained using historical data of software defects from the same software project. A major drawback of WP setting occurs when a project has little or insufficient historical data, which is often the case when a new project starts. Due to lack of sufficient training data, models may perform poorly.

Moreover, similar to SDP, JIT-SDP is a class-imbalanced problem [10]. In class-imbalanced problems, one class has significantly more training examples than the other. Because of very few examples of minority class, the model can not be trained properly to predict both majority and minority classes well. This may also result in poor model performance. To deal with such issues, prediction models can benefit from learning existing available resources of other projects [11]. This is known as Cross-Project (CP) learning. When there are not enough WP training data at the initial phase, or there are very few examples of the minority class, CP learning may provide the model with enough training data, including sufficient examples of the minority class, which may help the model achieve better predictive performance.

CP data was investigated for offline JIT-SDP [12, 13]. In existing offline CP JIT-SDP studies, only CP data were used to build the classifier and the classifier was never trained with any WP data. However, in a realistic online scenario, both CP and WP training examples arrive over time in the form of a data stream. If incoming CP data is used for training a CP classifier, there is no reason to exclude incoming WP data from the training process of this classifier, as such WP data should not harm its predictive performance. Hence, a CP JIT-SDP model that considers an online scenario and is able to update with both CP and WP data over time, would be more suitable to adopt in practice than an offline CP JIT-SDP model. However, CP JIT-SDP approaches have never been investigated in realistic online

learning scenarios, where WP software changes arrive continuously over time and can be used to update the classifiers. This thesis will investigate the benefit of CP data in online JIT-SDP.

Based on the learning process, prediction models can be offline and online. The offline learning models are trained once and never get updated, whereas online learning models can be continuously updated with each new incoming data instance. Both online and offline learning models can be used to learn additional data received over time in online scenarios. Online learning models naturally fit into online scenario. Online learning models do not use retraining, hence, they can be faster to run. But such models may provide poor predictive performance than offline learning models when the training data is not large. Offline learning models can be adapted to the online scenario by retraining the model with new data combined with a sufficient amount of past data. Offline learning models could be robust compared to online learning models. As no existing study used offline learning models for online JIT-SDP, it is unknown when and to what extent the use of offline learning models in the online JIT-SDP scenario would be useful to improve the predictive performance. This thesis will thus investigate the benefit of online JIT-SDP learning compared to offline JIT-SDP learning.

In addition, characteristics of most defect prediction models are controlled by hyper-parameter configurations. Models can be sensitive to hyper-parameters. Recent studies show that prediction models can under-perform because of sub-optimal default parameter settings [14]. Hyper-parameters can be configured manually based on existing literature or trial and error [15]. But it is difficult to explore all the possible hyper-parameter settings manually. Hence, tuning hyper-parameters automatically is a possible solution. Though offline learning has procedures for automatically tune hyper-parameters, there is lack of research on automatically tuning of hyper-parameters in online learning.

It is particularly challenging to tune hyper-parameter in online learning scenario be-

cause if there is not enough data beforehand, different hyper-parameter choices can not be evaluated. Also, the best choice of hyper-parameter based on the initial portion of the data may not be the best choice in later periods because of the change in the data generation process [16]. It is important to make the tuning procedure automated for practitioners who may not be knowledgeable in machine learning. Online JIT-SDP could potentially achieve better predictive performance using an automated hyper-parameter tuning approach, which is still unexplored. This thesis proposes a novel online hyper-parameter tuning approach and investigate it in the context of JIT-SDP.

1.1 Definitions

This section presents some basic definitions for JIT-SDP used in this thesis.

1.1.1 Data Stream

A data stream is a potentially infinite sequence of training examples $\mathcal{S} = \{(\vec{x}_i, y_i)\}_{i=1}^{\infty}$, where i is a natural sequential number (time step) indicating the order with which the training examples were labeled, $\vec{x}_i$ are the input features describing example received at time step i, and y_i is the label of the example which can be defect-inducing or clean. In JIT-SDP, the input features are features describing the software change. In this thesis, we have used 14 input features recommended by Kamei et al. [7]: NS (number of modified subsystems), ND (number of modified directories), NF (number of modified files), Entropy (distribution of modified code across each file), LA (lines of code added), LD (lines of code deleted), LT (lines of code in a file before the change), FIX (whether or not the change is a defect fix), NDEV (number of developers that changed the modified files), AGE (average time interval between the last and the current change), NUC (number of unique changes to the

modified files), EXP (developer experience), REXP (recent developer experience) and SEXP (developer experience on a subsystem). These will be explained in Chapter 4.

1.1.2 Test Example

A test example is a software change for which a JIT-SDP model needs to provide a prediction about whether it is clean or defect-inducing. A test example can be represented by a vector of input features $\vec{x_t}$ related to that change. Details about the input features used for this study can be found in the Chapter 4. Each software change in a target project corresponds to a test example created at commit time, producing a test data stream.

1.1.3 Training Example

A training example d_t=($\vec{x}_t, y_t$) consists of a vector of input features $\vec{x}_t$ and the class label y_t associated with it, where t is the training time step. Class label y_t is represented by 0 (if clean) or 1 (if defect-inducing). Here, training time step is the sequential index representing the order of arrival of training examples. Software changes compose training examples when their labels become available, producing a training data stream ordered by the labeling time.

1.1.4 Online Scenarios with Verification Latency

An online scenario is a scenario where training examples are produced over time, forming a data stream. JIT-SDP operates in an online scenario where the labels of the software changes arrive with a delay, which is referred to as verification latency [17]. Specifically for JIT-SDP, labelled examples can be produced following the procedure defined by Cabral et al. [18]. When a change is committed to the repository, the developers hope it to be clean,

but it may, instead, induce a defect. To label this change as clean, we need to wait for a period of time (waiting period w) to be confident that the change is really clean. If no defect is reported to be associated to this change within w days, the change is labeled as clean at the end of the waiting period, producing a training example. If, on the other hand, a defect is found to be linked to this change during these w days, the change is immediately labeled as defect-inducing, without having to wait until the end of the waiting period. It may also happen that a change that was initially labeled as clean is found to be defect-inducing after the w days. When this happens, this change is relabeled as defect-inducing, producing a new labeled training example of the defect-inducing class. This procedure respects chronology, being able to capture a realistic scenario that reflects the labelling procedure that would be observed in practice. The waiting period w can be considered as a pre-defined parameter.

1.1.5 Concept Drift

A concept is a joint probability distribution ($p_t(\vec{x}, y) = p_t(y|\vec{x})p_t(\vec{x}) = p_t(\vec{x}|y)p_t(y)$) underlying a machine learning problem at a given time step t. In JIT-SDP, this represents the state of the defect-generating process at commit time step t. It represents the underlying function that encapsulates the relationship between features and classes, the chance of observing examples from each class, and the chance of observing each feature value [19].

Concept drift refers to a change in the concept over the time. When there are concept drifts, the joint probability distribution between two different time steps changes. For example, if for a time step t and $t + \Delta$, $p_t(\vec{x}, y) \neq p_{t+\Delta}(\vec{x}, y)$, then we can say that concept drift has occurred. There are different types of concept drift that occurs in JIT-SDP [19]: 1) Change in the ratio of examples of each class being predicted $p(y)$. 2) Evolving probability of observing different feature values given the class, i.e., change in the posterior probability $p_t(\vec{x}, y)$. This evolution can be of following types: a)Changes in how likely an example be-

longs to a given class, given its feature values, i.e., $p_t(y|\vec{x})$. An example of such change is – the features that would typically be associated with clean class, now may be associated with defect-inducing class. b) Changes in how frequently different feature values are being observed, i.e., changes in the probability distribution of the features $p(\vec{x})$. This means typical feature values vary over time.

Concept drift typically causes drops in the predictive performance resulting in unstable predictive performance over time, which have been observed in previous JIT-SDP works [18, 20]. Therefore, JIT-SDP models should attempt to minimise these drops in the predictive performance in order to produce a stable predictive performance over time.

1.1.6 Online learning

Consider a data stream $D_i = (x_i, y_i)_{i=1}^{\infty}$, where $(x_i, y_i) \sim p_i(x, y)$, $x_i \, \epsilon \, X$ and $y_i \, \epsilon \, Y$. At time step i, a machine learning algorithm has access to the model $f_{i-1} : X \rightarrow Y$ trained on past data D_{i-1}. Learning in an online scenario considers building an updated model $f_i : X \rightarrow Y$ that is able to generalise to unseen examples drawn from $p_i(x, y)$.

Given a data stream formed by training examples ordered by the time they were produced $\mathcal{S} = \{(\vec{x}_i, y_i)\}_{i=1}^{\infty}$, an online learning model is a model that is immediately updated whenever a new training example $(\vec{x}_i, y_i) \in \mathcal{S}$ becomes available. Strict online learning models must be able to process (learn) each training example only once. So, the classifier is always updated with new examples, without requiring any retraining on past examples. This is useful to speed up learning for cases where storing and reprocessing past training examples may be computationally infeasible, e.g., for very large data streams, or data streams where the frequency of incoming data is very large. However, some (non-strict) online learning algorithms may access a memory containing past training examples to support the learning

process. The classifier may also have strategies to speed up adaptation to changes (a.k.a., concept drifts) that may affect the underlying data generating process.

1.1.7 Offline learning

Consider a finite training set $\tau_0 = \{(\vec{x}_i, y_i)\}_{i=1}^{n}$ containing n examples. Being a set and not a stream, the time order of these examples is ignored. An offline learning model is trained on τ such thatthe training and testing phases cannot intersect in time, i.e., the classifier is only available to use when the training procedure has ended.

Even though the time order of the examples within τ_0 is ignored by the offline learning procedure, it is still possible to create a sequence of training sets τ_j, $j \geq 0$, where each training set τ_j is updated with new training examples that may become available until time step j, where j is a time step representing the order with which these training sets are created. Each training set j could contain all the training examples observed so far at time j, or a subset of these, depending on the approach adopted. offline learning models could then be created and rebuilt from scratch whenever a new training set becomes available. Even though the training process of the classifier would ignore the time order of examples *within* a given training set, this process would still ensure that only training data that is really available at a given point in time would be used for training, i.e., the JIT-SDP online scenario described in this section would still be respected.

As the data stream generated by the software changes submitted to a software repository is not a high frequency stream, it may be computationally acceptable to store past changes and rebuild classifiers from scratch when new training sets become available. Moreover, managing the whole historical data stream enables us to access all the benefits of offline learning over online learning. In particular, by ignoring the time order of examples, offline

learning models frequently process the training set several times, which can help to produce stronger (more accurate) classifiers. In face of verification latency, revisiting all software changes labeled so far allows us to delete any training examples whose label was incorrectly assigned as clean for software changes that have now been found to be defect inducing. This prevents the classifier to learn noisy information, different from online learning models such as ORB [18], where the mislabeled training example is definitely incorporated into the classifier. Nevertheless, a potential disadvantage of using offline learning for online scenarios is that, this could make it more difficult to deal with certain types of change in the defect generating process, as each given training set may contain a mix of examples produced by different defect generating processes.

1.1.8 Within-Project (WP) Scenario

In the within-project (WP) scenario, the JIT-SDP model is trained by the labeled examples from the target project to predict the test examples from the target project. So, both the test and training data streams are composed of software changes from the same project. We use the term WP when referring to approaches that only use WP data.

1.1.9 Cross-Project (CP) Scenario

In the cross-project (CP) scenario, training examples include both examples from the target project and from other projects. However, the model is used to predict only the test examples from the target project. In online scenarios, both WP and CP training data arrive over time as their collection can be automated. If incoming CP data is used for updating a CP classifier over time, it is natural and reasonable to also use any WP data that may arrive over time. This is because such WP data should not hurt (and could potentially improve) predictive

performance. Therefore, in online learning, we use the term CP when referring to approaches that make use of both incoming CP and WP data for training. This term is adopted with this meaning in our RQs and throughout the rest of this thesis. This means that, for online learning scenarios, when we refer to the benefits of CP data, we do not mean the benefits of CP data in isolation, but the benefits of CP data used along with incoming WP data. Indeed the term CP means Cross-Project, i.e., data coming from multiple projects, and thus there is no reason for this term to exclude WP data in the context of online learning.

1.1.10 Other-Project (OP) Scenario

For offline approaches reviewed in Chapter 3, CP approaches assume that there is no WP data available at the time of model creation. When discussing realistic online learning scenarios in this thesis, we will use the term Other-Project (OP) to refer to approaches that use data that come only from other projects than the project we are interested in. In this thesis, these approaches are CP approaches trained only on CP data that are available before the project of interest commences, i.e., they are CP approaches that have no access to WP data.

1.2 Aims and Objectives

Ideally, JIT-SDP approaches should be such that they can be easily adopted in practice by being able to tackle real-world challenges and also provide improved predictive performance. Therefore, the aim of this thesis is to address some major challenges to build a realistic JIT-SDP approach, which are listed below:

- **Challenge 1:** The lack of an online JIT-SDP approach that can maintain stable predictive performance when there is no or very little WP data at the initial phase

of a project and also during concept drift. In a realistic scenario, often there is lack of training data at the initial phase of the project, which leads to poor predictive performance. Over time, concept drift may also happen and negatively affect predictive performance. The fact that JIT-SDP is a class imbalanced problem further exacerbates this challenges, as there will typically be very few examples of the clean class to learn from and to adapt to concept drifts.

In a realistic online learning scenario, both CP and WP data arrive sequentially over time, and the model should be able to update with both incoming CP and WP data. However, in the existing offline CP JIT-SDP studies, JIT-SDP models are trained with CP data only, and models are never trained with any WP data. Hence, existing offline CP JIT-SDP approaches are not suitable for a realistic online scenario.

- **Challenge 2:** The lack of knowledge about how well online learning models perform compared to offline learning models, and which kind of online and offline base models are most competitive. Existing online JIT-SDP studies use online learning models which may have comparatively poor predictive power (specially, if the training data is not large) as it has only one pass over the data compared to offline models, which use multiple passes to retrain the model. It is unknown how good the existing online learning models are compared to offline learning models in online JIT-SDP scenario.

- **Challenge 3:** There is lack of an automated hyper-parameter tuning approach for online JIT-SDP. In online scenario, the initially chosen best hyper-parameter combination may change over time, which can prevent online JIT-SDP to achieve its optimal predictive performance.

The thesis aims to answer the following research questions with the objectives to find solutions for the above mentioned challenges:

- ***RQ1:*** *Can cross-project learning help to improve predictive performance in online JIT-*

SDP? How?

Objective to tackle challenge 1: Propose an online CP JIT-SDP approach that is able to operate in a realistic online scenario, and can reduce performance drops caused by lack of WP data, class imbalance and concept drift by incorporating CP and WP data together, leading to improved stable predictive performance of the JIT-SDP model.

- ***RQ2:*** *How online base learners perform compared to offline base learners in terms of predictive performance? Can use of different base learners influence the predictive performance of cross-project JIT-SDP approaches?*

  Objective to tackle challenge 2: Provide a detailed understanding of how the predictive performance and computational costs of online learning models compare with offline learning models in online JIT-SDP scenario.

- ***RQ3:*** *Does the best hyper-parameter choice for online JIT-SDP change over time? If so, then how to tune the hyper parameters in online manner?*

  Objective to tackle challenge 3: Propose an efficient online Hyper-Parameter tuning approach for online JIT-SDP that is able to automatically suggest appropriate hyper-parameter combinations over time.

1.3 Thesis Contribution

This thesis makes following novel contributions towards an online cross-project JIT-SDP framework:

- **Novel online cross-project approaches for JIT-SDP:** We provide the first investigation of cross-project (CP) learning for online JIT-SDP. We show how to adapt CP JIT-SDP approaches so that they can be used in online JIT-SDP scenario, leading to

three novel online CP JIT-SDP approaches. We show that CP data combined with WP data can improve the overall predictive performance compared to WP learning in JIT-SDP. We also provide systematic approach to identify performance drop periods in JIT-SDP. We present a thorough analysis of the extent to which CP data might be helpful. We demonstrate that using CP data can significantly improve predictive performance during the initial period, prevent sudden performance drops over time, and improve predictive performance during stable periods of the project.

- **Comparison between offline and online models in a realistic online JIT-SDP scenario:** We provide the first investigation of offline models in online JIT-SDP scenario and the first comparison between offline base learners and online base learners in an online JIT-SDP scenario. We analyse both predictive performance and computational costs of offline and online learning with different base learners. We demonstrate that despite having improved prediction performance for several base learners independent of CP or WP data, the offline model had some fluctuations in predictive performance. Online model performed well with only one base learner which was found to be a more reliable approach than offline models since it offered stable prediction performance. Additionally, the computational cost of using the online model in practice would be feasible.

- **Novel online hyper-parameter tuning approach for online JIT-SDP:** We show that JIT-SDP algorithms do get influenced by hyper-parameter choice and that the best hyperparameter choice varies over time. We propose a novel online hyper-parameter tuning approach for JIT-SDP. This is the first investigation of online hyper-parameter tuning for JIT-SDP. We demonstrate that online hyper-parameter tuning can improve predictive performance of online JIT-SDP. We also show that the proposed online hyper-parameter tuning approach is feasible regarding the computational cost.

1.4 Publications

Partial content of this thesis has been published in the three papers listed below:

- Tabassum, S., Minku, L.L., Feng, D., Cabral, G.G. and Song, L., 2020, October. 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) (pp. 554-565). IEEE. DOI: 10.1145/3377811.3380403 (Chapter five)

- Tabassum, S., Minku, L.L. and Feng, D., 2022. Cross-Project Online Just-In-Time Software Defect Prediction. IEEE Transactions on Software Engineering (in press). DOI: 10.1109/TSE.2022.3150153 (Chapter five)

- Cabral, G.G., Minku, L., Oliveira, A.L., Pessoa, D. and Tabassum, S., 2023. An Investigation of Online and Offline Learning Models for Online Just-in-Time Software Defect Prediction. Empirical Software Engineering. DOI: 10.1007/s10664-023-10335-6 (Chapter six)

1.5 Thesis Roadmap

To understand the organisation of the chapters, we present the thesis road map below:

- **Chapter 2:** This chapter provides essential background information on base learners and their hyperparameters, ensemble methods, performance evaluation methods utilised in this thesis.

- **Chapter 3:** This chapter provides a comprehensive review of existing JIT-SDP studies considering different data combinations (WP, CP), learning scenario (online, offline)

and base models. This chapter also includes existing state-of-the-art practice and studies on hyper-parameter tuning.

- **Chapter 4:** This chapter describes the open source and proprietary data sets used in this thesis. It also includes details about the algorithm used for data collection.

- **Chapter 5:** We introduce a novel online cross-project framework for JIT-SDP. This includes three cross-project approaches adapted to operate in online mode for JIT-SDP. We develop a new methodology to systematically analyse approaches during periods of drop in predictive performance. This allows us to understand how robust proposed CP JIT-SDP approaches are in terms of preventing such drops. We provide an analysis of sensitivity to hyper-parameters for the most promising CP approach. We compared the proposed online CP JIT-SDP approaches with baseline online WP JIT-SDP approach and dummy classifier.

- **Chapter 6:** We show the impact of offline base learners when adapted in an online JIT-SDP scenario. We provide a comparison of predictive performance and computational costs of offline and online base learners for online JIT-SDP.

- **Chapter 7:** In this chapter, we show that predictive performance of online JIT-SDP gets influenced by hyper-parameter choices and that the best hyper-parameter values change over time. We introduce our novel online hyper-parameter tuning approach for JIT-SDP, which is able to suggest appropriate hyper-parameter choices over time by using evolutionary algorithm to create new combinations. This approach improves predictive performance of JIT-SDP compared to existing grid search method. We also provide an analysis of computational cost of the new approach and show that it is feasible enough to adopt in practice.

- **Chapter 8:** In the final chapter, we present a discussion on the major findings of the thesis with respect to the research questions. Also, we provide insights about future

directions of this research.

Chapter Two

Background

This chapter provides relevant background knowledge on different base learners and their hyperparameters, ensemble Learning, performance metrics, and statistical tests used in this thesis. This chapter is organised as follows: Section 2.1 provides an overview of both offline and online base learners and their hyperparameters explored in this thesis. Section 2.2 delves into ensemble learning and introduces two ensemble methods utilized in this thesis, Oversampling-based Online Bagging (OOB) and Oversampling Rate Boosting (ORB). Lastly, Section 2.3 discusses performance evaluation, including performance metrics and statistical tests employed in this thesis.

2.1 Base Learners

This section provides details about the base learners that are investigated with the BORB and ORB approaches in this thesis.

2.1.1 Offline Base Learners

- Logistic Regression (LR): Logistic regression is a well known offline linear classifier [21]. Its training requires iterating through all the training data multiple times (epochs), and it has been successfully used for offline JIT-SDP in previous work [7, 13].

- Multilayer Perceptron (MLP): MLP is an Artificial Neural Network that consists of three layers of interconnected nodes [22], being able to model any function. Training also requires iterating through all the training data multiple times (epochs). It has been included here for being a universal approximator, able to model any function.

- Iterative Random Forest (IRF): IRF consists of an ensemble of CART decision trees [23]. It is similar to a Random Forest [24], but the decision trees are trained with different subsets of the training data to encourage more diversity. It was included here as previous work has shown that diversity is important in ensembles for JIT-SDP [25].

- Iterative Hoeffding Forest (IHF): IHF is the same approach as IRF, but using online Hoeffding trees as the base learners instead of CARTs. As the iterative ensemble approach itself is an offline learning approach, we classify this approach as an offline learning approach. We have adopted it with BORB in this study so that we can evaluate the benefits of the approach BORB itself compared with ORB, without being affected by the benefits of the offline decision tree over the online one. This evaluation can be conducted by comparing BORB-IHF against ORB-OHT.

2.1.2 Online Base Learners

- Logistic Regression (LR): despite Logistic regression being an offline algorithm, it is possible to set the number of epochs to one so that the algorithm becomes online. The downside of using logistic regression as an online learning algorithm is that the

resulting model is likely to become weaker, i.e., to have poorer predictive performance.

- Multilayer Perceptron (MLP): similar to LR, it is also possible to set the number of epochs for training MLPs to one, so that MLPs become online learning models. The downside is similar to that of LR, i.e., its predictive performance may considerably reduce when a single epoch is used.

- Naïve Bayes (NB): NB is a well known Bayesian classifier that can be trained through one pass over the training data. It is included here because it has been successfully used in component-based software defect prediction [26].

- Hoeffding Trees: Many decision tree learners such as ID3, C4.5 and CART require all training examples to be stored simultaneously in the main memory. This limits the number of examples that can be learned by these learners. Disk-based decision tree learners that require examples to be stored in the disk, are also not suitable for complex trees and large datasets. To overcome these issues of decision tree learners, Domingos and Hulten [27] first proposed an algorithm called *Hoeffding tree* for inducing decision trees from a data stream incrementally, which inspects each example in the stream only once to update the tree, without needing to store it. Hoeffding Tree incrementally grows a decision tree based on the fact that a small number of examples can give enough evidence to choose an optimal splitting attribute. In a Hoeffding Tree, a node is expanded as soon as there is enough statistical evidence that an optimal splitting feature exists which is decided by a distribution-independent Hoeffding bound. Hoeffding bound quantifies the number of examples needed at each node [28]. A key property of the Hoeffding tree is that its performance is asymptotically nearly identical to that of a non-incremental learner and the incremental nature of the tree does not significantly affect the quality of trees in a negative way [27].

- Oza Bagging of Hoeffding Trees (OHT): Oza Bagging is an online version of the Bagging

ensemble learning algorithm. It requires a single pass through the training data to learn it. It is typically run with Hoeffding Trees, which are online decision trees suitable for large complex datasets [27]. Hoeffding Trees also learn through a single pass through the training data. Due to its theoretical foundations on the Hoeffding bound, Hoeffding trees are able to produce online models with strong performance guarantees, reason why this approach is being adopted in this and in previous JIT-SDP work [29, 18, 30].

2.1.3 Hyperparameters of the Base Learners

A discussion about different hyperparameters of the base learners used in the thesis is given below:

Hoeffding Tree has several hyperparameters. These hyperperparameters and their domain are listed below:

- Grace_period: number of instances a leaf should observe between split attempts. (positive integer value)
- Split_criterion: separates the data into two regions at a node of the tree. (choices= ['gini', 'info_gain', 'hellinger'])
- Split_confidence: allowed error in split decision. (real value)
- Tie_threshold: threshold below which a split will be forced to break ties. (real value)
- No_preprune: if True, pre-pruning is disabled. (boolean)
- Leaf_prediction: prediction mechanism used at leafs. (choices=['mc', 'nb', 'nba'])
- n-estimators: number of trees. (positive integer value)

Connor et al. [31] shows that if the grace period is set too high it will affect the accuracy, as the initial model built is over fitted to the data.

As the value of Split confidence tends to 0, Hoeffding Tree takes longer time in splitting decision [32]. Prasad et al. [32] showed that for lower values of tie threshold (≤ 0.2) with increments in confidence level, and a tie-threshold with a comparatively higher value (≥ 0.3) has a relatively smaller impact on the improvement of accuracy.

Iterative Hoeffding Forest is an iterative ensemble approach but uses Hoeffding Tree as base learners, and have similar hyperparameters as Hoeffding Trees such as grace_period, split_criterion, split_confidence, tie_threshold, tie_threshold, no_preprune,leaf_prediction and n-estimators.

Iterative Random Forest is based on Random Forest and inherits its hyperparameters as follows:

- criterion: the function to measure the quality of a split. (choices=['entropy', 'gini'])
- min-samples-leaf: the minimum number of samples required to be at a leaf node. (integer value)
- max-features: the number of features to consider when looking for the best split. (integer value)
- n-estimators: the number of trees in the forest. (integer value)

Van and Hutter [33] show that min-samples-leaf and max-features are most important hyperparameters for Random Forest. Trithipkaiwanpon and Taetragool [34] provided a sensitivity analysis of the hyperparameters for Random Forest and showed that accuracy of the model was sensitive to min-samples-leaf.

Following hyperparameters of Logistic regression tuned in this thesis are given below:

- alpha: constant that multiplies the regularization term. Also used to compute the learning rate. (real value)
- l1_ratio: the Elastic-Net mixing parameter. (real value)
- n_epochs: number of epochs performed by the training. (integer value)

Logistic Regression model is regularised by elastic net which combines l1 and l2 penalties. l1_ratio controls the contribution of l1 and l2. If l1_ratio=0, the regularisation term becomes l2 regularisation. When l1_ratio=1, the regularisation term becomes l1 regularisation. The effect of the l1_ratio on logistic regression depends on the dataset and the specific problem being addressed. l1 regularisation can be a useful approach when dealing with datasets where there are a large number of irrelevant features in comparison to the number of training examples [35]. The hyperparameter alpha controls how much weight is given to each of the l1 and l2 penalties. The value of alpha ranges between 0 and 1, and it is used to control the relative contribution of the L1 and L2 penalties in regularized models. Specifically, the L1 penalty is weighted by the alpha value, while the L2 penalty is weighted by one minus the alpha value [36].

Following hyperparameters of Multilayer Perceptron (MLP) tuned in this thesis are given below:

- learning-rate: learning rate for weight updates (real value)
- n-hidden-layers: number of hidden layers (integer value)
- hidden-layers-size: hidden layers size (integer value)
- n_epochs: number of epochs performed by the training

- batch-size: size of minibatches (integer value)
- dropout-input-layer: dropout probability of the input layer (real value)
- dropout-hidden-layers: dropout probability of the hidden layers (real value)

Tran et al. [37] showed that the hyperparameters of MLP can influence the quality of MLP model. For example, using too few nodes in a neural network may result in high error rates, while using too many nodes may cause overfitting. Using batches in training neural networks has advantages such as it requires less memory and improves training speed by updating the weights after each propagation within each batch. For Naive Bayes, a hyperparameter nb-n-updates has been tuned which denotes number of updates performed by the training.

As previous studies suggested that the performance of base learners can be influenced by the selection of hyperparameters, the hyperparameters of the base learners are tuned using Random search (Chapter Six). However, it should be emphasized that the impact of hyperparameter values on the model's performance is reliant on the dataset, which is a common trend observed in Machine Learning algorithms [38].

2.2 Ensemble Learning

In software engineering, many software defect prediction studies used ensemble learning [39, 40, 41]. An ensemble method trains multiple classifiers known as *base learners*, and combines the predictions from all of these base learners to achieve the final prediction. Base learners are usually weak and do not perform well alone because of their high variance or bias. Ensemble learning is effective because it can reduce the variance in the prediction performance without increasing any bias [42]. This results in ensemble classifiers that are typically stronger than their individual classifiers [43]. Some widely used ensemble learning methods for JIT-SDP

includes offline ensemble methods such as Bagging [44] , boosting [45], Random forest [24], and online ensemble methods such as Online bagging and boosting [46].

There are different types of base learners used as part of ensembles in the existing JIT-SDP studies. For offline JIT-SDP studies, Yang et al. [47] investigated 5 base learners - Naive Bayes, Support Vector Machine, Decision Trees, Linear Discriminant Analysis and Nearest Neighbor Classifier for JIT-SDP. They showed that Decision trees performed better than other base learners for 5 out of 6 datasets. Kamei et al. used Random forest in their study on JIT-SDP [9]. Li et al. [25] investigated the impact of different combinations of base learners such as Support Vector Machines (SVMs), Logistic Regression (LR), Random Forest (RF), Multi-layer Perceptron (MLP), Naive Bayes (NB) and Decision Trees (DTs). They showed that the diversity of base learners plays an important role for achieving promising performance. For online JIT-SDP studies, Hoeffding Tree was used as base learners in [18] and [29] and achieved good predictive performance.

Sections 2.2.1 and 2.2.2 will discuss the ensemble methods adopted in this thesis.

2.2.1 Oversampling-based Online Bagging (OOB)

Wang et al. [48] proposed Oversampling-based Online Bagging (OOB) ensemble to deal with imbalanced data streams in an online scenario for real-time prediction. As an online approach, it does not require storing the data and also does not need prior knowledge about the data. OOB is triggered when class imbalance is detected by their class imbalance detection method. OOB is based on Online Bagging which is an online version of a well-known ensemble algorithm Bagging. In Bagging, each training subset for a base learner f_m contains K copies of each original training example. The distribution of K converges to a $Poisson(\lambda)$ distribution with the parameter λ=1 as the number of examples tends to infinity. Therefore,

in online bagging, a training example is presented K times to each ensemble member f_m, where K is drawn from a $Poisson(1)$. OOB combines oversampling with Online Bagging. OOB adjusts the class distribution by increasing the number of minority class examples. If a new example belongs to minority-class, then OOB tunes the parameter λ of Poisson distribution to $1/w_k$ (here w_k is the size percentage of class c_k) which increases K which decides how many times to use this example for training. However, OOB has some weakness such as strategy of setting λ is not consistent with the imbalance degree and can vary with number of classes. For balanced 2-class problem, λ will be 0.5, and for 1-class it will be 2. Also, OOB depends on the class imbalance detection method to trigger resampling. To overcome these issues, Wang et al. [49] proposed an improved version of OOB which presented a better parameter setting strategy. In the improved OOB, they assume two possible classes $Y = +1, -1$. If currently the size of positive class is smaller than the size of the negative class, then λ for positive class will be $w_-^{(t)}/w_+^{(t)}$ for oversampling. This rule automatically results in resampling being performed only when the data becomes imbalanced, without the need for an explicit class imbalance detection method. Both versions of OOB tracks the current imbalance ratio, i.e. the rate $\rho_c^{(t)}$ of examples belonging to each class $c\epsilon\{0, 1\}$ as follows:

$$\rho_c^{(t)} = \acute{\theta}\rho_c^{(t-1)} + (1 - \acute{\theta})(y^{(t)} == c) \tag{2.1}$$

where t is the current time step; if the training example at time t is of class 1, $(y^{(t)} == c)$ returns 1, otherwise 0; and $\acute{\theta}$, $0 < \acute{\theta} < 1$, is a pre-defined time decay factor to tune the emphasis on the more recent data.

2.2.2 Oversampling Rate Boosting (ORB)

A study [18] investigated online JIT-SDP which showed that class imbalance evolution occurs in JIT-SDP and can negatively impact on the prediction performance. It also considers the

fact that there is a delay between the commit time of a change and the time to find its true label. This study proposed a novel online class imbalance learning algorithm called Oversampling Rate Boosting (ORB) which can improve the performance compared to the state-of-the art class imbalance evolution algorithms - Undersampling Online Bagging (UOB) and Improved Oversampling Online Bagging (OOB). ORB is built upon OOB. Problems with UOB and OOB are - these approaches assume that adjusting the ratio of training examples close to 1:1 is enough to obtain a balanced predictive performance. But in reality, one class may be more difficult to learn than the other class which may cause unbalanced prediction performance. Also, there can be noisy or outlier training examples that have same input features but different labels. The negative effect of noisy outliers may be magnified if oversampling is used.

Algorithm 1 Oversampling Rate Boosting (ORB) [18]

Input: Ensemble size n, incoming training example d, parameters of the adjustment function (th, l_0, l_1, m), noise mechanism parameter o, decay factor $\theta^{'}$, window size w_s

for each training example $d^{(t)} = (x^{(t)}, y^{(t)})$, $t \leftarrow 0$ to ∞ **do**
  Obtain the ensemble prediction $y^{(t)}$ for $x^{(t)}$
  Compute the average $ma^{(t)}$ over the predictions on the most recent w_s examples, including $d^{(t)}$
  Update the proportions $\rho_0^{(t)}$ and $\rho_1^{(t)}$ of each class using Eq. 2.1
  for $i \leftarrow 0$ to n **do**
    $\lambda = 1$
    if $y^{(t)} == 1$ and $\rho_1^{(t)} < \rho_0^{(t)}$ **then**
      $\lambda = \rho_0^{(t)} / \rho_1^{(t)}$
    end if
    if $y^{(t)} == 1$ and $\rho_0^{(t)} < \rho_1^{(t)}$ **then**
      $\lambda = \rho_1^{(t)} / \rho_0^{(t)}$
    end if
    Set $k \sim Poisson(\lambda)$
    Calculate $OBF^{(t)}$ $(ma^{(t)}, th, l0, l1, m)$ using Eq. 6.1 or Eq. 6.2
    $k = k \; . \; OBF^{(t)}$
    Run noise safety mechanism with parameter o
    /* Depending on the noise safety mechanism outcome, update the ith Hoeffding tree with k copies of d_t */
    Update(HT_i, k, d_t)
  end for
end for

ORB can deal with these problems. It provides not only a way of balancing the training datasets, but also adjusts the resampling ratio based on the moving average over

the predictions. ORB monitors the model predictions to support the adjustment of the resampling rate. If predictions of a class $c1$ are significantly higher than predictions of the other class $c2$, then the resampling rate of the class $c2$ should be increased to tackle balanced predictive performances. ORB also uses a safety mechanism to deal with noise or outliers that allows the algorithm to skip learning of the current defect-inducing training example.

Algorithm 1 shows the pseudocode of the ORB approach. ORB calculates moving average of the predictions $ma^{(t)}$ using a time window of size w_s. The problem of defect prediction is a binary problem where 0 represents the clean class and 1 represents the defect-inducing class. Calculating moving average allows us to detect prediction bias towards any particular class. Depending on the bias, the resampling rate of one of the classes is boosted (increased). After that, procedures similar to OOB are performed to update the proportions of each class $\rho_0^{(t)}$ and $\rho_1^{(t)}$, and resampling rate λ. The resampling rate determines k which denotes the number of times training example will be sampled for a given Hoeffding Tree. In ORB, k is taken from $Poisson(\lambda)$ distribution and multiplied by a boosting factor $OBF^{(t)}$, which boosts the number of times the training example is sampled. The following equations are used to calculate $OBF^{(t)}$:

$$OBF_0^{(t)}(P_0) = \begin{cases} \left(\frac{m^{ma^{(t)}} - m^{th}}{(m - m^{th})} * l_0\right) + 1, \text{if } ma^{(t)} > th \\ 1, \text{ otherwise} \end{cases} \tag{2.2}$$

$$OBF_1^{(t)}(P_1) = \begin{cases} \left(\frac{m^{(th - ma^{(t)})} - 1}{(m^{th} - 1)} * l_1\right) + 1, \text{if } ma^{(t)} \leq th \\ 1, \text{ otherwise} \end{cases} \tag{2.3}$$

Here, P_0 and P_1 are set of hyper-parameters containing the following hyper-parameters: m - determines the growth of the exponential function, th - stands for the threshold that

indicates which class must be boosted; ma^t - the predictions moving average at time t; l_0 and l_1 - control the maximum boosting factor values. The parameters th and ma^t must be scaled between 0 and 1 [18]. If $ma^{(t)} \leq th$, it suggests that less than $th\%$ of the commits are classified as defect-inducing, hence, resampling rate of the defect-inducing class should be boosted. If $ma^{(t)} > th$, then more than $th\%$ of the commits are classified as defect-inducing, hence, resampling rate of the clean class should be boosted.

2.3 Performance Evaluation

2.3.1 Predictive Performance

We adopt the Geometric Mean (G-Mean) of Recall0 and Recall1 as measures of predictive performance, where Recall0 is the recall on the clean class and Recall1 is the recall on the defect-inducing class. These metrics were computed prequentially and using a fading factor to enable tracking changes in predictive performance over time, as recommended for problems that may suffer concept drift [50]. If the current example belongs to class i, $Recall_i^{(t)} = \theta Recall_i^{(t-1)} + (1-\theta)\mathbb{1}_{\hat{y}=i}$, where i is zero or one, t is the current time step, θ is a fading factor set to 0.99 as in [51], $\hat{y}$ is the predicted class, and $\mathbb{1}_{\hat{y}=i}$ is the indicator function, which evaluates to one if $\hat{y} = i$ and to zero otherwise. If the current example does not belong to class i, $Recall_i^{(t)} = Recall_i^{(t-1)}$. Also, $G\text{-}Mean^{(t)} = \sqrt{Recall_0^{(t)} \times Recall_1^{(t)}}$. It is worth noting that $Recall_0 = 1 - FalseAlarmRate$, i.e., false alarms are taken into account through Recall0 and G-Mean. Appendix A shows how four quadrants of confusion matrix can be computed from Recall0 and Recall1. These metrics were chosen because they are the most recently recommended for online class imbalance learning [52].

To evaluate the performance of defect prediction models, several performance metrics

have been used in the existing literature such as Area Under Curve (AUC), accuracy, precision, F1, recall, G-Mean and MCC. Area Under the ROC Curve (AUC) has been discouraged in the context of software defect prediction [53] because it incorporates several threshold values that are not meaningful in practice, making the comparison among approaches difficult.

However, the predictive performance of a classification model evaluated using class imbalanced data can be biased when using accuracy, precision, F1 and MCC as performance measure [54, 55, 52]. Hence, these metrics are unsuitable for class imbalance scenario.

In particular, Zhu et al. [54] provided a detailed analysis of different performance metrics on imbalanced data, and showed that G-Mean performed consistently on both balanced and imbalanced data. Zhu et al. [54] also shows that MCC gets affected by imbalanced data. MCC deteriorates seriously when dataset is imbalanced, hence, not suitable for classification accuracy measurement on imbalanced datasets.

Luque et al. [55] showed that the presence of imbalance can significantly affect the interpretation and significance of accuracy, as well as other commonly used performance metrics. They analysed the behavior of various performance metrics that are based on binary confusion matrices when used with imbalanced data. They showed that G-Mean is Null biased and thus recommended for imbalanced data, whereas MCC and F1 are biased.

In particular, such bias makes the interpretation of the metrics difficult, as a classifier of the same quality gets different performance values depending on the level of imbalance of the dataset being used for evaluation.

For example, consider data with a sample size of 200. Let's assume a classifier can predict 70% of the negative examples, and 60% of the positive examples correctly. Performance evaluations of this classifier using different performance metrics (precision, recall, F1-score, G-Mean, MCC) for different imbalanced scenario are given below:

i) Balanced case: Let's consider a balanced case where the data has 100 positive and 100 negative example. So, TP=60, TN=70, FP=30, FN=40.

$$Precision = \frac{TP}{(TP + FP)} = \frac{60}{(60 + 30)} = 0.667$$

$$Recall1 = TPR = \frac{TP}{(TP + FN)} = \frac{30}{(30 + 70)} = 0.6$$

$$Recall0 = TNR = \frac{TN}{(TN + FP)} = \frac{70}{(70 + 30)} = 0.7$$

$$F1\text{-}score = \frac{2 * (Precision * Recall)}{(Precision + Recall)} = \frac{2 * (0.667 * 0.6)}{(0.667 + 0.6)} = 0.632$$

$$G - mean = \sqrt{TPR * TNR} = \sqrt{0.6 * 0.7} = 0.648$$

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

$$= \frac{(60 * 70 - 30 * 40)}{\sqrt{(60 + 30)(30 + 40)(70 + 30)(70 + 40)}} = 0.302$$

ii) Imbalanced case 1: Let's consider an imbalanced case where the data has 50 positive and 150 negative example. So, TP=30, TN=105, FP=45, FN=20.

$$Precision = \frac{TP}{(TP + FP)} = \frac{30}{(30 + 45)} = 0.4$$

$$Recall1 = TPR = \frac{TP}{(TP + FN)} = \frac{30}{(30 + 20)} = 0.6$$

$$Recall0 = TNR = \frac{TN}{(TN + FP)} = \frac{105}{(105 + 45)} = 0.7$$

$$F1 - score = \frac{2 * (Precision * Recall)}{(Precision + Recall)} = \frac{2 * (0.4 * 0.6)}{(0.4 + 0.6)} = 0.48$$

$$G - mean = \sqrt{TPR * TNR} = \sqrt{0.6 * 0.7} = 0.648$$

$$MCC = \frac{(TP * TN - FP * FN)}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}}$$
$$= \frac{(30 * 105 - 45 * 20)}{\sqrt{(30 + 45)(30 + 20)(105 + 45)(105 + 20)}} = 0.268$$

iii) Imbalanced case 2: Let's now consider an even more imbalanced scenario where the dataset contains 10 positive and 190 negative examples. So, TP=6, TN=133, FP=57, FN=4.

$$Precision = \frac{TP}{(TP + FP)} = \frac{6}{(6 + 57)} = 0.095$$
$$Recall1 = TPR = \frac{TP}{(TP + FN)} = \frac{6}{(6 + 4)} = 0.6$$
$$Recall0 = TNR = \frac{TN}{(TN + FP)} = \frac{133}{(133 + 57)} = 0.7$$
$$F1 - score = \frac{2 * (Precision * Recall)}{(Precision + Recall)} = \frac{2 * (0.095 * 0.6)}{(0.095 + 0.6)} = 0.164$$
$$G - mean = \sqrt{TPR * TNR} = \sqrt{0.6 * 0.7} = 0.648$$
$$MCC = \frac{(TP * TN - FP * FN)}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}}$$
$$= \frac{(6 * 133 - 57 * 4)}{\sqrt{(6 + 57)(6 + 4)(133 + 57)(133 + 4)}} = 0.141$$

The analysis of different performance metrics in various balanced and imbalanced scenarios shows that, for a given classifier, precision, F1-score and MCC vary with different degrees of class imbalance in the data. Values for three different imbalance scenario for these metrics are: precision (0.667, 0.4, 0.09), F1-score (0.632, 0.48, 0.164), and MCC (0.302, 0.268, 0.141). On the other hand, Recall1, Recall0 and G-Mean remain constant across different imbalance levels and consistently provide reliable predictive performance. Values for three different imbalance scenario for these metrics are: Recall1(0.6, 0.6, 0.6), Recall0

(0.7, 0.7, 0.7) and G-Mean (0.648, 0.648, 0.648). This demonstrates that precision, F1-score, and MCC are sensitive to class imbalance, while recalls and G-Mean are not affected by class imbalance and are robust measures of predictive performance.

This is particularly important for online learning studies, where the imbalance ratio is changing over time [18]. If a metric that is biased to the level of class imbalance was adopted, it would be difficult to interpret the metric values over time. This is because the metric would vary depending on the current imbalance ratio, even if the quality of the predictions given by the classifier remains the same.

Therefore, G-Mean is a more adequate metric than MCC, precision, F1 and AUC for the purpose of this thesis.

The main idea behind the G-mean is to provide a balanced measure of the classifier's performance on both classes in a binary classification problem [55]. A classifier that is able to predict both classes efficiently in a binary classification would achieve a higher G-Mean. On the other hand, a classifier that is unable to predict one or both classes would achieve a lower G-Mean. G-Mean does not distinguish between a perverse learner (learner that always predicts the wrong class in binary classification), and a learner that can predict only one class. For both learners, G-Mean is zero. This is a desirable feature as both classifiers are equally pointless from a practical perspective – neither of them is giving any meaningful, usable prediction. For such learners, the MCC and informedness values are also zero, even though we do not recommend MCC for the reasons mentioned earlier.

The goal of just-in-time software defect prediction (JIT-SDP) is to classify software changes into either clean or defect-inducing class. For the purpose of JIT-SDP, learner that predicts just one class is not helpful as it does not provide any useful information about the true class of the samples. Similarly, a perverse learner is also not helpful as it always provides wrong prediction and can not be relied on to identify defect-inducing changes. Therefore, a

G-Mean score of zero indicates that such learners are useless in the context of JIT-SDP.

The analysis of various class imbalance scenarios reveals that precision is more vulnerable to data imbalance than Recalls and G-Mean. As the imbalance increases, precision decreases even when the classifier is effective. This trend is also evident in MCC, although to a lesser extent than precision. G-Mean is unbiased to class imbalance and computes a score by combining Recall0 and Recall1, thereby incorporating true positive rate (sensitivity) and true negative rate (specificity) into a single score. It also captures the FPR and FNR, given the relationship between sensitivity and FNR, and specificity and FPR. Hence, G-Mean is a superior choice over precision (and MCC and F1) when it comes to dealing with class imbalance learning, as it can deliver consistent and reliable performance.

2.3.2 Statistical Tests and Effect Size

To compare different approaches in this thesis, Scott-Knott procedure [56], which ranks the models and separates them into clusters. This test is used to select the best subgroup among different models. Non-parametric bootstrap sampling is used to make the test non-parametric, as recommended by Menzies et al. [57]. As explained by Demsar [58], non-parametric tests are adequate for comparison across data sets.

In addition, the Scott-Knott test adopted in this chapter uses A12 effect size [59] to rule out any small differences in performance. Specifically, Scott-Knott only performed statistical tests to check whether groups should be separated if the A12 effect size was not insignificant [57]. If the A12 effect size was insignificant, groups were *not* separated. We will refer to Scott-Knott based on Bootstrap sampling and A12 as Scott-Knott.BA12. Smaller Scott-Knott.BA12 rankings are better rankings.

Chapter Three

Literature Review

This chapter provides a literature review of existing component-based SDP and JIT-SDP studies related to this thesis. It also reviews existing work on hyper-parameter tuning, including both methods adopted for SDP and work proposed in the machine learning literature. This chapter is organised as follows: Section 3.1 presents existing SDP studies based on their learning methods and scenarios (online vs offline and CP vs WP). It includes both component-based SDP and JIT-SDP studies. Section 3.2 discusses hyper-parameter tuning methods, including methods used in the general machine learning literature and in SDP studies.

3.1 Software Defect Prediction

3.1.1 Component-Based SDP

3.1.1.1 WP Component-Based SDP

The majority of the earlier SDP studies focused on within-project (WP) defect prediction where data from the same project is used to build the prediction models. These studies used various machine learning algorithms including Support Vector Machines (SVM), Decision Trees, Ensemble Learners and Neural Networks [60, 61, 62, 63]. Many existing WP SDP studies used Ensemble learning methods [64, 65, 66]. Frequently used ensemble learning algorithms are Random Forest (RF), Boosting, and Bagging [39]. Wang et al. [67] compared different ensemble learning methods with Naive Bayes and Random Forest models. They evaluated on 10 real-world datasets using five performance measures – Probability of detection (PD), Probability of false alarm (PF), balance, Geometric mean G-mean and Area under the curve (AUC). They showed that among all the ensemble methods, AdaBoost.NC [68] with C4.5 decision trees as base learner outperformed NB and Random Forest.

Some of the WP SDP studies that used ensemble techniques also investigated the class imbalance problem of the defect data. Chen et al. [64] proposed neighbour cleaning learning (NCL) + ensemble random under-sampling (ERUS) that can eliminate the overlapping non-defective instances through the NCL algorithm and also resample non-defective instances to construct balanced training subsets through ERUS algorithm. They also build an ensemble prediction model using Adaptive Boosting (AdaBoost). Wang et al. [65] proposed a multiple kernel ensemble learning (MKEL) approach for defect prediction. MKEL can represent defect data in a high-dimensional feature space by multiple kernel learning, and reduce the bias caused by the majority class by use of ensemble learning. They also

designed a sample weight vector updating strategy to tackle class imbalance problem, that increase the weights of the defective samples and decrease the weights of the non-defective samples according to the classification results. Some WP SDP studies put emphasize on feature selection [69, 70, 71, 72]. Laradji et al. [69] proposed an ensemble learning method called average probability ensemble (APE) combined with efficient feature selection that enhanced the results by handling redundant and irrelevant features in the software defect datasets. Resampling approaches such as simple Random Over-Sampling (ROS), Random Under- Sampling (RUS), and synthetic methods such as SMOTE have been proposed to alleviate the negative effects of class imbalance on the performance of Within Project Defect Prediction (WPDP) models [73].

In addition, some studies used semi-supervised learning to deal with shortage of labelled data [74, 75, 76, 77]. Wang et al. [78] combined semi-supervised learning with ensemble methods to utilise both labeled and unlabeled data, and also to deal with class-imbalance problem via ensemble learning. There are also some studies that used unsupervised methods to build defect prediction models using unlabelled data [79, 80, 81].

In WP SDP, models are trained using historical data on software defects from the same software project. When a project has little to no previous data, the predictive performance of WP SDP models suffers greatly. Typically, there are not enough training data available at the early stages of a project. As a result, the WP SDP models exhibit poor predictive performance at the beginning of a project. Furthermore, there is no way to develop a model to predict the first version of the software because there is no training data at this stage. To address the issue of insufficient training data, many existing studies investigated cross-project (CP) SDP where data from projects other than the target project is used to train the prediction model.

3.1.1.2 CP Component-Based SDP

There have been several studies on CP SDP at the component level [82, 83, 84, 85, 86]. Zimmermann et al. [87] provided a large scale study on CP defect prediction models. They ran 622 CP predictions on 12 real-world datasets and found that only 21 predictions satisfied their success criteria of having precision, recall, and accuracy values over 75%. They also investigated the influence of different factors on CP prediction and argued that characteristics of data and process factors are more important than the domain of the project. Turhan et al. [82] proposed a CP defect prediction approach based on 10 publicly available datasets. They used only static code features (e.g. branchcount, numberoflines) to build the defect predictors. They showed that CP data dramatically increases probability of detecting defective modules, but it also increases false alarm rate. Their results showed that filtering can help to avoid high false alarm rate. However, their CP approach could not outperform WP approach.

He et al. [85] extended the work of [87] and [82] and provided further empirical evidences for the study of CP defect prediction. In their experiment, they employed 5 machine learning algorithms – Naive Bayes (NB), J48 (a C4.5 decision tree), Support Vector Machine (SVM), Decision Table (DT), and Logistic Regression. They investigated 34 different defect data sets and 20 static code attributes and a labeled attribute. Their results showed that CP data can provide better prediction results than data from same projects. Prediction result for CP data is related to distributional characteristics of data sets, which is valuable for training data selection. They also proposed an approach for automatically selecting suitable CP training data, which provided comparable performance with those provided by training data from the same project.

Some studies focused on minimising differences in the data distribution for CP SDP. Peters et al. [88] focused on selecting suitable CP training data based on the similarities

between the distribution of the test and potential training data. They used a similarity metric and feature subset selection to remove irrelevant training data. Nam et al. [89] applied Transfer Component Analysis (TCA) to CP SDP. TCA is a transfer learning approach that maps the data to a common latent space where CP and WP data are similar to each other. They also proposed a new approach called TCA+, which selects suitable normalisation options for TCA. Liu et al. [90] proposed a two-phased transfer learning method (TPTL) which can solve the instability problem of TCA+ [89]. In the first phase, a source project estimator (SPE) is used to choose two projects with highly similar distributions, and in the second phase, they leverage TCA+ to build two models and combines their predictions. Krishna and Menzies [91] introduced the concept of *bellwethers*, where bellwether is the project whose data yields the best predictions on all others. Sun et al. [92] proposed Collaborative Filtering based source Projects Selection (CFPS) to select source projects based on similarity. Herbold [93] investigated the election of suitable training data for CP defect prediction using 44 data sets. Their CP approach with training data selection outperforms training with all available data, but could not outperform WP prediction.

Similar to most WP SDP work, CP SDP also suffers class-imbalance problem and class-imbalanced sources may lead to misclassification of defective instances [94]. Hence, many studies [95, 94, 96, 97] focused on coping with the class-imbalance problem in CP SDP. Various approaches, e.g., data resampling [98], cost sensitive learning [99], and ensemble methods[3] have been suggested. Bennin et al. [98] assessed impact of eight commonly used data resampling methods (MAHAKIL, SMOTE, Borderline-SMOTE, ADASYN, ROS, RUS, Tomek links and OSS) for CP SDP using 34 versions of 22 open source projects. They showed that data resampling methods improve the prediction performance by significantly improving the recall (pd) and g-mean performance measures. Ryu et al. [96] proposed a transfer cost-sensitive boosting method (TCSBoost) that assigns cost to correct/incorrect classification based on the distributional characteristics and the class imbalance. They also

investigated whether WP data can be used to improve the prediction performance for CPDP. Ryu et al. [95] proposed an approach that uses similarity weight drawn from distributional characteristics and the asymmetric misclassification cost to balance imbalanced distributions. Jing et al. [94] proposed a prediction approach called SSTCA+ISDA for CP class imbalance problem. A transfer learning method SSTCA is used to make the distribution of target project data similar to that of source project data. Then ISDA is applied on the transferred data to learn features with more powerful classification ability.

Canfora et al. [100] proposed a multi-objective approach for CP defect prediction. They attempted to achieve a compromise between amount of code to inspect and number of defect-prone artifacts. This approach performed better than WP models. Panichella et al. [101] analysed the equivalence of different defect predictors and proposed a combined approach CODEP (COmbined DEfect Predictor) that uses machine learning to combine different and complementary classifiers. This combination performed better than the stand-alone CP technique. Jin et al. [102] proposed a semi-supervised transfer learning approach called DA-KTSVM for CP models, that tries to match the distribution of the data of different software projects.

Overall, these studies demonstrate that data distributional characteristics are important for CP SDP. They proposed approaches to select CP data that are similar to WP data, or to map CP and WP data into a latent space where they are similar. However, none of these studies were in the context of JIT-SDP or online SDP. JIT-SDP offers change-level prediction, which has a number of benefits over component-based SDP, which is described in Chapter 1. Despite the benefits of JIT-SDP models, they require a lot of training data, much like any prediction model, and may perform poorly in terms of prediction if there is not enough training data available at the initial phase of a project. Using CP data for JIT-SDP can be helpful in resolving this problem.

3.1.2 Offline JIT-SDP

3.1.2.1 WP JIT-SDP

Mockus et al. [103] is one of the first studies to consider software development process as a series of changes and investigate prediction of the defect-proneness of such changes. In particular, they investigated failure probability of IMR (initial maintenance request), which consists of several MR. Each MR needs changes to several source code files. They used input features based on the properties of a change such as size, duration, diffusion, type, experience of the developer, etc. Input features used in this study are easily obtainable. The drawback of this approach is they did not use any change management system which could provide considerable amount of data. Also, IMR contains multiple changes which makes it coarse grained. Hence, inspection effort to fix defects could not be minimized.

Kim et al. [104] conducted one of the first studies on JIT-SDP. They proposed an approach to classify software changes based on features such as change metadata including author, commit hour, LOC, lines of comments, cyclomatic complexity, etc. Their approach classifies changes as buggy or clean with 78% accuracy on average in a study involving 12 open source software projects. Many approaches that followed investigated features that may be correlated with inducing defects. Śliwerski et al. [105] investigated linking of bug database with version archive to identify 'fix-inducing changes' (changes that lead to problems indicated by fixes). They investigated which properties of change correlate with inducing fixes. They showed that if the change is larger, then it is more likely to induce a fix. Eyolfson et al. [106] showed that commits submitted during certain time or day of the week and daily commit frequency of the developer may influence the "bugginess" of a commit.

Kamei et al. [7] performed a large scale empirical study on change-level defect prediction which is called Just-in-Time Software Defect Prediction (JIT-SDP). They used 6 large

well-known open source projects (i.e., Bugzilla, Columba, Mozilla, Eclipse JDT, Eclipse Platform, and PostgreSQL) and 5 commercial projects. They extracted information from the CVS repositories of the projects and combined it with bug reports. To know whether or not a change introduces a defect, we use the SZZ algorithm which links each defect fix to the original defect-introducing code change from the version archive (such as CVS) with the bug tracking system (such as Bugzilla). To deal with class imbalance, they used a resampling approach for the training data where they randomly deleted and decreased majority class instance (non-defect-inducing changes) to the same level as the minority class (defect-inducing class). They used 14 different factors grouped into five dimensions extracted from code changes to predict defect inducing changes. For defect prediction, they built a logistic regression model. For each change, the logistic regression model outputs a probability (value between 0 and 1) and a threshold value 0.5. If the model predicts a probability greater than 0.5, then the change is classified as defect-inducing, otherwise it is classified as non-defect inducing/clean. Their approach achieved an average accuracy of 68 percent and an average recall of 64 percent. They also found that 20 percent of all effort (for review) suffices to identify 35 percent of all predicted defect-inducing changes.

Pascarella et al. [107] explained in their study that in existing JIT-SDP an entire commit is considered as defective. They argued that in real-world scenarios, a commit may be partially defective (composed of both defective and non-defective files). They investigated to what extent commits are partially defective and proposed a novel fine-grained just-in-time defect prediction model that is able to predict specific defective files contained in a commit using 10 open source projects.

Chen et al. [13] considered JIT-SDP as a multi objective problem by maximising the number of identified buggy changes and minimising efforts to fix it. They used a logistic regression model to perform the prediction using 6 open source datasets considering the 14 metrics described in [7]. Their approach managed to identify 63.8% buggy changes on

average when using only 20% efforts. Yang et al. [47] proposed a two-layer ensemble learning approach TLEL. In the inner layer, they used Decision Tree and Bagging to create a Random Forest model. In the outer layer, they ensemble many different Random Forest models using stacking [108]. They have also investigated other base learners than Decision Trees, including Naive Bayes, Support Vector Machine, Linear Discriminant Analysis and Nearest Neighbor Classifier. They showed that ensembling Decision Trees to create Random Forests within their model performed better than ensembling other base learners in 5 out of 6 open source projects. Their approach enabled to detect 70% of bugs by reviewing 20% of the code. TLEL also achieved higher F1-score compared to three baseline approaches – Deeper, DNC and MKEL. Yang et al. [109] proposed a deep learning method called 'Deeper' for JIT-SDP. They compared their approach with the approach proposed by Kamei et al. [7] and showed that their approach was able to discover 32.22% more defects, based on a study with 6 open source datasets. Li et al. [25] investigated the impact of different combinations of base learners such as Support Vector Machines (SVMs), Logistic Regression (LR), Random Forest (RF), Multi-layer Perceptron (MLP), Naive Bayes (NB) and Decision Trees (DTs). They showed that the diversity of base learners plays an important role for achieving promising performance.

Yang et al. [110] considered simple unsupervised methods for JIT-SDP and found that these unsupervised methods perform better than previous supervised methods. They build an unsupervised model for each of the change metrics (except LA, and LD), that ranks changes in descendant order according to predicted risk value. Their idea is inspired by Koru et al.'s finding that smaller modules should be inspected first as they tend to be proportionally more defect-prone. Yang et al. hypothesizes that changes with lower metric (e.g. LT) value are in smaller modules and should be inspected earlier. However, Fu and Menzies [111] revisited Yang et al.'s empirical results and found that not all unsupervised methods have better performance than supervised methods. Therefore, Fu and Menzies

proposed OneWay method, which can automatically selects the potential best method. Their results showed that OneWay nearly always outperforms the unsupervised predictors found by Yang et al. They concluded that there are clear advantages to use supervised approaches over unsupervised ones which is against Yang et al's recommendation of using unsupervised learning for JIT-SDP.

Misirli et al. [112] argues that all fix-inducing changes should not be treated equally. They proposed an unsupervised clustering technique using 6 large open source projects to classify changes as high and low impact based on the the amount of churn, the number of modified files and the number of subsystems where the modified files span affected during a fix. Their approach managed to save the inspection efforts by 34 % compared to a typical binary model. McIntosh and Kamei [20] performed a longitudinal case study of 37,524 changes from the rapidly evolving QT and OPENSTACK systems. They trained model with data from short and long period. Short-period JIT-SDP models are trained with changes from one time period to avoid older changes which may not be relevant at present. Long-period JIT-SDP models are trained with all changes that occurred during or prior to a specific period. They found that model trained with data period that is closer to testing period outperforms models that are trained using older periods. Their results showed that fluctuations in the properties of fix-inducing changes can impact the performance of JIT-SDP models. They showed that JIT-SDP models typically lose predictive power after one year, possibly as a result of concept drift. Hence, the JIT model should be updated with more recent data.

All the above studies considered JIT-SDP as an offline learning problem and the investigations were carried out in WP context. In reality, WP models may perform poorly especially in the beginning of a project, when there is no or little training data available. To overcome this issue, existing studies investigated CP data for JIT-SDP.

3.1.2.2 CP JIT-SDP

JIT-SDP classifiers require sufficient amount of training data, which is not available at the beginning of a software project as data arrives sequentially over time. CP JIT-SDP can overcome this issue by using data from other projects to build the classifier. Several studies investigated CP JIT-SDP. The first CP JIT-SDP study [12] carried out an empirical evaluation of CP JIT-SDP performance by using data from 11 open source projects. They proposed three CP JIT-SDP approaches based on project similarity, data merging and ensembles of models. All approaches were based on random forests as base learners.

In the first approach, they take into account training the model with similar projects. This is due to the fact that the metrics' distribution can differ between projects which may cause poor prediction performance in CP models. Hence, the similarity of CP data is measured with respect to the WP data. They explored domain agnostic similarity which is measured using predictor metric value, and domain-aware similarity which is measured using project details. First, they trained a model with project that has similar domain-agnostic characteristics to the test project. They found that domain-agnostic similarity approach is threshold-sensitive and requires access to all CP data which is not practical due to privacy concern of the companies involved. To overcome such limitations, they considered domain-aware similarity that focuses measuring similarity based on the characteristics of the software projects (e.g. company, programming language). Their finding shows that models chosen using either similarity technique have similar CP performance. However, while compared to WP JIT-SDP, prediction performance was not improved when using training datasets that were highly similar to the test dataset.

The second approach contains three sub areas: a) all CP data is placed into a single dataset and a model is built using this merged training dataset, b) a model is trained with a dataset assembled by selecting only the CP instances that are most similar to the WP ones

and c) transform the training data of each project using a context-aware rank transformation and train a model on this CP data [113]. Approaches (b) and (c) do not provide better performance than (a). No evidence of difference in the overall performance has been found between the performances of these approaches and the WP approach.

In the third approach, a simple ensemble composed of models, each trained on a training dataset corresponding to a different project, is used. The mean of the prediction probabilities of the models is calculated and used to give a prediction. This approach is extended by applying similarity where more weight is given to projects that are more similar to the target project. To measure similarity, they calculated Spearman correlation between predictors and output variables of training and testing data; then the 3 predictors with highest Spearman correlation value are selected from training and testing data and finally Euclidean distance between predictors of training and testing data is calculated to obtain the similarity measure. However, they found that weighing down dissimilar projects did not improve the performance.

Overall, they found that simple merging of all CP data into a single training set and ensemble approaches obtained similar predictive performance to that of WP models. Different from SDP at the component level, other more complex approaches, including similarity-based approaches, did not offer any additional advantage compared to these.

Catolino et al. [114] investigated CP JIT-SDP in mobile platforms using 14 apps extracted from the Commit.Guru platform [115]. They compared the CP performance of four different well-known classifiers and four ensemble techniques. Naive Bayes performed best compared to other classifiers and some ensemble techniques. They did not check how CP compared against WP results. Chen et al. [13] considered JIT-SDP as a multi-objective problem to maximise the number of identified defect-inducing changes while minimising the effort required to fix the defects. They proposed a multi-objective optimization-based

supervised method called MULTI to build logistic regression JIT-SDP models. They used six open source projects. MULTI was evaluated on three different model performance evaluation scenarios (cross-validation, CP-validation, and timewise-cross-validation) against 43 state-of-the-art supervised and unsupervised methods. They found that it can perform significantly better than WP methods.

Li et al. [25] investigated a semi-supervised approach for effort-aware JIT-SDP on six open source projects. This approach outperformed existing supervised and unsupervised models for effort-aware JIT-SDP and required small amount of labeled training data. Hoang et al. [116] proposed a deep learning method for JIT-SDP called DeepJIT. Zhu et al. [117] proposed a JIT-SDP approach named DAECNN-JDP based on denoising autoencoder and convolutional neural network. WP and CP defect prediction experiments were performed on six large open source projects and DAECNN-JDP was compared with 11 baseline models, including eight machine learning models, EALR, Deeper and CNN-JDP. The results show that DAECNN-JDP achieved better predictive performance than the baseline models for both CP and WP defect prediction. However, the predictive performance of CP and WP approaches was not compared against each other.

All of the studies mentioned above were investigated in an offline learning scenario. Offline JIT-SDP models are created using a pre-existing training set and are never updated after that. However, online learning differs from offline learning in that training examples (bot CP and WP) arrive over time, and the model must be updated every time a new training example is received. As a result, it is unknown if the approaches explored in the offline learning scenario would produce similar results in the online learning scenario.

3.1.3 Online JIT-SDP

Tan et al. [10] investigated JIT-SDP in a scenario where new batches of training examples arrive over time and can be used for updating the classifiers. They considered the fact that the labels of training data may arrive much later than the commit time (verification latency). They used 7 updatable learning algorithms based on Naive Bayes (Bayes, LWL), instance-based learning (IBK, KStar), boosting (LogitBoost), nearest-neighbors (NNge), and Support Vector Machines (SPegasos) to learn over time. They used resampling techniques to deal with the class imbalanced. Class imbalance [67] occurs when the class distributions are highly imbalanced (e.g. for JIT-SDP, the number of clean changes can be very large compared to number of defect-inducing changes), which often leads the model to struggle to identify examples of the minority class. Based on a study with one proprietary and six open source projects, they showed that both resampling techniques and updatable classification improve the precision by 12.2-89.5% or 6.4–34.8 percentage points. They showed that not considering chronology and verification latency can create a false impression of higher predictive performance. Therefore, it is important to take chronology and verification latency into account in realistic scenarios. However, their approach assumes that there is no concept drift, i.e., that the defect generating process does not suffer variations over time.

Cabral et al. [18] proposed a method called Oversampling Rate Boosting (ORB) to tackle a type of concept drift called class imbalance evolution, where the proportion of examples of the defect-inducing and clean classes change over time. Their work investigates an online JIT-SDP scenario taking verification latency into account. They considered a waiting time (w) after the commit time to decide whether a change should be labeled as defect-inducing or clean. If a defect is found within w, the change is labeled as defect-inducing and used for training. If no defect associated to a change has been found in w days from its commit, this gives confidence that this change is clean and it is labeled as

clean and used for training. If a change that has already been labeled as clean is found to be defect-inducing after w days, the training example corresponding to that change will be updated with the label defect-inducing and presented again for learning.

ORB has a resampling rate to tackle class imbalance evolution based on a moving average over the predictions given by the JIT-SDP model, being able to improve predictive performance over JIT-SDP approaches that assume a fixed level of class imbalance and approaches that adjust the resampling rate based on the imbalance ratio. ORB achieved better $|R_0 - R_1|$ up to 45.38% and 63.59% compared to the state-of-the art class imbalance evolution algorithms Undersampling Online Bagging (UOB) and Improved Oversampling Online Bagging (OOB), respectively.

In a recent study, Cabral et al. [19] considered the fact that Just-in-Time Software Defect Prediction (JIT-SDP) are affected by changes in the defect generating process (concept drifts). They provided a detailed analysis of concept drift in online WP JITSDP. They investiagted 10 open source datasets and proposed a novel online approach called Prediction-Based Sampling Adjustment (PBSA) that is able to cope with concept drifts and provide a reliable predictive performance online WP JIT-SDP. Based on 13 GitHub projects, Song et al. [118] provided discussion on how to continuously evaluate predictive performance in JIT-SDP over time during the software development. They showed that waiting time to label software change can have impact on the prediction performance of JIT-SDP. They recommended to use smaller waiting time if frequent concept drift is expected, and otherwise larger waiting time is more suitable.

The existing online JIT-SDP studies build JIT-SDP models that are trained only on the WP data. They do not use any CP software changes for training. Such JIT-SDP models may suffer from poor predictive performance due to lack of insufficient training data at the beginning of the project. These models may also suffer from concept drift [20] which can

cause sudden drops in the predictive performance of the models. To overcome these issues, CP data can be useful to improve the predictive performance of online JIT-SDP models. However, none of the online JIT-SDP studies above investigated CP JIT-SDP.

3.2 Hyper-Parameter Tuning

Characteristics of a machine learning algorithm are controlled by its hyper-parameters which need to be configured before running the algorithm [119]. Existing studies showed that hyper-parameter choices could influence the predictive performance of machine learning-based defect prediction models [120, 121, 104, 122].

There are several ways to configure hyper-parameters such as using default values from existing literature, manual configuration by the user or tuning procedure for finding optimal values. Using default values may result in sub-optimal predictive performance [119]. Manual tuning is also ineffective for many problems due to some limitations such as large number of hyper-parameters, complex models, time-consuming model evaluations, and non-linear hyper-parameter interactions [119]. Sub-optimal predictive performance obtained by default values and inefficiency of manual tuning procedure necessitated the research of automatic hyper-parameter tuning for machine learning models. Automated hyper-parameter tuning is advantageous because i) it reduces human effort necessary for applying machine learning in practice; ii) it improves prediction performance of the machine learning algorithms; iii) it improves reproducibility of scientific studies, hence, can facilitate fair comparisons between different approaches [123].

This section discusses existing hyper-parameter tuning approaches, including AutoML, approaches from the general machine learning literature and studies investigating hyper-parameter tuning for software defect prediction.

3.2.1 AutoML

Automated Machine Learning (AutoML) has become an important research area in recent years due to its ability to automate various stages of the machine learning process, including hyperparameter tuning [124, 125, 126]. AutoML aims to automate the entire machine learning pipeline [124]. According to yao et al. [125], AutoML has gained popularity as a promising solution to simplify the application of machine learning techniques and alleviate the need for skilled human experts. The aim of AutoML is to make machine learning more accessible and user-friendly. An all-encompassing AutoML system has the ability to create a dynamic combination of different techniques, resulting in a user-friendly end-to-end ML pipeline system [124].

The AutoML pipeline is composed of multiple processes, such as data preparation, feature engineering, model selection and hyperparameter optimisation [124, 125, 127]. Data preprocessing is usually the first task in Machine Learning pipeline. Current AutoML tools still require considerable human intervention for this task [128]. Feature engineering in AutoML refers to automatically construct features from the data, which can lead to improved performance of subsequent machine learning models [125]. Several studies investigated constructing new feature from the original ones based on some pre-defined operations [129, 130, 131].

In AutoML, the task of model selection and hyperparameter tuning involves the automatic selection of classifiers and configuration of their hyperparameters to achieve optimal learning performance [125]. One of the first AutoML system is Auto-Weka [132] which investigates the problem of simultaneously selecting a learning algorithm and setting its hyperparameters using the WEKA framework. They used Bayesian Optimization, specifically Sequential Model Based Optimisation for tuning hyperparameters in a study using 21 benchmark datasets. The results show that Auto-WEKA was able to achieve competitive

performance compared to manually tuned models while requiring significantly less human effort. Bergstra et al. [133] proposed Hyperopt, a python library for model selection and hyperparameter optimisation. Hyperopt uses Bayesian optimisation to efficiently search the hyperparameter space and can support parallel computation. Komer et al. [134] proposed Hyperopt-sklearn, a tool to automate the process of hyperparameter tuning for machine learning models built for Scikit-learn machine learning library. Feurer et al. [135] proposed an AutoML system called Auto-sklearn, which improves on existing AutoML methods by automatically taking into account past performance on similar datasets. Although Auto-sklearn automates the process of hyperparameter tuning for users, it also has its own set of hyperparameters that can affect its performance within a given time budget. Feurer and Hutter [126] provided a survey on hyperparameter optimisation as part of AutoML, explaining that it can lead to improved performance while saving substantial amounts of resources. Efforts to automate hyperparameter tuning include Random Search-based Approach [136], Bayesian Optimization-based Approach [137, 138, 139, 140] and Gradient Descent-based Approach [141]. These approaches are described in Section 3.2.2.

Online machine learning algorithms have emerged to deal with challenges of offline learning algorithms such as limited memory. Online algorithms also may come with multiple hyperparameters, which need to be tuned for obtaining optimal performance. However, the learning task of online algorithms is dynamic in nature, hence, concept drift may happen. So the hyperparamter tuning for online learning needs to consider its dynamic nature and concept drift. Most of the AutoML studies for hyper-parameter tuning have not considered a fully online learning scenario. Existing approaches for online hyperparameter tuning are described in Section 3.2.3.4.

3.2.2 Offline Hyper-Parameter Tuning Approaches

Existing offline learning studies used different approaches for tuning hyper-parameters. Some commonly used hyper-parameter tuning methods are following:

3.2.2.1 Grid search

Grid search is one of the most common methods for hyper-parameter tuning [142]. It is an exhaustive search or common brute force method that evaluates predefined finite set of hyper-parameter configurations. Grid search is easy to implement. However, the main drawback is it is inefficient for high-dimensional hyper-parameter configuration space, because the number of function evaluation grows exponentially as the number of hyper-parameters increases [143].

3.2.2.2 Random search

Random search for hyper-parameter tuning which is a simple alternative of Grid search proposed by Bergstra et al. [136]. Random search generates random hyper-parameter combinations as candidate hyper-parameter values until a fixed budget (e.g. number of function evaluation) is exhausted. It then selects the best hyper-parameter combination among these. Not defining an absolute grid allows random search to explore other values in the given distribution. Bergstra et al. [136] shows that random search is able to find models that are as good or better than grid search within less computational time. It is therefore more efficient in high dimensional space compared to grid search. When given large budget, Random search is able to find the global optimum or near-global optimum.

However, the main disadvantage of both Grid search and Random search is they may

perform unnecessary function evaluations in the poor-performing region by not exploiting better-performing region of the search space.

3.2.2.3 Gradient-based Optimisation

Another optimisation technique is Gradient descent optimisation [144] that calculates the gradient of the hyper-parameters and move towards the optimum direction. These methods use local information about the loss function of the machine learning algorithm in order to compute the gradient of the cost function with respect to hyper-parameters. A study [141] used reversed stochastic gradient descent with momentum to compute gradient with respect to all continuous hyper-parameters. Another study [145] uses approximate gradient instead of true gradient because computing an exact gradient can be computationally demanding. This approach provided fast convergence, but the drawback is the algorithm requires to configure additional parameter (tolerance parameters). However, gradient descent is only suitable for cases where it is possible to obtain the gradient such as continuous hyper-parameters, and can not be used for categorical hyper-parameters.

3.2.2.4 Bayesian optimisation

Bayesian optimisation is an iterative algorithm that is often used for hyper-parameter tuning. Unlike grid search and random search, Bayesian optimisation utilise previously obtained results to determine promising future evaluation points. Bayesian optimisation uses a surrogate model to approximate the objective function and an acquisition function to direct the samples towards areas where an improvement over the current best observation is likely [146, 147]. Common surrogate models include Gaussian process [148], Random forest [149], and the Tree Parzen Estimator [139]. After estimating the distribution of the surrogate model, the acquisition function is used to balance the exploration (sample unexplored area) and

exploitation (sample current promising region).

Bayesian optimisation is more efficient than grid search and random search because it can use previous results to find the optimal hyper-parameter values and the surrogate model is usually less computationally expensive than the objective function. However, the main drawback is that Bayesian optimisation can be stuck in local optima if the exploration and exploitation are not balanced correctly and can be difficult to parallelize.

3.2.2.5 Evolutionary Algorithms

Evolutionary Algorithms are widely used for optimisation problems including hyper-parameter optimisation. Similar to the principles of biological evolution, within the paradigm of an evolutionary algorithm, finding the solution of a problem is considered as a survival task. Possible solutions compete with each other for survival and eventually an approximate best solution is found.

- **Genetic Algorithm (GA):** GA is one of the most common evolutionary algorithm which solves optimisation problem by simulating evolution of solutions over multiple generations in such a way that fittest solutions can go to the next generation, which eventually leads to finding optimal solution. Genetic algorithm maintains a population of candidate solutions. In each generation, a set of genetic operators such as cross-over and mutation are used to evolve new candidate solution from the population. The algorithm evaluates fitness of each solution using a fitness function. This process is repeated until a stopping criteria is reached.
- **CMA-ES:** Covariance matrix adaption evolutionary strategy (CMA-ES) [150] is an evolutionary strategy that samples candidate solutions from a Multivariate Gaussian Distribution (MGD)and evaluates each solution in each generation. Based on the

ranking of the solution after evaluation, mean and covariance matrix of the sample distribution are updated to increase or decrease the search space for the next generation. This allows CMA-ES to learn to sample solutions from the promising region [151].

- **Differential Evolution (DE):** DE is a population-based stochastic search technique. It follows the common procedure of evolutionary algorithm. DE randomly generates an initial population. After initialization, it performs a loop of evolutionary operations such as mutation, cross-over and selection. Based on current population, the algorithm creates mutation vector using different mutation strategies. After mutation, a binomial crossover operation forms the trial offspring vector. The selection operator chooses the better vector between the parent vector and the trial vector based on their fitness values. DE has itself only a few algorithm parameter which can be tuned. There are several recent adaptive and self-adaptive DE algorithms available [152, 153] that can dynamically update these parameters automatically without the need for pre-defining them.

3.2.3 Studies Analysing Hyper-Parameter Tuning Approaches

Frank and Hutter [123] provided an overview of the existing prominent automated hyper-parameter optimisation approaches such as model-free methods and Bayesian optimisation. They also discussed challenges of hyper-parameter tuning such as expensive function evaluation for large models and datasets, complex and high-dimensional configuration space and problem of determining which hyper-parameter should be optimised. Elshawi et al. [154] provided a comprehensive survey for state-of-the-art approaches to tackle problems combining algorithm selection and hyper-parameter tuning (CASH process) defined in [155]. The goal of this technique is to minimise the need for human experts, enabling even non-expert users to create machine learning (ML) systems that are tailored to their own requirements. Yang

and Shami [119] provided a comprehensive discussion on the state-of-the art hyper-parameter optimisation techniques and how to apply them to different ML algorithms. Villalobos-Arias and Quesada-López [156] compared 12 state-of-the-art hyper-parameter tuning algorithms (6 bio-inspired algorithms, 2 heuristic tuners, and 3 model-based algorithms) against random search, on 9 datasets of the PROMISE repository for software effort estimation and 4 sub-datasets from the ISBSG R18 datasets. Their results showed that, for software effort estimation, these methods were more effective than random search in only 6% of the studied model and dataset combinations.

Tabu Search is a metaheuristic algorithm proposed by Glover et al. [157] to efficiently search for good solutions in complex search spaces. This approach relies on adaptive memory and responsive exploration of the search space, as explained in [158]. The algorithm iteratively moves through the search space by making small changes to the current solution and comparing it with neighboring solutions. If a neighboring solution is found to be better, it replaces the current solution and is used to explore a new neighborhood in the next iteration. If not, the search continues by generating other moves starting from the original solution. To prevent revisiting already explored solutions, Tabu Search employs a tabu list that keeps track of recently visited solutions marked as “taboo". These solutions are not allowed to be visited again for a certain number of iterations, thus allowing the algorithm to explore new areas of the search space [158]. Tabu search has been used for hyperparameter tuning in several studies. Lebrun et al. [159] proposed a model selection method where they employ tabu search to optimise hyperparameters for Support Vector Machine. Their objective was to merge feature selection, streamline the training set, and hyperparameter tuning into a single global optimization problem. This approach aims to generate Binary Decision Functions (BDF) with SVM that are both efficient and have low complexities. Their experiment results show that the Tabu search selection method achieves these goals. Corazza et al. [158] investigated use of Tabu search to optimise hyperparameters of Support Vector Regression (SVR)

using a cross-company data set of Web applications, the Tukutuku database. Their results show that by applying Tabu Search, significant improvement in performance was achieved. They extended this study in [160], where they investigated Tabu search [161] for optimising hyper-parameters of Support Vector Regression (SVR) for software effort estimation using 21 open source datasets. They found that estimation provided SVR using Tabu search to optimise hyper-parameters outperformed estimation provided by SVR with other methods such as random configuration, default configuration and grid search configuration. Yao et al. [162] employed Tabu search to optimise hyperparameter for Support Vector Machine and showed that tabu search algorithm can effectively provide better parameter values for this machine learning approach.

Other hyper-parameter tuning approaches are based on meta-heuristic optimisation. Among meta-heuristic algorithms, Genetic algorithm (GA) is widely used for hyper-parameter optimisation [163, 164, 165, 119]. GA searches for the optimal hyper-parameter combination by passing well-performing hyper-parameter combinations of each generation to the next generation [119]. Oliveira et al. [164] investigated Genetic Algorithm based feature selection and hyper-parameter optimisation for software effort estimation using three machine learning technique (i) support vector regression (SVR), (ii) MLP neural networks and (iii) decision trees (M5P) on 6 benchmark data sets of software projects. Their results showed that their GA-based method was able to improve performance of the three machine learning techniques considered in the problem of software effort estimation. Xia et al. [166] used Differential Evolution to tune hyper-parameters of Analogy-Based software effort estimation. Their method led to significantly better results than untuned Analogy-Based SEE and a baseline called Automatically Transformed Linear Model. Fu and Menzies [167] investigated Differential Evolution to tune hyper-parameters of Support Vector Machines (SVM) to the problem of finding which questions in the Stack Overflow programmer discussion forum can be linked together. Their approach achieved similar or better results than a state-of-the-

art Deep Learning Convolutional Neural Network in terms of precision, recall and F1-score, while being 84 times faster to run.

Nelder-Mead optimisation method, where the search for the minimum is guided by the values of the cost function at a particular simplex has also been used for hyper-parameter tuning [168, 169]. Gilles et al. [170] proposed an approach called Direct Simplex Search (DSS) based on Nelder-Mead [171] method to tune hyper-parameters of Support vector Machine. Compared to Grid search, DSS was able to reduce the number of function evaluations. The accuracy attained by DSS and that of the Grid search did not, however, show any significant differences. HyperOpt [133] is a Bayesian optimisation tool for model selection and hyper-parameter optimisation. It provides an optimisation interface where the user can specify configuration space as probability distribution. A framework for classification problems called Auto-Weka is presented by Thornton et al. [155] that performs hyper-parameter tuning using a Bayesian optimiser.

Reinforcement learning has also been used for hyper-parameter tuning. Bandit-based algorithms such as Successive Halving [172] and Hyperband [173] have shown success in hyper-parameter optimisation. Successive halving eliminates half of the poorly performing hyper-parameters and allows more budget to the better-performing half to be evaluated in next iteration. However, allocating budget efficiently is a concern for Successive Halving. Hyperband regularly adjusts the trade-off between the available computational budget and the number of candidates.

3.2.3.1 Offline Hyper-Parameter Tuning in SDP

Over the last few decades, a number of studies investigated software defect prediction using machine learning classifiers. In most studies, the hyper-parameters of these machine learning classifiers have used their default values. Hall et al. showed that classification models may

under-perform because of default hyper-parameter settings [62]. Fu et al. [174] pointed that 80% of the 50 highest cited defect prediction studies rely on a default setting and only a few percent worked on some of the tuning techniques.

Nevendra et al. [175] investigated impact of hyper-parameter optimisation for defect count prediction using 15 software defect datasets. They optimised hyper-parameters of various regression techniques using Grid Search and Random Search. Their results showed that hyper-parameter optimisation improved the prediction performance for MLPR, Lasso, DTR, Hubber, and SVR, by 16.96%, 8.31%, 8.16%, 6.01% and 5.22%.

Most existing offline SDP studies used either default values or grid search to tune hyper-parameters. Alazba and Aljamaan [176] used grid search to tune hyeper-parameters of seven tree-based ensembles including random forest, extra trees, AdaBoost, gradient boosting, histogram-based gradient boosting, XGBoost and CatBoost using 21 defect datasets. Their result showed that hyper-parameter tuning had a large impact on extra trees and random forest ensembles. Shan et al. [177] used grid search for tuning hyper-parameters of defect prediction approach called LLE-SVM which outperformed the SVM model. However, their result showed that time cost of grid search was too high. Bahaweres et al. [178] used Random search to optimise hyper-parameters of a their defect prediction approach called Neural Network based SMOTE improve neural network performance on class imbalance problem. Neural Network-based SMOTE can increase significantly (bal by 25.48%, and recall 45.99%) compared to the original Neural Network without SMOTE. Agrawal and Menzies [179] proposed an automatic method SMOTUNED based on Differential Evolution for tuning hyper-parameters of SMOTE algorithm for 7 datasets. Their experiments showed that hyper-parameter tuning increased area under the curve (AUC) by 60%.

Tantithamthavorn et al. [14] explored an off-the-shelf automated hyper-parameter optimisation technique called Caret [180] for defect prediction. Caret evaluates candidate

parameter settings and suggests the optimized setting that obtains the best performance. They investigated 26 classification techniques as well as using the out-of-sample bootstrap validation technique on 18 datasets (including both proprietary and open source). Their result showed that Caret improves accuracy of defect prediction models up to 40 percentage points. Performance improvement provided by applying Caret is large for 9 of the 26 studied classification techniques.

In another study, Tantithamthavorn et al. [121] investigated 4 automated hyper-parameter optimization techniques - grid search, random search, genetic algorithm, differential evolution for defect prediction models. They investigated 26 commonly-used classification techniques using 18 datasets from both open source and proprietary domains. Their results showed that hyper-parameter optimisation improves the AUC performance of defect prediction models by up to 40 percentage points. 16 of the 26 studied classification techniques had non-negligible performance improvement. Grid search, random search, genetic algorithm, and differential evolution techniques yield similar benefits of AUC performance improvement when applying to defect prediction models.

Pan et al. [181] analysed Convolutional Neural Network model performance for WP SDP under different hyper-parameter settings and showed that performance of individual project varies with different hyper-parameter settings. Li et al. [182] investigated the impact of automated hyper-parameter optimisation on CP SDP. They investigated 62 combinations of the transfer learning algorithms and classifiers using 20 real-world software projects data, leading to a total of 37 different parameters to be tuned. For tuning, they used Hyperopt that provides two options to choose from, i.e., random search [136] and Tree-of-ParzenEstimators (TPE) algorithm [139], to carry out the parameter optimisation. They showed that automated hyper-parameter optimisation improved performance of 77% of the CP SDP models within manageable computational cost.

Osman et al. [183] tuned hyper-parameters and compared the performances of SVM and IBK (k-nearest neighbours algorithm) models using 5 open source projects. They used Multisearch Weka to search for the optimal hyper-parameter values. For each project, data is divided into tuning set and experiment set. Prediction errors between model with default parameter value and tuned parameter value are compared. Tuning hyper-parameters improved IBK performance significantly compared to SVM. The prediction accuracy was improved by up to 20% in IBK and by up to 10% in SVM. The default values change during tuning. Also, it makes IBK statistically equivalent to SVM than before. This indicates that tuning parameters not only improves prediction accuracy, but also have an effect on how the models compared to each other. IBK was more sensitive to its hyper parameter values than SVM, which indicates that models may differ in their sensitivity to their hyper parameters.

Khan et al. [184] explored the Artificial Immune Network (AIN) optimisation method called opt-aiNet for hyper-parameter optimisation for software defect prediction models base on 7 machine learning classifier such as support vector machine Radial base function (SVM-RBF), K-nearest neighbor (KNN) (Minkowski metric), KNN (Euclidean metric), Naive Bayes (NB), Decision Tree (DT), Linear discriminate analysis (LDA), Random forest (RF) and adaptive boosting (AdaBoost). They compared the results obtained by optimised and default hyper-parameter values and showed that AUC performance is improved by up to 41%.

3.2.3.2 Offline Hyper-Parameter Tuning in JIT-SDP

Several existing offline JIT-SDP studies are based on default hyper-parameter values [185, 186]. Very few studies considered hyper-parameter tuning for offline JIT-SDP. A study [187] used Sequential Bayesian Model-based Optimisation for tuning 6 hyper-parameters of an offline JIT-SDP approach based on temporal convolutional networks, which can improve the performance of JIT-SDP model. Kang et al. showed that performance of JIT-SDP can be

improved by optimising hyper-parameter using Harmony Search optimisation method.

All studies above were about offline hyper-parameter tuning. They require a pre-existing validation set to evaluate different hyper-parameter choices. When dealing with data stream problems, such choices may become inadequate over time. Therefore, these are not suitable for online hyper-parameter tuning.

3.2.3.3 Chunk-based Hyper-Parameter Tuning Approaches

There are few existing studies that investigated hyper-parameter tuning approaches that used chunk/batch processing of the data for training. Joy et al. [188] used Bayesian optimisation for hyper-parameter tuning in data streams, by dividing the data stream into chunks/batches and generating hyper-parameter combinations for the chunks using Bayesian Optimisation. Despite being applicable to data streams, this approach is still an offline hyper-parameter tuning approach, where hyper-parameters to be used for predicting a new chunk of data are tuned based on a past existing chunk of data. If the best hyper-parameter choice changes over time, this approach still becomes ineffective. Lin et al. [189] performed hyper-parameter optimisation for auto-augmentation inspired by the forward hyper-gradient proposed in [190] that eliminates the need of re-training. This study also used batch processing for training. Lawal et al. [191] proposed an incremental learning-model selection method for support vector machine (SVM) for data streams, which relies on incremental k-fold cross-validation to perform the incremental tuning of the support vector machine hyper-parameters. Pfaffe et al. [192] used different strategies (e.g. e-greedy) to select algorithms and also tuned the algorithm hyper-parameters using Nelder-Mead optimisation technique for raytracing application. Their approach also used batch processing for training.

However, these approaches are not applicable in online learning scenario as these approaches use batch processing.

3.2.3.4 Online Hyper-Parameter Tuning Approaches

Online hyper-parameter tuning requires continuously tuning hyper-parameter values over time without needing a pre-existing training and validation set. Hyper-parameter tuning in online learning scenario can be challenging due to the constraints that it presents such as continuous learning from data streams and dealing with concept drifts caused by change in the relationship between the input data and the output variable. There are fewer existing studies on online hyper-parameter tuning compared to offline hyper-parameter tuning.

Zhan et al. [193] proposed an algorithm that computes the hyper-gradient on the fly when a new datum is observed and then average of the historical hyper-gradient is computed. Song et al. [194] investigated to what extent parameter settings affect the performance of learning machines in software effort estimation in online learning scenario. Minku [16] proposed an online supervised tuning procedure that operates without fixed dataset with known output attribute values. They applied automated tuning procedure to a cross-company software effort estimation approach Dycom. This tuning procedure can adapt to changes that occurs in the data generation process over the time. However, the hyper-parameter values investigated by the approach are restricted to a pre-defined set of hyper-parameter combinations. Veloso et al. [195] proposed an online hyper-parameter tuning approach called SSPT, by transforming Nelder-Mead optimisation procedure to work with data stream. Their approach creates n + 1 learning models for n hyper-parameters in the exploration phase, these models are updated with each training example until the learning models converge to an optimal configuration. The best learning model obtained in the exploration stage is continuously updated and monitored for the occurrence of concept drifts. The detection of a concept drift triggers a new exploration phase.

However, there is no existing investigation of online hyper-parameter tuning for JIT-SDP. Hence, it is unknown whether online JIT-SDP is influenced by hyper-parameter choice,

if the best hyper-parameter choice would vary over time, and if tuning hyper-parameters would improve predictive performance of JIT-SDP over time.

3.3 Summary

This chapter discussed a number of component-based SDP and JIT-SDP studies. Earlier SDP studies considered WP context and assumed an offline scenario, as discussed in section 3.1.1.1. Most of the WP SDP aimed at predicting defect-prone modules. However, some later studies showed that change-level defect prediction (JIT-SDP) could be more advantageous than module-level defect prediction (Section 3.1.2.1). A major weakness of WP studies is that there is not enough training data available at the beginning of a project, hence, the WP SDP models may perform poorly. To deal with such issue, many researchers focused on investigating SDP in the CP context by using data from other projects (section 3.1.1.2). In particular, some studies explored CP JIT-SDP.

Despite showing that CP JIT-SDP can obtain promising results compared to WP JIT-SDP, a major drawback of these studies is that none of them considered JIT-SDP in an online learning scenario, whereas in practice JIT-SDP operates in an online scenario. A very few studies explored online JIT-SDP, but they are in the WP context. No study investigated online CP JIT-SDP. It is unclear how helpful CP data would be in online JIT-SDP. In particular, both WP and CP incoming examples may arrive over time in online scenario and the model can be trained with both CP and WP data. The availability of incoming WP data over time means that the role of CP data may be different in online scenarios compared to offline ones.

Furthermore, from the above literature review on hyper-parameter tuning, it can be seen that machine learning models are likely sensitive to hyper-parameters and may under

perform when adopting default hyper-parameter settings. Hyper-parameter tuning tends to improve performance compared to default parameter settings. So, it is important to choose an appropriate tuning procedure. Though there are many studies on hyper-parameter tuning in different areas of software engineering and some in the area of SDP, though most SDP studies used default settings. Among the JIT-SDP studies, none has adopted online hyper-parameter tuning procedures. Moreover, when dealing with data streams, the best hyper-parameter choice may (or may not [194]) change over time. Even though it is likely that JIT-SDP is influenced by hyper-parameter choice, it is unknown whether the best hyper-parameter choice *changes over time* in JIT-SDP. As a result, it is also unknown whether an online hyper-parameter tuning approach is necessary for JIT-SDP, or whether an offline hyper-parameter tuning approach would suffice.

Chapter Four

Datasets

This chapter describes the algorithm used in the data collection process, and also the open source and proprietary data sets used in our investigations.

4.1 SZZ Algorithm

This thesis uses software defect data collected using Commit Guru [115], a tool that analyses and provides change level analytics. Commit Guru extracts features to describe software changes and uses the "SZZ" algorithm, which was proposed by Sliwersk et al. [105] to identify defect-inducing change(s). SZZ identifies a defect-fixing commit, and compares the lines in the source code that differ in most recent and previous revisions of the same file. Lines that are modified or removed are considered to induce defect. SZZ contains two main parts. The first part is related to linking version control system (e.g. CVS) and issue tracking system (e.g. BUGZILLA) to find defect-fixing commits. The second part is related to identifying the defect-inducing commits.

In the first part, SZZ searches the change log for FIXED/CLOSED issues, regular expressions to identify defect number and relevant keywords in the commit messages that

are likely to point out a real defect-fixing change. For this, it breaks down the commit message in to a sequence of tokens and gives them a confidence value by applying first a syntactic analysis on the log message, and then a semantic analysis between the log message and the defect report.

The second part of the SZZ algorithm is concerned with defect-inducing commits. SZZ takes the location of the identified defect-fixing commit, and searches back for the commit that initially introduced the defect. The algorithm uses the *diff* command from git to detect lines that have been modified in the file revision. Then, by using the built-in annotate/blame functionality of the version control system, SZZ locates the commit that modified or deleted those lines for the last time in the previous version of the change. Finally, these change(s) are flagged as suspicious as defect-inducing commits only if they are committed before the defect was reported.

4.2 Open source projects

We have used ten open source datasets extracted from open source GitHub repositories same as [18] , which are made available at https://github.com/sadiaTab/datasets_jitsdp. The initial count of commits for the 10 open source datasets was 200190. However, 1722 commits were eliminated due to the presence of empty value for all input features which Commit Guru failed to collect. Consequently, the remaining count of commits for all 10 datasets, following the processing phase, is 198468. Table 4.2 shows the number of changes before processing, the number of changes that were eliminated during processing, and the number of changes after processing, for each of the datasets. Summary statistics and plots for the input features for these datasets can be found in the Section B.1 in Appendix B.

We compute Cohen's Kappa score for the open source datasets. Cohen's Kappa is a

statistical measure of the level of agreement between two observers of a similar phenomenon, while compensating for agreements that can be attributed to chance [196]. The formula for calculating Cohen's Kappa is:

$$K = \frac{P_0 - P_e}{1 - P_e} \tag{4.1}$$

where P_0 is the total agreement probability, or the accuracy, and P_e is the agreement probability which is due to chance [196].

Kappa values range from -1 to 1.00, with larger values indicating better reliability. According to Hall et al. [197], the following interpretation of Cohen's Kappa score can be given: [-1..0] less than chance agreement; [0.01..0.20] slight agreement; [0.21..0.40] fair agreement; [0.41..0.60] moderate agreement; [0.61..0.80] substantial agreement; [0.81..0.99] almost perfect agreement.

We randomly selected 50 defect-fixing and 50 non-defect-fixing software changes from each of four datasets, namely Tomcat, Broadleaf, Npm and Nova to calculate the Kappa score. Defect-fixing changes are used to identify defect-inducing changes by SZZ. Here, defect-fixing and non-defect-fixing changes are considered to calculate the kappa score instead of checking the agreement with regards to clean and defect-inducing changes. This is because manually checking whether a change is defect-inducing directly for the purpose of computing the kappa score would be extremely time consuming, if even possible at all. The Tomcat and Broadleaf datasets were chosen for having the largest (27.54%) and smallest (16.86%) percentage of defect-inducing changes, respectively. For Npm and Nova, percentages of defect-inducing change are 17.77% and 25.37%, respectively.

These commit messages have been manually checked by three expert human reviewers. To measure the level of agreements, Kappa scores have been calculated based on 6 comparisons: 1) between reviewer 1 and SZZ, 2) between reviewer 2 and SZZ 3) between reviewer 3 and SZZ, 4) between reviewer1 and reviewer 2, 5) between reviewer1 and reviewer

Table 4.1: Kappa scores

Data	Rev 1 with SZZ	Rev 2 with SZZ	Rev 3 with SZZ	Rev 1 with Rev 2	Rev 1 with Rev 3	Rev 2 with Rev3	Considering Unsure Commit
tomcat	0.76	0.76	0.62	0.76	0.58	0.69	yes
broadleaf	0.7	0.86	0.72	0.72	0.59	0.74	yes
npm	0.66	0.66	0.64	0.64	0.62	0.7	yes
nova	0.54	0.66	0.6	0.72	0.56	0.7	yes
tomcat	0.8	0.81	0.65	0.88	0.76	0.8	no
broadleaf	0.7	0.91	0.83	0.81	0.72	0.87	no
npm	0.73	0.72	0.73	0.69	0.7	0.76	no
nova	0.57	0.65	0.66	0.75	0.62	0.76	no

3, and 6) between reviewer2 and reviewer 3.

In the event that a reviewer is uncertain whether a commit message is a defect-fixing commit or not, they are instructed to express their opinion as either TRUE_but_unsure or False_but_unsure. The former is used if they believe the commit is a defect-fixing one but are unsure, while the latter is used if they believe it is not a defect-fixing commit but are unsure.

Table 4.1 shows Kappa scores calculated both including and excluding the software changes that reviewers were unsure about, which is shown in the “Considering Unsure Commit" column in the table. In this column, “yes" indicates that the unsure commits were included in the calculation, and “no" indicates that they were not considered while calculating the Kappa scores.

While comparing between the reviewers and SZZ (Table 4.1 columns 2,3,4), table 4.1 shows that, most datasets achieved substantial agreement, with Kappa scores in the range of 0.61 to 0.80. Some cases even showed almost perfect agreement with Kappa scores in the range of 0.81 to 0.99, such as Broadleaf (Kappa scores 0.86 and 0.91 with reviewer 2 and the SZZ, for both including and excluding unsure commits). In two cases for Nova with Reviewer 1 and SZZ, moderate agreement was achieved, with Kappa scores in the range of 0.41 to 0.60.

While comparing between reviewers with each other (table 4.1 columns 5,6,7), for all datasets, there was at least substantial agreement between reviewer 1 and reviewer 2, and between reviewer 2 and reviewer 3. In some cases, there was almost perfect agreement (such as tomcat, broadleaf with reviewer 1 and reviewer 2, excluding unsure commits). For reviewer 1 and reviewer 3, including unsure commits, the level of agreement was at least moderate, while excluding such commits resulted in substantial agreement.

Excluding unsure commits usually resulted in higher Kappa scores compared to including unsure commits. This could be attributed to the fact that some commit messages are not clearly indicative of whether they are related to defect-fixing or not, leading to divergent opinions among reviewers. Therefore, removing these uncertain commits naturally leads to increased agreement levels between the reviewers.

Based on the results presented above, there is evidence of a large Kappa score and higher level of agreement between SZZ and the reviewers. These findings suggest that SZZ's measurement approach is comparable to human judgment, suggesting that SZZ's assignment of labels to the software changes was in line with the assignment that would have been performed by human labelers if they had been given access to the same information as SZZ. In particular, it is encouraging to see that the Kappa between reviewer and SZZ was not smaller than the Kappa between two reviewers. This suggests that the dataset's quality is acceptable. However, dataset noise remains a threat to validity as SZZ may generate mislabeling noise [198, 199].

All datasets were extracted based on Commit Guru [115]. The input features include 14 change metrics (Table 4.3) grouped in to 5 dimensions derived from the source control repository data of a project. The choice of metrics depends on the specific requirements and goals of the classification task. This thesis is focused on JIT-SDP. Hence, it is very reasonable to use studies based on JIT-SDP to justify choice of metrics instead of other SDP

Table 4.2: An overview of the projects (Open source and Proprietary)

Project	**Total Change (before processing)**	**Eliminated (during processing)**	**Total Changes (after processing)**	**# Defect-inducing Changes**	**% Defect-inducing Changes**	**Median Defect Discovery Delay**	**Time Period**	**Main Language**	**Project Type**
Tomcat	18956	49	18907	5207	27.54	200.5798	27-03-2006 - 06-12-2017	Java	Open source [18]
JGroups	18432	107	18325	3153	17.21	116.1565	09-09-2003 - 05-12-2017	Java	Open source [18]
Spring-Int	9026	276	8750	2333	26.66	415.1201	14-11-2007 - 16-01-2018	Java	Open source [18]
Camel	30738	163	30575	6255	20.46	28.1947	19-03-2007 - 07-12-2017	Java	Open source [18]
Brackets	17562	198	17364	4047	23.31	14.454	07-12-2011 - 07-12-2017	JavaScript	Open source [18]
Nova	49275	286	48989	12430	25.37	88.5615	28-05-2010 - 28-01-2018	Python	Open source [18]
Fabric8	13487	381	13106	2589	19.75	39.1833	13-04-2011 - 06-12-2017	Java	Open source [18]
Neutron	19690	168	19522	4607	23.6	82.5097	01-01-2011 - 27-12-2017	Python	Open source [18]
Npm	7936	16	7920	1407	17.77	111.514	29-09-2009 - 28-11-2017	JavaScript	Open source [18]
Broadleaf	15088	78	15010	2531	16.86	42.5818	19-12-2008 - 21-12-2017	Java	Open source [18]
C1	1030	0	1030	confidential	confidential	18.17	06-09-2018 - 17-07-2019	Javascript	Proprietary
C2	601	0	601	confidential	confidential	1.93	16-10-2018 - 19-07-2019	Javascript & 3D studio	Proprietary
C3	417	0	417	confidential	confidential	2.98	05-09-2018 - 19-07-2019	Python	Proprietary
C4	341	0	341	confidential	confidential	4.13	17-04-2019 - 17-10-2019	Javascript & 3D studio	Proprietary
C5	323	0	323	confidential	confidential	4.24	22-05-2019 - 13-11-2019	Javascript & 3D studio	Proprietary
C6	555	0	555	confidential	confidential	2.92	13-05-2019 - 13-12-2019	Javascript & 3D studio	Proprietary
C7	546	0	546	confidential	confidential	3.33	16-04-2019 - 18-11-2019	Javascript & 3D studio	Proprietary
C8	798	0	798	confidential	confidential	0.81	22-05-2019 - 16-09-2019	Python	Proprietary
C9	694	0	694	confidential	confidential	0.77	30-04-2019 - 21-08-2019	Python	Proprietary

studies. These 14 change metrics have been shown to be adequate for JIT-SDP in previous work [6] and have been adopted in previous online and offline JIT-SDP studies [18, 20, 200, 201, 186, 202, 203, 13, 204]. We describe each dimension and its relevant factors for the 14 change metrics:

- **Diffusion dimension:** According to [6], diffusion dimension can be leveraged to determine the likelihood of a change being defect-inducing. The diffusion dimension consists of 4 metrics: NS (number of modified subsystems), ND (number of modified directories), NF (number of modified files), Entropy (distribution of modified code across each file).

- **Size dimension:** Size of a change can be an important feature for defect-prone modules [205]. A larger has higher chance of inducing defects. The size dimension consists of three different metrics: LA (lines of code added), LD (lines of code deleted), LT (lines of code in a file before the change).

- **Purpose dimension:** The purpose of a change is also important to provide information about defect inducing change. A change that fixes a defect is usually more likely to introduce a new defect, as a file that was defective in past, may have more chance to have defect in future [206, 207]. Keywords like "bug", "fix", "defect" or "patch" are usually used in the commit messages in the version control system to determine whether or not the change fixes a defect. The following metric is used to determine whether the change is defect-fixing or not: FIX (whether or not the change is a defect fix).

- **History dimension:** The history of a change can provide useful information about whether a change can be defect inducing or not. We considered 3 metrics for history dimension: NDEV (number of developers that changed the modified files), AGE (average time interval between the last and the current change), NUC (number of unique changes to the modified files).

- **Experience dimension:** Higher programming experience by a developer may decrease probability of a change to be defect inducing [103]. There are 3 metrics considered for this dimension: EXP (developer experience), REXP (recent developer experience) and SEXP (developer experience on a subsystem).

Table 4.3: Change Metrics (adapted from [6])

Dimension	Metric	Description
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	Fix	Whether or not the change is a defect fix
History	NDEV	Number of developers that changed the files
	AGE	Average time interval between the last and the current change
	NUC	Number of unique last changes to the files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

All software changes were chronologically ordered based on the author timestamp, which is provided by Git as the timestamp when the commit was first created by the author. Author timestamp is recommended over commiter timestamp for studies involving time-based Git data [208]. This timestamp is used in our work as the moment in time when a JIT-SDP model is required to provide a prediction for this software change. The timestamp when a software change is found to be defect-inducing is the author timestamp of the commit that fixes this software change, as retrieved by Commit Guru [115]. Such timestamps and the waiting period w are used to determine when the training examples become available for training. Training examples are generated using the online procedure recommended by Cabral et al. [51] to take verification latency into account for all approaches studied in this thesis. In particular, a software change becomes a training example either when a defect

is found to be associated to it, or once a pre-defined waiting period of w days has passed, whichever is earlier. This waiting period represents the amount of time that it takes for one to be confident that the software change in question is clean. In other words, if no defect is found to be associated to the software change during the waiting period, a training example of the clean class is created to represent this software change. Otherwise, a training example of the defect-inducing class is created immediately after the defect is found. If, after the waiting period, a defect is found to be associated to a change that was previously considered clean, a new defect-inducing training example is created for it. Training examples are used to update the classifier as soon as they are created. In this way, both the moment when software changes are predicted and the moment when software changes become available for training fully respect chronology. Commit guru uses git log to get the commits. Git log lists all commits in the current branch. If another branch is merged into it, the commits from that other branch will also show, and are chronologically ordered with the other commits based on author timestamp.

4.3 Proprietary projects

The experimental results for Chapter 5, we have investigated 9 proprietary datasets along with previously described 10 open source datasets. Details of the proprietary datasets can be found in 4.2. We used same 14 change metrics described in previous section for the proprietary projects. To collect the proprietary data for this study, all commit messages from the three repositories were extracted using the git log command by a senior software engineering academic. A Chinese language native speaker knowledgeable of programming was asked to read the commit messages to identify keywords that can be used to identify *corrective* commits. Keywords, representative commit messages of corrective and non-corrective cases, and commit messages for which the native speaker was uncertain about were stored in a separate

file. This gave a total of 57 commit messages. The senior software engineering academic then went through the file providing his independent classification of the commits as corrective or non-corrective, by using Google Translate to translate the commit messages into English. Unclear Google translations were discussed with the native speaker. Commit messages for which there was still some doubt after this discussion were further discussed by the senior software engineering academic with the Company, who confirmed whether they were corrective or non-corrective. Upon discussion with the native speaker, a list of keywords to identify commits was created, which was presented to the Company. The Company confirmed that the list of keywords was adequate for their projects.

The list of keywords was used by the senior software engineering academic as input for Commit Guru to identify corrective commits, which were then used to identify which changes are defect-inducing or clean [115] to generate the data sets. As a data quality assurance procedure, all commit messages considered by Commit Guru as corrective and a sample of non-corrective commit messages were double checked by the native speaker. There was a disagreement between Commit Guru's classification and the native speaker in only 3 cases, and the native speaker was unsure of the correct classification in 7 cases. Therefore, Commit Guru's classification of commits as corrective and non-corrective was deemed appropriate.

4.4 Reasons for concept drift and performance drops in JIT-SDP models

Existing work has shown strong evidence that JIT-SDP suffers from concept drift [20, 18, 19] (i.e. changes in the defect generating process). Due to change in the underlying defect generating process, JIT-SDP model trained on data from certain period of time may no longer suitable in later period of time, which may cause drops in the predictive performance.

Mcintosh and Kamei [20] show that characteristics (input feature values, e.g. size, experience) of defect-inducing software changes can fluctuate during the lifecycle of a software project. For example, the magnitude of the change (size) or the experience of the developer may increase as the software system grows. This fluctuations can negatively impact the predictive performance of the defect prediction model leading to concept drift. This is because due to such fluctuations or change in the data, past data may become irrelevant at present, and model trained with such irrelevant data may perform poorly.

Cabral et al. [18] conducted a study on concept drift affecting the prior probabilities of the classes in JIT-SDP, using the same 10 open source datasets as this thesis. Their detailed analysis of the data indicates potential reasons behind the concept drift such as refactoring of the project. They observed that the ratio of defect-inducing software changes for Fabric8, Jgroups, Camel, Spring-Integration, Nova, and NPM fluctuates over time, with certain periods of increase followed by subsequent decrease. In the case of Fabric8, Cabral et al. [18] identified increases in defect-inducing software changes at certain time periods , which aligned with major refactorings reported on the official website at http://fabric8.io/gitbook/overview.html#history. The RefactoringMiner tool was used to confirm the time steps corresponding to these refactorings. This suggests that refactoring may be linked to large variations in the problem's imbalance status, which can disturb machine learning algorithms that are unable to adapt rapidly to such changes [52]. Spring-Integration also exhibits a period where the ratio of defect-inducing software changes significantly decreases, resulting in an extremely high level of imbalance within the problem. This presents a challenge for machine learning algorithms, which tend to struggle with highly imbalanced SDP problems [209].

For some other datasets, such as Camel and Neutron, the ratio of defect-inducing changes present decreasing trend over time. Cabral et al. [18] shows that the ratio for Camel starts at 40% and decreases to less than 10% at the end of the project, which increases the

challenge posed by class imbalance [52]. Cabral et al. [18] argued that the ratio of defect-inducing changes become smaller over time as the project matures and becomes less defective. Also, defects may be found more quickly at the beginning of the project, which may cause increased rate of the defect-inducing changes.

Another study by Cabral et al. [19] (using same 10 open source datasets as this thesis) provided a detailed analysis of the factors contributing towards concept drift in JIT-SDP. They show that the combination of various concept drifts and verification latency substantially hinders the consistency of predictive performance in JIT-SDP approaches, leading to a significant decline in their reliability over time. They observed drop in the predictive performance for some of these datasets were related to two types of concept drift $\rho(y|\bar{x})$ (changes in the associated class described by features) and $\rho(\bar{x})$ (changes in the probability distribution of the features), where (x, y) is software change, $\bar{x}$ is the vector of features and y is its respective class label. Their proposed approach that can tackle class imbalance evolution led to significantly more stable and improved predictive performances for these datasets. This suggests that the presence of class imbalance evolution can indeed hinder predictive performance of JIT-SDP causing concept drift.

Based on the above analysis, it is apparent that changes in the characteristics of the data, changes in the class imbalance ratio, which may occur due to change in data generation process such as project refactoring and project maturity are some of the potential reasons contributing to the concept drift causing sudden drop in the predictive performance of JIT-SDP models.

Chapter Five

Online Cross-project JIT-SDP

This chapter is based on the first and second publications (ICSE [29] and IEEE Transactions on Software Engineering [210]) listed in Chapter 1.

5.1 Introduction

Similar to SDP at the module/file level, JIT-SDP classifiers require sufficient amount of training data which is not available during the initial phase of a project. To overcome this problem, previous work has proposed Cross-Project (CP) JIT-SDP, where historical data from other projects are used to train the classifier [12]. Existing CP JIT-SDP work [12] assumes an offline learning scenario, where classifiers are built based on a pre-existing training set and never get updated after that. This means that the CP classifier is never trained on Within-Project (WP) data. However, in practice, JIT-SDP is an online learning problem [51], i.e., a problem where both additional CP and WP training examples arrive over time, forming a data stream that can be used to update JIT-SDP models whenever a new labelled software change arrives.

The role of CP data in such online learning scenario is unclear. CP JIT-SDP has never

been investigated in online mode before. In particular, it is unknown whether CP data is only helpful at the very early stages of the project when there is little WP data, or if it brings a prolonged benefit to the predictive performance of the classifier. For instance, it may be that CP approaches using an augmented training data stream formed by both WP and CP examples lead to increased predictive performance even at later stages of the project, given that classifiers are built using more data than WP classifiers trained only with WP data. Or, it may be that CP data cause such approaches to obtain worse predictive performance than WP classifiers once enough WP data is used for training. Besides, prediction quality can fluctuate due to variations (concept drifts) in the underlying defect generating process [20, 51], rendering JIT-SDP classifiers unreliable. The use of CP training data could potentially help to handle concept drift. This is specially the case considering that JIT-SDP is a class imbalance problem, where the number of defect-inducing software changes is typically much smaller than that of clean changes. In such kind of problem, it would take a lot of time to collect new WP defect-inducing examples to recover well from concept drift. CP training examples could potentially help to recover from concept drift more quickly.

This chapter thus aims at investigating when and to what extent CP JIT-SDP data can be helpful in a realistic online learning scenario. In this context, this chapter aims to answer the first research question introduced in Chapter 1:

RQ1: Can cross-project learning help to improve predictive performance in online JIT-SDP? How?

This research question can be further divided into following sub-research questions:

RQ1.1 Can CP data help to improve predictive performance in the initial phase of a project, when there is little or no WP training data available? For how long and to what extent?

RQ1.2 Can CP data help to prevent sudden drops in predictive performance, which may be caused by concept drift? To what extent?

RQ1.3 What are the effects of CP data on the predictive performance during stable periods of the projects? Could the CP data be detrimental during stable periods, given that enough WP data may be available for training?

RQ1.4 What is the overall effect of using CP and WP data together on the predictive performance throughout the development of software projects? In particular, do classifiers trained on both CP and WP data improve, deteriorate or retain similar overall predictive performance to WP classifiers?

RQ1.5 Is it necessary to update CP approaches over time once a project starts, or is it enough to use models trained only with CP data produced before the project being predicted commences? If it is enough, online learning is not really necessary.

To answer our RQs, we investigate four approaches: (1) a single online learning CP classifier trained on incoming WP and CP training examples, (2) an online learning CP ensemble where each classifier is trained on incoming training examples from a different project, (3) a single online learning CP classifier that filters out CP training examples that are likely to be very different from recent WP examples and (4) an OP classifier that is trained only with CP training data which arrive before the project of interest commences, and is never updated with incoming WP data. The first 3 approaches are online and are enhancements of the approaches used in the JIT-SDP literature [12], so that they can operate in online mode. The last approach represents CP approaches from the offline learning literature, which make use of only OP data. All of these approaches are compared against online WP [51, 52] approaches. Our experiments based on 9 proprietary and 10 open source software repositories show that the first and third approaches are helpful to improve predictive performance in JIT-SDP compared to WP classifiers, while the second and fourth are not. The main

contributions of this chapter are the following:

- We provide the first investigation of CP JIT-SDP in a realistic online learning scenario.
- We show how to adapt CP JIT-SDP approaches so that they can be used in an online learning scenario.
- We develop a new methodology to systematically analyse approaches during periods of drop in predictive performance (and conversely also periods of stability) in online learning. This enables us to know how much more robust CP JIT-SDP approaches are in terms of preventing such drops (and thus leading to more reliable predictions over time) compared to WP JIT-SDP approaches.
- By answering RQ1.1-1.3, we provide a detailed analysis revealing that the extent to which CP can be helpful in JIT-SDP is much larger than previously thought. It can not only provide significant improvements in predictive performance during the initial stages of a project (RQ1.1), but also can prevent drops in predictive performance over time (RQ1.2) and improve predictive performance during stable periods (RQ1.3).
- By answering RQ 1.4, we reveal that the use of CP data combined with WP data can improve overall predictive performance (rather than just achieving similar predictive performance) compared to WP learning in JIT-SDP.
- We show that online CP approaches outperform OP approaches (RQ1.5), highlighting the benefits of online learning in JIT-SDP.
- We show that it is better to use CP and WP data together to build a single classifier, rather than building different classifiers with disjoint subsets of the data.
- We provide an analysis of sensitivity to hyperparameters for the most promising CP approach (Filtering), providing an understanding of their effect on predictive performance.

The chapter is further organized as follows. Section 5.2 introduces our online CP JIT-SDP approaches. Section 5.3 explains the experimental setup for answering the RQs. Section 5.4 explains the results of the experiments. Section 5.5 presents the sensitivity analysis. Section 5.6 presents the analysis of computational cost. Section 5.7 explains the implications to practice. Section 5.8 explains the threats to validity. Section 5.9 presents the summary.

5.2 Online CP JIT-SDP Approaches

In this section, we modify and enhance three CP JIT-SDP approaches adopted in [12] to enable them to be applied to online JIT-SDP. All our algorithms fully respect chronology following the procedure described in Chapter 4. In particular, they never use future CP/WP training examples, future knowledge about labels (i.e., defect-inducing or clean), or test examples, for training a model used for testing the present. Our CP approaches can be trained not only with CP and WP training examples that are made available over time after the first WP commit, but also with CP training examples produced before the first WP commit.

5.2.1 All-in-One (AIO) Approach

Existing offline JIT-SDP work [12] assumes that CP classifiers are not trained with any WP data. Unlike offline approaches, the AIO online CP approach includes WP data for training. All incoming CP and WP training data are considered as part of a single data stream of training examples, which are used to train a single online classifier as soon as they are produced. The data stream is in chronological order, i.e., the training examples are sorted based on the Unix timestamp of their author creation. A new incoming change

Algorithm 2 All-in-One (AIO) Approach

Input: S = stream of incoming changes from several projects, b = index identifying the target project, w = waiting period

```
1: Initialise predictive model m
2: for each incoming change x_p^t ∈ S do          ▷ x_p^t is a change arriving from project p at timestamp t
3:     if p = b then
4:         ŷ ← predict(m, x_p^t)
5:     end if
6:     store x_p^t in a queue WFL-Q          ▷ WFL-Q is a queue of incoming examples waiting to be used for training
7:     for each item q^i in WFL-Q do
8:         if a defect was linked to q^i at a timestamp ≤ t then
9:             create a defect-inducing training_example for q^i
10:            train(m, training_example)
11:            remove q^i from WFL-Q
12:        else
13:            if q^i is older than w then
14:                create a clean training_example for q^i
15:                train(m, training_example)
16:                remove q^i from WFL-Q
17:                store training_example in CL-H          ▷ CL-H is a hash of clean training examples
18:            end if
19:        end if
20:    end for
21:    if a defect was linked to a training_example in CL-H at a timestamp ≤ t then
22:        Swap the label of training_example to defect-inducing
23:        train(m, training_example)
24:        remove training_example from CL-H
25:    end if
26: end for
```

that does not belong to the target project is used for training (respecting the chronology and verification latency), but not for prediction. Algorithm 2 shows the pseudocode for the AIO approach.

The classifier is initialised as an empty model that always predicts "clean". When a new incoming change x_p^t is received at time step t (line 3), the algorithm first checks if this change belongs to the target project, i.e., to the project whose changes are being predicted (line 4). If it does, a prediction is provided for this change. After that, x_p^t is stored in

a queue for a pre-defined waiting period (line 7). All software changes in this queue are checked to see whether they can be used for training (line 8 to 21). If a defect is found to be associated to a given software change in the queue during the waiting period (lines 9 to 12), a defect-inducing training example is created to represent this change and is used for training. If a defect is not found by the end of the waiting period of a given change (lines 13 to 20), a clean training example is created for this change and is used for training. After that the change is removed from the queue. The algorithm also checks whether there is any past change found to be defect-inducing, but that was previously considered as a clean training example (lines 22 to 26). If there is, the classifier is trained using that change as a defect-inducing training example.

The key difference between our approach and the data merging approaches used by Kamei et al. [12] is that their learning was offline (without taking into account incoming training examples and verification latency) and they did not include WP data for training. In our proposed approach, the learning is online, takes verification latency into account, and the classifier is trained on both CP and WP data whose labels are produced before the current timestamp.

5.2.2 Ensemble Approach

The Ensemble approach uses an ensemble of classifiers rather than a single classifier. A separate classifier is built from each project's separate training data stream (e.g., for 10 projects there will be 10 different classifiers). This includes both CP and WP data streams. Each change belonging to the target project is then predicted by all the classifiers, and the mean of the predicted probabilities retrieved by the classifiers is calculated. This mean is used to predict whether the change is clean or defect-inducing. Algorithm 3 shows the pseudocode for the Ensemble approach. As with the AIO approach, the chronological order

Algorithm 3 Ensemble approach

Inputs: S = stream of incoming changes from n projects, b = index identifying the test project, w = waiting period

initialise ensemble model M consisting of n models $\{m_1, m_2, ..., m_n\}$
for each incoming change $x_p^t \in S$ **do** ▷ x_p^t is a change arriving from project p at timestamp t

  if $p = b$ **then**
    $\hat{Y}$-$List \leftarrow$ get probability $\hat{y}_i$ of change x_p^t being defect-inducing from each model in M
    $MeanY$ = mean of values in $\hat{Y}$-$List$
    if $MeanY < 0.5$ **then**
      $\hat{y} = 0$
    else
      $\hat{y} = 1$
    end if
  end if
  store x_p^t in a queue WFL-Q_p for project p ▷ WFL -Q_p is the queue of incoming changes of project p waiting to be used for trained

  for each model m_p in M **do**
    for each change q^i in WFL-Q_p **do**
      if a defect was linked to q^i at a timestamp $\leq$ t **then**
        create defect-inducing $training_example$ for q^i
        train(m_p, $training_example$)
        remove q^i from WFL-Q_p
      else
        create a clean $training_example$ for q^i
        train(m_p, $training_example$)
        remove q^i from WFL-Q
        store $training_example$ in CL_p-H ▷ CL_p-H is a hash of clean training examples for project p
      end if
    end for
  end for

  if a defect was linked to a $training_example$ in CL_p-H before time t **then**
    Swap the label of $training_example$ to defect-inducing
    train(m_p, $training_example$)
    remove h from CL_p-H
  end if
end for

of the training examples is always respected.

The predictions given by the ensemble are based on simple voting. The key differences between our approach and the approach used by Kamei et al. [12] are that our approach is online, and our ensemble contains a classifier built from WP training examples that have been labelled up to the current timestamp, rather than using only CP data classifiers.

5.2.3 Filtering Approach

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

The average of the smallest K distances is calculated. If this average distance is equal to or lower than a maximum threshold, the CP training example is allowed to train the classifier. Discarded CP training examples are kept in a fixed-sized queue. This queue is checked in every iteration to see whether any old discarded CP training example has now become suitable for training. This can be useful in case concept drift causes such discarded

Algorithm 4 Filtering Approach

Input: S = stream of incoming changes from several projects, b = index identifying the test project, w = waiting period, windowSize = size of the WP-Q sliding window, K = number of top short distances to be used, $maxDist$ = distance threshold for similarity, $cpqSize$ = maximum size of the queue CP-Q of dissimilar CP instances to be re-checked for similarity in the future

```
1: Initialise predictive model m
2: for each incoming change x_p^t ∈ S do                ▷ x_p^t is a change arriving from project p at timestamp t
3:     if avgDist(training_example, WP-Q,K) ≤ maxDist for any training_example in CP-Q then
4:         train(m, training_example)
5:         remove training_example from CP-Q
6:     end if
7:     if p = b then
8:         ŷ = predict(m, x_p^t)
9:     end if
10:    store x_p^t in a queue WFL-Q        ▷ WFL-Q is a queue of incoming examples waiting to be used for training
11:    for each item q^i in WFL-Q do
12:        if a defect was linked to q^i at a timestamp ≤ t then
13:            Create a defect-inducing training_example for q^i
14:        else
15:            if q^i is older than w then
16:                Create a clean training_example for q^i
17:                store q^i in CL-H                    ▷ CL-H is a hash of clean training examples
18:            end if
19:        end if
20:        if a training_example was created for q^i then
21:            if avgDist(training_example, WP-Q,K) ≤ maxDist or this is a WP change then
22:                train(m, training_example)
23:            else
24:                Add training_example to CP-Q
25:            end if
26:            Remove q^i from WFL-Q
27:            Slide WP-Q if training_example is WP
28:        end if
29:    end for
30:    if a defect was linked to a training_example in CL-H before time t then
31:        swap label of training_example to defect-inducing
32:        remove training_example from CL-H
33:        if avgDist(training_example,WP-Q,K) ≤ maxDist then
34:            train(m, training_example)
35:        else
36:            add training_example to CP-Q
37:        end if
38:    end if
39: end for
```

examples to become relevant at some point.

Algorithm 4 shows the pseudocode for the Filtering approach. The classifier is initialised as an empty model that always predicts "clean". When a new incoming change x_p^t from project p is received at time step t (line 3), the algorithm checks whether there are any old CP training examples that were previously not used for training due to their dissimilarity to WP examples, but that are now suitable for training due to their similarity to the current WP sliding window (lines 4–7). If there are, they are used for training. Then, a prediction is given if the change x_p^t belongs to target project (lines 8–10). The change x_p^t is then stored in a queue (line 11), waiting to be labelled. All changes in this queue are checked to see whether they can be labelled (lines 12 to 30). If they can, corresponding training examples are created and used for training only if they are similar enough to the WP sliding window (lines 21–23). If they are not similar enough and are CP examples, they are stored in the queue $CP\text{-}Q$ of discarded CP examples for possible future use (line 25). The sliding window is updated (slided) if the training example is WP (line 28). The algorithm also checks whether any change that was previously considered as clean has now been associated to a defect and uses it for training, but only if it is similar enough to the WP window (lines 31–39). An analysis of the dropped CP instances by the Filtering approach can be found in 5.4.7.

5.3 Experimental Setup

This section explains the experimental setup for answering the RQs introduced in Section 5.1. The experiments in this chapter use 10 open source and 9 proprietary data sets described in the Table 4.2 of Chapter 4. Replication package for this chapter is available at https://zenodo.org/record/5994499

5.3.1 Compared Approaches

RQ1.1-1.4 will be answered by comparing the predictive performance of the online CP approaches presented in Section 5.2 against that of online WP approaches, providing a detailed understanding of when and to what extent CP data can be helpful to improve predictive performance in JIT-SDP. Comparisons of online CP against OP approaches will be used to answer RQ1.5, revealing whether online learning is really helpful in JIT-SDP. As with the online CP approaches, the WP and OP approaches take verification latency into account following Cabral et al. [51]'s waiting time procedure and always respect chronology.

A dummy classifier is also used for a preliminary investigation. Here, the dummy classifier always predicts 50% of each class, which means it randomly assigns each data point to a class with equal probability, without learning from the data. The dummy classifier does not learn from the data and its predictions are completely based on chance. The dummy classifier that predicts 50% for each class in a binary classification where class labels are clean and defect-inducing, can be formalised as follows: If X is the feature matrix, then for each data point $x \epsilon X$, the dummy classifier predicts class label of x as clean with probability 0.5, and as defect-inducing with probability 0.5. The dummy classifier that always predicts 50% for each class has a true positive rate and a true negative rate of 0.5, since there are an equal number of positive and negative predictions. Therefore, the G-Mean, which is the square root of the product of the true positive rate and the true negative rate, is also 0.5 or 50%.

The online WP approaches are approaches able to learn WP training examples over time, but learn no CP data. The OP approaches represent the core CP strategies adopted in the offline CP JIT-SDP literature [12]. They do not use WP data for training, except for when filtering is adopted, in which case WP data are only used to decide which CP data to filter out from the training process. As existing CP JIT-SDP work does not adopt

online learning, its corresponding approaches would only train JIT-SDP models with CP data produced prior to the development of the project of interest. Therefore, this is also the case for the OP approaches investigated in this study. Both an AIO and a Filtering OP approach are adopted. The OP AIO uses all the OP data to train the same predictive model, as in [12]. To ensure fair comparisons for the purpose of answering RQ1.5, the OP Filtering approach makes use of the same filtering mechanism explained in Section 5.2.3.

For simplicity, when referring to the CP approaches AIO and Filtering presented in Section 5.2, we will omit the term "CP". When referring to the OP AIO and Filtering approaches, we will make use of the term "OP" explicitly.

Given an open source repository corresponding to a project of interest, for CP approaches, all 10 open repositories are considered as the CP data. For the OP approaches, the other 9 open source repositories are considered as the OP data. Given a proprietary repository, the AIO approaches were investigated in two different ways: (a) combining both 10 open source and the 9 proprietary repositories as the CP data and (b) only with the 9 proprietary repositories as the CP data. For OP AIO, cases similar to (a) and (b) were used, but the OP data excluded the data from the project of interest. CP and OP Filtering approaches were investigated only in the scenario (a), again with the OP data excluding the data from the project of interest. They were not investigated using only proprietary data because filtering reduces the amount of training data by filtering dissimilar instances, and the amount of proprietary data is already small compared to the amount of open source data. Hence, using only proprietary data would likely be insufficient to train models for Filtering approaches. In addition, due to the poor results obtained by the Ensemble approach on the open source data, this approach was not run for the proprietary data.

JIT-SDP is a class imbalance problem [51, 6, 10, 12], and the open source data used in this study are known to be affected by this issue [51]. Therefore, learning approaches need

to use online learning classifiers that can deal with that. Two state-of-the-art approaches for online class imbalanced learning are Improved Oversampling Online Bagging (OOB) and Improved Undersampling Online Bagging (UOB) [52]. Also, Cabral et al. [51] proposed a new approach called Oversampling Rate Boosting (ORB), which improves the predictive performance further in the context of WP JIT-SDP. The three approaches are ensembles of Hoeffding Trees [211]. These are online learning base classifiers, which are updated incrementally with each new training example, preserving old knowledge without requiring storage of old examples. Previously, OOB and UOB achieved similar performance in WP JIT-SDP [51], and ORB performed the best. Thus, only OOB and ORB are selected as the base classifiers for the approaches investigated in our study.

5.3.2 Predictive Performance

We define a time step as a sequential number indicating the order of WP commits according to their author timestamp. Each WP commit requires a WP software change to be predicted as defect-inducing or clean. Therefore, to evaluate predictive performance, we investigate how well a classifier can predict the WP commit received at each time step. At this time step, the online CP and WP classifiers used to give a prediction will have been updated with all CP+WP and WP labelled training examples that became available *prior* to the Unix timestamp corresponding to this time step, respectively. Therefore, they correspond to classifiers that could really have been used in practice to make such prediction. The OP approach always uses the OP classifier that was created based on all the labelled CP training examples that were available prior to the beginning of the project being predicted. The analyses done for RQ1.1, RQ1.2, RQ1.3 and RQ1.4 will concentrate on the predictive performance (1) in the beginning of the projects, (2) during periods of time where we can observe sudden drops in predictive performance of the WP approach and (3) during periods

of stable predictive performance of the WP approach, and (4) average across time steps, respectively. The analysis for RQ1.5 will discuss (1-3). The procedure used to distinguish between (1), (2) and (3) is described in Section 5.3.3.

We adopt the Geometric Mean (G-Mean) of Recall0 and Recall1 as measures of predictive performance, where Recall0 is the recall on the clean class and Recall1 is the recall on the defect-inducing class. These metrics were chosen because they are the most recently recommended for online class imbalance learning [52]. More details about these measures can be found in Chapter 2.

5.3.3 Identifying Initial Periods and Periods of Sudden Drops in Predictive Performance

We define the initial phase of the open source projects as the period of time ranging from the first time step until the time step where the G-Mean of the WP approach reaches the value of its average G-Mean across time steps. It represents the time it takes for the G-Mean of this approach to reach its typical values for a given project. Knowledge of how long such period can be and how much it may vary from project to project can be useful to enable: (a) more informed decisions in terms of whether and when to trust the predictions given by a JIT-SDP model, and (b) to check which approaches can learn more quickly so as to present a better G-Mean during this initial stage.

For the proprietary data, the total number of time steps is too small to use the average G-Mean across time steps for this purpose. In particular, had longer periods of time been observed, the G-Mean values would be likely to improve further, given the trends in G-Mean at the end of the period analyzed for these projects (see Fig. 5.2, which will be discussed later in this section). Therefore, instead of using the average G-Mean across all time steps,

we have used the average G-Mean across the last 40 time steps to determine the initial phase. We chose to fix the value of 40 so that enough time steps are being used for the calculation while ensuring that only the very last time steps are used, as they would be the time steps most likely to have increased predictive performance.

Algorithm 5 Sudden Drop Detection Approach

Input: d_thr = Detecta threshold, $gmeans$ = G-mean for each time step t

```
drop_periods = {}
peaks = detecta_onset(d_thr, gmeans) // time steps with peaks
for time step t = 1 to gmeans.length() do
    if t ∈ peaks then
        latest_peak = t
    end if
    window_avg = average(gmeans[latest_peak : t])
    drop_thr = window_avg − (0.20 ∗ window_avg)
    if gmeans[t] < drop_thr then
        peak_left = latest_peak
        peak_right = Get next peak after timestamp t from peaks
        drop_periods.add([peak.left, peak.right])
    end if
end for
Return drop_periods
```

A systematic approach is applied to identify the periods of sudden drop in predictive performance. Its pseudocode is shown in the Algorithm 5. First, the peaks in the $G\text{-}Mean^{(t)}$ (Section 5.3.2) obtained by the WP approach across time steps are identified for each of the datasets using a python module called 'Detecta' [212]. This module provides a function to detect or identify the change or occurrence of a particular event in the data, based on a threshold. Then, we start analysing the $G\text{-}Mean^{(t)}$ for each time step t. In particular, we compare $G\text{-}Mean^{(t)}$ against the simple average G-Mean obtained based on the WP approach's predictions over all software changes between the immediate previous peak identified by Detecta and the current time step t. If $G\text{-}Mean^{(t)}$ drops below 20% of the simple average G-Mean, then the time period from the immediate previous peak to the next peak following this time step is considered as a period of sudden drop in G-Mean. The threshold 20% was

chosen after investigating several different threshold values. Smaller thresholds than 20% caused too many small drops in performance to be considered as sudden drops, whereas larger thresholds resulted in most sudden drops in performance to be missed. Examples of sudden drop periods identified by this systematic approach can be seen in the grey shaded areas of Figures 5.1 and 5.2. As we can see, the approach is able to identify periods of time where the G-Mean of the WP approach presents severe drops.

Considering only the sudden drop periods without knowing what happens in the periods of stable performance would give us incomplete insights. In particular, an approach that is able to prevent sudden drops in performance, but performs much worse during stable periods would not be so useful. Therefore, we also investigate periods of stable performance. Any period after the initial period excluding the sudden drop periods is considered as a stable period.

5.3.4 Statistical Tests and Effect Size

The predictive performances obtained during each period of the projects (initial phase, periods of sudden drop and stable periods) will be compared across data sets using the Scott-Knott procedure [56], which ranks the models and separates them into clusters. This test is used to select the best subgroup among different models. Non-parametric bootstrap sampling is used to make the test non-parametric, as recommended by Menzies et al. [57]. As explained by Demsar [58], non-parametric tests are adequate for comparison across data sets.

In addition, the Scott-Knott test adopted in this chapter uses A12 effect size [59] to rule out any small differences in performance.

We also report the A12 effect sizes against the dummy approach for each dataset individually to support the analysis. Symbols [*], [s], [m] and [b] represent insignificant,

small, medium and large A12 effect size. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the dummy approach.

5.3.5 Hyper-Parameter Tuning

The hyper-parameters for the Filtering approach were chosen by performing grid search on the initial portion (1000 commits and one-third commits respectively for open source and proprietary data) of the datasets using the following set of values: $windowSize$ ={500,600,700,1000}, number of top short distances tobe used K={50,100,200}, distance threshold for similarity $maxDist$={0.6,0.7,0.8} and maximum size of the queue of dissimilar CP instances $cpqSize$ = {500, 1000}. Two ORB hyper-parameters are also tuned by grid search: moving average window size={50, 100, 200} and n={3, 5, 7} for open source and proprietary data. The rest of the ORB hyper-parameters were kept to the same values as in [29] for open source datasets, as they have already been tuned for these datasets. However, for the proprietary datasets, OOB hyper-parameters were further tuned using the following set of values: ensemble={5,10,20,30,40} and theta={0.9,0.99,0.999}. We will also provide an analysis of sensitivity of the online CP approach that obtained the most promising results (Filtering) to its hyper-parameters in Section 5.5.

The waiting period was 90 days for the open source datasets as in [29] and 30 days for the proprietary datasets, due to their lower defect discovery delay (see Table 4.2). When open source data were used in combination with proprietary data, waiting period of 30 days was used for both types of datasets. Cabral et al. [51]'s study on WP JIT-SDP recommended to use waiting periods that are close to the expected median time to discover defects in the project. However, due to concept drift, it is advisable to use shorter waiting periods when some of the CP data have shorter defect discovery delay. This is because shorter waiting periods enable speeding up recovery from concept drift. The choice of waiting period may

cause some defect-inducing commits to be initially labelled as clean and then relabelled to defect-inducing once a defect is found to be associated to them. However, once an example is relabelled, it is immediately used for training again. When that happens, the resampling rates adopted by OOB and ORB are likely to help emphasising the new defect-inducing label, given that the defect-inducing class is usually a minority class. Thirty executions of each approach with each of the base classifiers have been performed on each dataset.

5.4 Experimental Results and Discussion

Tables 5.1 to 5.8, Fig. 5.3 and Fig. 5.4 show the results obtained by the CP, WP and OP approaches for the different periods of time investigated in this study. Some of the results obtained by the ORB-based approaches are shown in the appendix C. Section 5.4.1 presents a preliminary comparison of the approaches against the dummy classifier. Sections 5.4.2, 5.4.3, 5.4.4 and 5.4.5 will focus on the comparisons between the CP and WP approaches, whereas Section 5.4.6 will also consider the OP approaches.

5.4.1 Preliminary Analysis

We compared all approaches against a dummy classifier as a preliminary analysis. A JIT-SDP classifier that outperforms a dummy classifier in terms of overall predictive performance is a classifier that was able to learn relevant JIT-SDP knowledge. Filtering and AIO were always able to outperform a dummy classifier in terms of overall G-Mean across times steps, with large effect size. These results are shown in Table 5.8. The Scott-Knott.BA12 ranking of these approaches across datasets was much better than that of the dummy classifier. The WP and Ensemble approaches also achieved better overall G-Mean ranking than the dummy classifier, even though they were unable to consistently outperform the dummy classifier in

all individual datasets. The OP approaches obtained worse overall G-Mean ranking than the dummy classifier. This means that it was unable to learn enough JIT-SDP knowledge to be of practical use for these projects.

Table C.7 of the appendix C shows the overall G-Mean across time steps obtained by all approaches for the proprietary projects. The dummy classifier outperformed all approaches, achieving the top ranking in terms of overall G-Mean across datasets. It is known that a small number of training examples from the underlying problem distribution negatively affects machine learning models [213]. The small number of WP training examples (Table 4.2) is likely the reason for such poor results on the proprietary projects. Nevertheless, it is still useful to further compare the existing approaches on the proprietary datasets in the subsequent sections of this chapter. Approaches that perform better than other approaches on these datasets are likely to learn faster (with less training examples) than approaches that perform worse.

An ideal JIT-SDP approach would also outperform a dummy classifier at different individual stages of the learning process. However, this is unlikely to be achievable in practice. In particular, during the first time steps of the learning, there would always be a very small number of WP examples to learn from. Therefore, so long as WP data is relevant to achieve good predictive performance, there would always be at least some very initial time steps where a given approach performs worse than a dummy classifier. During latter periods of time, concept drifts may sometimes also have a similar effect to that of a small number of training examples. This is because concept drifts may uncover areas of the input space that have not yet been learned. To gain a better understanding of how well the approaches perform against the dummy classifier, we have compared each approach against the dummy classifier separately during the initial, performance drop and stable periods.

Table 5.1 (and Table C.1 of the appendix C) shows the results obtained during initial period for the opensource projects. From the Scott-Knott.BA12 test results, we can see that AIO-OOB and Filter-OOB were better ranked than the dummy classifier in terms of G-Mean across open source projects for both OOB and ORB. When analysing individual open source projects, AIO-OOB and Filter-OOB outperformed the dummy classifier with large effect size in 6 and 7 out of 10 projects, respectively. Similar results were achieved when adopting ORB. Absolute improvements in G-Mean were up to 25.09% and 24.98%, for Neutron using AIO-ORB and Filter-ORB, respectively. All other approaches were unable to rank better than the dummy classifier during the initial period.

From Table 5.2 (and Table C.2 in the appendix C) we can see that all approaches ranked worse than the dummy classifier on the initial period of proprietary projects. This is because the number of commits in the initial period for proprietary data is very small. In reality, the actual initial period might be much longer if we had more commits from these projects. It is possible that the whole period of time analysed for these projects is actually part of the initial period. In most cases, the whole period of the proprietary projects is smaller than the initial period of the open source projects. In particular, the median total number of software changes in the proprietary projects is 555, whereas the median initial period of the open source projects is 17844.5. This explains the poor performance of all approaches on the proprietary projects.

Table 5.3 (and Table C.3 of the appendix C) shows the results obtained during performance drop periods for open source projects. AIO and Filtering obtained better Scott-Knott.BA12 ranking than the dummy classifier, whereas the other approaches did not. When looking at individual datasets, AIO and Filtering were able to outperform the dummy classifier with large effect size in the performance drop periods of all projects except for JGroups when using OOB and Npm when using ORB. This is an encouraging result for AIO and Filtering, as the performance drop periods are periods where we expect the approaches to

struggle to perform well. The absolute improvements were up to 17.39% for BroadleafCommerce and up to 17.48% for Nova when using AIO-OOB and AIO-ORB, respectively in terms of G-Mean.

For the proprietary projects (Table 5.4 of the chapter and Table C.4 of the appendix C), all approaches obtained worse Scott-Knott.BA12 ranking than the dummy classifier during the performance drop period.

Table 5.6 (and Table C.5 of the appendix C) shows the results obtained during stable periods for open source projects. According to the Scott-Knott.BA12 tests, the AIO, Filtering, Ensemble and WP approaches performed better than the dummy classifier during the stable periods. There were improvements in G-Mean with large effect sizes for all datasets, except for the Ensemble approach on JGroups. In particular, Filtering-OOB led to absolute improvements in G-Mean of up to 32.22% (for Neutron) and AIO-ORB approach led to absolute improvements in G-Mean of up to 31.87% (for Neutron) compared to the dummy approach (Tables 5.6 and Table C.5 of the appendix C). The OP approaches were unable to achieve better ranking than the dummy classifier across datasets.

For proprietary projects (Table 5.7 of the chapter and Table C.6 of the appendix C), according to Scott-Knott.BA12, AIO-OOB-combined and Filter-ORB-combined were able to outperform the dummy classifier across datasets. Filter-OOB-combined achieved the largest absolute improvement in G-Mean of 9.42% for C2 among CP approaches. AIO-ORB-proprietary achieved absolute G-Mean improvements of up to 12.7% (for C3). These results are encouraging. They show that, at least during periods of stability, some AIO approach is able to outperform the dummy classifier, despite the overall G-Mean of all approaches having been worse than that of the dummy classifier. This strengthens our believe that, had they been given more WP training data for training, the proposed approaches would also perform better than the dummy classifier in terms of overall G-Mean.

The preliminary analysis shows that the proposed CP approaches are able to learn relevant JIT-SDP knowledge. They outperformed the dummy classifier in terms of overall G-Mean for the open source projects, despite not always outperforming the dummy classifier on every individual initial and performance drop periods. Other approaches struggled more to outperform the dummy classifier. All approaches struggled to perform well on the proprietary projects, due to the limited number of WP training examples.

5.4.2 RQ1.1: Initial Phase of the Project

Tables 5.1 and 5.2 show the number of time steps of the initial phase, the average G-Mean, the effect size A12 against the dummy approach, and the Scott-Knott.BA12 results during this period for the OOB approaches on the open source and proprietary data, respectively. Results for ORB can be found in the appendix C. Note that the time steps corresponding to the initial period (and also the stable and sudden drop periods) are different for OOB and ORB, such that we cannot compare these approaches against each other in these separate periods. A comparison between these two approaches across all time steps is shown in the Table 5.8 (for open source datasets) and appendix C (for proprietary datasets).

The initial phase of the open source projects lasted from 461 to 6271 time steps with a median of 1418 when using OOB (Table 5.1). This length of time was also considerable when using ORB, lasting from 900 to 6271 time steps with a median of 1553. The initial phase of the proprietary projects was typically much smaller than that of the open source projects, lasting from 8 to 581 time steps for OOB (Table 5.2) and from 62 to 671 time steps for ORB. This is probably related to the fact that the duration of these projects as a whole was in general much shorter than that of the open source projects.

For open source projects, AIO and Filtering ranked the best for both OOB and

Table 5.1: Number of initial time steps of the initial phase, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare OOB-based approaches on this initial phase for open source data

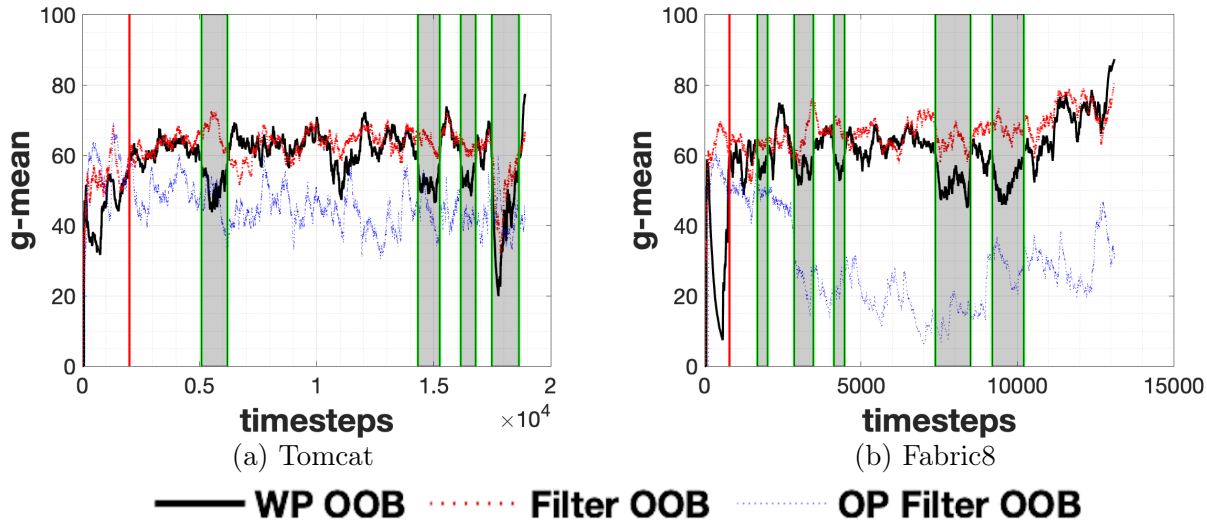
Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB	Filter-OOB	Ensemble-OOB	OP-AIO-OOB	OP-Filter-OOB
Tomcat	2006	50	43.62(4.84)[-b]	51.95(1.25)[b]	52.5(0.71)[b]	33.67(0.63)[-b]	53.12(3.28)[b]	52.86(1.15)[b]
JGroups	1268	50	38.6(0.83)[-b]	38.29(0.84)[-b]	39.01(0.75)[-b]	16.64(0.36)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	461	50	27.3(0.12)[-b]	24.28(0.25)[-b]	38.69(0.49)[-b]	19.48(0.22)[-b]	11.15(6.13)[-b]	29.16(6.34)[-b]
Camel	3112	50	46.82(1.71)[-b]	57.75(0.74)[b]	57.07(0.82)[b]	39.42(0.52)[-b]	47.0(1.06)[-b]	46.94(0.96)[-b]
Brackets	1569	50	21.37(0.03)[-b]	64.89(1.32)[b]	66.17(0.74)[b]	46.83(0.91)[-b]	50.46(1.05)[s]	59.45(1.33)[b]
Nova	6271	50	55.42(0.41)[b]	63.15(0.68)[b]	64.39(0.34)[b]	51.66(0.29)[b]	62.55(1.16)[b]	62.88(4.43)[b]
Fabric8	795	50	27.01(0.3)[-b]	51.51(1.42)[b]	58.63(1.57)[b]	40.3(0.69)[-b]	56.57(3.12)[b]	45.52(1.08)[-b]
Neutron	917	50	44.84(5.75)[-b]	73.55(0.99)[b]	67.29(2.45)[b]	55.07(1.1)[b]	67.03(0.41)[b]	73.15(0.7)[b]
Npm	2536	50	26.92(1.1)[-b]	48.37(1.34)[-b]	45.75(1.94)[-b]	42.02(0.34)[-b]	42.89(6.01)[-b]	15.7(1.15)[-b]
BroadleafCommerce	677	50	26.37(0.26)[-b]	50.32(1.55)[*]	53.16(1.99)[b]	37.26(1.58)[-b]	40.29(2.8)[-b]	30.63(3.6)[-b]
Ranking		2	4	**1**	**1**	4	3	3

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

Table 5.2: Number of initial time steps of the initial phase, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare OOB-based approaches on this initial phase for proprietary data

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB combined	AIO-OOB proprietary	Filter-OOB combined	OP-AIO-OOB combined	OP-AIO-OOB proprietary	OP-Filter-OOB combined
C1	284	50	15.76(4.12)[-b]	35.07(3.74)[-b]	15.97(3.9)[-b]	42.5(2.11)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	581	50	24.62(0.37)[-b]	31.03(0.43)[-b]	32.71(5.38)[-b]	46.82(0.76)[-b]	43.84(1.9)[-b]	5.62(6.26)[-b]	45.33(2.97)[-b]
C3	82	50	8.48(0.0)[-b]	23.36(0.0)[-b]	35.9(0.0)[-b]	8.62(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	24	50	7.59(0.13)[-b]	18.93(2.73)[-b]	18.63(3.19)[-b]	0.0(0.0)[-b]	20.1(0.14)[-b]	14.47(0.39)[-b]	0.0(0.0)[-b]
C5	8	50	11.62(0.0)[-b]	1.69(2.63)[-b]	31.66(4.57)[-b]	10.14(3.89)[-b]	5.16(6.01)[-b]	1.99(4.52)[-b]	39.83(0.0)[-b]
C6	20	50	3.6(0.0)[-b]	1.8(0.0)[-b]	1.8(0.0)[-b]	1.38(0.77)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C7	29	50	1.17(0.0)[-b]	42.76(2.42)[-b]	50.92(3.65)[-s]	0.0(0.0)[-b]	40.61(8.02)[-b]	23.0(10.38)[-b]	0.0(0.0)[-b]
C8	39	50	0.76(0.0)[-b]	5.87(4.91)[-b]	40.16(8.41)[-b]	0.66(0.26)[-b]	16.56(8.28)[-b]	37.57(6.39)[-b]	0.0(0.0)[-b]
C9	62	50	10.44(1.43)[-b]	15.4(6.94)[-b]	25.12(2.67)[-b]	0.55(0.0)[-b]	21.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Ranking		**1**	4	3	2	4	4	4	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

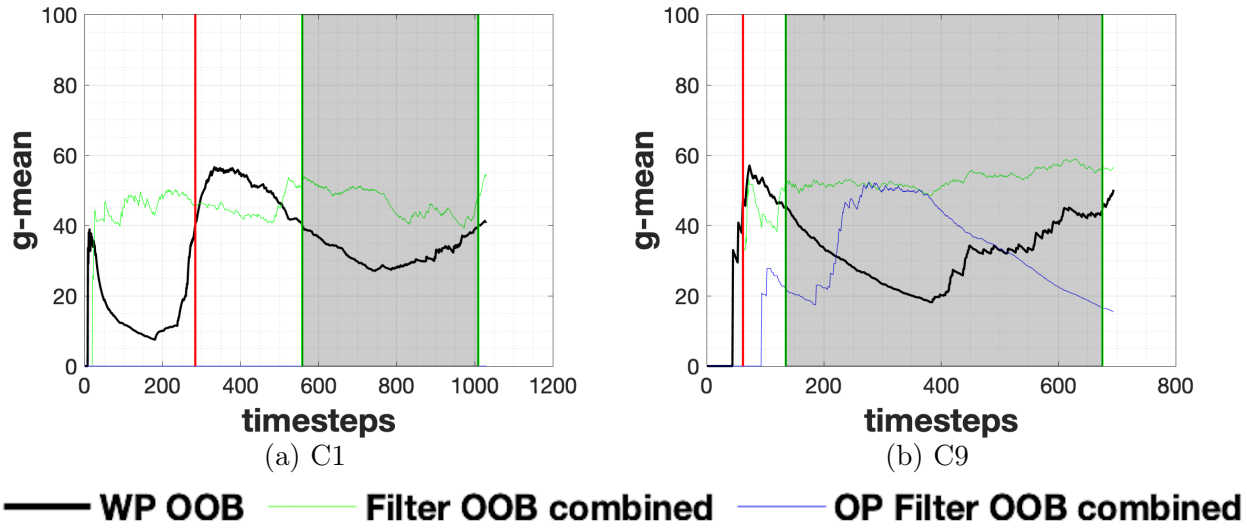


(a) Tomcat (b) Fabric8

Figure 5.1: G-Mean for open source datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods.

ORB according to Scott-Knott.BA12. A12 effect sizes against WP learning for individual datasets were typically large. The average G-Means of the AIO and Filtering approaches were typically higher during the initial phase than those of the WP approach (absolute improvements in G-Mean of up to 44.81% and 53.89%, for Brackets using Filtering-OOB and AIO-ORB respectively). For the Filtering approach, the G-Means were better than those of the WP classifiers for the initial period of all open source datasets. Fig. 5.1 illustrates some sample results (see G-Means up to the red vertical line). Other figures are in the appendix C. AIO also performed better than WP except for AIO-OOB on JGroups and Spring-integration. The Ensemble approach ranked worse than AIO and Filtering, even though it was able to improve over WP when using ORB according to Scott-Knott.BA12.

The fact that AIO and Filtering ranked best across datasets means that these approaches typically outperform the others. However, this does not mean that they will always outperform the others. For instance, WP-OOB performed better than AIO-OOB on JGroups and Spring-integration at the initial phase. In the case of JGroups, there were no CP training examples arriving during the initial period because of the chronological order. So, AIO



(a) C1 (b) C9

Figure 5.2: G-Mean for proprietary datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods.

was trained only with WP data during this period. Hence, for JGroups, the predictive performance obtained by WP and AIO is very similar.

For Spring-Integration, the initial period is small (461 commits for OOB) and has CP data from only 2 projects (Tomcat and JGroups). It is possible that these CP projects are dissimilar to Spring-Integration. A proportion of 82% of the CP commits in the initial phase of Spring-Integration are from project JGroups. There are large differences in the input features of JGroups and Spring-integration. For instance, the averages of the developers experience (EXP) is 4790 and 1039, of the number of defective commits is 1918 and 123, and of the time interval between last and current change (AGE) is 25 and 4.57 in JGroups and Spring-integration, respectively. This indicates that during this initial period, the JIT-SDP model is trained with very dissimilar commits, possibly causing AIO-OOB to perform worse than WP. Filter-OOB was able to overcome this issue by filtering out commits that are very dissimilar.

For proprietary projects, AIO-proprietary was better ranked than the WP approach

for both OOB and ORB, being followed by AIO-combined according to Scott-Knott.BA12. A possible reason why AIO-proprietary performed better than AIO-combined is the high dissimilarity between the open source and proprietary projects. Together with the very small number of WP training examples during the initial phase, such high dissimilarity can prevent the WP training examples from playing a significant role when training the AIO-combined classifiers. This is because they may be perceived almost as noise by the learning process. When using AIO-proprietary, there is not only smaller amount of CP data (enabling the WP data to play a more significant role), but also those CP data are more similar to WP data as they are all projects of similar nature. This may enable the AIO-proprietary approach to learn the underlying defect generating process better.

AIO-proprietary's G-Means were better than those of WP classifiers in the initial phase (absolute improvements in G-Mean of up to 49.75%, for C7) for most of the datasets except for datasets C6 (OOB) and C1 (ORB). Filtering with OOB's G-Means were lower than WP's for several datasets especially where the initial periods were very small (ranging from 8 to 62). Sample results when it performed better and worse are shown in Fig. 5.2. For Filtering with ORB, the G-Means were better than those of WP classifiers in the initial phase for all of the proprietary datasets.

It is also interesting to note that AIO performed better than Filtering for the proprietary projects. Specifically, when comparing AIO-OOB combined against Filter-OOB combined, we can see that filtering itself was detrimental. It is likely that filtering resulted in the removal of relevant training examples, which was particularly a problem given the small length of the initial phase of these datasets.

The G-Means obtained for proprietary datasets were in general low. As AIO-proprietary performed significantly better than WP and AIO-combined (see Scott-Knott.BA12 results in Table 5.2), the CP proprietary data were usually important for improving predictive per-

formance in the initial period. Therefore, the small number of CP proprietary software changes compared to the open source projects is likely to be part of the reason for the poor G-Means obtained for the proprietary projects, even though other factors such as the specific distribution of the WP data may also affect the G-Mean on individual projects.

The G-Means of all approaches during the initial periods were relatively poor compared to the stable periods. This is expected, because the initial period is by definition a period of time when JIT-SDP classifiers have not yet reached their typical average G-Mean, i.e., it corresponds to the period of time when the approaches perform below average. Reaching the average G-Mean value requires training on more WP data. Despite that, our experiments have shown that AIO-OOB outperformed other existing approaches during such periods, having obtained better initial period G-Mean ranking.

RQ1.1: The initial phase of the projects had considerably large length, lasting from 461 to 6271 and from 8 to 582 time steps for the open source and proprietary projects, respectively. CP data was helpful when there was no or little WP training data available, in particular when using AIO approaches for both open source projects and proprietary projects. Filtering was also top performing for the open source projects. Absolute improvements in average G-Mean were up to 53.89%, avoiding extremely low G-Means.

5.4.3 RQ1.2: Periods with Sudden Drops in WP Classifier's Predictive Performance

In several datasets, after the initial phase, the WP approach suffered periods of large drops in G-Means. Figs. 5.1 and 5.2 show some examples of that highlighted in grey background. Tables 5.3 and 5.4 show further information about the sudden drop periods suffered by OOB for open source and proprietary projects, respectively. Other tables and results for ORB are

Table 5.3: Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB for open source data during the drop periods

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB	Filter-OOB	Ensemble-OOB	OP-AIO-OOB	OP-Filter-OOB
Tomcat	3855	50	48.52(1.17)[-b]	58.86(0.72)[b]	60.22(1.0)[b]	52.26(0.32)[b]	36.42(6.24)[-b]	43.81(1.44)[-b]
JGroups	4353	50	43.95(1.44)[-b]	48.01(1.05)[-b]	48.08(0.89)[-b]	31.41(0.5)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	3973	50	32.98(0.7)[-b]	59.01(0.53)[b]	60.15(0.97)[b]	45.68(0.43)[-b]	52.05(5.34)[s]	34.89(4.74)[-b]
Camel	6009	50	53.15(1.68)[b]	56.17(0.98)[b]	57.0(0.92)[b]	52.4(0.41)[b]	50.09(1.54)[*]	50.68(0.69)[b]
Nova	4427	50	66.99(0.06)[b]	66.99(0.18)[b]	66.85(0.26)[b]	57.19(0.66)[b]	65.4(3.73)[b]	67.1(6.49)[b]
Fabric8	3443	50	53.79(2.18)[b]	56.34(0.76)[b]	65.57(1.04)[b]	53.98(0.4)[b]	23.23(6.54)[-b]	24.48(6.65)[-b]
Npm	1202	50	38.64(1.95)[-b]	62.97(1.47)[b]	65.37(1.25)[b]	51.13(0.62)[b]	44.42(7.52)[-s]	17.36(3.61)[-b]
BroadleafCommerce	3579	50	50.18(0.55)[s]	67.39(1.05)[b]	66.05(0.66)[b]	54.38(0.25)[b]	52.72(1.17)[b]	56.08(2.43)[b]
Ranking		3	3	2	**1**	3	4	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

Table 5.4: Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB for proprietary data during the drop periods

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB combined	AIO-OOB proprietary	Filter-OOB combined	OP-AIO-OOB combined	OP-AIO-OOB proprietary	OP-Filter-OOB combined
C1	452	50	32.27(4.27)[-b]	41.37(1.45)[-b]	43.4(4.45)[-b]	46.66(1.08)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C3	210	50	42.79(1.72)[-b]	47.75(1.31)[-b]	39.1(1.42)[-b]	52.17(1.3)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	249	50	33.0(0.95)[-b]	51.47(1.74)[b]	60.05(2.89)[b]	56.78(1.76)[b]	57.74(1.11)[b]	47.1(1.74)[-b]	51.35(0.0)[b]
C6	466	50	21.34(1.35)[-b]	43.64(1.49)[-b]	45.89(5.02)[-b]	44.32(1.98)[-b]	43.53(0.41)[-b]	37.47(0.18)[-b]	45.1(0.47)[-b]
C7	255	50	25.74(1.19)[-b]	41.5(5.94)[-b]	32.34(1.51)[-b]	40.34(1.02)[-b]	50.43(2.6)[-*]	36.26(0.52)[-b]	30.73(0.4)[-b]
C8	287	50	7.3(0.89)[-b]	44.65(0.96)[-b]	34.6(5.43)[-b]	40.14(1.82)[-b]	44.3(1.81)[-b]	45.49(9.63)[*]	25.57(8.7)[-b]
C9	541	50	31.75(1.77)[-b]	51.34(1.76)[b]	50.08(2.77)[s]	53.25(1.58)[b]	40.3(5.22)[-b]	34.92(3.87)[-b]	34.36(0.55)[-b]
Ranking		**1**	5	3	4	2	4	5	5

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

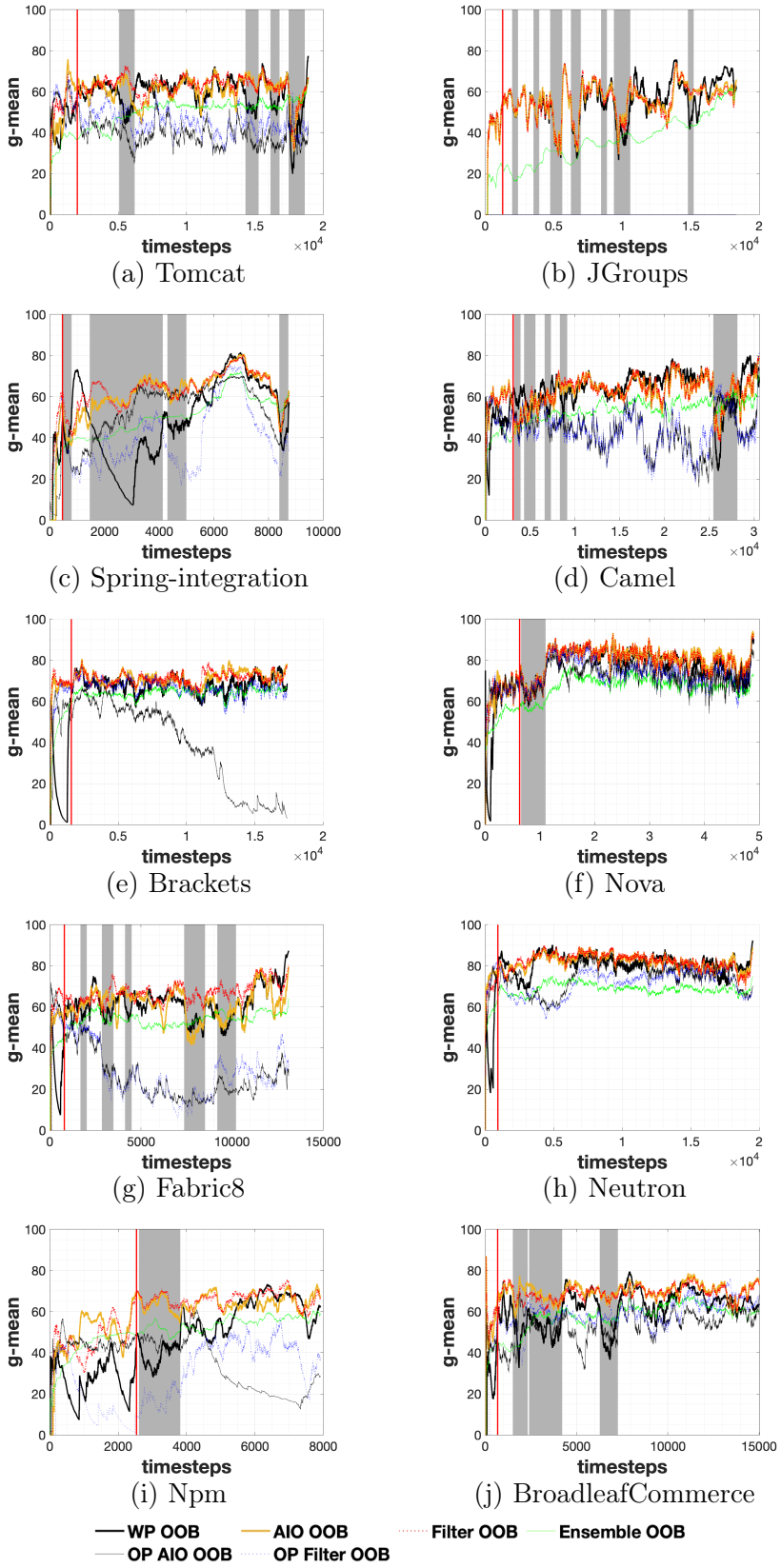


Figure 5.3: G-Mean for all datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods.

in the appendix C.

The periods of sudden drop have considerable length for the open source projects, covering in total from 1202 to 6009 time steps when using OOB (Table 5.3) and from 936 to 4408 when using ORB. Due to the shorter duration of the proprietary projects, the total length of the sudden drop periods in those projects was smaller, ranging from 210 to 541 time steps for OOB (Table. 5.4) and 23 to 238 time steps for ORB.

For open source projects, when using OOB, Filtering was the best ranked approach, followed by AIO, according to Scott-Knott.BA12. When using ORB, AIO was top ranked together with Filtering. Drop periods are likely to occur as result of concept drifts, meaning that most recent WP data are dissimilar to past WP data. Filtering out dissimilar examples may help to focus more on the current concept being learned, enabling the model to adapt quicker than with AIO. However, ORB is more robust to concept drift than OOB, due to its automatically boosted resampling rate [51]. So it may not always require the extra focus on the current concept. The Ensemble approach was unable to consistently perform better than the WP approach across open source datasets, obtaining the same Scott-Knott.BA12 ranking as the WP approach.

AIO and Filtering outperformed the WP approach in all open source datasets except Nova during such periods (absolute improvements of up to 27.17% and up to 29.01% when using Filtering-OOB and Filtering-ORB, respectively, for Spring-integration). Fig. 5.1 shows some examples of that, highlighted in grey background. As we can see, Filtering managed to reduce or even eliminate the drops in G-Mean suffered by the WP approach.

For proprietary projects, Filtering was also better than the WP, AIO and OP approaches, based on Scott-Knott.BA12. Therefore, even when using ORB, Filtering was important during the performance drop periods of the proprietary projects. The effect size of the Filtering and AIO approaches against WP were frequently large. The absolute im-

provements in average G-Mean over the WP approach were up to 37.35% for OOB when using AIO-combined and up to 30.34% for ORB when using Filtering-ORB (for C8). Fig. 5.2 shows some examples of results achieved by Filtering, where it is clear that this approach outperformed the WP approach. The AIO approach also outperformed WP approach in most of the datasets. AIO-combined achieved better results than AIO-proprietary when using OOB, but similar when using ORB. This shows that open source data can sometimes help to improve predictive performance on proprietary projects during sudden drop periods.

Given that existing literature indicates that JIT-SDP suffers from concept drift [20], it is likely that WP classifiers suffer drops in performance due to changes in the characteristics of WP training data over time. However, in CP learning, training data comes from different projects. Some of the CP training data may have similar distribution as the target project currently has, helping to reduce the negative effect of differences in the distribution over time. This is a potential reason for CP data (including mixed proprietary and open source data) to be helpful in case of sudden performance drops.

Interestingly, the Filtering approach typically achieved better results than the AIO approach during the sudden drop periods. Improvements in G-Means achieved by Filtering-OOB compared to AIO-OOB were up to 9.21% (for Fabric8) for open source and up to 13.16% (for C3) for the proprietary data. This suggests that even though CP data may prevent performance drops resulting from changes in characteristics of the data, it might introduce other performance drops due to the use of too dissimilar CP data. This indicates that filtering dissimilar instances can be particularly helpful to reduce or prevent sudden drops of performance, which has not been captured by the offline JIT-SDP literature when they concluded that AIO and Filtering approaches perform similarly [12]. Unlike in the initial period, CP approaches benefited from a combination of both open source and proprietary data during drop periods, specially for OOB, where combined data outperformed proprietary-only approaches both through Filtering and AIO. This suggests that the diversity of data

from multiple sources may help with dealing with concept drift in JIT-SDP.

Brackets, Neutron, C2 and C5 did not present drops in predictive performance. Together with the fact that no large sudden changes in the values of the input features were found for these projects over time, we believe that these projects are more stable than the others, being less negatively affected by concept drift.

RQ1.2 Answer: The periods of sudden drop in performance were of considerable length, covering in total from 936 to 6009 time steps for open source, and from 23 to 541 time steps for proprietary data. CP approaches frequently helped to reduce or even prevent sudden drops in G-Mean compared to WP approaches, with absolute improvements of up to 37.35% during such periods. Filtering was the top ranked approach across datasets.

Table 5.5: Number of time steps of the stable periods, and average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare OOB-based approaches on stable periods for open source data

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB	Filter-OOB	Ensemble-OOB	OP-AIO-OOB	OP-Filter-OOB
Tomcat	13046	50	62.69(0.81)[b]	61.36(0.5)[b]	63.36(0.64)[b]	50.57(0.22)[b]	39.68(6.07)[-b]	45.23(1.56)[-b]
JGroups	12704	50	60.08(0.66)[b]	58.74(0.72)[b]	58.51(0.54)[b]	38.8(0.3)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	4316	50	64.86(1.01)[b]	65.99(0.35)[b]	66.96(0.77)[b]	56.02(0.33)[b]	55.88(8.08)[b]	45.51(8.94)[-*]
Camel	21454	50	67.33(0.56)[b]	64.48(0.67)[b]	65.05(0.62)[b]	54.61(0.35)[b]	41.45(1.55)[-b]	41.52(1.26)[-b]
Brackets	15795	50	68.27(0.12)[b]	71.44(0.48)[b]	71.38(0.42)[b]	64.25(0.54)[b]	38.19(13.68)[-b]	66.67(0.36)[b]
Nova	38291	50	79.84(0.26)[b]	82.03(0.21)[b]	81.54(0.2)[b]	68.72(0.79)[b]	73.99(16.63)[b]	75.19(0.5)[b]
Fabric8	8868	50	65.0(0.77)[b]	63.53(0.62)[b]	67.64(0.64)[b]	54.15(0.4)[b]	26.11(4.94)[-b]	28.23(4.47)[-b]
Neutron	18605	50	81.13(0.27)[b]	81.87(0.23)[b]	82.22(0.25)[b]	68.93(1.5)[b]	74.53(1.79)[b]	71.53(3.95)[b]
Npm	4182	50	59.52(1.71)[b]	64.96(0.63)[b]	66.49(0.99)[b]	55.3(0.46)[b]	29.24(2.24)[-b]	40.11(1.15)[-b]
BroadleafCommerce	10754	50	65.96(1.12)[b]	70.73(0.62)[b]	70.13(0.55)[b]	59.55(0.2)[b]	54.26(0.8)[b]	60.42(1.07)[b]
Ranking		4	2	**1**	**1**	3	5	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

5.4.4 RQ1.3: Effects of CP Data on the Predictive Performance During Stable Periods of The Projects

Tables 5.6 and 5.7 show information about the stable periods for the open source and proprietary projects, respectively, when using OOB. Tables for ORB are in the appendix C.

The smallest and largest total length of stable periods for open source projects are 4182 and 38291 for OOB (Table 5.6), and 5490 and 38310 for ORB. For proprietary projects, they are 20 and 472 for OOB (Table 5.7) and 19 and 486 for ORB. Due to the long duration of the open source projects, it is expected that their stable periods are longer than the periods of sudden drop in performance, even though the periods of sudden drop are still considerable periods worth addressing. For the proprietary data, the stable periods were short compared to the sudden drop periods. Possibly, due to the relatively small number of software changes in the proprietary projects, the whole period of time covered by the proprietary projects is more unstable.

Given the length of time covered by stable periods, it is important that the better results obtained by CP approaches during the initial and sudden drop periods are not counteracted by poor performance during stable periods.

Except for the Ensemble approach, Scott-Knott.BA12 shows that all CP approaches performed at least as well as the WP approach across datasets during the stable periods. This was the case both when adopting OOB and ORB, for both open source and proprietary data. This is very encouraging, as it shows that the G-Mean improvements achieved by Filtering and AIO during the initial and sudden drop periods are not cancelled out by worse G-Mean during stable periods.

For the open source data, according to Scott-Knott.BA12, both AIO and Filtering approaches were ranked the best for OOB and ORB during stable periods in terms of G-

Mean. Moreover, except for Ensemble, the CP approaches were frequently able to outperform the WP approach in terms of G-Mean during the stable period, though by a smaller amount than they did for the initial and sudden drop periods. The Filtering approach performed up to 6.97% (for Npm) and 8.39% (for Spring-integration) better compared to the WP approach for OOB (Table 5.6) and ORB, respectively.

Table 5.6: Number of time steps of the stable periods, and average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare OOB-based approaches on stable periods for open source data

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB	Filter-OOB	Ensemble-OOB	OP-AIO-OOB	OP-Filter-OOB
Tomcat	13046	50	62.69(0.81)[b]	61.36(0.5)[b]	63.36(0.64)[b]	50.57(0.22)[b]	39.68(6.07)[-b]	45.23(1.56)[-b]
JGroups	12704	50	60.08(0.66)[b]	58.74(0.72)[b]	58.51(0.54)[b]	38.8(0.3)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	4316	50	64.86(1.01)[b]	65.99(0.35)[b]	66.96(0.77)[b]	56.02(0.33)[b]	55.88(8.08)[b]	45.51(8.94)[-*]
Camel	21454	50	67.33(0.56)[b]	64.48(0.67)[b]	65.05(0.62)[b]	54.61(0.35)[b]	41.45(1.55)[-b]	41.52(1.26)[-b]
Brackets	15795	50	68.27(0.12)[b]	71.44(0.48)[b]	71.38(0.42)[b]	64.25(0.54)[b]	38.19(13.68)[-b]	66.67(0.36)[b]
Nova	38291	50	79.84(0.26)[b]	82.03(0.21)[b]	81.54(0.2)[b]	68.72(0.79)[b]	73.99(16.63)[b]	75.19(0.5)[b]
Fabric8	8868	50	65.0(0.77)[b]	63.53(0.62)[b]	67.64(0.64)[b]	54.15(0.4)[b]	26.11(4.94)[-b]	28.23(4.47)[-b]
Neutron	18605	50	81.13(0.27)[b]	81.87(0.23)[b]	82.22(0.25)[b]	68.93(1.5)[b]	74.53(1.79)[b]	71.53(3.95)[b]
Npm	4182	50	59.52(1.71)[b]	64.96(0.63)[b]	66.49(0.99)[b]	55.3(0.46)[b]	29.24(2.24)[-b]	40.11(1.15)[-b]
BroadleafCommerce	10754	50	65.96(1.12)[b]	70.73(0.62)[b]	70.13(0.55)[b]	59.55(0.2)[b]	54.26(0.8)[b]	60.42(1.07)[b]
Ranking		4	2	**1**	**1**	3	5	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

Table 5.7: Number of time steps of the stable periods, and average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare OOB-based approaches on stable periods for proprietary data

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB combined	AIO-OOB proprietary	Filter-OOB combined	OP-AIO-OOB combined	OP-AIO-OOB proprietary	OP-Filter-OOB combined
C1	294	50	49.55(1.47)[-*]	43.65(2.33)[-b]	43.4(6.94)[-b]	45.75(2.02)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	20	50	64.59(0.23)[b]	52.86(1.75)[b]	55.58(2.72)[b]	59.42(0.62)[b]	36.49(0.93)[-b]	2.8(3.22)[-b]	37.36(1.46)[-b]
C3	125	50	54.54(1.05)[b]	54.01(0.43)[b]	33.69(2.6)[-b]	57.0(0.61)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	68	50	45.72(1.21)[-b]	53.4(1.23)[b]	56.98(4.85)[b]	27.16(8.92)[-b]	46.76(0.61)[-b]	44.62(2.6)[-b]	21.15(0.0)[-b]
C5	315	50	23.2(0.43)[-b]	46.55(1.35)[-b]	42.6(0.99)[-b]	49.93(1.71)[-s]	44.14(5.2)[-b]	49.53(2.92)[-s]	41.15(4.14)[-b]
C6	69	50	41.71(0.64)[-b]	43.05(0.65)[-b]	49.84(2.73)[-*]	41.83(3.36)[-b]	24.67(0.34)[-b]	14.76(0.07)[-b]	21.1(0.26)[-b]
C7	262	50	45.55(1.0)[-b]	52.02(5.6)[s]	52.62(2.11)[b]	52.35(2.27)[b]	57.67(1.5)[b]	52.6(1.18)[b]	40.38(0.7)[-b]
C8	472	50	26.13(0.22)[-b]	55.16(0.91)[b]	47.01(3.71)[-b]	40.38(1.65)[-b]	45.29(1.94)[-b]	48.99(5.2)[s]	41.14(2.6)[-b]
C9	91	50	49.63(1.59)[-b]	56.46(2.26)[b]	59.12(6.12)[b]	46.6(2.06)[-b]	29.03(2.98)[-b]	8.62(1.08)[-b]	14.33(2.11)[-b]
Ranking		2	3	**1**	2	3	4	5	5

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

For proprietary data, according to Scott-Knott.BA12, AIO-OOB-combined was ranked the best for OOB (Table 5.7). The largest improvement in G-Mean compared to WP ap-

proaches was achieved by AIO-OOB-combined, which had an improvement of 29.03% for C8. When using ORB, Filtering was the top ranked approach. Filtering-ORB achieved G-Mean improvements of up to 15.92% (for C8).

Filtering was helpful for open source projects, but not so much for proprietary projects, where Filtering-OOB performed similar to WP-OOB. Therefore, the more competitive performance achieved by Filtering compared to AIO during the sudden drop periods can possibly be counteracted by less competitive results during stable periods compared to AIO. Nevertheless, it is worth reiterating that both Filtering and AIO performed at least as well as the WP approach during the stable periods across datasets, and that both of them outperformed WP approaches during the periods of sudden drop, based on Scott-Knott-BA12.

Although the Ensemble approach performed similar to the WP approach for initial and drop periods, it was worse for stable periods. A further analysis of the results obtained by each classifier within the ensemble reveals that their individual G-Means were not high, which resulted in the poor G-Mean of the ensemble as a whole. This could be due to lack of enough training data for the individual classifiers, given that each classifier in the ensemble is trained with data from one project. This suggests that the larger amount of varied data to train a model was crucial to improve predictive performance in online JIT-SDP. It also explains why the Ensemble approach worked in previous offline mode but not in online mode, as in offline mode chronology would have been ignored, enabling each base model to be trained with larger amounts of data that would not have been available in practice. In addition, these results suggest that using a weighted ensemble instead of simple votes as in this study would also unlikely work well. This kind of strategy would emphasise the individual models that perform best for the project being predicted, but if all models underperform, emphasising the best models would not help in this case.

RQ1.3 Answer: Except for the Ensemble approach, CP approaches performed at least as well as WP approaches in terms of G-Mean during stable periods. In particular, Filtering with ORB obtained up to around 8.39% improvement in G-Mean for open source data and AIO-OOB-combined achieved up to 29.03% absolute improvement in G-Mean for proprietary data than the WP approach during such periods.

5.4.5 RQ1.4: Overall effect of using CP and WP data together on the predictive performance

According to the Scott-Knott.BA12 test to rank the overall G-Mean of all OOB- and ORB-based approaches across open source datasets (Table 5.8), All-in-One-OOB, All-in-One-ORB, Filtering-OOB and Filtering-ORB ranked best, WP-OOB and WP-ORB ranked second, Ensemble-ORB ranked third, Ensemble-OOB ranked fourth, OP-AIO-ORB ranked fifth, and OP-AIO-OOB, OP-Filter-OOB and OP-Filter-ORB ranked the worst.

Table 5.8 shows that All-in-One-OOB's G-Means were up to 13.43% better (for Npm) and All-in-One-ORB's were up to 16.04% better (for Spring-integration) compared to WP-OOB and WP-ORB, respectively. Filtering performed similarly to All-in-One. Table 5.8 shows that Filtering-OOB's G-Means were up to 13.97% better than WP-OOB's (for spring-integration), and Filtering-ORB's G-Means were up to 16.88% better than WP-ORB's (for spring-integration). Therefore, even though Filtering improved the results over the WP approach, filtering instances dissimilar to the target project does not have a major impact on the performance of the classifier compared to All-in-One. As All-in-One merges all data together to build one classifier, it is possible that classifier performance mainly depends on the amount and potentially variety of training data rather than CP project similarity. From that we can see that merging CP and WP data together to train a classifier (through All-In-

Table 5.8: Overall predictive performance, A12 effect sizes and Scott-Knott.BA12 statistical tests to compare learning approaches for open source data

Dataset	Approach	Recall0	Recall1	G-Mean
Tomcat	Dummy	50	50	50
	WP-OOB	58.25(1.85)[b]	63.59(1.56)[b]	57.78(0.62)[b]
	AIO-OOB	61.07(1.35)[b]	63.76(1.15)[b]	59.85(0.48)[b]
	Filter-OOB	66.99(1.24)[b]	60.05(0.99)[b]	61.57(0.53)[b]
	Ensemble-OOB	63.78(0.71)[b]	43.06(0.74)[-b]	49.12(0.2)[-b]
	OP-AIO-OOB	22.09(4.78)[-b]	85.86(2.49)[b]	40.44(5.71)[-b]
	OP-Filter-OOB	27.61(1.73)[-b]	82.4(1.5)[b]	45.75(1.3)[-b]
Jgroups	Dummy	50	50	50
	WP-OOB	57.55(1.71)[b]	59.01(1.53)[b]	54.76(0.69)[b]
	AIO-OOB	64.54(1.6)[b]	52.85(1.48)[b]	54.77(0.65)[b]
	Filter-OOB	65.94(1.34)[b]	51.59(1.24)[b]	54.68(0.43)[b]
	Ensemble-OOB	84.03(0.4)[b]	18.6(0.37)[-b]	35.51(0.3)[-b]
	OP-AIO-OOB	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-Filter-OOB	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	Dummy	50	50	50
	WP-OOB	61.13(1.67)[b]	56.09(1.33)[b]	48.41(0.71)[-b]
	AIO-OOB	65.32(0.62)[b]	61.57(0.88)[b]	60.62(0.29)[b]
	Filter-OOB	69.82(0.76)[b]	59.96(1.21)[b]	62.38(0.69)[b]
	Ensemble-OOB	73.09(0.58)[b]	37.6(0.64)[-b]	49.4(0.28)[-b]
	OP-AIO-OOB	47.2(9.1)[*]	70.94(7.57)[b]	51.79(6.1)[m]
	OP-Filter-OOB	24.67(5.81)[-b]	89.93(1.22)[b]	39.82(6.35)[-b]
Camel	Dummy	50	50	50
	WP-OOB	56.04(1.25)[b]	74.62(0.97)[b]	62.45(0.58)[b]
	AIO-OOB	53.89(1.15)[b]	75.29(0.73)[b]	62.16(0.62)[b]
	Filter-OOB	55.52(0.82)[b]	74.29(0.72)[b]	62.66(0.56)[b]
	Ensemble-OOB	55.25(0.88)[b]	55.17(0.78)[b]	52.63(0.25)[b]
	OP-AIO-OOB	25.72(2.44)[-b]	82.82(3.08)[b]	43.71(1.43)[-b]
	OP-Filter-OOB	25.06(1.21)[-b]	85.75(0.85)[b]	43.87(1.03)[-b]
Brackets	Dummy	50	50	50
	WP-OOB	49.16(0.28)[-b]	89.6(0.24)[b]	64.03(0.11)[b]
	AIO-OOB	65.47(1.01)[b]	79.68(1.32)[b]	70.85(0.48)[b]
	Filter-OOB	69.15(0.84)[b]	75.44(1.31)[b]	70.91(0.4)[b]
	Ensemble-OOB	64.89(0.77)[b]	62.81(1.13)[b]	62.68(0.48)[b]
	OP-AIO-OOB	31.33(9.76)[-b]	89.04(2.47)[b]	39.3(12.45)[-b]
	OP-Filter-OOB	49.42(1.23)[-m]	90.25(1.67)[b]	66.02(0.36)[b]
Nova	Dummy	50	50	50
	WP-OOB	68.2(0.24)[b]	86.95(0.53)[b]	75.55(0.21)[b]
	AIO-OOB	70.04(0.32)[b]	88.9(0.54)[b]	78.25(0.2)[b]
	Filter-OOB	70.91(0.31)[b]	87.24(0.38)[b]	78.01(0.18)[b]
	Ensemble-OOB	75.88(0.79)[b]	57.23(1.61)[b]	65.5(0.61)[b]
	OP-AIO-OOB	59.07(12.18)[b]	92.65(2.2)[b]	71.75(13.26)[b]
	OP-Filter-OOB	71.16(1.79)[b]	76.16(3.22)[b]	72.89(1.28)[b]
Fabric8	Dummy	50	50	50
	WP-OOB	50.56(2.72)[s]	75.72(2.21)[b]	59.75(1.01)[b]
	AIO-OOB	55.91(1.01)[b]	70.76(1.46)[b]	60.92(0.48)[b]
	Filter-OOB	61.94(1.89)[b]	73.25(1.58)[b]	66.55(0.63)[b]
	Ensemble-OOB	48.84(1.22)[-b]	61.42(1.39)[b]	53.27(0.36)[b]
	OP-AIO-OOB	10.88(2.39)[-b]	96.88(1.22)[b]	27.2(5.03)[-b]
	OP-Filter-OOB	15.42(3.65)[-b]	87.62(4.89)[b]	28.3(4.48)[-b]
Neutron	Dummy	50	50	50
	WP-OOB	70.03(0.7)[b]	91.81(0.61)[b]	79.43(0.36)[b]
	AIO-OOB	73.24(0.42)[b]	91.39(0.5)[b]	81.48(0.24)[b]
	Filter-OOB	74.89(0.45)[b]	89.63(0.54)[b]	81.51(0.26)[b]
	Ensemble-OOB	78.58(1.03)[b]	59.78(3.32)[b]	68.28(1.42)[b]
	OP-AIO-OOB	70.23(0.95)[b]	81.61(2.92)[b]	74.18(1.7)[b]
	OP-Filter-OOB	76.02(3.05)[b]	71.15(8.76)[b]	71.6(3.77)[b]
Npm	Dummy	50	50	50
	WP-OOB	36.99(2.19)[-b]	75.74(1.52)[b]	45.91(0.91)[-b]
	AIO-OOB	54.8(1.61)[b]	68.69(2.03)[b]	59.34(0.61)[b]
	Filter-OOB	55.39(1.61)[b]	68.59(1.48)[b]	59.68(0.54)[b]
	Ensemble-OOB	54.22(1.01)[b]	51.89(1.05)[b]	50.42(0.29)[b]
	OP-AIO-OOB	61.08(6.38)[b]	37.31(6.76)[-b]	35.91(2.78)[-b]
	OP-Filter-OOB	17.06(0.89)[-b]	81.46(0.61)[b]	28.84(1.22)[-b]
BroadleafCommerce	Dummy	50	50	50
	WP-OOB	58.44(1.58)[b]	69.82(2.13)[b]	60.41(0.83)[b]
	AIO-OOB	67.28(1.37)[b]	72.11(1.54)[b]	69.01(0.58)[b]
	Filter-OOB	67.41(0.94)[b]	70.53(0.98)[b]	68.39(0.42)[b]
	Ensemble-OOB	61.96(0.89)[b]	56.48(1.08)[b]	57.31(0.18)[b]
	OP-AIO-OOB	42.99(1.3)[-b]	71.99(2.46)[b]	53.27(0.67)[b]
	OP-Filter-OOB	54.22(4.85)[b]	68.75(5.18)[b]	58.04(1.23)[b]
Ranking	Dummy	4	4	5
	WP-OOB	3	**1**	2
	AIO-OOB	2	**1**	**1**
	Filter-OOB	2	**1**	**1**
	Ensemble-OOB	2	4	4
	OP-AIO-OOB	5	**1**	6
	OP-Filter-OOB	5	**1**	6

Dataset	Approach	Recall0	Recall1	G-Mean
Tomcat	Dummy	50	50	50
	WP-ORB	58.41(1.72)[b]	64.49(1.21)[b]	59.78(0.83)[b]
	AIO-ORB	61.98(1.19)[b]	63.38(0.74)[b]	61.72(0.82)[b]
	Filter-ORB	61.07(0.89)[b]	63.31(0.74)[b]	61.25(0.48)[b]
	Ensemble-ORB	61.75(1.23)[b]	47.67(1.08)[-b]	50.82(0.34)[b]
	OP-AIO-ORB	46.92(5.41)[-b]	54.07(4.56)[b]	42.31(2.63)[-b]
	OP-Filter-ORB	49.5(5.02)[*]	45.05(3.66)[-b]	36.03(3.38)[-b]
Jgroups	Dummy	50	50	50
	WP-ORB	61.18(0.71)[b]	56.93(1.28)[b]	57.2(0.81)[b]
	AIO-ORB	62.77(0.94)[b]	56.67(0.96)[b]	57.92(0.66)[b]
	Filter-ORB	63.05(1.07)[b]	57.21(1.26)[b]	58.29(0.68)[b]
	Ensemble-ORB	83.72(0.33)[b]	20.1(0.42)[-b]	37.35(0.33)[-b]
	OP-AIO-ORB	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-Filter-ORB	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	Dummy	50	50	50
	WP-ORB	71.92(1.0)[b]	45.83(1.31)[-b]	51.73(0.78)[b]
	AIO-ORB	67.51(0.86)[b]	69.48(1.05)[b]	67.77(0.75)[b]
	Filter-ORB	68.57(0.84)[b]	69.93(0.83)[b]	68.61(0.58)[b]
	Ensemble-ORB	74.61(0.48)[b]	39.98(0.67)[-b]	52.4(0.54)[b]
	OP-AIO-ORB	76.29(2.16)[b]	61.45(4.22)[b]	67.5(1.79)[b]
	OP-Filter-ORB	81.69(8.13)[b]	42.21(7.74)[-b]	56.2(3.9)[b]
Camel	Dummy	50	50	50
	WP-ORB	59.59(1.18)[b]	70.11(1.15)[b]	62.92(0.98)[b]
	AIO-ORB	59.12(0.52)[b]	73.1(0.56)[b]	65.01(0.37)[b]
	Filter-ORB	58.58(0.62)[b]	72.7(0.53)[b]	64.48(0.43)[b]
	Ensemble-ORB	61.46(0.88)[b]	52.33(1.13)[b]	54.26(0.51)[b]
	OP-AIO-ORB	60.0(8.59)[b]	47.9(7.2)[-m]	51.05(2.66)[b]
	OP-Filter-ORB	52.14(10.01)[s]	46.55(11.0)[-*]	46.44(3.82)[-b]
Brackets	Dummy	50	50	50
	WP-ORB	61.31(0.74)[b]	76.66(1.43)[b]	63.31(0.49)[b]
	AIO-ORB	66.9(1.03)[b]	74.94(1.71)[b]	69.98(0.87)[b]
	Filter-ORB	66.34(0.92)[b]	75.35(1.33)[b]	69.78(0.74)[b]
	Ensemble-ORB	61.7(1.09)[b]	69.14(1.23)[b]	64.71(0.75)[b]
	OP-AIO-ORB	52.23(1.53)[b]	84.23(2.0)[b]	64.42(0.55)[b]
	OP-Filter-ORB	49.77(2.14)[-s]	85.76(4.24)[b]	63.91(1.56)[b]
Nova	Dummy	50	50	50
	WP-ORB	74.25(1.49)[b]	80.57(2.64)[b]	75.42(0.66)[b]
	AIO-ORB	74.15(0.89)[b]	81.95(1.27)[b]	77.28(0.32)[b]
	Filter-ORB	72.72(0.21)[b]	82.85(0.54)[b]	76.83(0.35)[b]
	Ensemble-ORB	76.76(0.76)[b]	68.42(1.8)[b]	71.7(0.59)[b]
	OP-AIO-ORB	53.97(38.63)[m]	67.03(25.53)[b]	37.92(27.73)[*]
	OP-Filter-ORB	69.64(3.22)[b]	79.47(7.14)[b]	72.83(2.93)[b]
Fabric8	Dummy	50	50	50
	WP-ORB	61.32(2.17)[b]	67.83(1.32)[b]	61.42(1.04)[b]
	AIO-ORB	57.39(1.44)[b]	73.78(1.42)[b]	63.7(1.03)[b]
	Filter-ORB	64.14(0.62)[b]	71.8(1.09)[b]	67.42(0.42)[b]
	Ensemble-ORB	49.61(1.24)[-s]	65.68(1.22)[b]	55.91(0.59)[b]
	OP-AIO-ORB	72.26(5.51)[b]	41.86(8.81)[-b]	50.3(5.86)[-*]
	OP-Filter-ORB	77.01(18.09)[b]	31.63(19.81)[-b]	40.84(4.19)[-b]
Neutron	Dummy	50	50	50
	WP-ORB	80.47(1.71)[b]	79.52(1.88)[b]	79.52(0.58)[b]
	AIO-ORB	75.34(0.57)[b]	87.53(0.89)[b]	80.72(0.39)[b]
	Filter-ORB	74.01(0.81)[b]	88.98(1.02)[b]	80.67(0.54)[b]
	Ensemble-ORB	77.01(1.55)[b]	72.02(3.41)[b]	74.28(1.1)[b]
	OP-AIO-ORB	70.35(18.03)[b]	66.39(7.53)[b]	66.55(12.39)[b]
	OP-Filter-ORB	74.34(9.05)[b]	70.43(21.56)[b]	70.28(6.99)[b]
Npm	Dummy	50	50	50
	WP-ORB	54.93(1.17)[b]	63.67(1.32)[b]	53.81(0.95)[b]
	AIO-ORB	50.95(1.56)[b]	76.91(1.92)[b]	60.03(1.11)[b]
	Filter-ORB	51.23(4.02)[-s]	73.26(3.94)[b]	59.15(1.94)[b]
	Ensemble-ORB	52.48(1.25)[b]	56.11(0.97)[b]	52.41(0.58)[b]
	OP-AIO-ORB	73.38(16.16)[b]	44.16(24.46)[-*]	47.99(19.35)[b]
	OP-Filter-ORB	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
BroadleafCommerce	Dummy	50	50	50
	WP-ORB	59.43(2.96)[b]	65.93(3.01)[b]	59.72(1.29)[b]
	AIO-ORB	66.78(0.85)[b]	70.12(1.02)[b]	67.85(0.58)[b]
	Filter-ORB	66.83(0.64)[b]	69.2(1.48)[b]	67.51(0.85)[b]
	Ensemble-ORB	61.57(1.2)[b]	59.13(1.46)[b]	58.94(0.45)[b]
	OP-AIO-ORB	84.29(4.22)[b]	26.63(7.15)[-b]	44.42(7.7)[-b]
	OP-Filter-ORB	76.78(5.08)[b]	35.82(4.08)[-b]	50.57(2.62)[s]
Ranking	Dummy	4	4	5
	WP-ORB	2	2	2
	AIO-ORB	2	**1**	**1**
	Filter-ORB	2	**1**	**1**
	Ensemble-ORB	2	3	3
	OP-AIO-ORB	**1**	4	5
	OP-Filter-ORB	**1**	5	6

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. Scott-Knott.BA12 was run for all OOB- and ORB-based approaches together. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

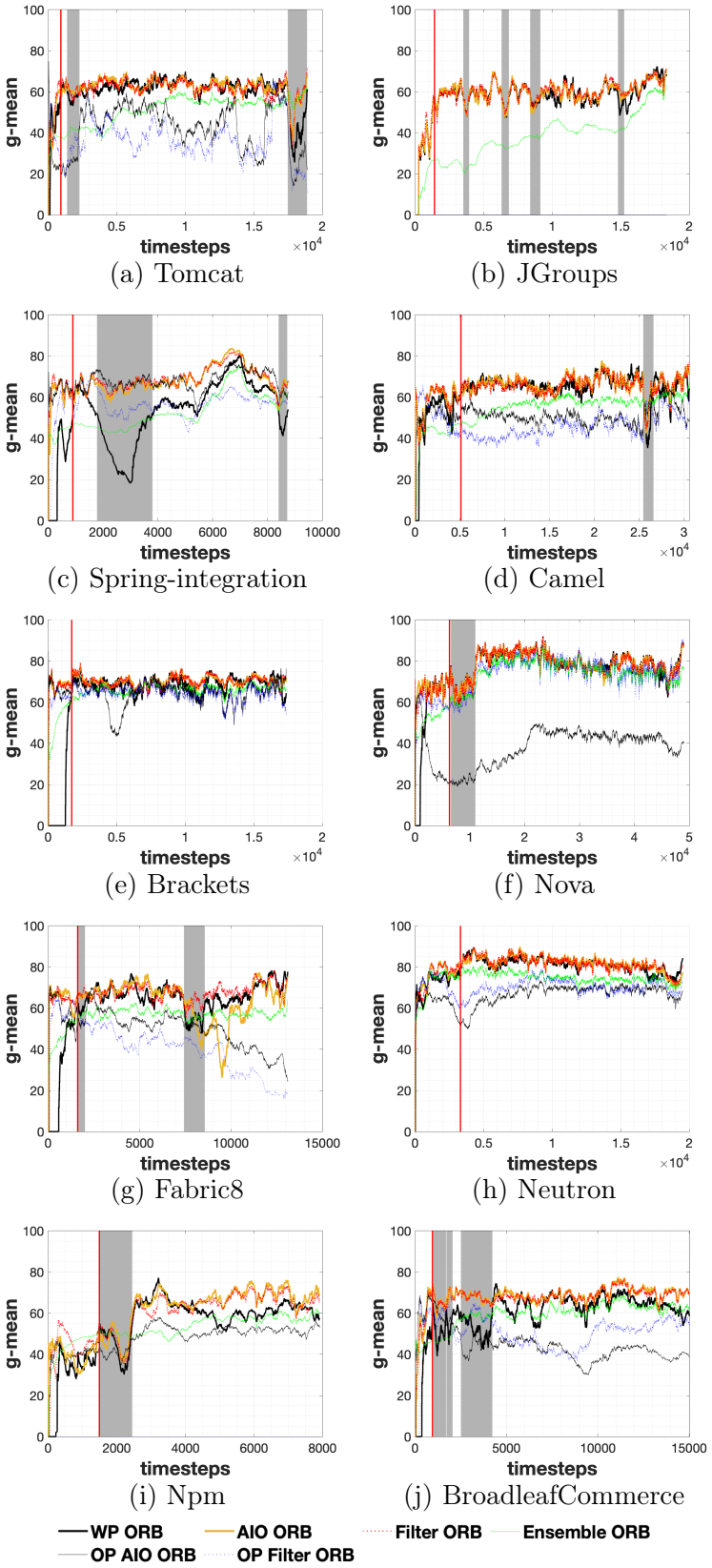


Figure 5.4: G-Mean for all datasets through time using ORB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods.

One or Filtering) improved overall predictive performance in terms of G-Mean, rather than maintaining similar predictive performance as in offline [12] scenarios.

Although Ensemble approaches performed well in offline mode [12], our study shows that ensembles of classifiers trained on separate projects did not perform well in a realistic online scenario. Ensemble-OOB's G-Means were up to 19.25% worse than WP's (for JGroups), and Ensemble-ORB's were up to 19.85% worse than WP's (for JGroups). A further analysis of the results obtained by each classifier within the ensemble reveals that their individual G-Means were not high, which resulted in the poor G-Mean of the ensemble as a whole. This could be due to lack of enough training data for the individual classifiers, given that each classifier in the ensemble is trained with data from one project. This again suggests that the larger amount of varied data was crucial to improve predictive performance in online JIT-SDP. It also explains why the Ensemble approach worked in offline mode but not in online mode. Specifically, studies in offline scenarios ignore the chronology of the projects. When the target and other projects have an overlap in their development period, offline CP approaches train their individual classifiers with a considerably larger amount of data that would still not have been available for training in practice, leading to overoptimistic estimates of predictive performance.

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

those obtained by WP and Ensemble.

Table C.7 in the appendix C shows the overall results for proprietary data. For proprietary data, all WP and CP approaches ranked worse than dummy approach using both OOB and ORB. AIO-OOB (combined), AIO-OOB (proprietary) and Filter-OOB (combined) achieved up to 30.86% (C8), 24.1% (C8) and 26.03% (C5) improvement in terms of G-Mean compared to WP-OOB. AIO-ORB (combined), AIO-ORB (proprietary) and Filter-ORB (combined) achieved up to 23.87% (C8), 22.27% (C8) and 25.1% (C5) improvement in terms of G-Mean compared to WP-ORB.

The G-Mean values obtained when predicting the proprietary data were in general much lower than when predicting the open source data, even when using all the open source data as CP data. This suggests that having a good amount of WP data is also important. The importance of the number of WP training examples is also supported by the (lower) overall G-Mean during the initial phase of the open source projects (Table 5.1) against the (higher) overall G-Mean across the whole open source projects (Table 5.8). So, both open source and proprietary projects need a good number of WP training examples to perform well.

False Positive Rate (FPR), False Negative Rate (FNR), Balanced Accuracy and Informedness are also reported for this chapter in the Appendix C Section C.2 (Table C.20 to Table C.35). Table C.20 shows the FPR and FNR of all WP and CP approaches using OOB for opensource data. It is desirable for a good classification model to minimise both FPR and FNR as much as possible. For AIO and Filter approach, FPR and FNR are comparatively low for most datasets. FPR is up to 18.4% lower for Filter-OOB compared to WP approach when using the dataset NPM, suggesting that it has a good ability to identify true negatives. FNR is up to 5.48% lower for AIO-OOB compared to WP approach when using the dataset Spring-Integration, which means it is correctly identifying most of the defect-

inducing instances. Low FPR and FNR suggests that AIO and Filtering approaches are capable in identifying the clean and defect-inducing instances and has a low rate of failing to detect them, which is desirable from a good JIT-SDP model. In contrast, the Ensemble, Offline-AIO, and Offline-Filter approaches demonstrate either higher FPR and/or higher FNR, suggesting that they are not so good for correctly identifying both classes.

Table C.20 also shows the Balanced Accuracy and Informedness for all approaches for opensource data using OOB. AIO and Filtering approach have higher Balanced Accuracy and Informedness value for all datasets except Camel compared to WP approach. Ensemble, Offline-AIO, and Offline-Filter approaches have lower Balanced Accuracy and Informedness value compared to AIO and Filtering for all datasets (Table C.20).

To summarize, AIO and Filtering approaches performed well compared to WP approaches, not only for G-Mean, but also for additional performance metrics (Balanced Accuracy and Informedness) reported here. Moreover, by considering metrics such as Recall0 and Recall1 (reported in the Appendix C), and FPR and FNR, it is possible to obtain a comprehensive understanding of the performance of these JIT-SDP models.

RQ1.4 Answer: Merging CP and WP data together (All-in-One or Filtering) to train a classifier achieved up to 16.88% higher overall G-Mean than WP classifiers for the open source, and up to 30.86% higher overall G-Mean for the proprietary data in an online scenario. There was no evidence that filtering out training examples dissimilar to the recent software changes of the target project is helpful to improve overall predictive performance. Building separate classifiers from individual projects (Ensemble approach) was detrimental.

5.4.6 RQ1.5: The Need for Updating CP Approaches Over Time

In traditional offline JIT-SDP, models are built with only CP training data and are never updated with new incoming training data over time once they start predicting. Such OP approaches have been successful in achieving similar predictive performance to WP approaches in the offline learning literature [12]. Even though there has been work on online JIT-SDP [10, 51], it is unclear whether OP approaches would perform well enough such that online learning is after all unnecessary. In particular, if OP approaches trained on data received before a given project perform well enough when making predictions in realistic online scenarios, then it would be unnecessary to update CP or WP models over time with incoming WP examples. It would be sufficient for the model to be trained just once with all the available CP data in offline mode.

We investigated this using a JIT-SDP OP model only with CP data that are available until the first WP instance arrives. Once the first WP instance arrives for prediction, training of the OP model is ceased. Comparing the results between OP and CP approaches during initial period, drift period and stable period will provide a better understanding of the necessity of updating CP approaches over time. There are several cases when OP approaches have G-Mean of zero. This happened because the OP model did not have enough training data. When calculating differences in G-Mean between OP and CP, these cases are excluded because they may not reflect the actual magnitude of improvement.

During initial phase, for open source data, the OP AIO and Filtering approaches usually performed worse than the AIO and Filtering approaches (Table 5.1). CP approaches obtained absolute G-Mean improvements of up to 32.15% (AIO-ORB for Nova) for open source data and up to 35.02% (AIO-ORB-proprietary for C9) for proprietary data during this period, when compared to OP approaches. However, OP's G-Means were frequently better than those of WP classifiers in the initial phase. This is supported by the Scott-

Knott.BA12 which ranked the OP-AIO and OP-Filtering better (third) than WP (fourth) for OOB (Table 5.1). For proprietary data, when using OOB, whereas Filtering outperformed the WP approach according to ScottKnott.BA12, all OP approaches performed similarly to WP approaches. In general, we can see that performing online CP learning over time is important to improve G-Means in the initial period, even though data from other projects was also useful to improve G-Means over WP approaches through OP.

During the sudden drop periods, CP approaches obtained absolute improvements in G-Mean of up to 48.16% (Filter-OOB for Npm) compared to OP for open source data, and up to 23.9% (AIO-ORB-proprietary for C9) for proprietary data. While AIO and Filtering were very successful in preventing sudden drops in predictive performance suffered by the WP approach for the open source data, the OP approaches did not perform so well during these periods, being ranked worse than the WP approach based on Scott-Knott.BA12 (Table 5.3). The OP results were a bit more competitive for the proprietary data (Table 5.4), but still not in the top ranked group. These results show that OP data are not enough. Possibly, even though OP data may be prepared to reduce the sudden drop in predictive performance caused by concept drifts, in general it cannot learn well the underlying defect generating process of the project of interest, due to the lack of data coming from it. This suggests that the combined use of WP and data from other projects is important to prevent sudden drops in predictive performance.

For stable periods, OP approaches ranked worse than all CP and WP approaches for both open source and proprietary data (Tables 5.6 and 5.7). There were also some datasets (Fabric8 and Npm for OOB) where the differences in G-Means between OP and WP approaches were quite large. CP approaches obtained absolute G-Mean improvements of up to 41.25% (AIO-ORB for Nova) compared to OP for open source data, and 37.94% (AIO-OOB-proprietary for C9) for proprietary data during the stable period. These results further confirm that using OP data is not enough to achieve good predictive performance

in JIT-SDP when considering realistic online learning scenarios. For JGroups dataset, OP approaches obtained G-Mean of zero. This is because the first change of this project was created before any training data from other projects was available. Hence, the models were not trained, resulting in them always predicting the non-defect inducing class, which led to a G-Mean of zero.

Overall, these results suggest that a combination of CP and WP data is important to improve predictive performance in JIT-SDP. Using only WP data can lead to very poor performance during the initial phase of the projects, and severe drops in predictive performance over time. Using only OP data can help during the initial period, but overall is not enough to achieve competitive results. Tables 5.8 in this chapter and C.7 of the appendix C contain a further comparison of all approaches across the whole data streams formed by the projects. It confirms that OP approaches are not well ranked in terms of G-Mean when considering the data streams as a whole. OP approaches performed up to 39.36% (for Nova with ORB) and 36.64% (for C9 with ORB) worse compared to corresponding CP approaches across all time steps.

Online approaches can make use of both CP and WP incoming data whereas OP approaches only train with data from other projects. As a result, online approaches use more training data than OP approaches, which contributes towards better predictive performance. The presence of WP data is also important along with larger amount of data. This can be realised when we observe the experiments with proprietary projects as proprietary projects have very small amount of WP data and their predictive performance was much worse. This indicates that data from other projects help, but we still need WP data. Therefore, online approaches achieved better predictive performance compared to OP approaches most of the time.

RQ1.5 Answer: CP approaches achieved absolute G-Mean improvements of up to 32.15%, 48.16% and 41.25% for open source data and up to 35.02%, 23.9% and 37.94% for proprietary data compared to OP approaches during initial, sudden drop and stable periods respectively. This highlights the importance of updating the JIT-SDP model with CP (including WP) data received over time.

5.4.7 Characteristics of Dropped CP Instances in Filtering approach

According to the proposed Filtering approach in Chapter Five, if a CP instance is dissimilar to a window of recent WP instances of same label as the CP instance, then it is dropped (initially dropped). This dissimilar CP instance is then stored in a fixed sized waiting queue, for comparing again with WP instances that may arrive at a later period. The CP instance in the waiting queue may become more similar to the recent WP instances due to change in the concept, and can be used for training (reused). However, the CP instance does not remain in the waiting queue forever, as the queue is continuously updated with more recent dropped CP instances, and older CP instances get removed (completely dropped).

Table 5.9 shows the number of CP data points that have been initially dropped, reused and completely dropped (i.e. dropped data points that were never reused for training). The table also shows the percentage of completely dropped CP instances with regards to the whole datastream combining CP and WP data. The number of completely dropped CP data points are between 9442 (4.56%) and 39789 (19.35%), for Brackets and JGroups, respectively (Table 5.9). Moreover, total number of reused data points are between 1494 (Tomcat) and 6270 (Nova).

Table 5.10 shows the percentage of completely dropped CP instances whose numeric

Table 5.9: Number of dropped, reused and completely dropped CP instances for open source datasets

dataset	initially dropped CP	reused CP	completely dropped CP	completely dropped CP %
tomcat	12313	1494	10714	5.26
JGroups	42604	2475	39789	19.35
spring-int.	24115	5499	18191	8.45
camel	16554	5558	9755	5.07
brackets	13097	3440	9442	4.56
nova	20318	6270	13724	8.14
fabric8	19223	5330	12966	6.13
neutron	21053	3652	15681	7.71
npm	22945	4826	17745	8.17
Broadleaf.	19259	4493	14116	6.73

input features are above 75th, below 25th and between 25th and 75th percentile of the corresponding WP data feature. Table 5.11 reports the corresponding total number of instances that are above the 75th percentile and below 25th percentile of the WP data for each feature. Table 5.12 shows the min, max and median for each of the percentiles for all numeric input features.

If a CP instance has a feature whose value is above the 75th percentile or below the 25th percentile of the values presented by the WP data for this feature, this indicates that this instance is considerably distinct from the WP data in terms of this feature. In contrast, if a CP instance has a feature whose value is between the 75th and the 25th percentiles, this CP instance is similar to the WP instances in terms of this feature. However, when a given CP instance has a feature between the 25th and 75th percentiles, this does not mean that the whole instance is similar to the CP instances. Even though some features may be between these percentiles, others may not be. So, the CP instance could still be different from the typical WP instances.

The results of our analysis indicate that for all ten datasets, the percentage of dropped CP instances for the above 75th percentile was higher (up to 87.05%, Table 5.12) than that of below 25th percentile. In other words, cross-project instances were dropped more often for having features with higher values than the typical values presented by the WP

Table 5.10: Summary statistics for dropped CP instances for open source datasets compared to wp instances (in percentage)

dataset	percentile	ns	nd	nf	entropy	la	ld	lt	ndev	age	nuc	exp	rexp	sexp
tomcat	above 75th	16.19	62.24	100	50.99	95.9	89.14	34.59	98.56	29.9	46.33	31.72	4.25	24.49
	below 25th	83.81	37.76	0	0	0	0	24.94	1.04	22.54	0	53.7	0.09	37.87
	between 25th and 75th	0	0	0	49.01	4.1	10.86	40.47	0.4	47.55	53.67	14.58	95.66	37.64
JGroups	above 75th	16.3	56.27	100	62.37	97.63	90.4	14.9	82.17	45.46	30.08	14.3	99.91	10.46
	below 25th	83.7	43.73	0	0	0	0	38.21	4.46	20.32	0	72.08	0.09	60.97
	between 25th and 75th	0	0	0	37.63	2.37	9.6	46.89	13.38	34.22	69.92	13.62	0	28.57
spring-int.	above 75th	13.07	54.98	14.34	29.66	96.47	88.98	59.78	91.15	41.62	61.04	37.77	99.92	62.55
	below 25th	86.93	45.02	0	0	0	0	0.01	1.47	21.23	0	29.82	0.08	0
	between 25th and 75th	0	0	85.66	70.34	3.53	11.02	40.21	7.38	37.14	38.96	32.41	0	37.45
camel	above 75th	18.91	67.74	100	32.09	99.28	90.8	62.36	62.27	21.99	41.54	1.1	99.88	3.84
	below 25th	81.09	32.26	0	0	0	0	7.93	11.84	21.14	0	64.22	0.12	45.41
	between 25th and 75th	0	0	0	67.91	0.72	9.2	29.71	25.89	56.87	58.46	34.68	0	50.74
brackets	above 75th	16.36	57.73	100	67.57	97.33	88.79	33.79	64.6	66.7	47.19	84.13	99.97	83.43
	below 25th	0	0	0	0	0	0	0	0	19.21	0	0	0.03	0
	between 25th and 75th	83.64	42.27	0	32.43	2.67	11.21	66.21	35.4	14.09	52.81	15.87	0	16.57
nova	above 75th	18.84	49.24	100	59.63	97.46	87.95	63.95	13.71	80.42	54.38	96.87	99.95	94.72
	below 25th	0	0	0	0	0	0	0.01	0	19.58	0	0	0.05	0
	between 25th and 75th	81.16	50.76	0	40.37	2.54	12.05	36.05	86.29	0	45.62	3.13	0	5.28
fabric8	above 75th	16.72	61.9	100	38.53	97.78	90.22	52.85	75.98	55.68	45.14	29.7	99.92	42.19
	below 25th	83.28	38.1	0	0	0	0	10.57	8.95	20.11	0	34.95	0.08	24.39
	between 25th and 75th	0	0	0	61.47	2.22	9.78	36.59	15.07	24.21	54.86	35.35	0	33.42
neutron	above 75th	15	43.94	100	53.64	95.68	87.05	85.72	12.86	48.93	97.95	92.59	99.98	98.92
	below 25th	0	0	0	0	0	0	0.08	0	22.5	0	0	0.02	0
	between 25th and 75th	85	56.06	0	46.36	4.32	12.95	14.21	87.14	28.57	2.05	7.41	0	1.08
npm	above 75th	16.88	55.38	100	43.62	97.42	89.68	55.86	65.73	38.74	28.14	37.53	99.93	42.54
	below 25th	83.12	44.62	0	0	0	0	15.66	4.65	21.88	0	42.98	0.07	29.06
	between 25th and 75th	0	0	0	56.38	2.58	10.32	28.48	29.62	39.38	71.86	19.49	0	28.41
Broadleaf.	above 75th	16.87	62.36	100	45.77	97.04	89.65	62.46	88.33	48.85	66.22	34.95	99.94	38.01
	below 25th	83.13	37.64	0	0	0	0	0.04	0	21.69	0	0	0.06	0
	between 25th and 75th	0	0	0	54.23	2.96	10.35	37.5	11.67	29.46	33.78	65.05	0	61.99

instances. For example, for Brackets (WP data), the feature 'exp' has normalised median value of 0.001350672. For dropped CP instances, the normalised median value for 'exp' is 0.620232316. This shows that for the feature 'exp', there is large difference in values between the WP data and the dropped CP data.

Further examination of the characteristics of the dropped CP instances that fall between the 25th and 75th percentile revealed that no instances were dropped that had values for all features between the 25th and 75th percentile. This finding means that there is at least

Table 5.11: Summary statistics for dropped CP instances for open source datasets compared to wp instances (in total count)

dataset	percentile	ns	nd	nf	entropy	la	ld	lt	ndev	age	nuc	exp	rexp	sexp
tomcat	above 75th	1735	6668	10714	5463	10275	9550	3706	10560	3204	4964	3399	455	2624
	below 25th	8979	4046	0	0	0	0	2672	111	2415	0	5753	10	4057
	between 25th and 75th	0	0	0	0	439	1164	4336	43	5095	5750	1562	10249	4033
JGroups	above 75th	6484	22391	39789	24818	38846	35971	5928	32693	18087	11969	5689	39753	4163
	below 25th	33305	17398	0	0	0	0	15202	1774	8087	0	28679	36	24258
	between 25th and 75th	0	0	0	0	943	3818	18659	5322	13615	27820	5421	0	11368
spring-int.	above 75th	2378	10001	2609	5396	17548	16187	10875	16582	7572	11104	6871	18177	11378
	below 25th	15813	8190	0	0	0	0	1	267	3862	0	5425	14	0
	between 25th and 75th	0	0	15582	0	643	2004	7315	1342	6757	7087	5895	0	6813
camel	above 75th	1845	6608	9755	3130	9685	8858	6083	6074	2145	4052	107	9743	375
	below 25th	7910	3147	0	0	0	0	774	1155	2062	0	6265	12	4430
	between 25th and 75th	0	0	0	0	70	897	2898	2526	5548	5703	3383	0	4950
brackets	above 75th	1545	5451	9442	6380	9190	8384	3190	6100	6298	4456	7944	9439	7877
	below 25th	0	0	0	0	0	0	0	0	1814	0	0	3	0
	between 25th and 75th	7897	3991	0	0	252	1058	6252	3342	1330	4986	1498	0	1565
nova	above 75th	2586	6758	13724	8183	13375	12070	8776	1881	11037	7463	13294	13717	13000
	below 25th	0	0	0	0	0	0	1	0	2687	0	0	7	0
	between 25th and 75th	11138	6966	0	0	349	1654	4947	11843	0	6261	430	0	724
fabric8	above 75th	2168	8026	12966	4996	12678	11698	6852	9852	7219	5853	3851	12956	5470
	below 25th	10798	4940	0	0	0	0	1370	1160	2608	0	4532	10	3163
	between 25th and 75th	0	0	0	0	288	1268	4744	1954	3139	7113	4583	0	4333
neutron	above 75th	2352	6891	15681	8412	15004	13651	13441	2017	7672	15359	14519	15678	15511
	below 25th	0	0	0	0	0	0	12	0	3529	0	0	3	0
	between 25th and 75th	13329	8790	0	0	677	2030	2228	13664	4480	322	1162	0	170
npm	above 75th	2996	9827	17745	7741	17288	15914	9913	11664	6874	4993	6660	17733	7548
	below 25th	14749	7918	0	0	0	0	2779	825	3883	0	7627	12	5156
	between 25th and 75th	0	0	0	0	457	1831	5053	5256	6988	12752	3458	0	5041
Broadleaf	above 75th	2381	8803	14116	6461	13698	12655	8817	12469	6896	9347	4934	14107	5366
	below 25th	11735	5313	0	0	0	0	6	0	3062	0	0	9	0
	between 25th and 75th	0	0	0	0	418	1461	5293	1647	4158	4769	9182	0	8750

one input feature where the values are considerably dissimilar to the WP data, which leads to the filtering approach identifying these instances as having distinct characteristics and filtering them. This shows that the filtering approach is effective in removing CP instances that exhibit dissimilar characteristics to WP data.

In the proposed filtering approach in Chapter 5, an incoming CP instance is compared with the window of those WP instances where the label of the WP instances are same as the CP instance. For example, if a CP instance that arrived previously that has a class label 'clean', then it will be compared with most recent window of WP instances in the window that has the class label 'clean'. If another CP instance arrives that has same input feature as the previous clean CP instance, but different class label such as 'defect-inducing', then this later arrived CP instance will be compared with most recent window of WP instances with the class label 'defect-inducing'.

Table 5.12: Min, max, median percentage for dropped CP instances for open source datasets compared to wp instances

dataset	percentile	min	max	median
tomcat	above 75th	4.25	100	46.33
	below 25th	0	83.81	1.04
	between 25th and 75th	0	95.66	14.58
JGroups	above 75th	10.46	100	56.27
	below 25th	0	83.7	4.46
	between 25th and 75th	0	69.92	13.38
spring-integration	above 75th	13.07	99.92	59.78
	below 25th	0	86.93	0.01
	between 25th and 75th	0	85.66	32.41
camel	above 75th	1.1	100	62.27
	below 25th	0	81.09	7.93
	between 25th and 75th	0	67.91	25.89
brackets	above 75th	16.36	100	67.57
	below 25th	0	19.21	0
	between 25th and 75th	0	83.64	16.57
nova	above 75th	13.71	100	80.42
	below 25th	0	19.58	0
	between 25th and 75th	0	86.29	12.05
fabric8	above 75th	16.72	100	55.68
	below 25th	0	83.28	8.95
	between 25th and 75th	0	61.47	15.07
neutron	above 75th	12.86	100	87.05
	below 25th	0	22.5	0
	between 25th and 75th	0	87.14	12.95
npm	above 75th	16.88	100	55.38
	below 25th	0	83.12	4.65
	between 25th and 75th	0	71.86	19.49
BroadleafCommerce	above 75th	16.87	100	62.46
	below 25th	0	83.13	0
	between 25th and 75th	0	65.05	11.67

If there were two CP instances with same input features but different labels, this should not hinder the filtering process, as the Filtering approach will consider the label of the incoming CP instance and compare it with the current window of WP instances of same class label as that of the CP instance. However, after exploring the data, we also found that there were no contradictory CP instance in the 10 open source datasets which have the same input features but different classification label. However, ORB [18] algorithm has a safety mechanism to cope with instances with exactly same input features. If such instance is found more than certain number of time, the ORB algorithm skips learning that defect-inducing training example.

5.5 Sensitivity Analysis

Filtering and AIO were the most competitive approaches according to our experiments. Filtering should be preferred over AIO especially if one wants to strongly focus on prevention of sudden drops in G-Mean. However, Filtering has more hyper-parameters to be tuned than AIO, and it is unclear how these hyper-parameters affect its performance. To avoid loosing such advantage, it is important to understand the effect of these hyper-parameters better. We will focus on Filtering-OOB, as overall it achieved similar or better rankings than Filtering-ORB (see Tables 5.8 in this chapter and C.7 of the Appendix). Results for Filtering-ORB can be found in the appendix C. To investigate this, Analysis of Variance (ANOVA) was performed to analyse the influence of each hyper-parameter as well as their interactions with each other on the average G-Mean. The following within-subject factors are investigated: $windowSize \in \{500, 700, 1000\}$, number of top short distances to be used $K \in \{50, 100, 200\}$, distance threshold for similarity $maxDist \in \{0.6, 0.7, 0.8\}$ and maximum size of the queue of dissimilar CP instances $cpqSize \in \{500, 1000\}$. The *dataset* is a between-subjects factor. Therefore, a Split-Plot (mixed) ANOVA design was used. Mauchly's sphericity test were adopted with a level of significance of 0.05 to evaluate whether or not the sphericity assumption is violated. If violated, Greenhouse-Geisser corrections were adopted to take that into account. Partial eta-squared η_p^2 was used as a measure of effect size.

The analysis was based on Filtering-OOB with six datasets: JGroups, Fabric8 and Neutron (for open source) and C5, C6 and C8 (for proprietary data). These datasets were chosen because they are the open source and proprietary datatsets where this approach achieved the best, worst and average G-Mean values, respectively. Thirty runs for each combination of hyper-parameter values were carried out on each dataset.

Table 5.13 shows the sensitivity analysis for the open source projects with OOB. Several factors and interactions have large ($\eta_p^2 \geq .244$) effect size. Many of them involve the

Table 5.13: ANOVA and Effect Size Results for Open Source Data

Within Subject Test for Filtering-OOB (Open Source Data)					
Factor/Int.	SS	DF	MS	F	η_p^2
K*Dataset	1791.279	4	447.82	2038.001	0.979
K	905.532	1.815	498.873	2060.511	0.959
maxDist*Dataset	304.904	4	76.226	363.86	0.893
K*maxDist*Dataset	326.35	8	40.794	248.202	0.851
windowSize*K*Dataset	254.511	8	31.814	168.637	0.795
K*maxDist	213.227	4	53.307	324.334	0.788
windowSize*Dataset	124.758	4	31.189	160.438	0.787
windowSize*K	242.986	3.467	70.079	322.001	0.787
windowSize	101.327	1.806	56.103	260.612	0.75
windowSize * maxDist * Dataset	60.926	8	7.616	39.002	0.473
windowSize*K*maxDist*Dataset	116.255	16	7.266	38.747	0.471
windowSize*K*maxDist	112.351	8	14.044	74.891	0.463
windowSize*maxDist	36.793	4	9.198	47.106	0.351
maxDist	11.785	2	5.893	28.128	0.244
K*maxDist*cpqSize	7.657	4	1.914	11.486	0.117
K*maxDist*cpqSize*Dataset	7.695	8	0.962	5.772	0.117
windowSize*K*cpqSize*Dataset	8.666	8	1.083	5.527	0.113
K*cpqSize*Dataset	3.712	4	0.928	4.284	0.09
maxDist*cpqSize*Dataset	3.019	4	0.755	3.226	0.069
windowSize*cpqSize*Dataset	1.642	4	0.411	2.825	0.061
maxDist*cpqSize	1.498	1.725	0.869	3.202	0.036
windowSize*maxDist*cpqSize	0.757	3.596	0.21	0.988	0.011
windowSize*cpqSize	0.171	2	0.085	0.588	0.007
K*cpqSize	0.271	2	0.136	0.626	0.007
Between Subject Test for Filtering-OOB					
Dataset	56.040 $\times 10^4$	2	28.020 $\times 10^4$	122.836 $\times 10^4$	1

factors K, *maxDist* and their interaction with dataset. Therefore, we examine them further through Fig. 5.5. From this figure we can see that despite the choice of hyper-parameters leading to significantly different G-Means, such differences did not have large magnitude except for Fabric8. Therefore, in general, small K with small *maxDist* led to top or close to top G-Means for the open source projects, even when this combination of values did not lead to the best G-Mean values. Smaller K means that Filtering can use examples for training so long as they are similar to a small number of recent WP software changes.

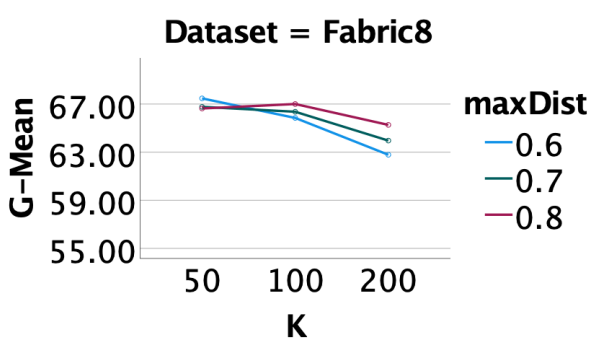
Table 5.14: ANOVA and Effect Size Results for Proprietary Data

Within Subject Test for Filtering-OOB (Proprietary Data)					
Factor/Int.	SS	DF	MS	F	η_p^2
maxDist*dataset	35880.183	4	8970.046	3254.373	0.986
K*dataset	7718.661	4	1929.665	729.317	0.942
K*maxDist*dataset	4795.522	8	599.44	232.216	0.838
maxDist	1289.891	2	644.946	233.989	0.722
K	1076.273	1.858	579.372	203.389	0.693
K*maxDist	989.154	3.577	276.565	95.796	0.516
cpqSize*dataset	44.216	2	22.108	6.963	0.134
cpqSize	33.326	1	33.326	10.497	0.104
maxDist*cpqSize	56.189	2	28.095	8.667	0.088
maxDist*cpqSize*dataset	55.978	4	13.995	4.317	0.088
K*cpqSize*dataset	34.787	4	8.697	2.972	0.062
K*maxDist*cpqSize	63.415	3.472	18.265	5.56	0.058
Between Subject Test for Filtering-OOB					
dataset	74431.101	2	37215.55	14019.740	0.997

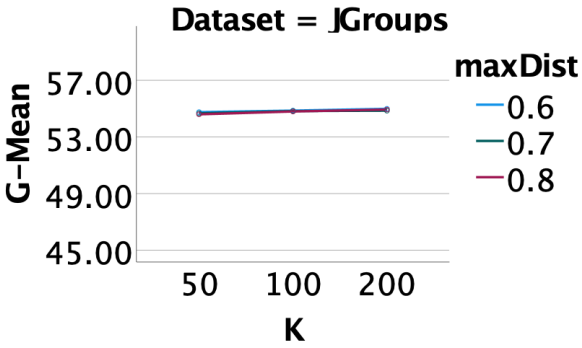
Small *maxDist* means that the similarity measure is more strict, not allowing very dissimilar training examples. Small K with small *maxDist* also led to top or close to top G-Means when using ORB.

The hyper-parameter values selected based on the grid search used for hyper-parameter tuning in Section 5.3.5 generally matched the best hyper-parameter values obtained in the sensitivity analysis. This indicates that the hyper-parameter tuning procedure worked well for the open source projects.

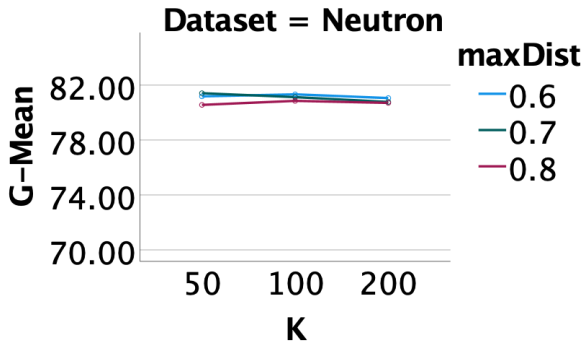
Table 5.14 shows the sensitivity for proprietary projects with OOB. hyper-parameters *maxDist*, K, interaction between K*$maxDist$, $maxDist$*$dataset$ and K*$maxDist$*$dataset$ had very large effect size ($\eta_p^2 \geq .516$) compared to the other hyper-parameters and interactions, whose effect sizes were always $\eta_p^2 \leq .134$. As the largest effect sizes involve hyper-parameters *maxDist*, K and their interaction with *dataset*, we also examine them further through Fig. 5.6.



(a) Effect of $K * maxDist$ with Fabric8



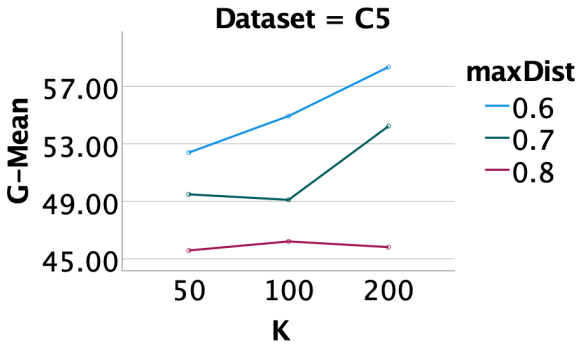
(b) Effect of $K * maxDist$ with JGroups



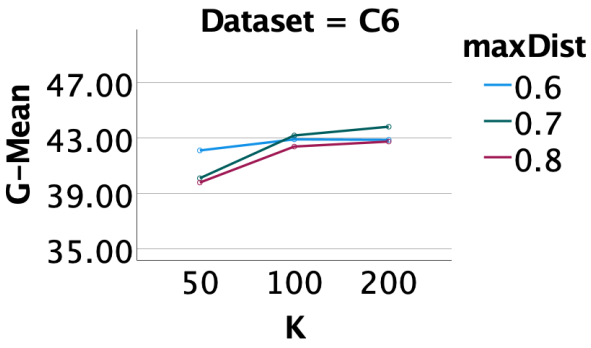
(c) Effect of $K * maxDist$ with Neutron

Figure 5.5: Plots of Marginal Means for Filtering-OOB's Factors K, $maxDist$ and $dataset$ (Opensource data).

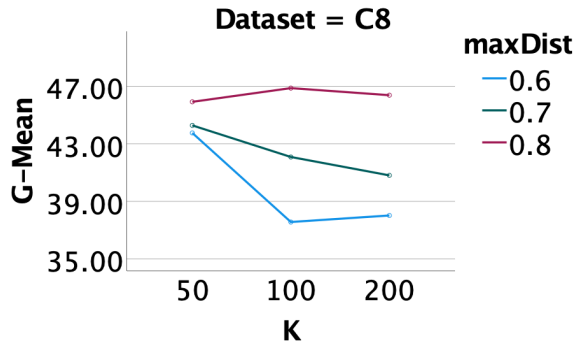
From this figure, we can see that the magnitudes of the differences in G-Mean were larger than for the open source projects. This suggests that Filtering becomes more sensitive to hyper-parameter choice when dealing with smaller datasets. Larger values of K usually led to top or close to top G-Means. A larger K requires CP training examples to be similar to a larger number of WP examples. This may have helped the learning process to focus more on learning the core aspects of the current defect generating process, which are shared by a larger proportion of WP training examples. As the number of WP training examples is small in the proprietary datasets, such additional focus may be important. The results for different values of $maxDist$ were varied, indicating that this hyper-parameter was more dependent on the characteristics of specific datasets. The findings for ORB were similar for K, but large $maxDist$ led to good results in all datasets when using ORB.



(a) Effect of $K * maxDist$ with C5



(b) Effect of $K * maxDist$ with C6



(c) Effect of $K * maxDist$ with C8

Figure 5.6: Plots of Marginal Means for Filtering-OOB's Factors K, $maxDist$ and *dataset* (Proprietary data).

The best values of hyper-parameter *maxDist* selected by the grid search were either small or large for datasets C5, C6 and C8, which aligns with the above finding. The value of hyper-parameter K selected by the grid search was not large for datasets C5 and C6. This means that Filtering was able to achieve competitive results using sub-optimal K value, and it is possible for it to achieve better predictive performance for these datasets if this parameter is better tuned for the dataset of interest.

It is worth noting that the hyper-parameter tuning done in this chapter was based on G-Mean, so as to achieve a good trade-off between Recall0 and Recall1. However, one could focus on another performance metric should they wish to focus more on a specific class. For instance, they could focus on Recall0 (or false positive rate) to target a given pre-defined false positive rate such as the value of 30% suggested in [214]. We have performed

an additional investigation to check whether different hyper-parameter configurations could enable to tune the false positive rate of the approaches. In 8 out of 10 open source datasets, at least one of these configurations led to an average overall false positive rate that is close to 30%: Spring-Integration: 21.63%, Neutron: 25.73%, Tomcat: 27.45%, BroadLeaf: 29.54%, Nova: 29.67%, Brackets: 29.79%, Fabric8: 33.79%, JGroups: 34.08%, Camel: 39.61%, Npm: 40.34%.

The hyper-parameter sensitivity analysis suggests that the G-Mean obtained by Filtering is more sensitive to hyper-parameters when the datasets are smaller. Smaller/larger values for K may be preferable for larger/smaller datasets for the purpose of improving G-Mean. While smaller maxDist led to acceptable G-Means on the larger datasets, the choice of maxDist was very dependent on the dataset and/or underlying machine learning algorithm when the dataset was small.

5.6 Analysis of Computational Cost

Table 5.15 shows total average runtime of AIO, Filter, OP-AIO and OP-Filter with OOB in columns 2–5, computed across 5 runs for 3 open source projects on an Intel(R) Xeon(R) CPU E5-2690 v3 at 2.60GHz and 16Gb of RAM. Tomcat, Npm and Brackets were chosen as they have small, medium and large number of OP projects, respectively.

Table 5.15: Runtime Analysis of CP Approaches

Dataset	**AIO-OOB** (s)	**Filter-OOB** (s)	**OP-AIO-OOB** (s)	**OP-Filter-OOB** (s)	**Duration (year)**	**Runtime Per Year** (AIO)	**Runtime Per Year** (Filter)	**Runtime Per Day** (AIO)	**Runtime Per Day** (Filter)	**# Training Examples After Target Project Starts** (For AIO and Filter)
Tomcat	98.8	2414.6	3	4.6	11.7	8.188	205.983	0.02	0.56	227286
Brackets	111	3185.6	8.6	146	6	17.067	506.6	0.05	1.39	166978
Npm	102.6	3358.8	3.4	51.8	8.16	12.157	405.27	0.03	1.11	204398

The total runtime of the OP approaches is mostly spend for training prior to the beginning of the project being predicted. The total runtime of the AIO and Filter approaches includes both such prior training and the training performed over time, with training examples generated after the target project commenced. As we can see, updating the JIT-SDP classifier with additional training examples received over time leads to large increases in the total runtime in the AIO and Filter approaches compared to the OP approaches. The Filter approaches are slower than their corresponding AIO approaches, because they require the training process to compare the similarity between software changes to decide which training examples to filter out.

Nevertheless, the runtime of *all* approaches is actually very small compared to the number of software changes being learned. Columns 7–10 of Table 5.15 list the average runtime per year and per day for the AIO and Filter approaches. This value is computed by dividing the runtime spent after the target project commenced by the duration of the target project in years and days, respectively. This is done because the learning process of these approaches occurs continuously over time. OP was not included as all its training process is spent before the target project commences. We can see that the highest runtimes are 17.067 and 506.6 seconds per year for AIO and Filter, respectively. They correspond to only 0.05 and 1.39 seconds per day, respectively. As these runtimes are very small, these approaches can be used in practice without causing much computational overhead.

Overall, Filter and AIO had much higher computational cost than the OP approaches. However, such higher computational cost is still very small (less than 1.5 seconds per day).

5.7 Implications to Practice

Our work has practical implications as described below:

- The combined use of both WP and CP data through All-in-One and Filtering improves overall predictive performance of JIT-SDP compared to WP classifiers (RQ1.4). Our study demonstrates the importance of the amount of training data, with both WP and CP data being important to improve JIT-SDP. Practitioners should consider collecting large amounts of both CP and WP training data when adopting JIT-SDP.

- The G-Means obtained by all JIT-SDP approaches were low on the proprietary projects, due to the small number of WP software changes that can be used as training examples in these projects. Therefore, if a software development company expects their projects to be very small like the proprietary software projects used in this study, current JIT-SDP approaches may not be recommended.

- In online JIT-SDP, if practitioners use CP data (along with WP data), this can help improving the performance of the model at the initial phase of a project. This can help practitioners to use JIT-SDP earlier during the development of a project (RQ1.1,RQ1.5).

- WP models can suffer performance drops which cause them to be unsuitable during certain periods of time. These drops mean that, at any given point in time, models may be performing very well or very poorly, being unreliable for practitioners. Using CP data (along with WP data) can help to overcome this issue by preventing or reducing such drops, enabling practitioners to more continuously use JIT-SDP throughout the lifetime of the project (RQ1.2,RQ1.5).

- The use of CP (along with WP data) also improves predictive performance of JIT-SDP during stable periods (RQ1.3,RQ1.5). Therefore, the benefits provided by CP data

during the initial and sudden drop periods are not cancelled out by poor predictive performance during stable periods. This further confirms that CP approaches could be adopted throughout the software development process.

- In practice, it is not possible to know beforehand how long the initial period will last, or whether we are currently in a performance drop period. In particular, tracking predictive performance over time will not enable practitioners to know that. Due to verification latency, any tracking of the predictive performance over time in practice would have a delay of "waiting time" days. This means that the performance being calculated "now" by practitioners would correspond to the performance obtained "waiting time" days ago. Therefore, it is important to adopt an approach that is competitive during the stable periods, while not being so negatively affected during the initial or drop periods. As shown in Sections 5.4.2 to 5.4.6, the proposed approaches AIO and Filtering achieved the best results in terms of that.

- Practitioners may still wish to track such predictive performance over time despite the delay caused by verification latency. Once the drop in performance is noticed, practitioners may opt for stopping to use JIT-SDP until an improvement in performance is noticed. The caveat is that this also means that a JIT-SDP model that is good will only start being used "waiting period" days after its predictive performance becomes suitable. A similar caveat occurs when monitoring the performance during the initial phase to decide when to start using the JIT-SDP classifier. An alternative strategy for the initial phase is to start using the JIT-SDP classifiers after 1418 software changes have been produced, as this was the median length of the initial phase according to Table 5.1. After this length, JIT-SDP classifiers are likely to perform better. Even though this number will not work for all projects, it is a suitable strategy if used together with an approach such as AIO and Filtering, which reduce the risk of underperforming a dummy classifier during this period.

5.8 Threats to Validity

Internal validity: Each approach for each dataset with each classifier has been executed 30 times to mitigate threats to internal validity. Also, results can be influenced by hyper-parameter choices. To ensure a fair comparison, a grid search was performed on a set of possible values for the hyper-parameters of each approach based on an initial portion of the data stream (see Section 6.3). All approaches take verification latency into account and fully respect chronology. There is some chance that the author timestamps used to chronologically order the data contain errors resulting from, e.g., misconfigured clocks or problems when converting old repositories into GitHub. However, such issues tend to affect a very small proportion of commits [208], being unlikely to negatively affect our analyses. An additional threat is the use of Commit Guru to collect labelled software changes. This tool is based on SZZ, which can lead to label noise [215]. This threat is partly mitigated by the finding that the performance reductions caused by the mislabeling incurred by the original SZZ algorithm are not significant compared to the most recent SZZ version [215]. However, even the most recent SZZ version is likely to incur some label noise.

Construct validity: This work is mainly based on G-Mean as an evaluation metric. Recall0 and Recall1 are also reported in the appendix C. Predictive performance is calculated in a prequential way with fading factor to discount older information across time, so that plots of predictive performance reflect the variations in predictive performance observed over time. G-Mean with fading factor is a widely used metric appropriate for class imbalance learning [52]. Even though G-Mean has been criticised in [53], such criticism does not affect the comparisons between classifiers on the same datasets.

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

size.

External validity: This study used 10 open source projects and 9 proprietary projects of various characteristics such as programming language, starting date, number of commits per day, etc. As with any study involving machine learning, the results may not generalise to other types of projects. This study is based on the threshold-dependent performance metric G-Mean. Other threshold values could be used to increase recall on one class at the cost worsening the recall on the other class. As we have adopted OOB and ORB to increase recall on the minority class while attempting not to worsen recall on the majority class too much, we have kept the default threshold of 0.5. Conclusions may not generalise to other threshold values.

5.9 Summary

This chapter provided the first investigation of CP learning for JIT-SDP in a realistic online learning scenario, using both open source and proprietary data. We showed that CP approaches trained with incoming CP and WP data can help to improve predictive performance over WP approaches trained only with WP data and over OP approaches trained only with data from other projects. The AIO and Filtering CP approaches were particularly helpful during the initial phase of the project when there is not enough WP data available (RQ1), leading to up to 53.89% and 35.02% absolute improvements in G-Mean over WP and OP models, respectively. These approaches also helped to reduce or even eliminate sudden drops in predictive performance suffered by the WP model (RQ2) after the initial phase of the project, achieving up to around 37.35% and 48.02% absolute improvements in G-Mean during such periods of time compared with WP and OP models, respectively. Also, during stable periods (RQ3), these approaches achieved up to 29.03% and 41.25% absolute improve-

ments in G-Mean compared to WP and OP models, respectively. These results highlight the importance of using CP data (including data from other projects and data from the project being predicted) over time for training JIT-SDP models.

When comparing CP, WP and OP, the best ranked approaches for initial, drift, non-drift and overall periods were always online CP approaches. For most cases, OP approaches were worse than and for few cases they were similar to online CP approaches. In general, Filtering is recommended over AIO if we wish to strongly focus on preventing sudden drops in predictive performance. If the focus is on the initial or stable periods, we recommend AIO, as it performed competitively and requires less hyper-parameters. Even though the Ensemble approach was shown to perform well in offline learning [12], it was the worst approach when considering a realistic online learning scenario, obtaining average G-Means that were even lower than those of the WP approach. This indicates that splitting data from different projects may not be appropriate in online scenarios. Training a single model combining CP and WP together (AIO) significantly improved performance, hence is more suitable in online JIT-SDP. Our results indicate that both the number of CP and WP training examples is important for achieving good predictive performance in JIT-SDP. This is confirmed by the fact that G-Means of CP approaches obtained during initial phase, when there is less WP training examples, are lower compared to the drop and stable periods.

Chapter Six

Impact of Online and Offline Base Learners on Online Cross-project JIT-SDP Scenarios

6.1 Introduction

Most of the existing JIT-SDP studies leveraged Machine Learning for defect prediction. Many of these JIT-SDP studies used ensemble learners to build the defect prediction models [18, 19, 30, 29]. Ensemble methods are learning algorithms that construct a set of classifiers and then classify new data points by taking a (weighted) vote of their predictions. There are different ensemble methods such as Random Forest, Bagging, and Boosting. Usually, an ensemble consists of multiple weak learners – also known as base learners. The generalization ability of an ensemble is usually much stronger than that of base learners. Ensemble learning is appealing because that it is able to boost weak learners which are slightly better than random guess to strong learners which can make very accurate predictions. Based on the learning process, ensembles can be offline or online. In existing offline JIT-SDP studies,

different offline base learners are used such as decision trees [47], naive bayes [114], logistic regression [7, 13] etc. However, few studies explored online JIT-SDP [18, 19, 29]. The number of online base learners are relatively small compared to offline base learners. Most of the online JIT-SDP studies used online bagging ensemble method [46] composed of Hoeffding Tree (HT) as base learners [27].

Both offline and online base learners have some advantages and limitations. For example, offline base learners iterate over the data several times during the training process, which enables them to learn the data distribution well. Hence, it is likely that offline base learners are stronger and less prone to over-fitting. However, offline base learners use fixed training set and require access to all the training data at once. As the training process is offline, they do not get updated with any new training data over time. The model may perform poorly if there is any change in the underlying data generation process as the old training data may become irrelevant. Moreover, due to the retraining, the computational cost of offline learners can be high.

Unlike offline base learners, online base learners are continuously trained on incoming data; they do not need access to all data beforehand. Online base learners are likely to adapt well to any changes in the relationship between input and output changes over time as they get updated with the most recent data. Also, the training process of online base learners is usually faster because, unlike offline base learners, online base learners do not retrain with the data. However, this may become a weakness for online base learners. Due to having only one pass over the data, online base learners may not learn the data distribution well enough and provide poorer predictive performance.

It may be possible to leverage some of the advantages of both offline and online learning if offline learning can be adapted to operate in an online scenario (defined in chapter 1) by retraining with new data along with past data. However, no investigation so far has

compared the performance of offline models adapted to operate in online scenarios against online learning in JIT-SDP.

In Chapter 5, online CP JIT-SDP approaches performed better than online WP approaches [29]. This indicates that use of CP data can make online classifiers stronger. However, the usefulness of CP data to train offline models applied to online scenarios has not been previously investigated. It might be that the use of CP data makes the performance of the offline learning even better, or CP data may not be so helpful for improving predictive performance when adopting offline learning in online scenarios. If offline models are already stronger, it is possible that WP data is enough and there may be no additional benefit of using CP data. To understand whether CP data helps offline learning in online scenario, it is important to investigate and compare offline learning models in online scenario with both CP and WP data.

Moreover, retraining offline models with CP data may increase the computational cost (run time). Hence, it is also important to investigate the computational cost of the offline approach in online scenario to understand whether such approach would be feasible enough to adopt in practice.

In addition, as most existing online JIT-SDP used Hoeffding Trees (HTs) [27] as base learners, it is still unknown how well approaches based on Hoeffding Trees (HTs) perform compared to the use of other online or offline base models in JIT-SDP, both in terms of predictive performance and training time.

This chapter thus aims at investigating when and to what extent the use of offline learning models in the online JIT-SDP scenario is useful to improve the predictive performance, and how well such models perform compared to different online learning models. This investigation is done both for Within-Project (WP) and Cross-Project (CP) JIT-SDP. In particular, these two contexts may lead to different conclusions in terms of which type

of learning models perform better, due to the different amounts of training data used. In this context, this chapter aims to address the second research question (RQ2) introduced in Chapter 1.

RQ2: How online base learners perform compared to offline base learners in terms of predictive performance? Can use of different base learners influence the predictive performance of cross-project JIT-SDP approaches?

RQ2 can be further divided into following sub research questions:

RQ2.1 Can offline learning help to improve predictive performance compared to online learning in online WP JIT-SDP scenarios? Which base learners usually perform best?

RQ2.2 How beneficial is CP data to improve predictive performance of offline models compared to online models in online CP JIT-SDP scenarios?

RQ2.3 How high is the computational cost of offline learning in online JIT-SDP scenarios compared to that of online learning models?

To answer the above research questions, we adopted an approach called Batch Oversampling Rate Boosting (BORB) [216] [1] that is able to use different offline base learners and can operate in the online JIT-SDP scenario by learning from new training data. BORB [216] and ORB [18] approaches are compared using 5 and 4 different base learners, respectively. The comparisons are performed for both WP and CP learning. Our experiments based on 10 open source software projects show that BORB with WP and CP and ORB with CP obtained similar predictive performance. Whereas BORB was quite robust to the choice

[1]The BORB algorithm was proposed by a collaborator [216] and is not itself part of this thesis. A preliminary evaluation of of BORB in the WP context was performed in [216]. However, all of the experiments and analysis reported in this chapter were done as part of this thesis. This chapter includes following additional works which are not part of the preliminary version in [216]: i) experiments using additional online base learners– ORB-LR, ORB-NB and ORB-MLP; ii) all of the experiments and analysis had to be re-done to include the new online base learners; iii) experiments with cross-project data (all-in-one approach); iv) runtime analysis of offline and online approaches.

of base learner, the predictive performance of ORB was better when using Oza Ensembles of Hoeffding Trees (OHTs) than when using other base learners. The benefit of CP data was smaller for BORB compared to ORB, as BORB achieved similar predictive performance when using CP and WP data, whereas ORB achieved better predictive performance when using CP data compared to WP data. Moreover, the use of CP data with ORB enabled it to achieve more stable predictive performance through time than ORB-WP and BORB, indicating that ORB-OHT-CP is a more reliable approach. The computational cost for BORB with WP and CP is higher than ORB with WP and CP, respectively. Despite the computational cost of ORB-CP being higher than that of BORB-WP, this computational cost was still very low, averaging up to only 1.14 seconds per day in the projects investigated in this study.

The contributions of this work are following:

- We show that offline learning can help JIT-SDP to become less sensitive to the choice of base learner. In particular, most base learners investigated in this study led to similar and top predictive performance when adopting offline learning.
- We show that WP JIT-SDP with offline base learners provide better predictive performance compared to WP JIT-SDP with online base learners.
- We provide the first WP vs CP comparison between offline base learners and online base learners in an online JIT-SDP scenario.
- We show that offline learning reduces the benefit of CP data, such that the CP offline approach obtained similar predictive performance to the offline WP approach. When adopting online learning, CP data was helpful to improve both the average predictive performance and the stability of the predictive performance through time.
- We show that BORB-WP and BORB-CP have similar average predictive performance

to ORB-OHT-CP, but that BORB suffered severe performance drops or variance in some of the datasets. ORB-OHT-CP did not suffer from such drops, being able to maintain stable predictive performance over time. Hence, if we are more concerned with reducing the performance drops, online ORB-OHT-CP is recommended.

- We compare the computational costs of offline (BORB) and online (ORB) models with different base learners and show that the computational cost for BORB with WP and CP is higher than that of ORB with WP and CP, respectively. The computational cost of ORB-OHT-CP is higher than that of BORB-WP approaches.

This chapter is further organized as follows. Section 6.2 introduces the BORB approach proposed in [216]. Section 6.3 explains the experimental setup for answering the RQs. Section 6.4 explains the results of the experiments. Section 6.5 presents threats to validity. Section 6.6 presents the summary.

6.2 BORB Algorithm

This section explains the BORB approach [216] used in this study. To investigate the influence of offline learning in online JIT-SDP scenarios, an approach called Batch Oversampling Rate Boosting (BORB) was proposed in [216], which consists of an adaptation of Oversampling Rate Boosting (ORB) [18]. Following the definitions given in Chapter 1, BORB uses a "batch" training process, where the model is periodically re-trained on historical data using an offline learning algorithm. Different from the offline learning approaches presented in Chapter 1, such batch training process ensures that only training examples whose labels are already available with respect to the dataset chronology can be used for training (i.e., it takes the verification latency problem into account and follows the online JIT-SDP scenario explained in Chapter 1.

Algorithm 6 BORB's testing and training procedure

Input: data stream $\mathcal{S}$, ORB parameters $(th, l0, l1, m)$, window size w_s, training sample size n, retraining time period rp, number of training iterations it, waiting time period w

$\theta \leftarrow 0.5$ ▷ decision threshold adopted by the classifier
$\mathcal{X} \leftarrow \emptyset$ ▷ set of software changes received so far
for $x^{(t)} \in \mathcal{S}$ **do** ▷ $x^{(t)}$ change (example) committed at timestep t
  if clf is already trained **then**
    $\vec{c} \leftarrow \emptyset$
    for $i \leftarrow (w_s - t)$ to t **do**
      $\vec{c} \leftarrow \vec{c} \cup score(clf, \theta, x^{(i)})$
    end for
    $\theta \leftarrow quantile(\vec{c}, th)$
    $\hat{y}^{(t)} \leftarrow pred(clf, x^{(t)}, \theta)$
  else
    $\hat{y}^{(t)} \leftarrow 0$ ▷ for this problem, class 0 refers to the clean class
  end if
  $\mathcal{X}.add(x^{(t)})$
  $\tau \leftarrow updateTrainingSet(\tau, \mathcal{X}, w)$ ▷ update the training set with any new label
  if modulo$(t, rp) == 0$ **then** ▷ an entire training procedure is performed
    $clf \leftarrow$ restart(clf) ▷ restart or instantiate a new classifier
    $obf_0 \leftarrow 1$
    $obf_1 \leftarrow 1$
    for $i \leftarrow 0$ to it **do**
      $S \leftarrow skewedSample(\tau, obf_0, obf_1, n)$
      $clf \leftarrow train(clf, S)$
      $ma \leftarrow \frac{1}{w_s} \sum_{j=t-w_s}^{t} pred(clf, x^{(j)}, 0.5)$
      $obf_0 \leftarrow OBF_0(ma, th, l_0, m)$
      $obf_1 \leftarrow OBF_1(ma, th, l_1, m)$
    end for
  end if
end for

Despite BORB and ORB sharing some key concepts, they operate very differently in terms of the following aspects:

- ORB is an online learning approach that processes each training example once at its arrival and then discards it. BORB maintains a training set containing all labeled software changes received so far. This set is used to periodically re-train the classifier, enabling offline base learners and procedures to be adopted.
- As BORB maintains a training set, whenever it is found that a given training example was previously mislabeled due to a corresponding defect having been found only after the waiting period, this example can be replaced by a new training example with the

correct label. This can reduce noise in the training set, potentially helping to improve predictive performance.

- While ORB computes an oversampling rate to tackle class imbalance in an online manner, BORB computes the oversampling rate by repeatedly performing iterations through the training data to assess the effect of this oversampling rate and further adjust it accordingly. This may result in a more adequate oversampling rate.

Algorithm 6 presents BORB's pseudocode. For each new incoming software change ($x^{(t)}$) received at timestep t, the base learner clf, if already trained, provides a class prediction (line 10). Note that clf is not useful until τ contains at least one labeled software change from each class (clean and defect-inducing). Before that, all provided predictions are assigned to the clean class.

Different from online learning models, BORB stores all historical software changes in $\mathcal{X}$. As new class labels arrive (following the procedure described in Chapter 1 and using waiting period w), they are immediately used to create training examples corresponding to their respective software changes in $\mathcal{X}$ (line 15). These training examples are added to the training set τ. If a given new defect-inducing class label corresponds to a software change that was previously labeled as clean, the previous training example in τ is replaced by the new one with the defect-inducing label. The base learner is periodically reset at every rp timesteps, and the training set τ is used to retrain it (lines 16 to 27).

BORB tackles the class imbalance problem at two different moments: in the test phase by picking a presumable adequate classifier prediction threshold (lines 5 to 9) and in the training phase by means of an oversampling mechanism (lines 16 to 27). For the testing phase, BORB considers that classifiers usually make predictions based on a prediction threshold θ. The value of θ is typically set to 0.5. If the score given by the classifier is smaller than 0.5, class 0 is predicted. Otherwise, class 1 is predicted. However, this threshold can

potentially be adjusted to help dealing with class imbalance. BORB does that in lines 5 to 9. In particular, $\vec{c}$ (line 7) stores the prediction scores over the last w_s software changes. The decision threshold θ to be adopted by the base learner is a quantile in $\vec{c}$ corresponding to a hyperparameter th. This hyperparameter represents the proportion of the predictions that is targeted to be defect-inducing predictions (th). E.g., if $th = 0.5$, $\theta = \text{median}(\vec{c})$. As JIT-SDP is a class imbalanced problem, the target proportion should normally be less than 0.5.

For the training phase, similar to ORB [18], BORB deals with class imbalance based on oversampling as follows. The oversampling rate is used to decide whether and by how much to oversample examples of a given class for training the base learner. The oversampling rate is adjusted based on the predictions given to the most recent software changes. This enables adjustments on the base learners without having to wait for the labels of these software changes. In particular, a proportion ma of predictions given to the defect-inducing class over the most recent w_s software changes is determined (line 23). This proportion is compared to the same hyperparameter th used in the test phase. If ma indicates that BORB is predicting the defect-inducing class more/less often than the target proportion th, we need to oversample the clean/defect-inducing class, so that BORB focuses more on learning how to identify examples of this class. The idea is that if S is skewed towards the defect-inducing class, the classifier should also incorporate this skewness in its predictions.

The iterations from lines 20 to 26 are responsible for updating the base learner based on the oversampling rate. The function *skewedSample* (line 21) retrieves a sample S containing n training examples from τ, based on the oversampling rate, which is determined according to the factors obf_0 and obf_1, as detailed in Algorithm 7. Therefore, many training iterations will be performed on different training sets S in order to make the base learner converge to a skew respecting th. The idea is that if ma (line 23) distantiates from th, the obfs are adjusted so that S contains the necessary class imbalance to make the base learner

Algorithm 7 *skewedSample* function

Input: n - the number of examples in the training set S, obf_0 and obf_1 - the absolute values for computing the desired proportions of clean and defect-inducing examples in S, $\tau_{(lab)}$

1: $S \leftarrow \emptyset$
2: **for** $i \leftarrow 0$ to n **do**
3: $r = \text{rand}(0, obf_0 + obf_1)$ ▷ r is a random number between 0 and $obf_0 + 0bf_1$
4: **if** $r < obf_0$ **then**
5: $S \leftarrow S \cup pickRandom(\tau, 0)$ ▷ S is incremented with a random example of the clean class from τ
6: **else**
7: $S \leftarrow S \cup pickRandom(\tau, 1)$ ▷ S is incremented with a random example of the defect-inducing class from τ
8: **end if**
9: **end for**

Return: S

accumulate new biased information such that at the last training iterations ma approaches th.

Equations 6.1 and 6.2 [18] compute obf_0 and obf_1 (lines 24 and 25 of Algorithm 6), respectively. Figure 6.1 presents the behaviour of these equations. In these equations: m stands as the rate of growth; $l0$ and $l1$ defines the upper limits for the oversampling rates for classes clean and defect-inducing, respectively; and th, as previously mentioned, represents the target proportion of defect-inducing outcomes of the base learner. Figure 6.1 a) shows the $obf0$ and $obf1$ curves generated by the parameters ($th = 0.4$, $l0 = 5$, $l1 = 12$, $m = 1000$ and $ma \in 0..1$) while in Fig. 6.1 b) the parameters $l0 = 9$ and $m = 10$. Due to different values for parameter m, in Fig. 6.1 a), when $ma \approx th$, obf_0 and obf_1 are less impacted than in Fig. 6.1 b). As in JIT-SDP the class 0 (clean class) is usually the majority class, it is advisable to use obf upper limits values (l_0) lower than the ones for defect-inducing class (l_1).

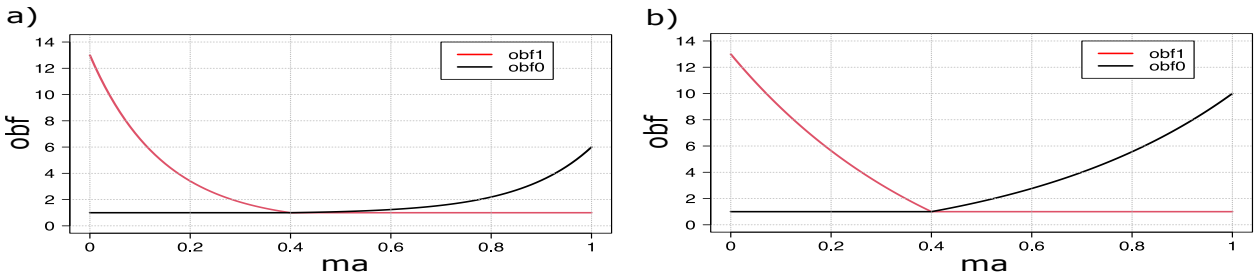


Figure 6.1: OBF

$$OBF_0(ma, th, l_0, m) = \begin{cases} \left(\frac{m^{ma} - m^{th}}{m - m^{th}} \times l_0\right) + 1, & \text{if } ma > th \\ 1, & \text{otherwise} \end{cases} \tag{6.1}$$

$$OBF_1(ma, th, l_1, m) = \begin{cases} \left(\frac{m^{(th-ma)} - 1}{m^{th} - 1} \times l_1\right) + 1, & \text{if } ma \leq th \\ 1, & \text{otherwise} \end{cases} \tag{6.2}$$

BORB was originally proposed in [216] as a WP approach respecting the online scenario introduced in Chapter 1. However, we have investigated BORB for both WP and CP learning in this thesis. For WP learning, the JIT-SDP model is trained with data from the target project only. For CP learning, the JIT-SDP model is trained both with data from the target project (WP data) and from all other available projects (CP data). Hence, for CP learning, BORB model is trained with both CP and WP data together similar to the All-in-One approach from Chapter 5. This means that any benefits of CP data mentioned in this study refer to the benefits obtained from combining both CP and WP data. This is reasonable in online scenarios, as both CP and WP data become available over time during the course of a project .

6.3 Experimental Setup

This section explains the experimental setup for answering the RQs introduced in Section 6.1. To perform the analysis for RQ2.1, we compare the predictive performances of BORB-WP and ORB-WP [18] approaches; for RQ2.2, we compare the predictive performances of BORB-WP, BORB-CP and ORB-CP (Chapter 5) approaches; and for RQ2.3, runtimes for BORB-WP, BORB-CP and ORB-CP are compared. Our analyses are based on various online and offline base learners, as listed in Section 2.1. For RQ2.1 and RQ2.2, we compared all approaches against a dummy classifier that randomly predicts defect-inducing or clean. This is because being able to outperform a dummy classifier in terms of overall predictive performance means the JIT-SDP model was able to learn relevant JIT-SDP knowledge.

Given a certain project P, we are interested in using JIT-SDP to predict the software changes of P as defect-inducing or clean. Such predictions should respect chronological order according to the scenario explained in Chapter 1. Chronology is determined based on author timestamp, as recommended in [208].

When creating a predictive model for a given project P, WP approaches make use of only WP data from P for training. CP approaches make use of data from all projects for training, including P. The training procedure of all approaches at a given timestamp t ensures that only training examples that have already been labeled by timestamp t based on their chronology and waiting period are used for training, as explained in Chapter 1 [18]. Waiting period of 90 is used as in previous studies [18, 29] for open source data.

All approaches have been executed 30 times on each data set. A replication package will be provided in Github and Zenodo.

6.3.1 Predictive Performance

The metrics adopted for measuring predictive performance are Geometric Mean (G-Mean) of Recall0 and Recall1, where Recall0 is the recall on the clean class and Recall1 is the recall on the defect-inducing class. These metrics are recommended for class imbalance learning due to being unbiased metrics in recent study [54]. More details can be found in Chapter 2.

6.3.2 Statistical Tests and Effect Size

Scott-Knott procedure [56] is used to compare the predictive performance obtained by BORB and ORB approaches across datasets, which ranks the models and separate them into sub-groups. As recommended by Menzies et al. [57], this test uses Non-parametric bootstrap sampling. This makes this a non-parametric test which is adequate for comparison across data sets [58]. To rule out insignificant differences in the predictive performance, this test uses A12 effect size [59]. Approaches are placed in separate groups by Scott-Knott test only if the A12 size is significant [57]. We will refer to Scott-Knott based on Bootstrap sampling and A12 as Scott-Knott.BA12 in this chapter. Smaller Scott-Knott.BA12 rankings are better rankings.

We also report the A12 effect sizes against the dummy approach for each dataset individually to support the analysis. Symbols [*], [s], [m] and [b] represent insignificant (A12 < 0.56), small (A12 ≥ 0.56), medium (A12 ≥ 0.64) and large (A12 ≥ 0.71) A12 effect size. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the corresponding WP approach.

6.3.3 Hyper-Parameter Tuning

Random search is used for hyper-parameter tuning as suggested in [136] [217], which show that random search performed similar or better compared to grid search for hyper-parameter optimisation. For each hyper-parameter of each configuration of a classifier, a random value is chosen regarding the probability distribution specified (either uniform or log-uniform, depending on the hyperparameter being tuned). ORB and BORB (meta-models) are associated with OHT, IHF, LR, MLP, NB and IRF (base learners). So, the overall configuration of the classifier includes the BORB, ORB and the base learner's hyper-parameters. The first 3000 commits for each datasets are used for hyper-parameter tuning for both WP and CP approaches. For BORB and ORB approaches with each dataset, base learner and hyper-parameter configuration, 3 executions have been performed for tuning purposes. For each dataset and classifier, 128 configurations were evaluated. More details of the investigated hyper-parameter values are given in Table D.1 in the appendix D. This chapter used 10 open source data sets described in the Table 4.2 of Chapter 4.

6.4 Experimental Results and Discussion

Tables 6.1, 6.2, 6.3 and 6.4 present the average G-Mean, A12 effect size and Scott-Knott.A12 ranking of BORB and ORB approaches with different base learners (listed in section 2.1) compared against a dummy classifier, using WP and CP data. Section 6.4.1 focuses on the comparison between WP approaches to answer RQ2.1; section 6.4.2 focuses on the comparison between CP and WP approaches to answer RQ2.2; and section 6.4.3 focuses on the computational cost analysis of each approach to answer RQ2.3.

Recall0, Recall1, False Positive Rate (FPR) and False Negative Rate (FNR) for this chapter are reported in section D.3 of Appendix D. Additionally, a discussion on the comparison between online and offline approaches, with a focus on the number of data points required by online techniques, such as Hoeffding trees, to achieve a performance level that is comparable to offline techniques is included in section D.2 of Appendix D.

6.4.1 RQ2.1: Can offline learning help to improve predictive performance in online WP JIT-SDP scenarios? Which base learners usually perform best?

To answer RQ2.1, we compared predictive performances of offline and online models with different base learners. As existing online WP JIT-SDP studies never explored any other base learners except OHT, it is interesting to know whether using different base learners would improve the predictive performance not only of offine WP approaches, but also of online WP approaches.

Scott-Knott.A12 test was performed on all BORB-WP, ORB-WP, BORB-CP and ORB-CP approaches together. The test shows that MLP, LR and IRF are the top ranked approaches for BORB-WP (Table 6.1), whereas for ORB-WP, no base learners were able to achieve top rank (Table 6.2). The best performing base learner for ORB-WP was OHT, which ranked third based on Scott-Knott.A12 ranking. For BORB-WP, all base learners outperformed dummy approach with large A12 effect size for all 10 datasets. For ORB-WP, ORB-OHT-WP and ORB-NB-WP outperformed dummy approach with large A12 effect size in 9 and 6 datasets, respectively. However, base learners with BORB-WP obtained higher G-Mean compared to base learners with ORB-WP for all datasets except Nova, where ORB-OHT-WP performed slightly better. The absolute improvement in G-Mean was up to 27.97%

(Spring-Integration) when comparing the best ranked base learners (MLP, LR and IRF) for BORB with best ranked base learner (OHT) for ORB.

Table 6.1: Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare BORB with WP data using different base learners

Dataset	**BORB-IHF-WP**	**BORB-IRF-WP**	**BORB-LR-WP**	**BORB-MLP-WP**	**BORB-NB-WP**
Brackets	67.86 (00.25)[b]	72.29 (00.10)[b]	71.86 (00.04)[b]	73.15 (00.10)[b]	62.89 (00.11)[b]
Broadleaf	67.46 (00.25)[b]	70.01 (00.15)[b]	71.72 (00.11)[b]	71.12 (00.25)[b]	61.10 (00.20)[b]
Camel	66.58 (00.15)[b]	70.13 (00.12)[b]	70.14 (00.04)[b]	70.51 (00.12)[b]	58.12 (00.12)[b]
Fabric8	63.75 (00.50)[b]	67.40 (00.14)[b]	66.78 (00.04)[b]	70.03 (00.17)[b]	60.21 (00.18)[b]
JGroups	61.15 (00.37)[b]	62.77 (00.16)[b]	65.05 (00.09)[b]	66.29 (00.21)[b]	57.80 (00.17)[b]
Neutron	73.87 (00.23)[b]	68.90 (00.06)[b]	74.85 (00.03)[b]	77.29 (00.22)[b]	60.79 (00.14)[b]
Nova	76.47 (00.15)[b]	**77.64** (00.06)[b]	69.85 (00.01)[b]	77.51 (00.21)[b]	64.99 (00.11)[b]
Npm	54.14 (01.13)[b]	62.75 (00.43)[b]	66.70 (00.12)[b]	65.79 (00.40)[b]	51.99 (00.35)[b]
Spring-Int	58.99 (00.68)[b]	67.32 (00.13)[b]	69.05 (00.06)[b]	67.91 (00.39)[b]	52.83 (00.15)[b]
Tomcat	65.37 (00.22)[b]	69.27 (00.08)[b]	69.62 (00.04)[b]	69.62 (00.16)[b]	57.93 (00.24)[b]
Ranking	2	**1**	**1**	**1**	4

Table 6.2: Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare ORB with WP data using different base learners

Dataset	**ORB-LR-WP**	**ORB-MLP-WP**	**ORB-NB-WP**	**ORB-OHT-WP**
Brackets	44.70 (00.57)[-b]	42.42 (12.67)[-m]	61.39 (01.90)[b]	67.81 (00.36)[b]
Broadleaf	46.76 (00.53)[-b]	46.61 (05.92)[-b]	55.70 (02.32)[b]	64.25 (00.83)[b]
Camel	48.27 (00.37)[-b]	48.65 (04.49)[-s]	51.90 (01.86)[b]	53.17 (00.80)[b]
Fabric8	42.92 (00.88)[-b]	40.60 (11.88)[-b]	56.68 (01.67)[b]	63.27 (00.71)[b]
JGroups	47.21 (00.55)[-b]	46.60 (04.85)[-b]	55.47 (01.11)[b]	60.89 (00.48)[b]
Neutron	49.52 (00.51)[-b]	49.14 (11.45)[-*]	39.99 (05.40)[-b]	77.03 (00.47)[b]
Nova	46.31 (00.32)[-b]	43.50 (12.56)[-b]	68.63 (01.24)[b]	**77.86** (00.20)[b]
Npm	49.90 (00.66)[*]	48.77 (05.02)[-s]	45.45 (01.80)[-b]	57.45 (00.83)[b]
Spring-Int	37.82 (00.93)[-b]	39.71 (08.32)[-b]	50.24 (01.92)[m]	41.08 (00.70)[-b]
Tomcat	46.68 (00.59)[-b]	49.21 (04.92)[-*]	48.33 (06.77)[*]	64.27 (00.41)[b]
Ranking	7	7	5	3

To investigate this further, we compared the performances of BORB and ORB with same type of base learners (for example, we compare BORB-MLP with ORB-MLP, BORB-LR with ORB-LR and BORB-NB with ORB-NB). Tables 6.1 and 6.2 show that, the G-Means obtained by BORB-MLP, BORB-LR and BORB-NB are much higher than G-Means obtained by ORB-MLP, ORB-LR and ORB-NB, respectively. BORB achieved better performance compared to ORB for all 10 datasets for MLP and LR, and 9 out of 10 datasets for NB. Absolute improvements in G-Mean for MLP, LR and NB were up to 31.23% (for

Table 6.3: Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare BORB with CP data using different base learners

Dataset	BORB-IHF-CP	BORB-IRF-CP	BORB-LR-CP	BORB-MLP-CP	BORB-NB-CP
Brackets	72.48 (00.18)[b]	76.73 (00.06)[b]	77.22 (00.04)[b]	77.31 (00.07)[b]	69.53 (00.16)[b]
Broadleaf	68.50 (00.26)[b]	70.24 (00.11)[b]	71.76 (00.10)[b]	70.87 (00.16)[b]	64.17 (00.19)[b]
Camel	66.35 (00.18)[b]	70.21 (00.09)[b]	69.74 (00.05)[b]	70.37 (00.12)[b]	57.67 (00.10)[b]
Fabric8	67.52 (00.44)[b]	67.57 (00.12)[b]	66.56 (00.07)[b]	66.63 (00.18)[b]	60.43 (00.34)[b]
JGroups	60.87 (00.29)[b]	62.95 (00.21)[b]	64.73 (00.06)[b]	66.07 (00.24)[b]	55.70 (00.28)[b]
Neutron	72.31 (00.10)[b]	75.08 (00.05)[b]	74.96 (00.03)[b]	**80.74** (00.13)[b]	75.64 (00.10)[b]
Nova	75.72 (00.14)[b]	71.19 (00.03)[b]	71.13 (00.02)[b]	79.13 (00.11)[b]	62.27 (00.13)[b]
Npm	63.52 (00.23)[b]	67.22 (00.16)[b]	67.59 (00.06)[b]	67.90 (00.27)[b]	58.81 (00.32)[b]
Spring-Int	69.87 (00.36)[b]	71.94 (00.12)[b]	71.95 (00.05)[b]	72.11 (00.15)[b]	57.47 (00.23)[b]
Tomcat	65.47 (00.18)[b]	68.82 (00.13)[b]	68.90 (00.06)[b]	68.66 (00.12)[b]	55.16 (00.24)[b]
Ranking	**1**	**1**	**1**	**1**	4

Table 6.4: Average G-Means, Standard deviation, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 ranking to compare ORB with CP data using different base learners

Dataset	ORB-LR-CP	ORB-MLP-CP	ORB-NB-CP	ORB-OHT-CP
Brackets	48.55 (00.43)[-b]	49.39 (07.41)[-*]	64.15 (03.88)[b]	75.94 (00.38)[b]
Broadleaf	48.24 (00.88)[-b]	50.39 (04.05)[-s]	52.84 (05.57)[b]	71.41 (00.34)[b]
Camel	49.26 (00.29)[-b]	47.54 (05.64)[-b]	55.66 (01.53)[b]	69.00 (00.25)[b]
Fabric8	49.77 (01.02)[-s]	48.55 (03.10)[-m]	54.87 (03.88)[b]	69.28 (00.44)[b]
JGroups	47.60 (00.61)[-b]	46.79 (04.38)[-b]	52.07 (01.78)[b]	60.95 (00.44)[b]
Neutron	49.55 (00.43)[-b]	51.78 (08.35)[s]	79.04 (03.35)[b]	**81.95** (00.25)[b]
Nova	49.49 (00.23)[-b]	48.91 (11.02)[-s]	69.52 (04.62)[b]	79.90 (00.21)[b]
Npm	48.31 (00.98)[-b]	47.41 (03.76)[-b]	43.32 (04.03)[-b]	65.18 (00.60)[b]
Spring-Int	49.37 (01.04)[-m]	48.29 (04.37)[-s]	52.20 (03.87)[b]	70.38 (00.63)[b]
Tomcat	49.63 (00.69)[-b]	48.16 (03.87)[-m]	49.94 (02.41)[-*]	66.66 (00.28)[b]
Ranking	6	6	5	**1**

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach (always predicts 50%). Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the corresponding Dummy approach. Scott-Knott.BA12 was run for all BORB and ORB-based WP and CP approaches together. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings. Dummy classifier ranked 6th according to Scott-Knott test.

Spring-Integration), 34.01% (for Nova) and 20.8% (for Neutron), respectively.

This indicates that the offline mechanism of BORB helps to strengthen these base learners. BORB re-trains the model periodically with all of the available training data. Hence, the base learners are trained with several iterations of the data. The multiple iterations can simulate a situation where there is more training data. As a result, the base models can learn the data distribution better. It is worth noting that, if JIT-SDP often had concept drifts affecting the relationship between input features and the label (clean or defect inducing), iterating over all historical data multiple times would likely be detrimental to the predictive performance. This is because different portions of the training set would correspond to different relationships. The model would thus try to learn a mix of relationships, being unable to learn any of the individual relationships well enough. However, we have found in previous work [218] that, despite sometimes happening, changes in such relationship are much less common than changes in the values of the input features. Therefore, iterating through historical data multiple times may offer some benefits, as shown in this section.

Unlike BORB, ORB is an online learning approach and in online learning, each incoming data instance is used to train the model only once, i.e., model is never re-trained with historical data. Due to this, when using only WP training data, online models may not learn the data distribution so well as the offline models.

From Fig. 6.2, it is visible that the best 3 BORB-WP approaches (MLP, LR and IRF as shown in blue, orange and green, respectively) performed on average very similar to each other, and comparatively better than the best ORB-WP approach (OHT as shown in black). Chapter 5 showed that WP approaches can suffer low performance in the very beginning of a project, when there is lack of sufficient data. This initial period of low performance can be seen in most datasets for ORB-WP. BORB-WP approaches also suffered in such initial

period, but was sometimes able to improve the G-Mean during the initial period (e.g., for Npm and Neutron).

Apart from the initial period, some other large performance drops can be observed in Camel, Npm and Spring-Integration (Fig. 6.2c, 6.2h and 6.2i) for ORB-WP. For these datasets, all 3 BORB-WP approaches managed to maintain stable performance during the drop periods. For BORB-WP approaches, a number of severe and frequent recurring drops can be observed in Neutron and Nova (Fig. 6.2f and 6.2g). However, ORB-WP maintained stable performance during these periods for these datasets.

Hence, considering the average predictive performance, ORB-WP is not the best option compared to BORB-WP, as seen based on the Scott-Knott-A12 ranking. However, as seen in Fig. 6.2, for two datasets Neutron and Nova (Fig. 6.2f and 6.2g), BORB experienced severe and frequent performance drops. Due to this recurrent fluctuations in the performance, BORB's stability may seem unreliable.

RQ2.1: Offline learning helps to improve average G-Mean in the online WP JIT-SDP scenario. BORB-WP with MLP, LR and IRF ranked the best, and performed better than the top ORB-WP approach, which used OHT. BORB-WP achieved up to 27.97% absolute improvement in terms of average G-Mean when compared against ORB-WP. However, offline WP BORB approaches suffered severe and recurrent performance drops in two datasets, which reveals some instability of these approaches.

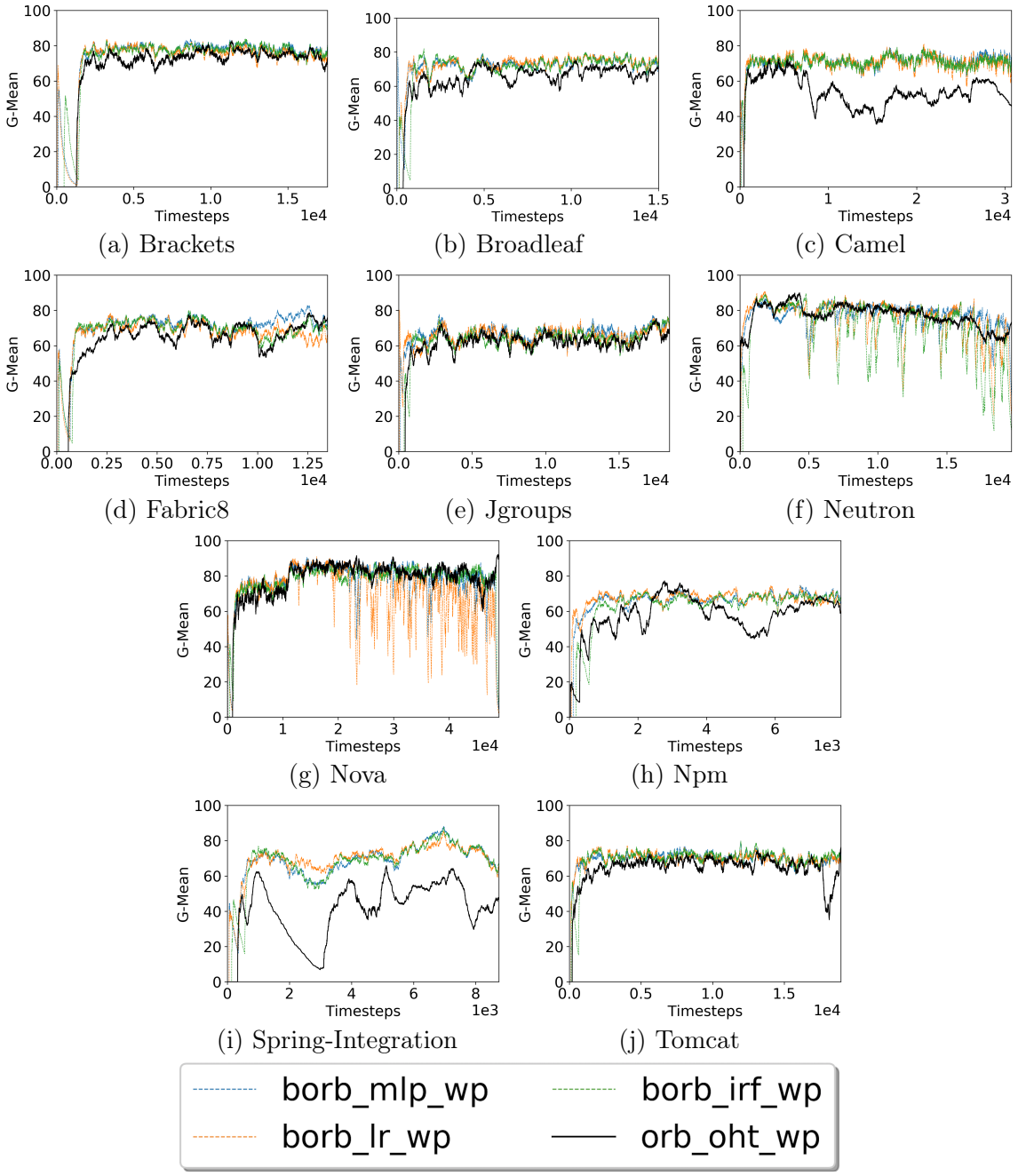


Figure 6.2: G-Mean for all datasets through time for best ranked BORB and ORB approaches with WP data.

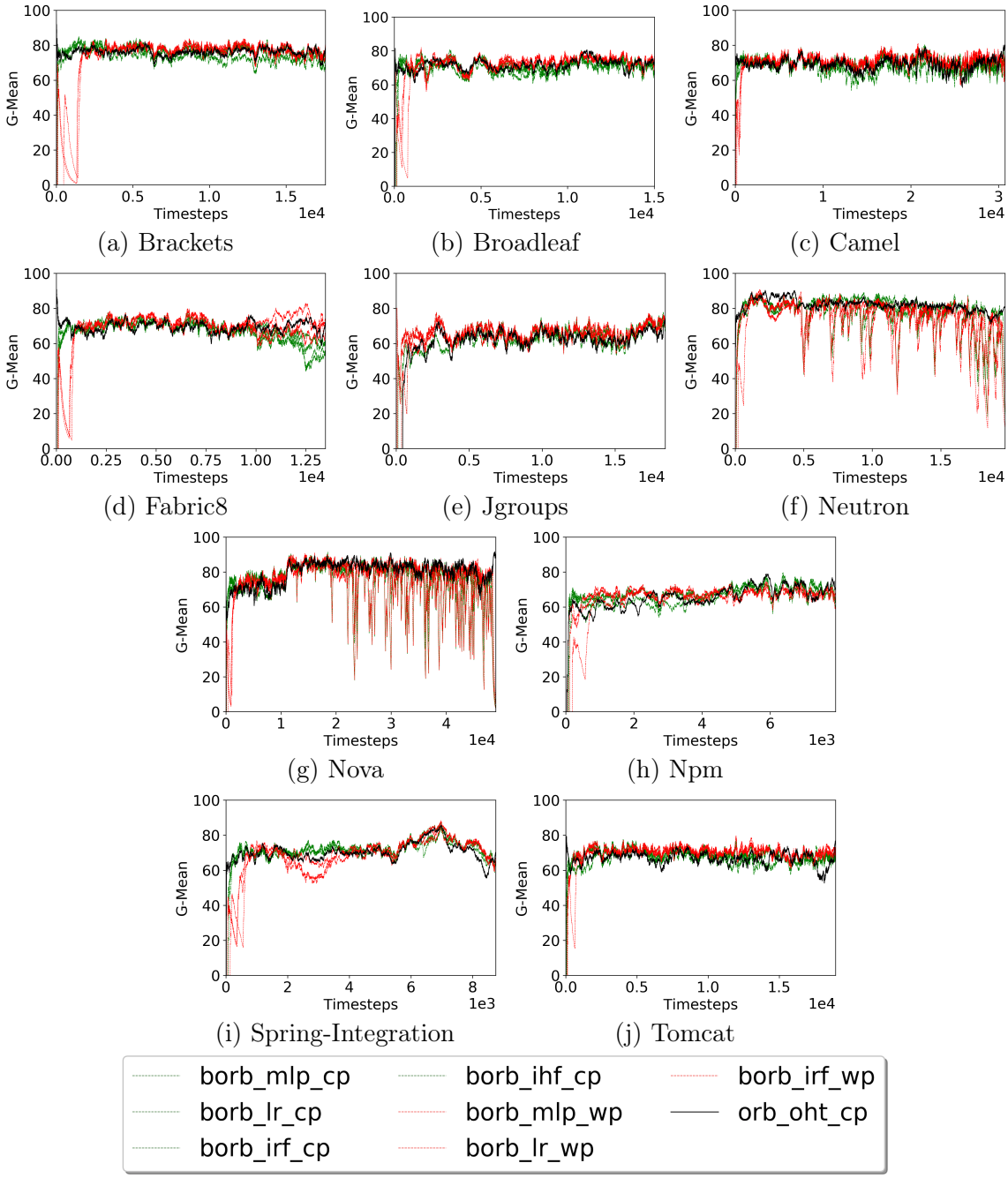


Figure 6.3: G-Mean for all datasets through time for best ranked BORB and ORB approaches with WP and CP data.

6.4.2 RQ2.2: How beneficial is CP data to improve predictive performance of offline models compared to online models in online CP JIT-SDP scenarios?

Previous Chapter 5 investigated the use of CP data for online JIT-SDP and found that, with the use of CP data, online learners are exposed to more data and are able to improve the predictive performance of the JIT-SDP model compared to using only WP data. Chapter 5 also showed that CP data was helpful to maintain stable predictive performance during the periods when the WP models typically suffered sudden performance drops. From section 6.4.1, we found that BORB's offline learners outperformed ORB's online learners with WP data. In particular, offline models iterate over the training data several times, simulating the existence of a larger data set. It is unknown whether incorporating CP data with BORB would further improve predictive performance for offline models. Moreover, no other base learners were explored for online CP JIT-SDP except OHT. It is not known whether CP data would still be useful for JIT-SDP using other online base learners. Hence, it is important to investigate the use of CP data not only for offline models, but also for other online models than OHT.

To answer RQ2.2, we compared 4 approaches – ORB-WP, ORB-CP, BORB-WP and BORB-CP. According to Scott-Knott.A12 test, all BORB-WP except IHF and NB, all BORB-CP except NB, and ORB-OHT-CP ranked the best (Tables 6.1, 6.3 and 6.2). The ranks obtained by BORB-CP were similar to BORB-WP for all base learners except for IHF, where BORB-CP ranked better than BORB-WP. ORB-OHT-CP ranked similar to the best BORB-WP and BORB-CP approaches. Rankings of other base learners with ORB-CP were worse than with BORB-WP and BORB-CP.

These results show that combinations of BORB with most of the base learners (IRF,

LR, MLP, NB) have no difference in the ranking between WP and CPmodels. For IHF, BORB-CP was ranked better than BORB-WP. Even though IHF is classified as an offline approach, it uses online Hoeffding Trees as base learners (Section 2.1). It is possible that using online Hoeffding Trees resulted into weaker BORB-WP models than BORB-CP.

However, OHT greatly benefits from use of CP data, hence, is the most suitable base learner for ORB. Therefore, CP data was typically helpful to improve average G-Mean in online learning, but not usually helpful in offline learning. This observation is in line with the fact that iterating through the training data several times as done in offline learning can work as a simulation of a larger training set. So, the additional data obtained from other projects was not so helpful for offline learning as it was for online learning. The absolute improvement in G-Mean obtained by ORB-CP over ORB-WP was up to 29.3% for Spring-Integration using OHT. When using OHT as base learners, the use of CP data enabled ORB to achieve top ranked average predictive performance, together with the BORB-CP approaches and BORB-IRF-WP, BORB-LR-WP and BORB-MLP-WP.

Fig.6.3 shows plots for top ranked ORB-CP, BORB-CP and BORB-WP approaches. From the plots it is visible that BORB-WP suffered from very low predictive performance in the beginning of the project for most of the datasets (see red lines in Fig.6.3a, Fig.6.3b, Fig.6.3d, Fig.6.3f, Fig.6.3h Fig.6.3i and Fig.6.3j). Performance drops can also be seen in other periods of the project (e.g., see performance drops shown by the red lines in Fig. 6.2f,6.2g, 6.2i, Fig.6.3f,6.3g and 6.3i). Although BORB-CP achieved better predictive performance during the initial period (as shown by the green lines in the figures), their performance for rest of the periods are often lower than that of BORB-WP. Moreover, BORB-CP suffered from sudden performance drops for Fabric8, Neutron and Nova (Fig.6.3d,6.3f,6.3g); particularly, drops for Neutron and Nova were large and frequent, which makes BORB-CP unreliable. ORB-CP never suffered such severe sudden drops in performance, being much more stable than BORB-WP and BORB-CP throughout the projects.

The results above suggest that CP data can be useful for both BORB and ORB during initial period of the project, when WP models struggle to perform well. However, for other periods, CP data could be very useful for ORB, but may not be so helpful for BORB as BORB (both CP and WP) suffered many severe drops in predictive performance for some data sets (Nova and Neutron). Given that ORB-OHT-CP obtained the same ranking as the top BORB-CP and BORB-WP approaches but did not present such drops, they may be more reliable approaches to adopt in practice.

RQ2.2: CP data was not so helpful for the offline learning as it was for the online learning approaches. In most cases, CP data did not lead to better average G-Mean with BORB. However, it enabled ORB-OHT-CP to be among the first ranked approaches according to Scott-Knott-A12. Moreover, it enabled ORB-OHT-CP to achieve more stable G-Mean through time than BORB-WP and BORB-CP. Therefore, ORB-OHT-CP is recommended over BORB in practice.

6.4.3 RQ2.3: How high is the computational cost of offline learning in online scenarios compared to that of online learning models?

An ideal JIT-SDP approach should not be computationally too expensive as such approaches are not suitable to use in practice. Hence, while comparing between offline and online JIT-SDP approaches, it is important to consider computational cost (run time) along with the predictive performance. Typically, offline learners require multiple iterations of the training data leading to higher computational cost compared to online learners. Offline CP approaches could be even more computationally expensive than offline WP approaches as they require retraining with larger amount of data from several projects.

To answer RQ2.3, the analysis of the computational cost of the offline and online models is divided in to two parts. The first part of the analysis focuses on investigating whether offline models have higher computational cost than their online counterparts, by comparing the computational cost of all offline and online models with equivalent base learners. Here, it is important that approaches with equivalent base learners are compared. This enables us to investigate the effect of offline and online learning itself, without being affected by other procedures that are specific to each different type of base learner. The second part of the analysis focuses on comparing the computational cost of the offline and online models that obtained the best predictive performance regardless of base learners. This will enable us to find out which approach would be more feasible to adopt in practice.

6.4.3.1 Comparison of computational cost between offline and online models

Tables 6.5, 6.6, 6.7 and 6.8 show the average computational cost for 30 executions of BORB-WP, ORB-WP, BORB-CP and ORB-CP approaches with different base learners. The tables include the Scott-Knott.A12 ranking obtained by comparing computational cost of all approaches together.

From fig. 6.4a, we can see that, offline WP approaches require larger computational cost than online WP for most base learners. This is also reflected in the results in Tables 6.5 and 6.6, which show that according to Scott-Knott.A12 ranking, when comparing equivalent base learners, ORB-WP with all base learners ranked better (faster) than BORB-WP except for NB for which they ranked similar. It is worth noting that NB is an online learning model, despite being used within the offline BORB approach here. This is likely the reason why BORB-NB-WP ranked similar. Online WP approaches are up to 16.02 times, 14.61 times and 2.46 times faster than offline WP approaches with the base learners LR, MLP and NB (for JGroups, Broadleaf and Neutron), respectively. Fig. 6.4b illustrates computational

Table 6.5: Average run time in seconds and Scott-Knott.BA12 ranking for BORB with WP data

Dataset	BORB-IHF-WP	BORB-IRF-WP	BORB-LR-WP	BORB-MLP-WP	BORB-NB-WP
Brackets	234.91	155.02	706.65	345.05	51.48
Broadleaf	886.19	524.6	231.5	1041.64	78.36
Camel	1816.6	327.34	1505.14	735.96	184.76
Fabric8	1058.21	439.57	513.88	258.26	51.2
JGroups	1533.76	257.92	789.46	386.7	78.07
Neutron	845.57	102.95	859.62	758.87	210.41
Nova	2081.5	791.48	2754.89	1349.57	152.78
Npm	204.59	60.62	265.63	135.82	27.85
Spring-Int	180.75	227.55	306.26	541.11	70.93
Tomcat	1997.68	499.21	592.35	271.4	84.62
Ranking	8	5	7	6	1

Table 6.6: Average run time in seconds and Scott-Knott.BA12 ranking for ORB with WP data

Dataset	ORB-LR-WP	ORB-MLP-WP	ORB-NB-WP	ORB-OHT-WP
Brackets	83.4	88.35	66.27	273.93
Broadleaf	57.35	71.25	64.67	243.39
Camel	169.82	195.31	140.59	534.45
Fabric8	56.49	60.37	39.97	213.41
JGroups	49.26	60.73	39.24	97.04
Neutron	114.5	125.12	85.47	183.77
Nova	299.92	352.14	336.03	916.09
Npm	29.7	34.57	24.4	54.53
Spring-Int	37.4	44.06	30	139.15
Tomcat	69.31	92.63	57.25	191.26
Ranking	1	2	1	4

Table 6.7: Average run time in seconds and Scott-Knott.BA12 ranking for BORB with CP data

Dataset	BORB-IHF-CP	BORB-IRF-CP	BORB-LR-CP	BORB-MLP-CP	BORB-NB-CP
Brackets	2638.92	2063.94	2217.43	3486.99	254.42
Broadleaf	1508.24	1921.99	946.89	3341.19	465.53
Camel	3657.26	3670.84	3855.49	6274.49	910.71
Fabric8	1038.82	1633.08	1417.51	2821.19	558.75
JGroups	1598.91	299.27	1085.37	569.25	135.4
Neutron	3060.56	2788.79	3095.75	2055.51	697.32
Nova	4892.24	6036.44	6355.17	10232.94	1829.93
Npm	1183	909.13	944.43	302.46	330.13
Spring-Int	442.81	836.66	779.62	1257.14	291.07
Tomcat	2842.89	2092.97	2102.43	3396.83	254.74
Ranking	9	9	9	10	6

Table 6.8: Average run time in seconds and Scott-Knott.BA12 ranking for ORB with CP data

Dataset	ORB-LR-CP	ORB-MLP-CP	ORB-NB-CP	ORB-OHT-CP
Brackets	234.23	396.96	153.93	1376.52
Broadleaf	253.9	454.57	159.55	879.41
Camel	399.76	561.27	274.79	1030.35
Fabric8	222.59	380.59	119.84	734.83
JGroups	220.49	379.28	98.19	761.51
Neutron	297.76	483.49	200.55	1528.3
Nova	647.73	845.17	543.88	3185.42
Npm	185.79	323.14	76.14	683.98
Spring-Int	221.59	411.33	96.15	566.4
Tomcat	279.59	442.08	151.25	1665.32
Ranking	4	6	3	8

Standard deviations are shown in brackets. Scott-Knott.BA12 was run for all BORB and ORB-based WP and CP approaches together. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

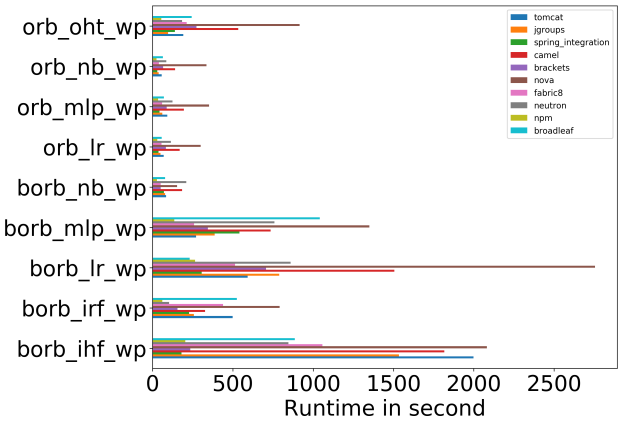
costs of BORB-CP and ORB-CP where it is clearly visible that online CP approaches are faster compared to the offline CP approaches in general. Tables 6.7 and 6.8 show that, according to Scott-Knott.A12 ranking, while comparing equivalent base learners, all ORB-CP approaches ranked better (faster) than BORB-CP approaches. Online CP approaches were up to 10.39 times, 12.10 times and 4.66 times faster than offline CP approaches with the base learners LR, MLP and NB (for Neutron, Nova and Fabric8), respectively.

From the above analysis, it can be seen that, online models are faster than offline models when compared under similar conditions such as equivalent base learners and similar type of data (WP or CP).

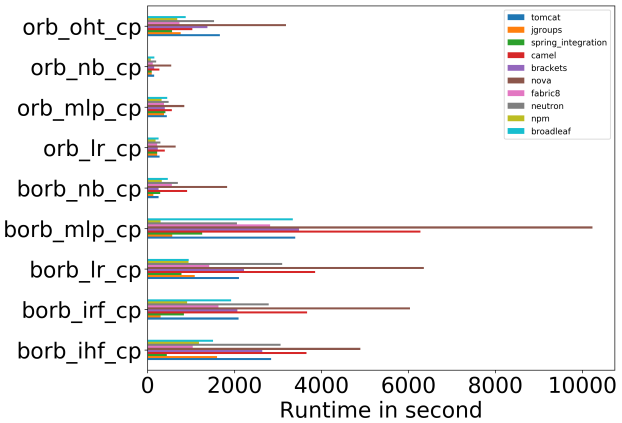
6.4.3.2 Comparison of computational cost between best offline and online models in terms of predictive performance

Ideally, an approach most suitable to be used in practice should provide high and stable predictive performance, and also should not be computationally too expensive. Hence, to discover which approach would be feasible to adopt in practice, we compared computational cost of the offline and online models that achieved best rank in terms of average G-Mean. Such best offline models were BORB-IRF-WP, BORB-LR-WP, BORB-MLP-WP, BORB-IHF-CP, BORB-IRF-CP, BORB-LR-CP, BORB-MLP-CP and best online approach was ORB-OHT-CP (Tables 6.1, 6.3 and 6.4).

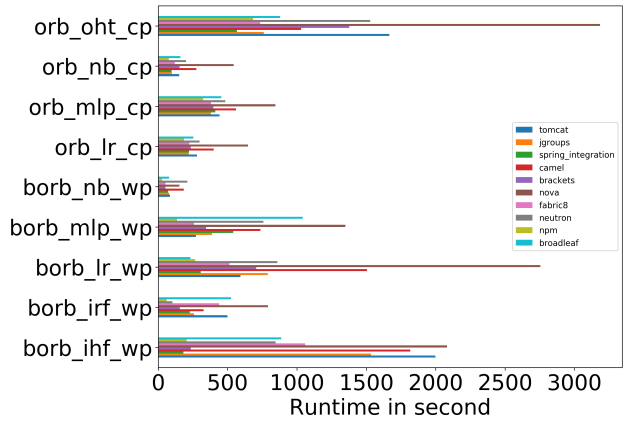
From Tables 6.5, 6.7 and 6.8, according to Scott-Knott.A12 ranking, BORB-IRF-WP ranked the best and BORB-MLP-CP ranked the worst in terms of computational cost. ORB-OHT-CP ranked better than all BORB-CP approaches and worse than all BORB-WP approaches mentioned previously. The higher computational cost of BORB-CP against BORB-WP was expected, given that BORB-CP is trained with more data. Despite not being expected, the higher computational cost of ORB-CP compared to BORB-WP approaches



(a) Runtime for BORB and ORB for WP data



(b) Runtime for BORB and ORB for CP data



(c) Runtime for WP BORB and CP ORB

Figure 6.4: Computational Cost Analysis for BORB and ORB

Table 6.9: Computational Cost Analysis for ORB-OHT-CP

Dataset	ORB-OHT-CP (s)	Duration (year)	Runtime Per Year (s)	Runtime Per Day (s)
Brackets	1376.52	6	229.42	0.63
Broadleaf	879.41	9.01	97.6	0.27
Camel	1030.35	10.7	96.29	0.26
Fabric8	734.83	6.65	110.5	0.3
JGroups	761.51	14.25	53.44	0.15
Neutron	1528.3	6.99	218.64	0.6
Nova	3185.42	7.68	414.77	1.14
Npm	683.98	8.17	83.72	0.23
Spring-Int	566.4	10.18	55.64	0.15
Tomcat	1665.32	11.7	142.34	0.39

(Fig. 6.4c) is understandable. Even though BORB-WP iterates through the WP several times whereas ORB-CP iterates through the CP data only once, the larger number of CP training examples contributed towards the larger computational cost of ORB-CP. BORB-WP approaches were up to 6.13 times faster than ORB-CP approaches (with BORB-MLP-WP and ORB-OHT-CP for Tomcat).

However, from fig. 6.2 we can see that these BORB-WP approaches suffered initial performance drops in almost all of the datasets. Apart from this, they also suffered severe recurrent performance drops in datasets Neutron and Nova (fig. 6.2f and 6.2g). Such performance drops indicate that these BORB-WP approaches are susceptible to concept drifts and may prone to become unstable in certain periods of time. On the contrary, ORB-OHT-CP usually did not suffer from such kind of performance drops, and managed to provide a continuous stable predictive performance through out the time, which makes it a robust approach. Therefore, it is important to check whether the higher computational cost of ORB-OHT-CP is still feasible for adoption in practice.

For that, as ORB-OHT-CP is run over time, we check what the average computational cost per day of this approach is. Column 3 in Table 6.9 shows total duration of each project. The computational cost of ORB-OHT-CP per year and per day are computed and

showed in columns 4 and 5. These values are computed by diving the total computational cost by duration of the project in years and days. As we can see, the computational cost for ORB-OHT-CP is very small (up to 414.77 second per year and up to 1.14 second per day). Therefore, ORB-OHT-CP is a feasible approach to adopt in practice in terms of computational cost and is recommended due to its top average predictive performance and higher stability of predictive performance compared to the other approaches.

RQ2.3: Online WP approaches were up to 16.02 times faster than offline WP approaches, and online CP approaches were up to 12.10 times faster than offline CP approaches. The online approach that achieved most competitive average G-Mean and that achieved the most stable G-Mean over time (ORB-OHT-CP) had a computational cost up to 6.13 slower than the the most competitive offline WP approaches (BORB-IRF-WP, BORB-LR-WP and BORB-MLP-WP). However, this computational cost can still be considered feasible, as the average computational cost was at most 1.14 second per day.

6.5 Threats to Validity

Internal validity: poor hyperparameter choices can affect the predictive performance of machine learning models. To mitigate this threat, random search was performed on a set of possible values for the hyperparameters of each approach and base learner based on an initial portion of the data stream. Verification latency was also taken in to account for all approaches respecting the chronology of the software changes. Besides, each of the approaches with each dataset was executed 30 times to mitigate the threats to internal validity.

Construct validity: the evaluation metrics used in this study were G-Mean, Recall0 and Recall1. These are widely used metrics appropriate for class imbalance learning [52] and

were computed prequentially considering a fading factor recommended in [50], that allows the model to give more importance to the most recent data.

Statistical conclusion validity: Scott-Knott test with non-parametric bootstrap sampling and A12 effect size were used to address conclusion validity. This avoids concluding that approaches have differences even though the difference in the predictive performance is very small which is presented by insignificant effect size.

External validity: this study used 10 open source projects from GitHub repositories. These projects are based on different programming languages and have different characteristics (e.g. number of commits per day, starting date, number of modified files). The results obtained by this study may not generalise for other type of projects (e.g. proprietary projects) and for problem domain other than JIT-SDP. The conclusions about offline learning drawn by this study are based on the proposed approach BORB. Other offline learning approaches may lead to different conclusions.

6.6 Summary

In this chapter, we investigate the influence of base learners on JIT-SDP, using open source data. The BORB approach proposed in [216] is used to adapt offline base learners to operate in a realistic online JIT-SDP scenario. Offline learning helped to improve average G-Mean in the online WP JIT-SDP scenario. Offline models performed better than online models for WP data (RQ1) achieving 27.97% absolute improvement in terms of G-Mean. MLP, LR and IRF were the best performing offline base learners using WP data (RQ1). Similar to online WP, offline WP approaches suffered from low G-Mean in the initial period of most projects and also experienced some frequent and severe drops in G-Mean over time in Neutron and Nova.

Our study found that, although CP data was very useful for ORB to improve predictive performance and to eliminate sudden drops in G-Mean, it was not so helpful for BORB. According to Scott-Knott ranking, both BORB-WP and BORB-CP approaches performed similarly for all base learners except for IHF, which is an offline ensemble based on online decision trees (RQ2). In terms of performance drops, CP data helped BORB mostly to reduce initial period of poor G-Means, but could not help to reduce some other drops, such as frequent and severe drops observed in Neutron and Nova. ORB-OHT-CP did not present such severe and frequent drops. Hence, ORB-OHT-CP could be preferred as an approach to obtain a stable and high predictive performance, as it is also among the best ranked approaches.

The analysis of computational cost showed that online WP and CP approaches are up to 16.02 and up to 12.10 time faster compared to corresponding offline models (RQ3). The offline WP approach (BORB-MLP-WP) achieving the best predictive performance was up to 6.13 times faster than the online CP approach (ORB-OHT-CP) achieving the best predictive performance. However, the average computational cost per day of ORB-OHT-CP is not very high. So, the approach is feasible for adoption in practice.

From our study, we have seen that OHT still is the best base learner for online learning as no other base learners were able to achieve a competitive predictive performance. But for offline models, there were several base learners that performed well, which makes BORB less rigid and provides multiple options (in terms of base learners) to choose from. offline models performed well on average with both WP and CP data, whereas online models performed well only with CP data, leading to a higher computational cost.

Chapter Seven

Online Hyper-Parameter Tuning in JIT-SDP

This chapter is based on the fourth publication (to be submitted to IEEE Transactions on Software Engineering) listed in Chapter 1.

7.1 Introduction

Predictive models built using machine learning algorithms usually involve a number of hyper-parameters. Unlike 'model parameters' which are learned during the training of the model, 'hyper-parameters' need to be carefully set before running the algorithm. Different options for setting hyper-parameters include using default values from existing studies, manually selecting values based on expert's experience, and tuning hyper-parameters based on existing data to select good values that will lead to better predictive performance. Hyper-parameters are critical for predictive models as different hyper-parameter choices can lead to different predictive performances. Predictive performance of offline machine learning models (including offline SDP [121],[219]) are known to be affected by hyper-parameters. Hence, tuning

hyper-parameters is important for choosing the optimal values.

Most of the existing hyper-parameter tuning approaches are for offline learning. They require a pre-existing data set to search for a good hyper-parameter configuration. Once the hyper-parameter configuration is chosen, it is fixed and never updated anymore. Tuning hyper-parameters for online learning is more difficult than offline learning, as the best hyper-parameter values may change over time. Even though such changes do not happen for all data stream learning problems [194], depending on the problem, an initially chosen hyper-parameter configuration may not be as effective in the later periods of the data stream.

However, effect of hyper-parameters in online JIT-SDP is unknown. If the best hyper-parameter choice changes over time in online JIT-SDP models, then an online hyper-parameter tuning approach would be necessary to improve the predictive performance of online JIT-SDP. Moreover, online JIT-SDP models are known to suffer performance drops because of changes (*concept drifts*) that may happen due to change in the underlying data generating process [19]. An online tuning approach may also help the model to maintain stable performance during concept drift by choosing suitable hyper-parameter values during that period. However, no existing study has investigated online hyper-parameter tuning for online JIT-SDP.

This chapter aims to investigate the impact of hyper-parameter choices on online JIT-SDP models over the course of software projects and whether an online hyper-parameter tuning approach would improve the predictive performance of online JIT-SDP models. It answers the following research questions:

RQ3.1 What is the impact of hyper-parameters on the predictive performance of online JIT-SDP?

RQ3.2 How to automatically tune hyper-parameters for JIT-SDP in an online manner? How

well does this approach perform in terms of suggesting appropriate hyper-parameters to improve JIT-SDP's predictive performance?

RQ3.3 How high is the computational cost of online hyper-parameter tuning approach? In particular, how feasible is the new approach considering the trade-off between predictive performance improvement and runtime?

We consider the online cross-project JIT-SDP strategy All-in-One (AIO) over Filtering approach proposed in Chapter 5 to address the aforementioned RQs because both approaches demonstrated comparable predictive performance. The computational cost of the Filtering approach, however, was greater than the AIO. As a result, the Filtering technique would incur additional computing costs if a hyper-parameter tuning was included. Also, this chapter will focus on the CP scenario, as it was shown to lead to better predictive performance than the WP scenario in online JIT-SDP in Chapter 5. We compare the predictive performances throughout the course of the software projects between AIO using different combination of hyper-parameters and AIO using a hyper-parameter combination selected using Grid Search based on an initial portion of the data stream. The results show that the best hyper-parameter configuration indeed changes over time, causing the initially chosen hyper-parameter combination to become ineffective (RQ3.1). Then, we propose a novel online hyper-parameter tuning approach for online JIT-SDP, that explores the hyper-parameter space and periodically updates the JIT-SDP model with appropriate hyper-parameters over time. Our experiments based on 10 open source software repositories described in Chapter 4 show that the proposed online hyper-parameter tuning approach significantly improves the predictive performance of online JIT-SDP compared to Grid Search (RQ3.2). Based on our computational cost analysis, we found that although our proposed approach has a higher computational cost compared to Grid search, the computational cost is still minimal (up to 33.34 seconds per day) making it suitable for adaption in practice (RQ3.3).

The overall contributions of this chapter are the following:

- We provide the first investigation of the impact of hyper-parameters on online JIT-SDP, showing that the best hyper-parameter choice changes over time.
- We show how to automatically tune hyper-parameters in an online manner and propose a novel online hyper-parameter tuning approach for online JIT-SDP.
- We show that online hyper-parameter tuning can improve predictive performance of online JIT-SDP.
- We show that the proposed online hyper-parameter tuning approach is feasible regarding the computational cost.

This chapter is further organized as follows. Section 7.2 introduces the proposed online hyper-parameter tuning approach. Section 7.3 explains the experimental setup for answering the RQs. Section 7.4 explains the results of the experiments. Section 7.5 explains the threats to validity. Section 7.6 presents the summary.

7.2 Proposed Approach

This section proposes a novel online hyper-parameter tuning approach – Hyperon, for tuning online JIT-SDP models.

7.2.1 Hyperon

Hyperon is an online meta-heuristic algorithm that searches for good hyper-parameter configurations over time through the data stream. It maintains a pool of JIT-SDP models trained based on different

Algorithm 8 Hyperon Algorithm

Input: S_{train} = training data stream b = test project index, w = waiting period, h = default hyper-parameter combination, $M : \{m1, m2, ..., m_n\}$ = Pool of models, **Algorithm Hyperparameters:** $minAge$ = minimum age of the model to be old enough, $windowSize$ = max size of the most recent instance window, $ea_interval$ = interval to run ADE-ACM, n = max number of models **Variables:** $InstWindow$ = window of most recent instances, $ErrorWindow$ = window containing the prediction errors of classifiers on $InstWindow$, $fitlist$ = hyperparameters of models in M with their fitness.

1: Generate set of random hyper-parameter combination and add default hyper-parameter combinations $H : \{h1, h2, ..., h_n\} \cup \{h\}$
2: Initialise model pool M consisting of n models $\{m_1, m_2, ..., m_n\}$ using n hyperparameter combinations from H
3: **for** each training example $d_t = (x_t, y_t)$ in S_{train} **do**
4: $\hat{Y}$ = List of predictions on x_t by each model m in M
5: **for** each model m_p in M **do**
6: Update $ErrorWindow[m_p]$ with $(\hat{Y}[m_p]! = y_t)$
7: $fitlist[m_p]$=$fitnessCalc(ErrorWindow[m_p])$
8: Train model m_p with d^t
9: **end for**
10: **if** t % $ea_interval == 0$ **then**
11: h', m_pnew, $fit_{h'}$, $errors_{m_pnew}$ = ADE-ACM($InstWindow$)
12: **if** $fit_{h'}$ better than the old enough worst model m_{wrst} in the pool M **then**
13: Delete m_pwrst from M, $ErrorWindow$ and $hpflist$
14: Add m_pnew, $fit_{h'}$ and $errors_{m_pnew}$ to M, $fitlist$ and $ErrorWindow$, respectively
15: **end if**
16: **end if**
17: Yield the best fitness model as the one to be used for testing purposes
18: **end for**
19: **function** MODELEVAL($InstWindow, m_pnew$)
20: $offspringError$ = []
21: **for** each training example $d_j = (x_j, y_j)$ in $InstWindow$ **do**
22: $\hat{y}_j = m_pnew$.predict(x_j)
23: $offspringError[m_pnew] = (\hat{y_j}! = y_j)$
24: m_pnew.train(d_j)
25: **end for**
26: fitness of $m_pnew = fitnessCalc(offspringError[m_pnew])$
27: **end function**
28: **function** FITNESSCALC($ErrorWindow[m_p]$,$fitlist[m_p]$)
29: **for** each err in $ErrorWindow[m_p]$ **do**
30: $totalError$ += err
31: **end for**
32: $totalError$ /= $ErrorWindow$
33: **end function**

Algorithm 9 ADE-ACM Algorithm

Input: current population P, dimension D, learning rate clr=0.01, mutation factor F=0.5, location parameter uF=0.5, scale parameter σF=0.1, correlation matrix CM

1: Randomly select a target vector x_i from NP
2: Create a mutant vector: $v_i = x_i + F * (x_{best} - x_i) + F * (x_{r1} - x_{r2})$
3: $F = randc(\mu F, \sigma F)$
4: $S =$ Get the sample correlation matrix from current population
5: Select a random coordinate m and random threshold c_{thr}
6: Get set of correlated variables from the correlation matrix $\{I_m\} = \forall j :\mid c_{mj} \mid > c_{thr}$ for each j in D
7: Create a trial vector using Eq 7.4
8: **if** (trial_vector fitness < target_vector fitness) **then**
9: Compute adaptive correlation matrix $CM = (1 - clr) * CM + clr * S$
10: Update $\mu F = (1 - clr)\mu F + clr * L2(F)$ ▷ //L2(F) is a contraharmonic mean of a set of all scale factors associated with the current population
11: return trial_vector as new hyperparameter combination
12: **else**
13: return null
14: **end if**

hyper-parameter combinations. This pool is created and maintained based on an evolutionary algorithm called Asynchronous Differential Evolution with Adaptive Correlation Matrix (ADE-ACM) [220]. The iterations of this algorithm are intertwined with the learning of incoming examples, such that the evolutionary algorithm is run in an online manner together with the online training of the JIT-SDP models. The models in the pool are evaluated based on a sliding window containing the most recent labelled examples. Such evaluation is used both as the fitness function of the evolutionary algorithm and to determine the most promising hyper-parameter configuration, which is then used for predicting new incoming software changes.

The ADE-ACM [220] algorithm is based on Asynchronous Differential Evolution (ADE) [221], which is a variant of Differential Evolution (DE) [222]. ADE-ACM combines ADE with a crossover based on adaptive correlation matrix inspired by another evolutionary algorithm CMA-ES [150]. Classical DE is a generation-based evolution strategy where evolutionary operations are performed on a large portion of the population to generate offspring solutions. Typically, the

number of generated offspring is the same as the population size. Offspring solutions then replace old solutions of the population. We used steady-state ADE-ACM, which is a variant of DE where evolutionary operations are performed on a (few) selected member(s) of the population instead of a large portion of the population. Typically, only one new solution can be inserted into the population, allowing the new better solution to take part in the evolution process without any time lag [220]. Vavak et al. [223] showed that the steady version outperforms the generational genetic algorithm regarding the convergence speed for an online/non-stationary environments. For our online JIT-SDP scenario, searching for a new combination must be done over time, requiring the optimisation algorithm to run multiple times. Hence, due to the faster convergence ability, a steady-state optimisation algorithm would be more suitable for the online JIT-SDP scenario. ADE-ACM showed faster convergence rates and can efficiently solve many global optimization problems (separable, partly-separable, and non-separable) because of its unique crossover mechanism. Also, ADE-ACM has a simple structure, and it is quasi-parameter-free from the user's perspective [224]. Such advantages make ADE-ACM a suitable option to adopt in Hyperon to search for new hyper-parameter combinations.

Hyperon considers a CP scenario where we are interested in predicting software changes of one project, while software changes from multiple projects could be used for training. However, it is possible to set the number of projects to be 1, corresponding to the single project being predicted. Therefore, this approach could be used both for CP and WP learning.

Algorithm 8 shows the pseudocode of the Hyperon approach. Hyperon generates an initial set of hyper-parameter combinations H (line 1). One of these combinations should be the default combination typically used with the underlying machine learning algorithm. In the case of the AIO algorithm (proposed in Chapter 5) used in our experiments, this is the default combination used for the ORB approach [18, 29, 210, 19]. The rest of the combinations are randomly sampled from a list of combinations investigated in [18] other than the default combination. A pool M of n models is created, where each model is associated with each hyper-parameter combination in H (line 2).

When a training example d^t (defined in Chapter 1) from the training data stream S_{train}

arrives, each model m_p in the pool M makes a prediction, which is stored in a list $\hat{Y}$ (line 4). For each model m_p in M, Hyperon maintains an error window $ErrorWindow$ which is a window containing the mistakes of classifiers on most recent $windowSize$ instances of that model (for each incorrect prediction by a model, we assign 1 to $ErrorWindow$ of that model for that instance), and a list $fitlist$ to keep track of the fitness of that model. Fitness is the rate of mistakes in the error window for a model, which is computed by dividing the total sum of mistakes in the $ErrorWindow$ for a model by the error window size (line 28). Next, $ErrorWindow$ and $fitlist$ are updated for each model in M (line 6-7). Then, each model m_p in M is trained with d^t (line 8).

After that, Hyperon checks whether it has seen a certain number of training examples, which is denoted by $EA_Interval$ (line 10). At every $EA_Interval$, ADE-ACM is run to create a new hyper-parameter combination h' (line 11). As part of the ADE-ACM algorithm, the fitness of h' is evaluated using the $windowSize$ most recent training examples stored in the $InstWindow$, as shown in the function $ModelEval$. If ADE-ACM produces a hyperparameter configuration with better fitness than its parent configuration, it returns the new hyperparameter combination h' and its associated model m_pnew, fitness value $fit_{h'}$ and errors $errors_{m_pnew}$ to Hyperon. Then, Hyperon selects the model with the worst fitness m_{wrst} from M, and compares it with $fit_{h'}$ (line 12). If $fit_{h'}$ is better than fitness of m_{wrst}, then m_{wrst} is deleted from M (line 13). Next, the new model m_pnew and its corresponding $fit_{h'}$ and errors $errors_{m_pnew}$ are added to M, $fitlist$ and $ErrorWindow$ (line 14).

7.2.2 ADE-ACM

Our proposed approach Hyperon uses an evolutionary algorithm called ADE-ACM [220] to generate new hyper-parameter combinations. ADE-ACM iteratively improves a candidate solution by applying evolutionary operations of mutation, crossover and selection over population members [220]. In the case of Hyperon, each candidate solution $\vec{x}_i$ is a vector whose positions correspond to each of the hyper-parameters of the underlying machine learning algorithm being tuned.

The pseudocode of ADE-ACM is shown in algorithm 9. ADE-ACM maintains a population P of vectors $\vec{x}_i$. In the case of Hyperon, we initialise this population with the hyperparameter combinations listed in line 1 of Algorithm 8. Population P from Hyperon is passed as an input to the ADE-ACM algorithm. A "target" vector $\vec{x}_i$ is selected randomly from the population. Mutation operation is applied on the target vector to create a mutant vector (line 1). Then a mutant vector (also known as donor vector) is created in line 2, which is used to promote diversity, resulting in exploration of novel areas of the search space. We have used the following mutation scheme proposed by Zhang et al. [225] to create a mutant vector:

$$\vec{v}_i = \vec{x}_i + F * (\vec{x}_{best} - \vec{x}_i) + F * (\vec{x}_{r1} - \vec{x}_{r2}). \tag{7.1}$$

Here, F is the mutation factor, $\vec{x}_i$ is the target vector, $\vec{x}_{best}$ is the current best vector in the population, $\vec{x}_{r1}$ and $\vec{x}_{r2}$ are two randomly selected vectors different from x_i and x_{best}. F is sampled from a Cauchy distribution as below using a location parameter $\mu \cdot F$ and a scale parameter $\sigma \cdot F$ (line 3):

$$F = randc(\mu F, \sigma F). \tag{7.2}$$

This mutation operator moves the vector $\vec{x}_i$ in the direction of the best vector to encourage improvement in fitness and in another random direction to encourage diversity.

Instead of using a crossover rate, ADE-ACM modifies the crossover operator to take into account pairwise dependencies between the variables. The current population is used to calculate a sample correlation matrix. The approximation of the correlation matrix is cumulatively adapted through successful evolutionary steps. The advantage of covariance matrix adaptation is that it can tackle ill-conditioned and non-separable problems. For crossover, first, we get the sample correlation matrix from the current population (line 4). By analysing all elements of the adaptive correlation matrix, the ADE-ACM algorithm identifies correlated variables. The set of correlated variables defines a subspace in the search domain. For a fixed index m, the algorithm identifies correlated variables (line 5-6):

$$\{I_m\} = \forall j :\mid c_{mj} \mid > c_{thr}, \tag{7.3}$$

where c_{thr} is a randomly selected threshold between 0 and 1. Through the modified crossover operation, all components of mutant vector v_i whose indices are in $\{I_m\}$ are moved to a trial vector u_i, and other components are taken from the target vector x_i (line 7) as shown below:

$$u_{i,j} = \begin{cases} v_{i,j} & \text{if } j\epsilon\{I_m\} \\ x_{i,j} & \text{otherwise.} \end{cases} \tag{7.4}$$

This crossover allows the algorithm to reduce the dimension of the problem to the dimension of the subspace of the correlated variables. Performance of the standard DE algorithm heavily depends on the parameter crossover rate C_r. ADE-ACM crossover mechanism eliminates the need of C_r. Next, we compare the fitness of the trial vector and the target vector. We get the fitness of the target vector from $fitlist$ from Hyperon, that contains fitness values of solutions in the current population. The fitness of the trial vector is computed over $InstWIndow$ using the evaluation function in line 19 of the Hyperon algorithm. If the trial vectors' fitness is better than the fitness of the target vector, CM and μF are updated, and the trial vector is returned as the new hyper-parameter combination (line 8-14).

Zhang et al. [225] investigated various DE variations and found that automatic adaptation of the control parameters can increase the algorithm's robustness by removing the need for prior user knowledge of the relation between the parameter settings and the characteristics of optimization problems. It allows the algorithm to be applied to various optimization problems without trial and error.

7.3 Experimental Setup

This section explains the experimental setup for answering the RQs introduced in Section 7.1.

7.3.1 Compared Approaches

This study is based on the state-of-the-art online CP JIT-SDP approach All-in-One Oversampling Rate Boosting (AIO-ORB) proposed in Chapter 5 of this thesis, which has achieved competitive results in online JIT-SDP scenarios. In particular, the use of CP data by this approach led to significant improvements over online WP JIT-SDP. However, this approach relies on a number of hyper-parameters. For RQ3.1, to investigate the impact of hyper-parameters over time in online JIT-SDP, 243 combinations of 5 AIO-ORB hyper-parameters ma_window_size, th, l_0, l_1 and m [18] are considered. Five independent runs of these combinations are executed with AIO-ORB for predicting each of the 10 projects described in Chapter 4. The hyper-parameter combination that achieved the best average predictive performance over an initial portion of the data stream (3000 commits) using AIO-ORB is compared with the predictive performances of rest of the 242 hyper-parameter combinations over time. This enables us to check whether the best hyper-parameter combination varies over time.

To answer RQ3.2, the predictive performance achieved through our proposed Hyperon is compared with (1) a Grid Search approach where the best hyper-parameter combination is selected based on the initial 3000 commits, and (2) the upper bound of the predictive performance achievable by Grid Search ("Upperbound"). Approach (1) has been chosen because it is the hyper-parameter tuning approach used in all previous online JIT-SDP studies [18, 29, 210, 19]. It is applicable in practice when the JIT-SDP models start being used for predictions after the first 3000 commits become training examples in a project. Approach (2) is a hypothetical approach used to represent the best possible results that could be achievable by running a Grid Search with the above mentioned 243 hyper-parameter configurations over time, throughout the data stream. For each timestep, we select the hyper-parameter combination that has the maximum average G-Mean among the set of 243 combinations explored in grid search. In practice, such selection is not possible, because the best average G-mean is computed using the true label of the software change being predicted, which would be unavailable in practice. However, if Hyperon is able to outperform this upperbound, this would represent a very positive result for Hyperon. A dummy classifier that predicts clean or defect-

inducing uniformly at random was also used to support the analysis. Thirty runs of Grid Search and Hyperon approach and five runs of Upperbound were executed.

To answer RQ3.3, computational cost of Hyperon is compared with Grid search, which will provide deeper understanding on the feasibility of the proposed approach. Hyperon algorithm has 4 hyper-parameters, which are tuned by executing 5 runs of the algorithm with different combinations, using 3 open source datasets (Django, Rails and Vscode) described in Table **??** of Chapter 4 which are different than the 10 open source datasets described in Table 4.2 of Chapter 4 used for the main experiments. Being a hyper-parameter tuning approach, Hyperon is only a valid approach if it can itself work well using default hyper-parameter values. Otherwise, the need for tuning the hyper-parameters of JIT-SDP models is replaced by the need for tuning the hyper-parameters of Hyperon. Therefore, these 3 datasets were used to decide what hyper-parameter values should be used as default values for Hyperon.

7.3.2 Predictive Performance

We define a time step as a sequential number indicating the order of commits of the project being predicted, according to their author timestamp. Each commit requires a WP software change to be predicted as defect-inducing or clean. Therefore, to evaluate predictive performance, we investigate how well a classifier can predict the WP commit received at each time step. At this time step, the AIO-ORB model used for prediction is the model trained with all CP+WP labelled training examples that became available *prior* to the Unix timestamp corresponding to this commit. This ensures that the most up-to-date model that fully respects chronology is used, corresponding to a model that could be adopted in practice. The only exception for respecting chronology is the one of the upper bound approach mentioned in Section 7.3.1, which is used to represent an ideal Grid Search tuning approach.

We adopt the Geometric Mean (G-Mean) of Recall0 and Recall1 as measures of predictive performance, where Recall0 is the recall on the clean class and Recall1 is the recall on the defect-

inducing class. More details can be found in Chapter 2. These metrics were chosen because they are the most recently recommended for online class imbalance learning [52, 53, 54].

7.3.3 ORB, Grid Search and Hyperon Hyper-parameters

The proposed approach Hyperon tunes 5 ORB hyper-parameters in an online manner. Initial hyper-parameter combinations for ORB are chosen from the following set of values: $ma_window_size = \{50, 100, 200\}$, $th = \{0.3, 0.4, 0.5\}$, $l0 = \{5, 10, 15\}$, $l1 = \{6, 12, 18\}$ and $m = \{1.5, 3, 5\}$. For initialisation of the pool in Hyperon, n number of combinations are sampled from these set of values. These combinations are used for Grid Search as well. The grid search procedure is a technique that systematically evaluates multiple hyperparameter combinations to identify the best-performing set of hyperparameters among a pre-defined set of values. As such, it does not investigate values beyond the borders of the pre-defined set of values. The hyperparameter combinations for ORB were selected from the following values: $ma_window_size = \{50, 100, 200\}$, $th = \{0.3, 0.4, 0.5\}$, $l0 = \{5, 10, 15\}$, $l1 = \{6, 12, 18\}$ and $m = \{1.5, 3, 5\}$. The grid search investigated all combinations of such hyperparameter values. These values were recommended by Cabral et al. [18] based on preliminary experiments that they conducted to check the range of values that usually lead to considerable changes in performance in the ORB approach. Since there are five hyperparameters with three sets of values each, the hyperparameter subspace consisted of 243 combinations. The predictive performance of ORB for each of the 243 combinations was evaluated using the initial portion of the data consisting of the first 3000 commits in terms of the performance metric G-Mean. Based on the evaluation results, the hyperparameter combination that achieved the best G-Mean was selected to be used with the ORB algorithm.

Apart from the ORB hyper-parameters, Hyperon has 4 hyper-parameters of its own which are tuned only once before the beginning of the algorithm run, by executing 5 runs with 144 different combinations from the following set of values $minAge = \{50, 500, 1000, 3000\}$, $windowSize = \{50, 500, 1000, 3000\}$, $EA_Interval = \{1000, 3000, 5000\}$ and $ns = \{20, 50, 70\}$ on each of the 3 datasets described in Chapter 4 using 7000 initial commits. We assign a rank for each combination

for each dataset based on average G-Mean of 5 runs. The combination with best average rank in all 3 datasets is selected, which is $\{500; 50; 3000; 70\}$. Unlike AIO-ORB's parameters, once this hyper-parameter combination is set, it does not need to be changed over time.

7.3.4 Statistical Tests and Effect Size

The predictive performances obtained using Hyperon will be compared with Grid Search across datasets using the Scott-Knott procedure [56], which ranks the models and separates them into clusters. This test is used to select the best subgroup among different models. Non-parametric bootstrap sampling is used to make the test non-parametric, as recommended by Menzies et al. [57]. As explained by Demsar [58], non-parametric tests are adequate for comparison across datasets. A12 effect size [59] to rule out any small differences in performance. More details can be found in Chapter 2.

We also report the A12 effect sizes against the Hyperon approach for each dataset individually to support the analysis. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Hyperon approach.

7.3.5 Labelling Procedure of the Training Examples

Our approaches fully respect chronology and never use any future CP/WP training examples. As in the other chapters of this thesis, training examples are generated using the online procedure recommended by Cabral et al. [51] to take verification latency into account for all approaches studied in this chapter. In particular, a software change becomes a training example either when a defect is found to be associated to it, or once a pre-defined waiting period of w days has passed, whichever is earlier. This waiting period represents the amount of time that it takes for one to be confident that the software change in question is clean. In other words, if no defect is found to be

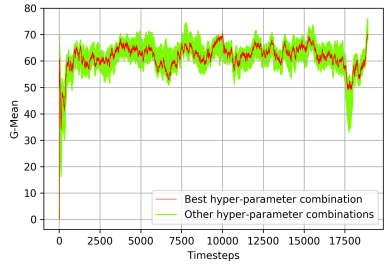
associated to the software change during the waiting period, a training example of the clean class is created to represent this software change. Otherwise, a training example of the defect-inducing class is created immediately after the defect is found. If, after the waiting period, a defect is found to be associated to a change that was previously considered clean, a new defect-inducing training example is created for it. Training examples are used to update the classifier as soon as they are created [210].

7.4 Experimental Results and Discussion

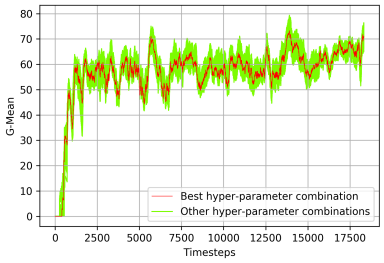
7.4.1 RQ3.1: What is the impact of hyper-parameters on the performance of online JIT-SDP?

For all of the existing online JIT-SDP studies [18, 29, 210, 19], Grid Search has been used to select a hyper-parameter combination based on initial portion of the data stream. These studies did not investigate whether the initially chosen combination can provide better predictive performance throughout time, or whether there could be some periods of time when other combinations would become more suitable. In this section, we investigate whether the best combination changes from the initially chosen one over time, impacting the predictive performance of the online JIT-SDP model over time (RQ3.1).

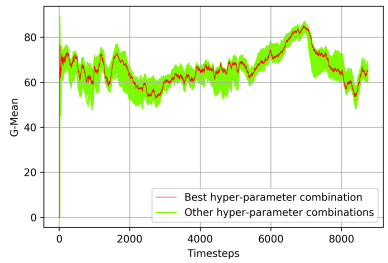
Figure 7.1 shows the average G-Means obtained by the 243 hyper-parameter combinations listed in Sections 7.3.1 and 7.3.3 for all time steps of 10 datasets. G-Mean over time using the combination selected by Grid Search based on the initial portion of the data stream is shown in red and G-Means obtained by the other 242 combinations are shown in green colour. From the plots, it is clear that, for all datasets, the hyper-parameter combination selected by Grid Search based on the initial portion did not always provide the best G-Mean throughout time, and other combinations (in green colour) frequently achieved better G-Mean.



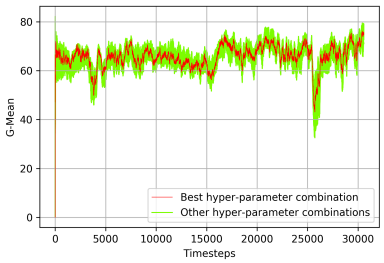
(a) Tomcat



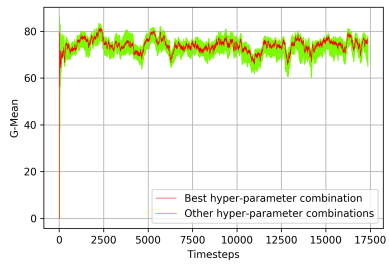
(b) JGroups



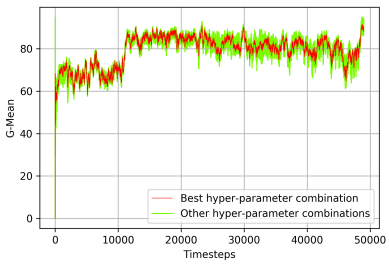
(c) Spring-integration



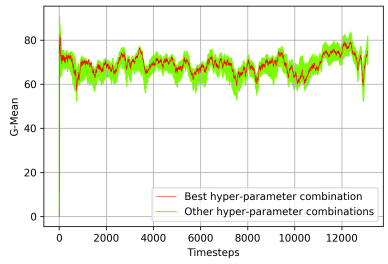
(d) Camel



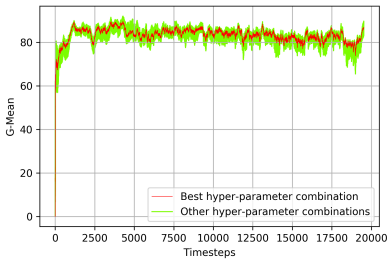
(e) Brackets



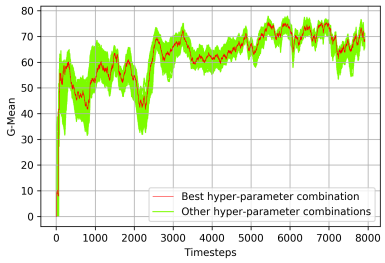
(f) Nova



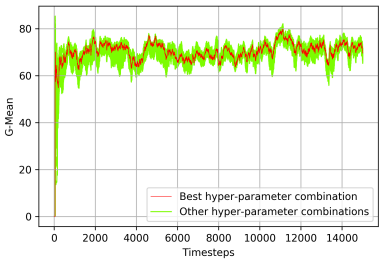
(g) Fabric8



(h) Neutron



(i) Npm



(j) BroadleafCommerce

Figure 7.1: G-Mean for all datasets through time for best hyper-parameter combination with other hyper-parameter combinations based on initial 3000 commits.

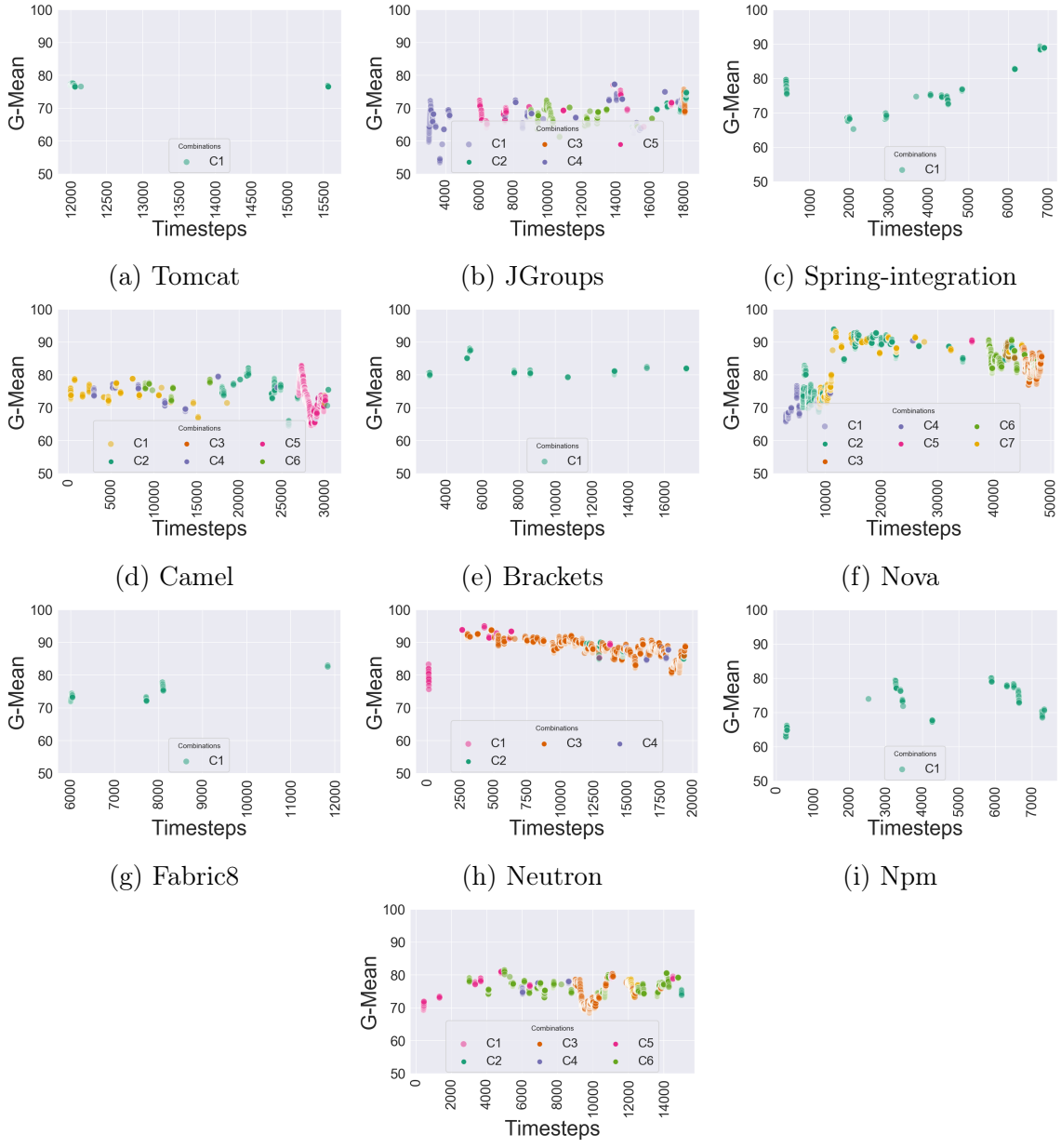
For some datasets, there are several periods where the G-Means obtained by grid-search on the initial portion were quite low. For Camel during 7000 to 16000 timesteps (fig. 7.1d), JGroups during 8000 to 13000 timesteps (fig. 7.1b), for Npm during 500 to 1500 timesteps (fig. 7.1i), for Spring-Integration during 2000 to 3800 timesteps (fig. 7.1c) and for Tomcat during 4500 to 8000 timesteps (fig. 7.1a), we can see that the initially chosen combination by Grid Search obtained considerably lower G-Mean than other configurations.

From the plots, it is visible that choice of the best hyper-parameter combination can change over time causing the model to obtain lower predictive performance when using the initially chosen combination. Hence, to avoid such low predictive performance, it is important to explore other hyper-parameter combinations throughout time and not just in the beginning of the project. To explore different hyper-parameter combinations over time for JIT-SDP, a hyper-parameter tuning approach is required that is able to operate in an online manner. A suitable online hyper-parameter tuning approach, which is particularly designed for JIT-SDP, would be very helpful to improve its predictive performance.

RQ3.1:The choice of the best hyper-parameter combination in online JIT-SDP can change over time causing the model to obtain lower predictive performance when using the initially chosen combination.

7.4.2 RQ3.2: How to automatically tune hyper-parameters for JITSDP in an online manner? How well does this approach perform in terms of suggesting appropriate hyper-parameters to improve JIT-SDP's predictive performance?

We proposed a novel online hyper-parameter tuning approach for JIT-SDP called Hyperon in Section 7.2. Hyperon is able to generate new hyper-parameter combinations over time using an evolutionary



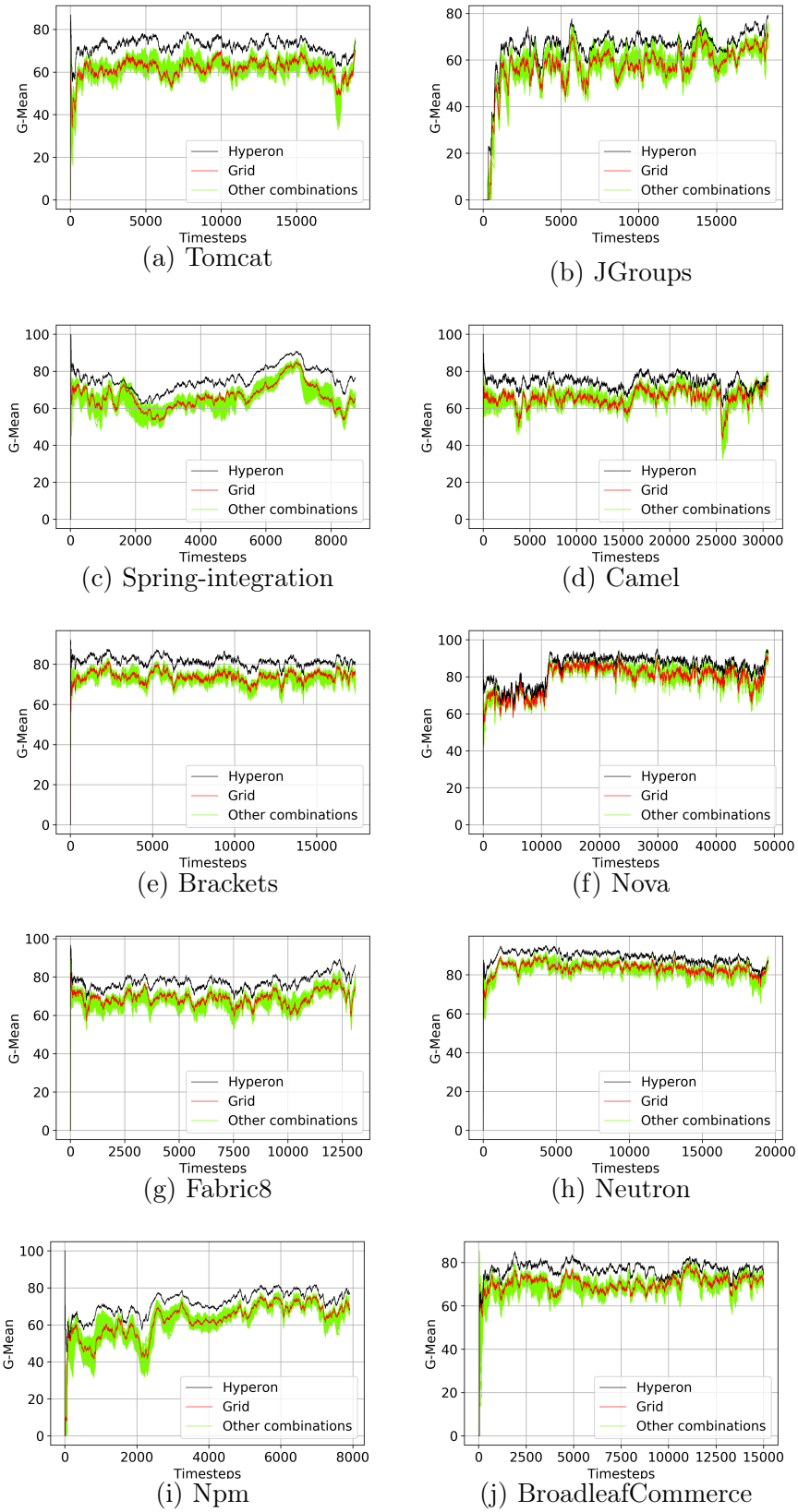
(a) Tomcat (b) JGroups (c) Spring-integration

(d) Camel (e) Brackets (f) Nova

(g) Fabric8 (h) Neutron (i) Npm

(j) BroadleafCommerce

Figure 7.2: New hyper-parameter combinations discovered by Hyperon over time that obtained best g-mean. The point corresponding to a given hyperparameter combination is plotted when this combination is selected to be used for predictions and is omitted in the consecutive time steps where it is used, to improve readability of the plots.



(a) Tomcat (b) JGroups

(c) Spring-integration (d) Camel

(e) Brackets (f) Nova

(g) Fabric8 (h) Neutron

(i) Npm (j) BroadleafCommerce

Figure 7.3: G-Mean for all datasets through time using Hyperon, Grid search and other 242 combinations.

algorithm (ADE-ACM) in an online way without being limited to exploring only a fixed set of hyper-parameter combinations. It is possible that new combinations do not provide better predictive performance than the already existing combinations explored by Grid Search. In such case, such new combinations will not be used for actual predictions. Not being able to generate any better performing combination would make the proposed approach inefficient. So, we investigated whether Hyperon can manage to generate better performing new combinations over time.

Fig. 7.2 shows new hyper-parameter combinations generated by Hyperon over time that were selected for use in JIT-SDP predictions at least at some point in time, due to their top average G-mean. Better performing new combinations in the plots are shown as different coloured points. From the plots, it is visible that for 5 datasets, Hyperon managed to generate at least one new useful combination, and for other 5 datasets, it was able to create more than one (up to 7) new hyper-parameter combinations (for Nova fig. 7.2f) over time that performed better than the existing ones. As Hyperon creates new combination in every $ea_interval = 3000$ time steps, for larger data sets such as Camel (30575 commits) and Nova (48989 commits), the number of new generated combinations is larger. Hence, possibility of generating better new combinations would be higher compared to smaller data. However, even for some of the smaller datasets like JGroups (18325 commits), Hyperon managed to create 5 new better hyper-parameter combinations. Moreover, from Fig. 7.2, it is visible that a new combination can be used not only once, but can be repeatedly useful in later periods of time. For example, in Spring-Integration, the combination C1, and in JGroups, combination C1 and C3 repeatedly performed better in several intercalated time periods.

These results show that Hyperon was able to suggest new useful hyper-parameter combinations for all datasets, demonstrating that it is possible to run evolutionary algorithms in an online way for the purpose of hyper-parameter tuning. However, to understand how much improvement in terms of the predictive performance of JIT-SDP can be obtained by Hyperon, we compared the predictive performance of Hyperon with the Grid Search on the initial portion of the data stream and the dummy classifier. Table 7.1 shows the average G-Mean, Recall0 and Recall1 of 30 runs obtained by Hyperon, Grid Search on initial portion and dummy classifier. Scott-Knott.BA12 is

Table 7.1: Average G-Means of Hyperon, Grid search and Dummy classifier, A12 effect sizes, Scott-Knott.BA12 to compare Hyperon, Grid search and Dummy classifier.

Dataset	Approach	G-Mean	R0	R1
Tomcat	Hyperon	72.24 (0.25)	81.26 (0.63)	64.81 (0.7)
	Grid	61.28 (0.57) [-b]	70.23 (0.87) [-b]	55.25 (0.6) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Jgroups	Hyperon	65.43 (1.07)	82.4 (1.15)	54.56 (2.14)
	Grid	57.17 (0.8) [-b]	69.09 (1.46) [-b]	51.17 (0.99) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Spring-Integration	Hyperon	75.73 (0.24)	81.08 (0.38)	71.82 (0.56)
	Grid	66.6 (0.88) [-b]	71.91 (0.81) [-b]	63.51 (1.51) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Camel	Hyperon	74.36 (0.38)	78.96 (0.61)	70.55 (1.09)
	Grid	65.78 (0.57) [-b]	67.56 (0.81) [-b]	65.35 (0.6) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Brackets	Hyperon	81.51 (0.17)	81.02 (0.37)	82.31 (0.47)
	Grid	73.77 (0.39) [-b]	72.5 (0.87) [-b]	75.6 (0.88) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Nova	Hyperon	85 (0.15)	80.44 (0.28)	90.34 (0.45)
	Grid	78.97 (0.15) [-b]	74.56 (0.37) [-b]	84.53 (0.5) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Fabric8	Hyperon	77.37 (0.24)	79.8 (0.33)	75.26 (0.61)
	Grid	69.01 (0.48) [-b]	71.09 (0.62) [-b]	67.61 (1.0) [-b]
	Dummy	71.59 [-b]	76.4 [-b]	81.64 [-b]
Neutron	Hyperon	89.09 (0.17)	86.55 (0.37)	91.83 (0.54)
	Grid	83.36 (0.24) [-b]	80.76 (0.56) [-b]	86.3 (0.53) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Npm	Hyperon	71.96 (0.39)	76.36 (0.46)	68.97 (0.98)
	Grid	62.64 (0.91) [-b]	62.42 (1.65) [-b]	66.51 (1.71) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Broadleafcommerce	Hyperon	76.76 (0.64)	81.55 (0.63)	72.84 (1.52)
	Grid	70.32 (0.58) [-b]	72.33 (0.51) [-b]	69.05 (0.9) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Ranking	Hyperon	1	1	1
	Grid	2	2	2
	Dummy	3	3	3

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Hyperon approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Hyperon approach. Scott-Knott.BA12 was run for all approaches together. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

used to compare Hyperon, grid and dummy classifier for 10 open source datasets. According to Scott-Knott.BA12, Hyperon ranked the best (lowest numbered ranking is better) compared to grid and dummy classifier. A12 effect sizes of Grid Search and dummy classifier against Hyperon were always large. Absolute improvements achieved by Hyperon in terms of G-Mean were up to 10.96% (Tomcat) and 39.09% (Neutron) compared to grid and dummy classifier, respectively. False Positive Rate (FPR) and False Negative Rate (FNR) are also reported in the Appendix E Section **??**.

We have seen that Hyperon performed better than Grid Search in terms of predictive performance. However, it is worth noting that Grid Search gets the opportunity to select best hyper-parameter combination only once, whereas Hyperon gets more chances as it selects best combination in each interval. It is possible that if Grid Search was repeatedly run over time, then it would be able to select better combinations leading to higher predictive performance. For an impartial comparison, we consider the upper bound explained in Section 7.3.1, which represents the highest possible predictive performance that could be obtained by Grid Search run over time.

We used Wilcoxon rank sum test [226] to compare Hyperon with Upperbound with a confidence level of 95% (i.e., p-value < 0.05). The Wilcoxon rank sum test is a non-parametric test that takes the null hypothesis that two approaches are same while the alternative hypothesis is that two approaches are significantly different. Wilcoxon rank sum test shows that the difference between G-Means of Hyperon and Upperbound is significant (p-value=9.605e-10 < 0.05). Hyperon achieved up to 6.9% (for Tomcat) improvement in terms of G-Mean over "Upperbound".

RQ3.2:Hyperon was able to find better performing new hyper-parameter combinations over time. It managed to find up to 7 new combinations and at least 1 new combination over time that improved predictive performance of the JIT-SDP model. Absolute differences in G-Mean were up to 10.96% and 6.9% in favour of Hyperon when compared, respectively, with Grid Search and Upperbound, which is the maximum possible G-Mean obtained if Grid Search was run over time.

7.4.3 RQ3.3: How high is the computational cost of the online hyper-parameter tuning approach? In particular, how feasible is the new approach considering the trade-off between predictive performance improvement and runtime?

Hyperon is an online hyperparameter tuning approach. As a single iteration of its underlying evolutionary algorithm is performed at every *ea_interval* time steps, its computational cost is expected to be much smaller than if a full evolutionary algorithm had been run at regular intervals. However, Hyperon still needs to train multiple JIT-SDP models simultaneously. Also, it requires training of a new model at regular intervals. Due to this, it is apparent that Hyperon would need higher computational cost compared to Grid Search. It is thus important to check how high this computational cost is, to understand whether Hyperon would be feasible enough to adopt in practice. Therefore, this section compares the computational cost of Hyperon with that of Grid Search run on the initial portion.

Dataset	AIO-Grid (s)	Hyperon (s)	Project Duration (year)	Runtime Per Year (AIO-Grid) (s)	Runtime Per Year (Hyperon) (s)	Runtime Per Day (AIO-Grid) (s)	Runtime Per Day (Hyperon) (s)	# Training Examples After Target Project Starts
Tomcat	1345.37	74077.76	11.7	114.99	6331.43	0.32	17.35	221752
JGroups	1138.37	85753.47	14.25	79.89	6017.79	0.22	16.49	226042
Spring-integration	1152.73	70823.3	10.18	113.23	6957.1	0.31	19.06	214379
Camel	1195.2	96433.35	10.73	111.39	8987.26	0.31	24.62	218173
Brackets	1341.43	66864.03	6	223.57	11144.01	0.61	30.53	160880
Nova	1451.5	93467.23	7.68	189	12170.21	0.52	33.34	192119
Fabric8	1252.27	76679.72	6.65	188.31	11530.78	0.52	31.59	175211
Neutron	1274.23	81804.29	6.99	182.29	11703.05	0.5	32.06	181659
Npm	1085.07	65264.52	8.16	132.97	7998.1	0.36	21.91	198864
BroadleafCommerce	1346.53	79799.2	9.01	149.45	8856.74	0.41	24.27	206093

Table 7.2: Computational Cost Analysis for Hyperon and Grid search

Table 7.2 shows total average runtime of Grid Search and Hyperon in columns 2 and 3, computed across 30 runs for each of the 10 open source projects on an Intel(R) Xeon(R) CPU E5-2690 v3 at 2.60GHz and 16Gb of RAM. Column 4 shows the duration of each project, and columns

5–8 of Table 7.2 list the average runtime per year and per day for the Grid Search and Hyperon approaches computed by dividing the total runtime by project duration. The maximum runtime of Grid Search and Hyperon were up to 223.57 and 12170.21 seconds per year, and up to 0.61 and 33.34 seconds per day. Even though the runtime of Hyperon was higher than that of Grid Search, it is still very small (less than a minute). Hence, the computational cost for Hyperon is feasible enough and would not add much computational overhead if adopted in practice.

RQ3.3: Hyperon had higher computational cost compared to Grid Search. However, the computational cost is still very small (up to 33.34 seconds per day), which makes Hyperon suitable for adaption in practice.

7.5 Threats to Validity

Internal validity: Each approach for each dataset has been executed 30 times to mitigate threats to internal validity. All approaches respect the chronology of the data and take verification latency into account. Being a hyper-parameter tuning approach, Hyperon should achieve good results with its own hyper-parameters set to default values. Therefore, Hyperon's hyper-parameters were set to default values in our experiments.

Construct validity: Evaluation metrics used in this work are G-Mean, Recall0 and Recall1. G-Mean with fading factor is a widely used and recommended metric appropriate for class imbalance learning [52, 53] due to being unbiased [54].

Statistical conclusion validity: Scott-Knott test for comparison of multiple groups was run with non-parametric bootstrap sampling considering A12 effect size to avoid concluding that there is a difference in predictive performance when this difference is likely to be irrelevant due to insignificant effect size. Also, Wilcoxon rank sum test is used to compare Hyperon with Upperbound as a non-parametric test for comparison of two groups.

External validity: This study used 10 open source projects of various characteristics such as programming language, starting date, number of commits per day, etc. Hence, the results may not generalise to other types of projects. Also, the proposed approach in this study was investigated for online JIT-SDP. We may observe differences in the performance of the proposed approach for different problem domains other than JIT-SDP.

7.6 Summary

This study investigated online hyper-parameter tuning for JIT-SDP using open source CP data. We showed that the best hyper-parameter combination changes over time in online JIT-SDP, having a significant impact on the predictive performance of online JIT-SDP models. In particular, the best hyper-parameter combination chosen based on initial portion of the data stream does not stay as the best choice over the time, becoming ineffective over the course of software projects. Our investigation on the impact of hyper-parameters shows the necessity of an online hyper-parameter tuning approach for online JIT-SDP. Our proposed approach Hyperon is an online hyper-parameter tuning approach that managed to suggest appropriate hyper-parameter combinations over time. Hyperon achieved up to 10.96% and 6.9% improvements in terms of G-Mean when compared with Grid Search run on the initial portion of the data stream, and the maximum possible G-Mean obtained by Grid Search if run over time (Upperbound). Although higher than Grid search, the computational cost for Hyperon is very small (less than a minute per day), which makes it a feasible enough approach to be adopted in practice.

Chapter Eight

Conclusions and Future Work

This thesis provides the first investigation of CP learning and automated hyper-parameter tuning for online JIT-SDP, with the aim of gaining a better understanding of when and how CP data can help to improve predictive performance of JIT-SDP in online learning scenario, and proposing an effective online cross-project JIT-SDP framework. In this chapter, we present the summary of the work based on the research questions outlined in Chapter 1 and directions for future work.

8.1 Conclusions

We summarise the conclusions of Chapters 5, 6 and 7 in Sections 8.1.1, 8.1.2 and 8.1.3, respectively.

8.1.1 Online Cross-project JIT-SDP

As aforementioned in Chapter 1, a major challenge of WP SDP and JIT-SDP studies is lack of training data at the beginning of a project. Also, due to concept drift, past data may become irrelevant during certain periods of the project, and can cause sudden drops in predictive performance of the defect prediction model. Another limitation of the existing offline CP JIT-SDP studies is that they assumed an offline scenario where the model never gets updated with any new data, which

contradicts a real-world scenario where data arrive sequentially over time. Hence, a practicable model should be able to learn new data over time. Moreover, the existing offline CP JIT-SDP approaches only use CP data to train the JIT-SDP models. Such models never use WP data for training. However, in a realistic online scenario, such WP data naturally arrives over time, such that models can be updated with both CP and WP data over time.

Chapter 5 focused on providing a deeper understanding of the role of CP data in online JIT-SDP scenario. This chapter proposed 3 novel online CP JIT-SDP approaches (AIO, Filtering and Ensemble) able to operate in a realistic online JIT-SDP scenario and the experimental results with these approaches to answer the following research question:

RQ1: Can cross-project learning help to improve predictive performance in online JIT-SDP? How?

We summarise the answers of RQ1 including the sub research questions RQ1.1 to RQ1.4 below:

- RQ1.1: Can CP data help to improve predictive performance in the initial phase of a project, when there is little or no WP training data available? For how long and to what extent?

  The initial phase of the projects investigated in Chapter 5 had considerably large length, lasting from 461 to 6271 and from 8 to 582 time steps for the open source and proprietary projects, respectively. Our experimental results showed that CP data was helpful when there was no or little WP training data available, in particular when using AIO approaches for both open source projects and proprietary projects. Filtering was also top performing for the open source projects. Absolute improvements in average G-Mean were up to 53.89%, avoiding extremely low G-Means.

- RQ1.2: Can CP data help to prevent sudden drops in predictive performance, which may be caused by concept drift? To what extent?

  The periods of sudden drop in performance were of considerable length, covering in total

from 936 to 6009 time steps for open source, and from 23 to 541 time steps for proprietary data. CP approaches frequently helped to reduce or even prevent such sudden drops in G-Mean compared to WP approaches, with absolute improvements of up to 37.35% during such periods. Filtering was the top ranked approach across datasets.

- RQ1.3: What are the effects of CP data on the predictive performance during stable periods of the projects? Could the CP data be detrimental during stable periods, given that enough WP data may be available for training?

  Except for the Ensemble approach, CP approaches performed at least as well as WP approaches in terms of G-Mean during stable periods. In particular, Filtering with ORB obtained up to around 8.39% improvement in G-Mean for open source data and AIO-OOB-combined achieved up to 29.03% absolute improvement in G-Mean for proprietary data than the WP approach during such periods.

- RQ1.4: What is the overall effect of using CP and WP data together on the predictive performance throughout the development of software projects? In particular, do classifiers trained on both CP and WP data improve, deteriorate or retain similar overall predictive performance to WP classifiers?

  Merging CP and WP data together (All-in-One or Filtering) to train a classifier achieved up to 16.88% higher overall G-Mean than WP classifiers for the open source, and up to 30.86% higher overall G-Mean for the proprietary data in an online scenario. There was no evidence that filtering out training examples dissimilar to the recent software changes of the target project is helpful to improve overall predictive performance. Building separate classifiers from individual projects (Ensemble approach) was detrimental.

- RQ1.5: Is it necessary to update CP approaches over time once a project starts, or is it enough to use models trained only with CP data produced before the project being predicted commences? If it is enough, online learning is not really necessary.

  CP approaches achieved absolute G-Mean improvements of up to 32.15%, 48.16% and 41.25% for open source data and up to 35.02%, 23.9% and 37.94% for proprietary data compared to

OP approaches during initial, sudden drop and stable periods respectively. This highlights the importance of updating the JIT-SDP model with CP (including WP) data received over time.

8.1.2 Impact of Online and Offline Base Learners on Online Cross-project JIT-SDP Scenarios

Existing JIT-SDP studies frequently used ensemble learners to build the defect prediction models. Depending on the learning process, these ensemble approaches can consist of several offline or online base learners, which are used for prediction. Both offline and online base learners have their advantages and disadvantages. For example, offline learners can be stronger but may be computationally expensive. They may also not tackle concept drift well caused by the change in the data generating process. On the other hand, online learners may be faster and able to deal with concept drift but may provide poorer prediction performance. Besides, most existing online JIT-SDP studies used Hoeffding Tree as base learners, and no other online or offline base learners were investigated for online JIT-SDP. Hence, it is not known how well approaches based on Hoeffding Tree perform compared to approaches based on other online and offline base learners for a realistic online scenario for CP JIT-SDP.

Chapter 6 investigated whether offline models adapted to operate in a realistic online JIT-SDP scenario can offer benefits compared to online models. The study also provides a comparison of predictive performance and computational costs of several offline and online base learners for online JIT-SDP. It answers the following research question:

RQ2: How online base learners perform compared to offline base learners in terms of predictive performance? Can use of different base learners influence the predictive performance of cross-project JIT-SDP approaches?

We summarise the answers of RQ2 including the sub research questions RQ2.1 to RQ2.3

below:

- RQ2.1: Can offline learning help to improve predictive performance compared to online learning in online WP JIT-SDP scenarios? Which base learners usually perform best?

  Offline learning helped to improve average G-Mean in the online WP JIT-SDP scenario. BORB-WP with MLP, LR and IRF ranked the best, and performed better than the top ORB-WP approach, which used OHT. BORB-WP achieved up to 27.97% absolute improvement in terms of average G-Mean when compared against ORB-WP. However, offline WP BORB approaches suffered severe and recurrent performance drops in two datasets, which reveals some instability of these approaches

- RQ2.2: How beneficial is CP data to improve predictive performance of offline models compared to online models in online CP JIT-SDP scenarios?

  CP data was not so helpful for the offline learning as it was for the online learning approaches. In most cases, CP data did not lead to better average G-Mean with BORB. How- ever, it enabled ORB-OHT-CP to be among the first ranked approaches according to Scott- Knott-A12. Moreover, it enabled ORB-OHT-CP to achieve more stable G-Mean through time than BORB-WP and BORB-CP. Therefore, ORB-OHT-CP is recommended over BORB in practice.

- RQ2.3: How high is the computational cost of offline learning in online JIT-SDP scenarios compared to that of online learning models?

  Online WP approaches were up to 16.02 times faster than offline WP approaches, and online CP approaches were up to 12.10 times faster than offline CP approaches. The online approach that achieved most competitive average G-Mean and that achieved the most stable G-Mean over time (ORB-OHT-CP) had a computational cost up to 6.13 slower than the the most competitive offline WP approaches (BORB-IRF-WP, BORB-LR-WP and BORB-MLP-WP). However, this computational cost can still be considered feasible, as the average computational cost was at most 1.14 second per day.

8.1.3 Online Hyper-parameter Tuning for JIT-SDP

The predictive performance of machine learning models is usually affected by their hyper-parameter choices. Default hyper-parameter choice frequently provides sub-optimal predictive performance. Hence, tuning is usually essential to select appropriate hyper-parameter combinations for the machine learning model. Although there are different hyper-parameter tuning approaches for offline learning, it is difficult to tune hyper-parameters for online learning because the initially chosen best hyper-parameter values may change over time. However, the effect of hyper-parameter choice on the predictive performance can vary for different problem domains and the effect of hyper-parameters in online JIT-SDP is unknown. If the initial best hyper-parameter choice changes over time for online JIT-SDP, then an online hyper-parameter tuning for online JIT-SDP would be necessary to improve its predictive performance.

Chapter 7 thus investigates the impact of hyper-parameter choices on online JIT-SDP models and proposes a novel online hyper-parameter tuning approach for online JIT-SDP to answer the third research question:

RQ3: Does the best hyper-parameter choice for online JIT-SDP change over time? If so, then how to tune the hyper parameters in online manner?

We summarise the answers of RQ3 including the sub research questions RQ3.1 to RQ3.3 below:

- RQ3.1: What is the impact of hyper-parameters on the predictive performance of online JIT-SDP?

  The choice of the best hyper-parameter combination in online JIT-SDP can change over time causing the model to obtain lower predictive performance when using the initially chosen combination. To avoid such low predictive performance, it is important to explore other hyper-parameter combinations throughout time and not just in the beginning of the project. To explore different hyper-parameter combinations over time for JIT-SDP, a hyper-parameter

tuning approach is required that is able to operate in an online manner.

- RQ3.2: How to automatically tune hyper-parameters for JIT-SDP in an online manner? How well does this approach perform in terms of suggesting appropriate hyper-parameters to improve JIT-SDP's predictive performance?

  Our proposed online hyper-parameter tuning approach Hyperon was able to find better performing new hyper-parameter combinations over time. It managed to find up to 7 new combinations and at least 1 new combination over time that improved predictive performance of the JIT-SDP model. Absolute differences in G-Mean were up to 10.96% and 6.9% in favour of Hyperon when compared, respectively, with Grid Search and Upperbound, which is the maximum possible G-Mean obtained if Grid Search was run over time.

- RQ3.3: How high is the computational cost of online hyper-parameter tuning approach? In particular, how feasible is the new approach considering the trade-off between predictive performance improvement and runtime?

  Hyperon had higher computational cost compared to Grid Search. However, the computational cost is still very small (up to 33.34 seconds per day), which makes Hyperon suitable for adaption in practice.

8.2 Future Work

This section presents potential research directions for further development of our proposed online CP JIT-SDP approaches in this thesis.

- Chapter 5 investigated when and to what extent CP data can be useful in online JIT-SDP scenario. The experiment results showed that combined use of CP and WP data can improve the overall predictive performance of the JIT-SDP model. The results indicated the importance of larger amount of training data to improve the predictive performance during different periods of time. The length of the proprietary projects used for the experiments

are much shorter than the open source projects. We found that the predictive performance of the JIT-SDP models on the proprietary projects was not high compared to that of open source projects, which could be due to the smaller amount of proprietary data. Hence, future work can consider incorporating additional longer running proprietary projects to investigate whether the predictive performance would improve.

- The experiment results of Chapter 5 showed that our online CP approaches achieved better predictive performance than OP approaches. Here, CP approaches were trained using both CP and WP data. Only CP data was used to train OP approaches; no WP data was used. This indicates that WP data is still necessary to improve predictive performance in JIT-SDP. In many cases, especially at the beginning stages of a project, there is not enough WP data accessible. It would be wise to look into methods that do not heavily rely on WP data. New transfer learning approaches to reduce the need for large amounts of WP data would be desirable. These approaches could focus especially on improving predictive performance during the initial phase of the projects, which was the most challenging for the proposed approaches.

- Our online CP JIT-SDP approaches were able to improve predictive performance during the initial and performance drop periods. However, the predictive performance was still much lower during these periods than during the stable periods. For example, G-Mean achieved by AIO-OOB for Nova during stable period was 82.03. During the initial periods the G-Mean was 63.15 and during drop periods the G-Mean was 66.99. These large differences in G-Mean for initial and drop periods with the stable period shows that there is room for improvements here. Also, practitioners should be aware of any such period of lower predictive performance. Therefore, mechanisms able to alert practitioners of potential reductions in predictive performance are desirable.

- Chapter 6 investigated impact of online and offline base learners on online CP JIT-SDP scenario. From Chapter 6 results, we found that online CP approaches achieved competitive G-Mean, but had a computational cost up to 6.13 slower than the the most competitive

offline WP approaches. Reducing computational cost would make the online CP approaches more effective in practice. Hence, future work can consider improving computational costs for online CP approaches.

- In Chapter 6, we found that the offline approach adapted for online JIT-SDP scenario (BORB) suffered some severe performance drops. Such severe and frequent performance drops in performance make the BORB approach unstable and unreliable to adopt in practice. Thus, future research could propose more stable offline models for online JIT-SDP scenario that can avoid such drops/variations in the predictive performance.

- Chapter 7 proposed an online hyper-parameter tuning approach for JIT-SDP. The proposed approach managed to suggest appropriate hyper-parameter combination over time, and thus improved the overall predictive performance of the JIT-SDP model. However, it is not known how this approach would work during any performance drop periods. Therefore, future work can consider investigating how the proposed online hyper-parameter tuning approach responds to sudden performance drops when there is lack of relevant data or a change in the data generating process. As CP data already helps in such scenario, it would be interesting to investigate whether the hyper-parameter tuning provides any additional advantage to improve the predictive performance along with CP data. Also, this work only considered open source projects. Future work might consider incorporating proprietary projects.

- Future work might also take into account case studies with the practitioners to gain a better understanding of the practicality of the proposed research. Based on such case studies, implementation of JIT-SDP tools may be investigated in the future.

Appendix One

Supplementary materials for Chapter Two

A.1 Computing four quadrants of confusion matrix

The performance measure G-Mean is calculated based on Recall0 and Recall1 as follows:

G-Mean = $\sqrt{Recall0 * Recall1}$.

Recall0 and Recall1 are specificity (true negative rate) and sensitivity (true positive rate) that measure the proportion of actual negative and positive cases that are correctly classified by the model. Equations for Recall0 and Recall1 are

$$Recall0 = \frac{TN}{N} = \frac{TN}{TN + FP} = 1 - FPR \tag{A.1}$$

and

$$Recall1 = \frac{TP}{P} = \frac{TP}{TP + FN} = 1 - FNR \tag{A.2}$$

where TP=Number of True Positives, TN=Number of True Negatives, FP=Number of False Pos-

itives, FN=Number of False Negatives, P=Total Number of Positive Examples, N=Total Number of Negative Examples, TPR=True Positive Rate, TNR=True Negative Rate, FPR=False Positive Rate, and FNR=False Negative Rate.

All experimental results in this thesis reported both Recall0 and Recall1 along with G-Mean. Recall0 and Recall1 already capture the four quadrants of the confusion matrix in the form of ratios. In particular, it is possible to calculate the True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR) and False Negative Rate (FNR) based on Recall0 and Recall1 as follows:

$$TNR = Recall0$$

$$TPR = Recall1$$

$$FNR = 1 - Recall1$$

$$FPR = 1 - Recall0$$

If one wishes to compute TP, TN, FP and FN to re-create the confusion matrix itself instead of the ratios, it is also possible to compute them as follows by using information about N and P from the corresponding dataset, which was also provided and/or can be computed from Table 4.2 in the Chapter Four of the thesis:

$$TN = Recall0 * N$$

$$TP = Recall1 * P$$

$$FN = P - TP$$

$$FP = N - TN$$

Appendix Two

Supplementary materials for Chapter Four

B.1 Summary statistics and input features plots for open source datasets

Summary statistics tables for open source datasets are included in section B.1.1. Box plots and bar plot to visualise the characteristics of the numerical and categorical input features for open source datasets are included in the section B.1.2.

In the summary statistics tables, the column labeled 'min' presents the smallest value observed for a given feature in the dataset, whereas the 'max' column displays the largest value. The 'median' column shows the midpoint of the feature values, with half the values falling below this point and half above it, while the 'mean' column shows the average value of that feature. The 'std' column shows the standard deviation. The 'Q1' and 'Q3' columns display the values of the first quartile and third quartile, respectively.

For the feature 'ns', the dataset Spring-Integration has higher standard deviation (3.072)

than other datasets indicating that the data values are spread out over a wider range from the mean value for this dataset. Also, this dataset contains comparatively more outliers for the feature 'ns' (Fig. B.3l). For feature 'nf', standard deviation for Broadleaf was the largest (92.856), which suggests that there are larger variations in the number of modified files. The feature 'entropy' has a median of 0 for all datasets except Camel and Spring-Integration.

Standard deviations are usually very high for 'la' (up to 4987.436 for Broadleaf) and 'ld' (up to 8060.365 for Neutron). For all features, there may be presence of outliers as mean is usually higher than the median. Maximum value for 'lt' is up to 85735 (for Brackets), which indicates the number of lines of code in a file before the change was largest for this dataset. The dataset Nova has the maximum value for 'ndev' (400), which is much higher than other datasets. The features 'exp', 'rexp' and 'sexp' are experience related features. Higher 'exp' means higher developer experience. For Nova and Neutron, 'exp' has much lower median values compared to other datasets.

For 'lt', 'age' and 'rexp', there are presence of negative values for 7 datasets. For 'lt', getting negative values may be possible. If a file comes from a different branch or is new and is combined with a file that has a lot of deleted lines, we may end up with a negative lines total value. The percentages of such negative values for the datasets are given in Table B.1. Even if such

Table B.1: Percentage of negative values for features lt, age and rexp

dataset	**lt**	**age**	**rexp**
brackets	0.09	0.63	0.14
camel	0.85	0.37	0.23
fabric8	0.14	1.48	0.77
neutron	1.26	6.54	0.63
nova	0.72	6.54	0.63
npm	1.41	2.79	0.61
tomcat	0.86	0.0	0.0

negative values are considered as noise in the data, the percentage for the noise is very small (for 'lt' up to 1.41%, for 'age' up to 6.54% and for 'rexp' up to 0.77%), and within acceptable range as recommended by Kim et al. [227]. Hence, the very small noisy values of the three features are unlikely to have a drastic impact on the performance of the JIT-SDP models in this thesis.

B.1.1 Summary statistics tables for opensource datasets

feature	min	max	median	mean	std	Q1 (1st quantile)	Q3 (3rd quantile)
ns	0	8	1	1.219	0.533	1	1
nd	0	161	1	1.752	2.786	1	2
nf	0	1157	1	3.245	16.197	1	3
entrophy	0	8.961	0	0.657	0.979	0	1
la	0	273329	8	101.083	2861.936	2	31
ld	0	273329	4	64.757	2060.842	1	15
lt	-293	7050	465	728.728	834.715	171	989.286
ndev	0	21	13	12.592	5.751	8	18
age	0	2574.668	6.468	65.574	149.62	0.106	58.385
nuc	0	5452	36	197.014	418.926	10	143
exp	0	32565	4214	9737.757	10335.517	1363	17250.5
rexp	0	86401	1.123	216.137	1536.585	1.01	6.544
sexp	0	23317	1251	4352.765	6479.589	231	4052.5

Table B.2: Summary statistics for dataset 'Tomcat'

feature	min	max	median	mean	std	Q1	Q3
ns	0	9	1	1.152	0.504	1	1
nd	0	126	1	1.423	1.713	1	1
nf	0	1043	1	2.469	12.984	1	2
entrophy	0	8.728	0	0.392	0.815	0	0.533
la	0	189749	10	90.029	2364.505	3	38
ld	0	87473	4	58.569	967.159	2	18
lt	-171	12869	792	1227.896	1357.735	290	1726
ndev	0	33	11	10.645	7.153	6	14
age	-33.048	2017.57	0.012	30.317	96.963	0	12.382
nuc	0	24930	46	166.836	472.105	12	171
exp	0	25048	7743	8149.322	6192.9	2830	12321
rexp	-233.783	86401	1.003	47.036	1221.31	0	1.167
sexp	0	14033	2468	3284.732	3170.391	572	5328

Table B.3: Data Statistics for JGroups

feature	min	max	median	mean	std	Q1	Q3
ns	0	21	1	1.261	0.834	1	1
nd	0	1931	2	4.151	22.866	1	3
nf	0	3740	2	6.697	41.094	1	4
entrophy	0	11.269	0.809	1.073	1.323	0	1.703
la	0	126926	16	144.265	1257.934	2	87
ld	0	161192	3	66.687	1378.706	1	15
lt	-378	5743	136	396.681	757.127	64	320
ndev	0	178	30	46.794	44.826	14	72
age	-481.772	2747.288	6.352	66.562	157.004	0.265	57.282
nuc	0	29861	26	273.838	1021.778	5	128
exp	0	66633.5	5541	15140.152	19468.701	742.75	27497.75
rexp	-5399	43201	1.059	62.731	679.046	1.006	2.166
sexp	0	33853	809	5091.662	8163.499	95	5932.5

Table B.4: Data Statistics for Camel

feature	min	max	median	mean	std	Q1	Q3
ns	0	4	1	0.747	0.595	0	1
nd	0	312	1	1.24	4.796	0	1
nf	0	1429	1	2.042	21.502	0	2
entrophy	0	9.067	0	0.327	0.68	0	0.377
la	0	169164	2	91.809	2178.328	0	15
ld	0	169809	1	55.465	1714.031	0	7
lt	-56	85735	200	425.605	1728.945	0	568
ndev	0	100	16	28.19	29.157	0	57
age	-460.335	1102.877	0.11	9.773	40.515	0	3.173
nuc	0	16952	16	93.05	285.184	0	91
exp	0	4976	53.5	446.849	889.123	0	469.125
rexp	-594.862	10801	1.076	29.236	216.948	0	2.449
sexp	0	2175	26	244.208	452.03	0	266

Table B.5: Data Statistics for Brackets

feature	min	max	median	mean	std	Q1	Q3
ns	0	7	1	0.624	0.659	0	1
nd	0	436	1	1.408	4.705	0	2
nf	0	3655	1	2.484	26.949	0	2
entrophy	0	9.655	0	0.484	0.904	0	0.863
la	0	430845	1	142.966	3634.853	0	19
ld	0	808411	0	129.919	4818.656	0	6
lt	-1409.8	20204.5	31	683.986	1651.507	0	578
ndev	0	400	17	92.489	121.26	0	174
age	-542.682	1548.882	0	11.336	47.345	0	6.659
nuc	0	23236	3	191.989	564.514	0	114
exp	0	8321	3	291.374	770.449	0	204.5
rexp	-1727	86401	0.507	67.279	1920.873	0	1.06
sexp	0	7690	2	199.772	579.966	0	120

Table B.6: Data Statistics for Nova

feature	min	max	median	mean	std	Q1	Q3
ns	0	39	1	1.466	1.78	1	1
nd	0	1971	1	5.239	26.253	1	3
nf	0	6856	1	9.027	82.598	1	4
entrophy	0	11.212	0	1.013	1.513	0	1.56
la	0	388880	10	241.836	4329.441	2	53
ld	0	388719	3	235.064	5389.66	1	19
lt	-271	16388	153	388.448	653.976	64	359.725
ndev	0	57	31	30.631	17.67	18	50
age	-71.951	918.023	1.807	15.712	41.039	0.067	12.359
nuc	0	15312	17	431.003	1578.003	4	89
exp	0	30413	3089.25	6598.303	8314.366	314.625	9946.375
rexp	-3084.714	86401	1.173	177.444	3172.979	1.018	3.284
sexp	0	7610	159	854.425	1448.894	29	927.75

Table B.7: Data Statistics for Fabric8

feature	min	max	median	mean	std	Q1	Q3
ns	0	35	1	1.671	3.072	1	1
nd	0	309	1	3.362	8.372	1	3
nf	0	1157	2	6.36	25.054	1	5.75
entrophy	0	9.499	0.811	1.143	1.349	0	1.948
la	0	38007	23	149.881	833.599	2	106
ld	0	38007	6	100.611	859.661	1	36
lt	-47.25	3834.333	103	178.922	273.298	33	218.65
ndev	0	35	11	10.531	9.038	3	13
age	-206.478	1716.157	6.258	47.852	109.022	0.066	42.678
nuc	0	7823	11	49.574	181.793	3	36
exp	0	21787	3598.75	5763.742	6308.655	493	9754.125
rexp	-113.184	86401	1.032	103.561	1430.077	1.003	1.612
sexp	0	5596	106	659.241	1314.519	22	385

Table B.8: Data Statistics for Spring-Integration

feature	min	max	median	mean	std	Q1	Q3
ns	0	7	1	1.274	0.554	1	1
nd	0	527	1	2.397	12.08	1	2
nf	0	1953	1	5.981	45.386	1	3
entrophy	0	9.209	0	0.66	1.142	0	1
la	0	136084	7	234.185	2767.105	1	35
ld	0	132800	2	170.109	2459.7	1	12
lt	-3778	8160	129.125	267.257	621.782	55	243
ndev	0	91	9	21.355	25.557	2	38
age	-530.745	1956.958	5.385	33.481	85.932	0.22	29.451
nuc	0	6117	43	211.577	385.874	9	173.25
exp	0	16025.5	2692	3977.046	4027.759	721.75	6679.375
rexp	-6645.154	43201	1.093	99.817	964.024	1.007	2.17
sexp	0	7116	312	727.81	966.014	69	1078

Table B.9: Data Statistics for Npm

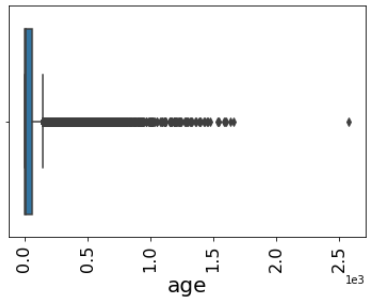
feature	min	max	median	mean	std	Q1	Q3
ns	0	11	1	0.614	0.684	0	1
nd	0	319	1	2.015	6.546	0	2
nf	0	1361	1	2.879	14.568	0	2
entrophy	0	9.073	0	0.561	1.033	0	0.926
la	0	285456	0	223.827	3410.051	0	21
ld	0	1059841	0	204.962	8060.365	0	7
lt	-678.222	17197.03	0	408.357	1444.157	0	303.619
ndev	0	253	4	65.936	81.136	0	132
age	-443.15	1247.036	0	17.885	54.318	0	18.367
nuc	0	7961	1	71.45	406.576	0	33
exp	0	8319	0.5	215.916	602.168	0	129
rexp	-703	3757.522	0	2.169	39.334	0	1.018
sexp	0	7403	0	136.802	451.57	0	57

Table B.10: Data Statistics for Neutron

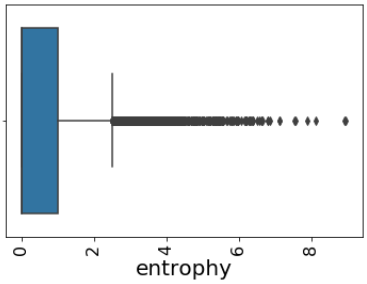
feature	min	max	median	mean	std	Q1	Q3
ns	0	11	1	1.097	1.008	1	1
nd	0	1358	1	3.399	20.505	1	2
nf	0	5444	1	8.039	92.856	1	3
entrophy	0	11.44	0	0.854	1.291	0	1.398
la	0	422681	8	224.152	4987.436	1	39
ld	0	436886	2	189.858	5061.702	0	14
lt	-240.5	9475	117	239.637	377.781	5.87	294
ndev	0	40	23	17.329	12.824	1	25
age	-273.431	1079.875	1.938	24.453	64.865	0	18.071
nuc	0	34182	13	164.817	768.475	1	64
exp	0	37310.5	699.75	3804.589	7094.405	40	4138.75
rexp	-1251.174	21601	1.041	106.302	799.252	0.717	1.558
sexp	0	32321	157	1351.18	3609.612	10	1066

Table B.11: Data Statistics for Broadleaf

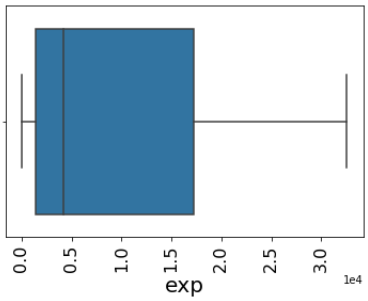
B.1.2 Plots for input features for opensource datasets



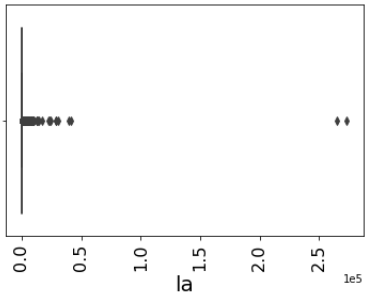
(a) age



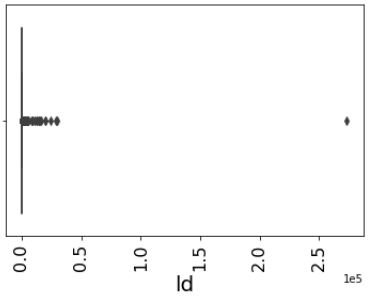
(b) entropy



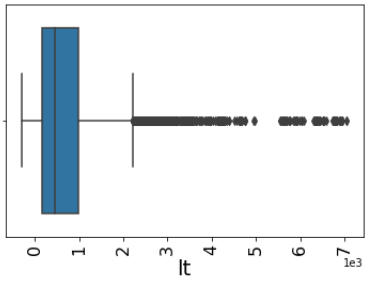
(c) exp



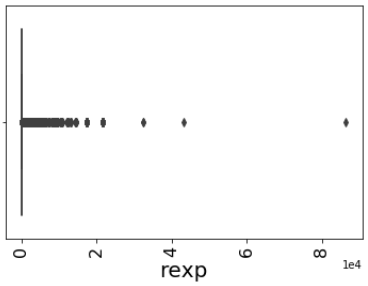
(d) la



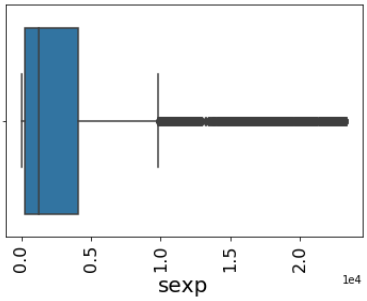
(e) ld



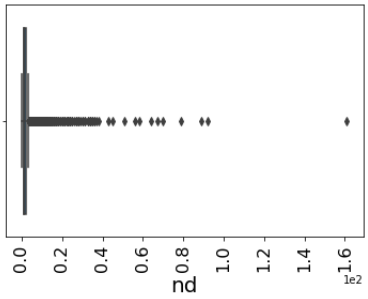
(f) lt



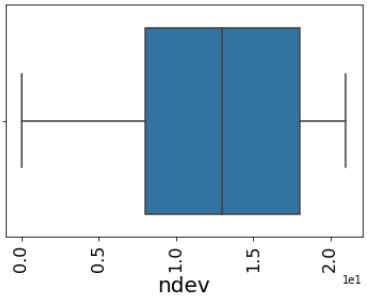
(g) rexp



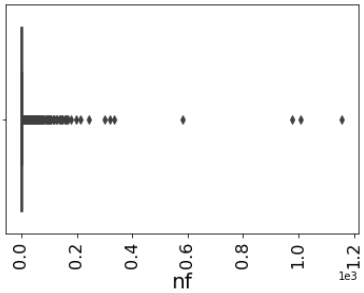
(h) sexp



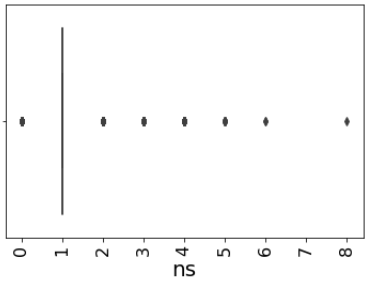
(i) nd



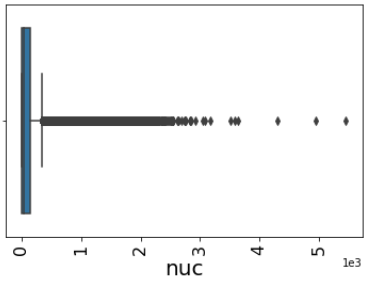
(j) ndev



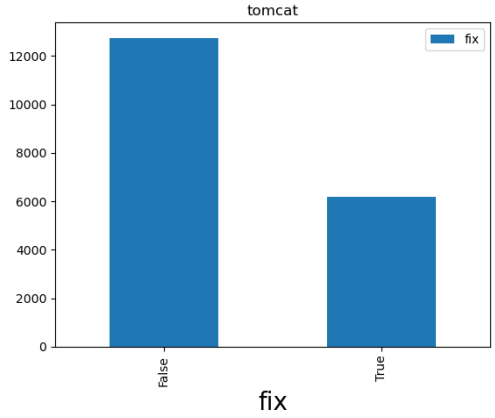
(k) nf



(l) ns

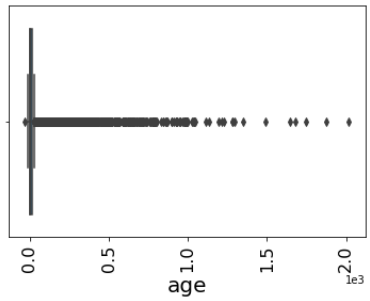


(m) nuc

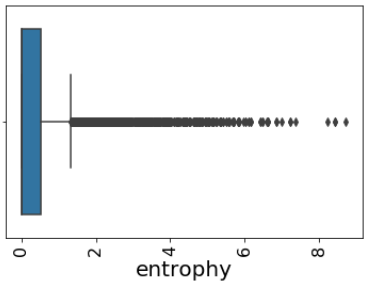


(n) Fix (Categorical Feature)

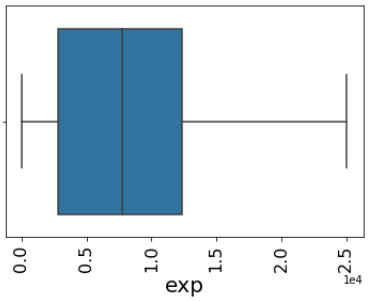
Figure B.1: Plots for numerical and categorical input features for Tomcat



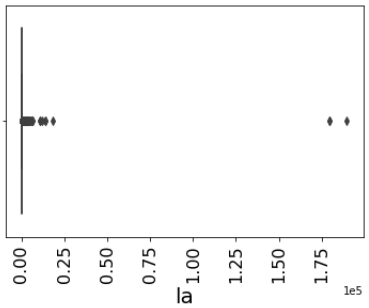
(a) age



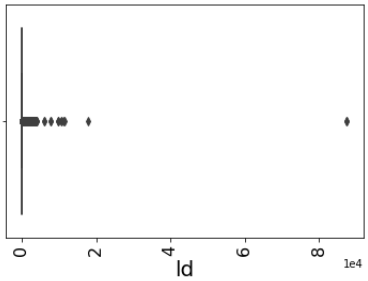
(b) entropy



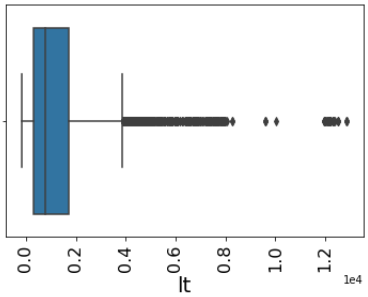
(c) exp



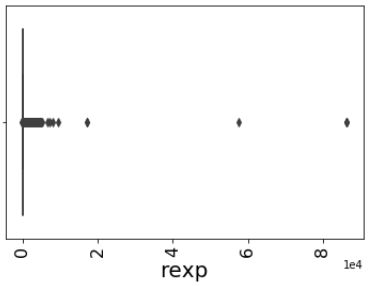
(d) la



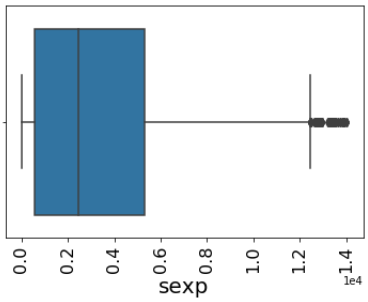
(e) ld



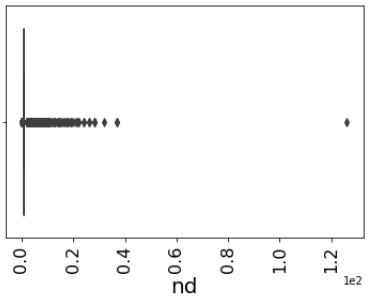
(f) lt



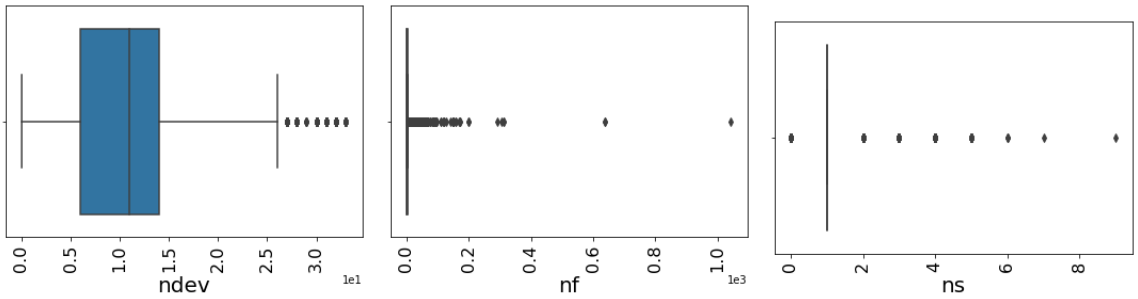
(g) rexp



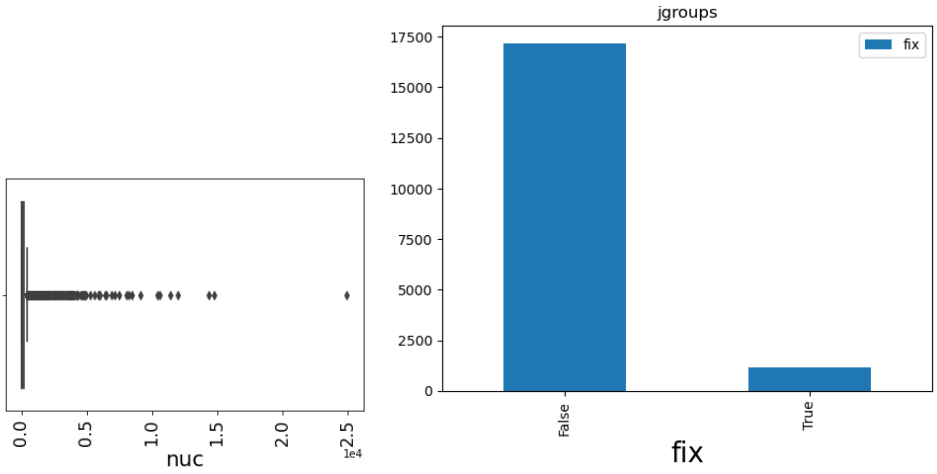
(h) sexp



(i) nd

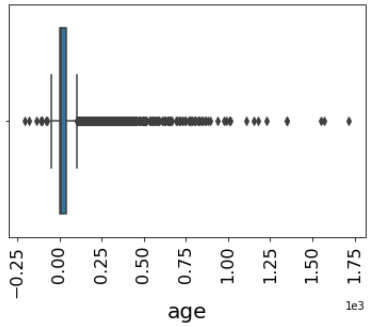


(j) ndev (k) nf (l) ns

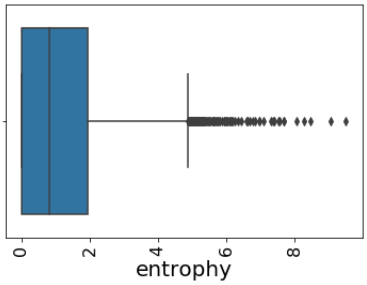


(m) nuc (n) Fix (Categorical Feature)

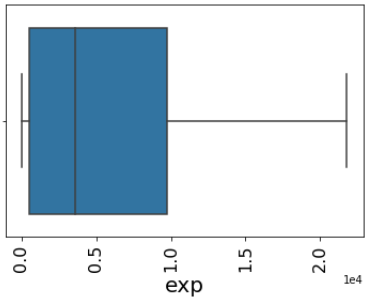
Figure B.2: Plots for numerical and categorical input features for JGroups



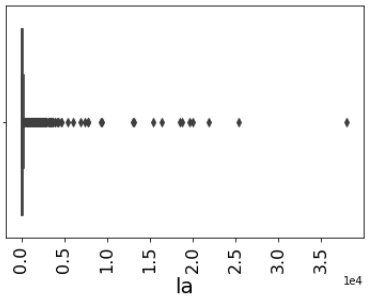
(a) age



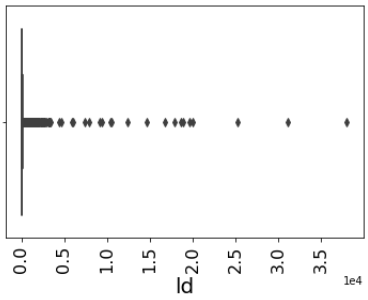
(b) entropy



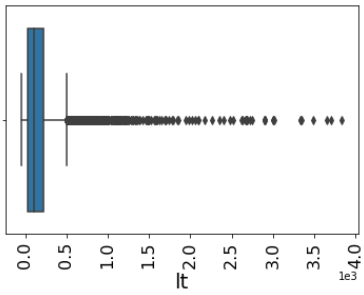
(c) exp



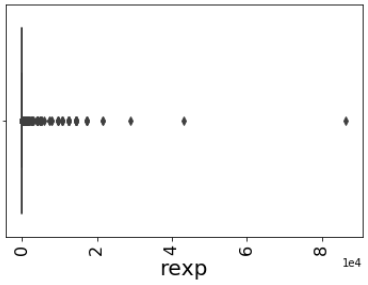
(d) la



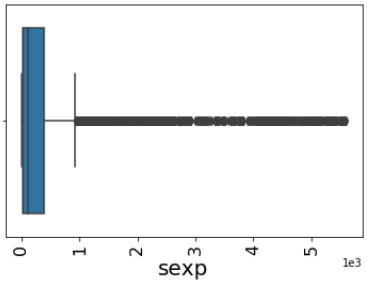
(e) ld



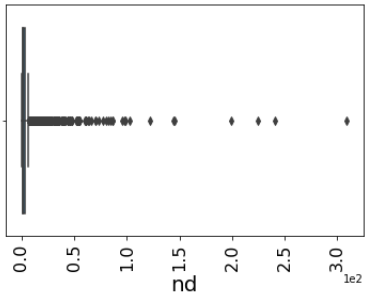
(f) lt



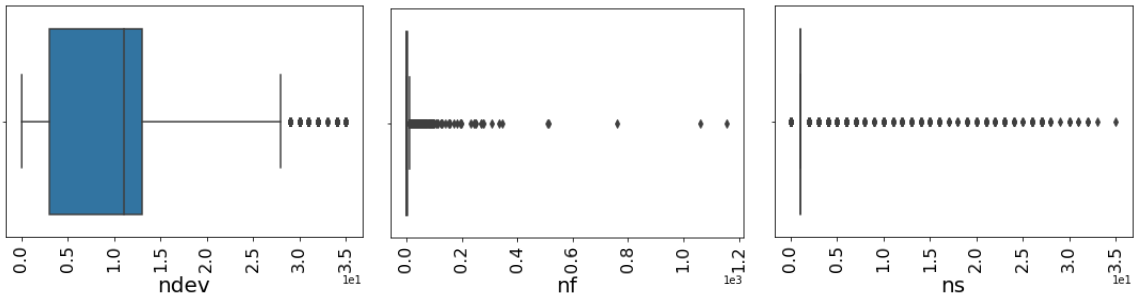
(g) rexp



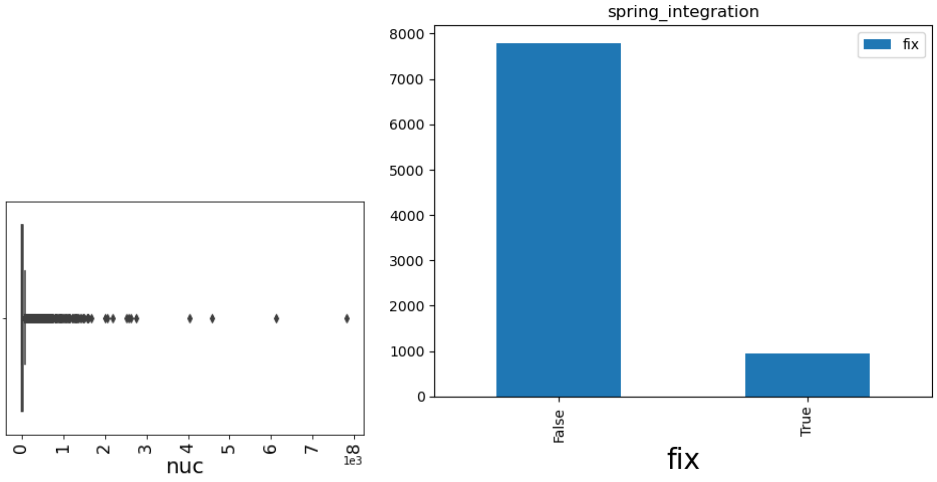
(h) sexp



(i) nd

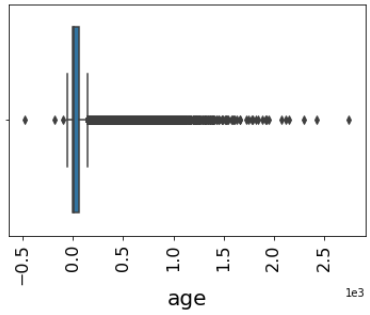


(j) ndev (k) nf (l) ns

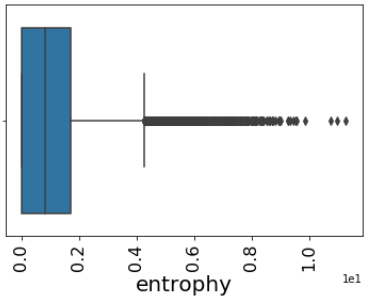


(m) nuc (n) Fix (Categorical Feature)

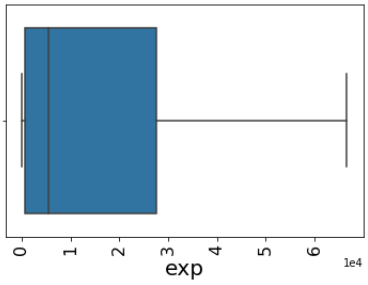
Figure B.3: Plots for numerical input features for spring-integration



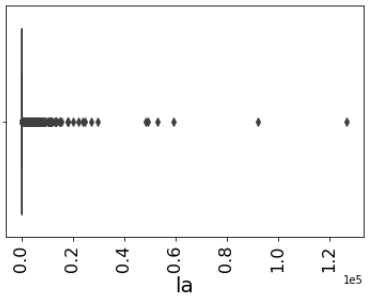
(a) age



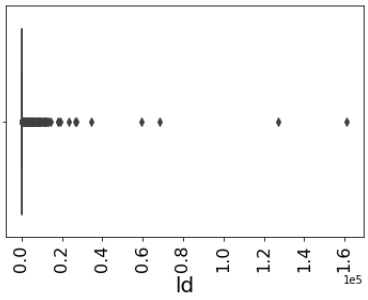
(b) entropy



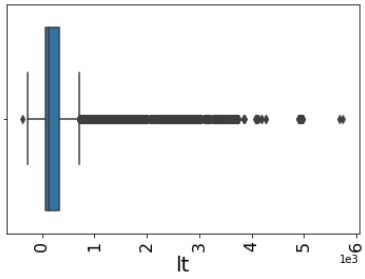
(c) exp



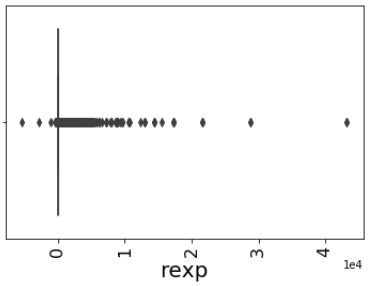
(d) la



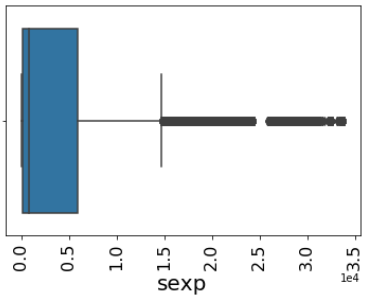
(e) ld



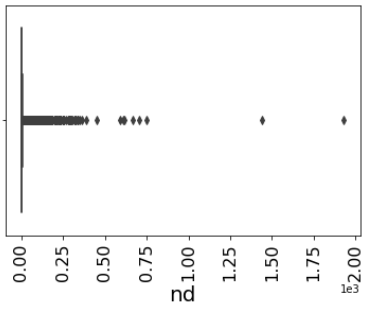
(f) lt



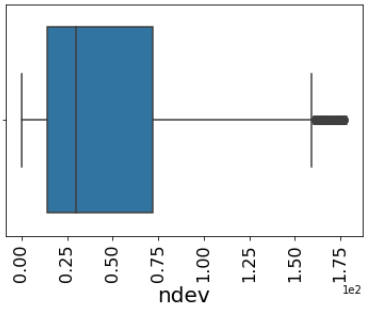
(g) rexp



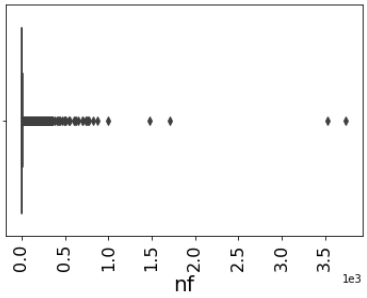
(h) sexp



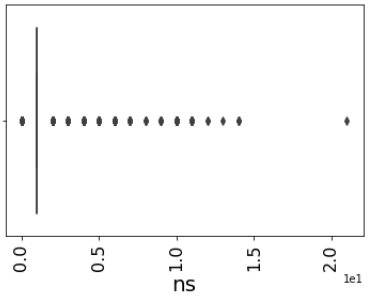
(i) nd



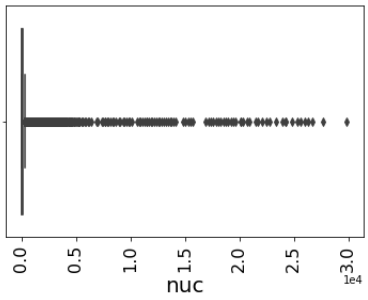
(j) ndev



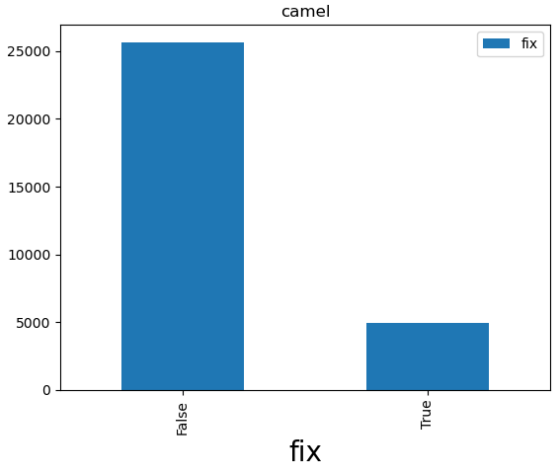
(k) nf



(l) ns

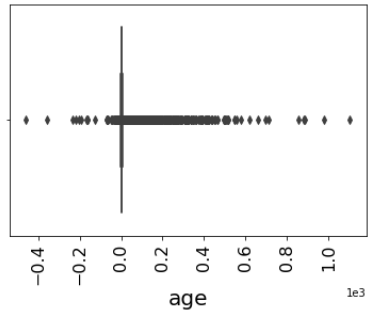


(m) nuc

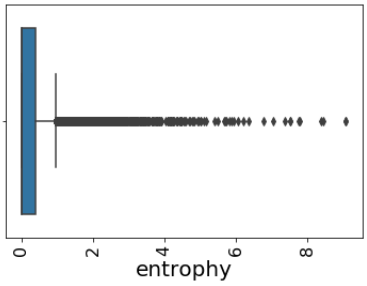


(n) Fix (Categorical Feature)

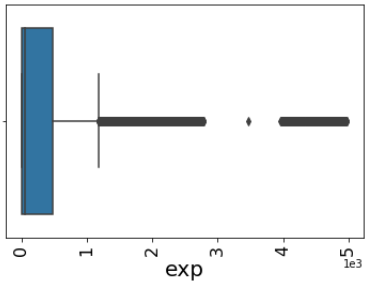
Figure B.4: Plots for numerical input features for camel



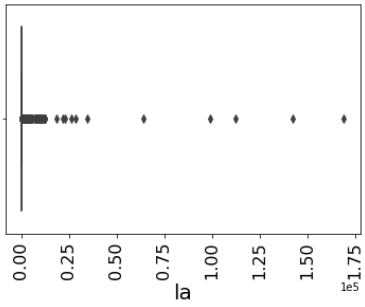
(a) age



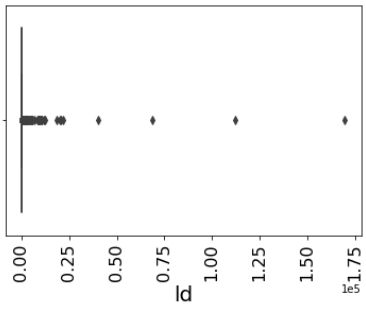
(b) entropy



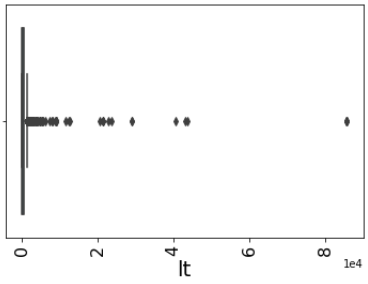
(c) exp



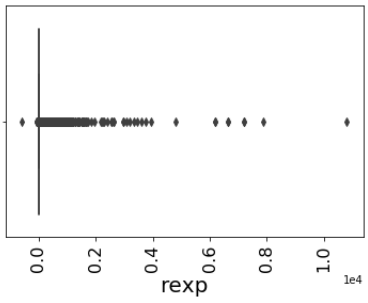
(d) la



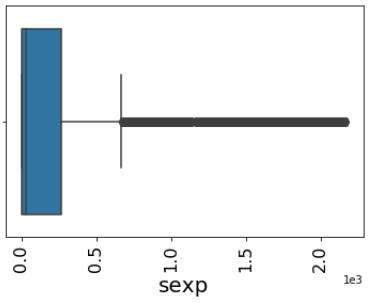
(e) ld



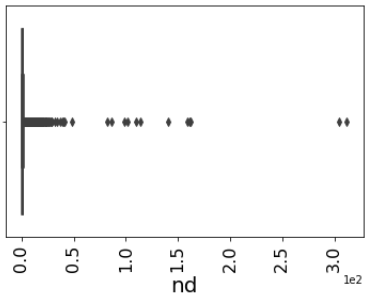
(f) lt



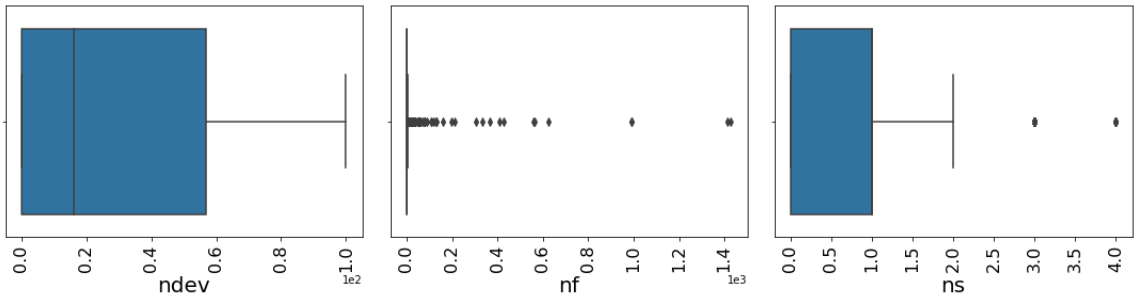
(g) rexp



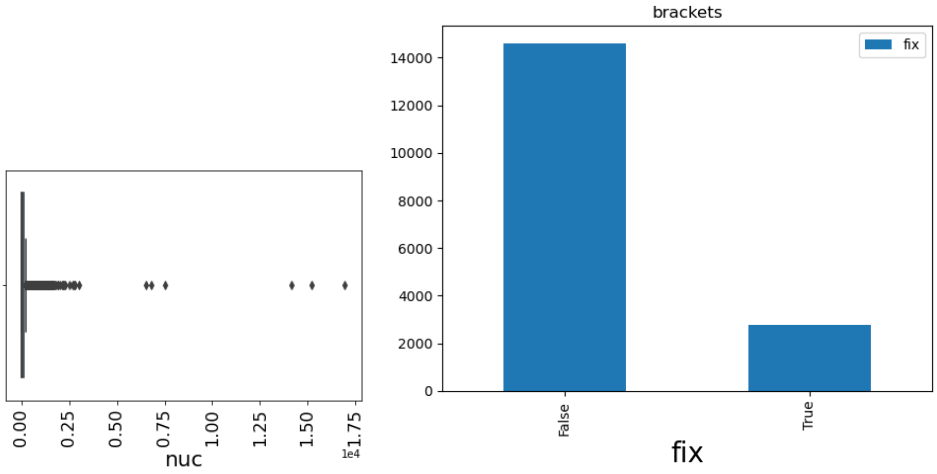
(h) sexp



(i) nd

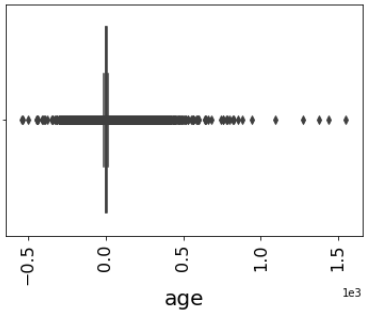


(j) ndev (k) nf (l) ns

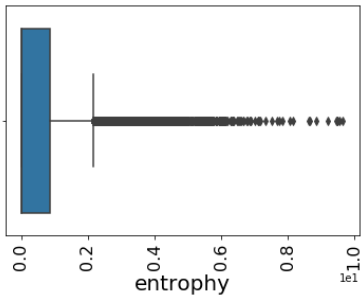


(m) nuc (n) Fix (Categorical Feature)

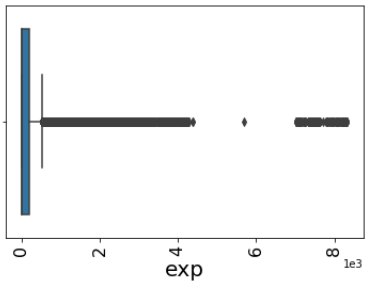
Figure B.5: Plots for numerical input features for brackets



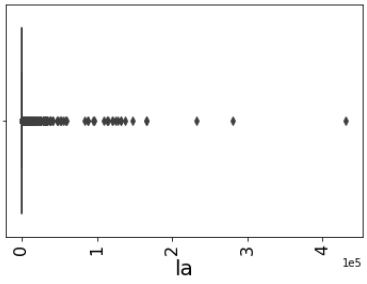
(a) age



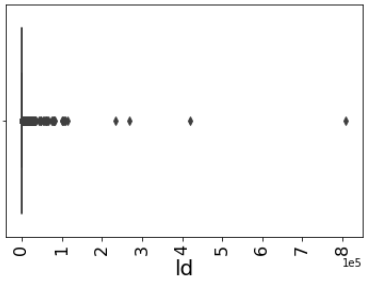
(b) entropy



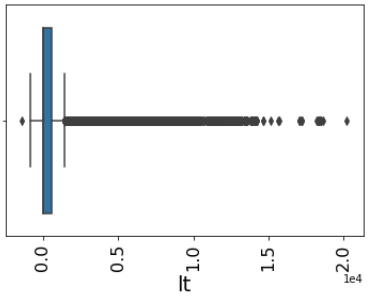
(c) exp



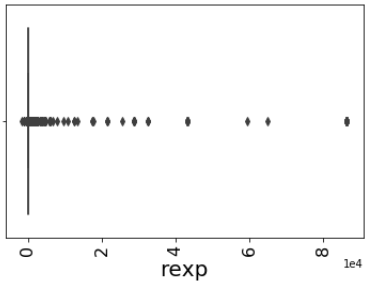
(d) la



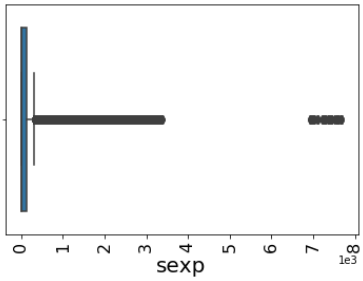
(e) ld



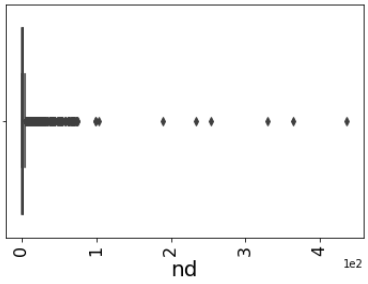
(f) lt



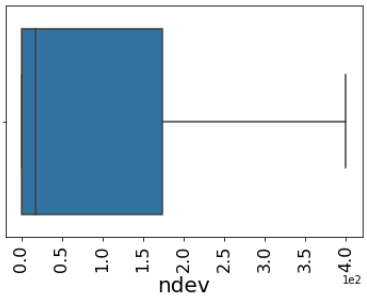
(g) rexp



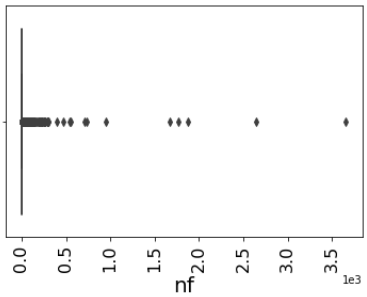
(h) sexp



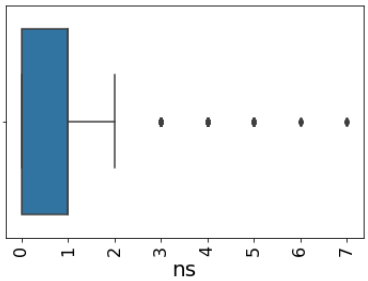
(i) nd



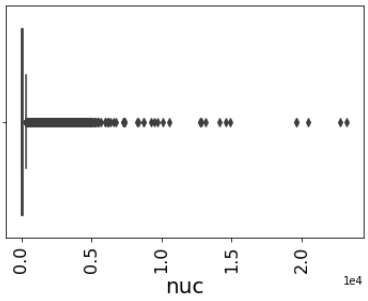
(j) ndev



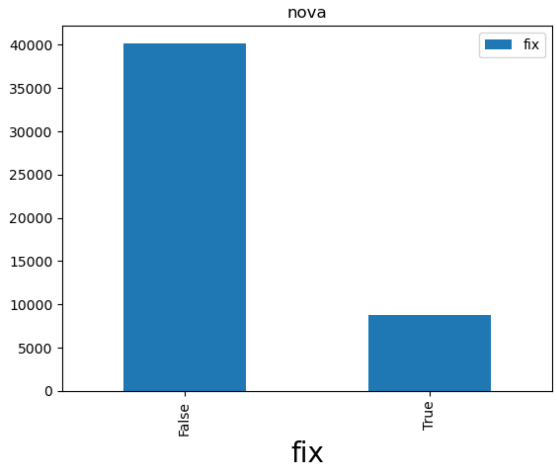
(k) nf



(l) ns

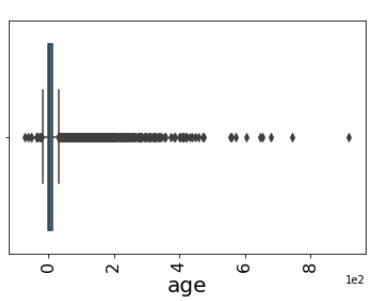


(m) nuc

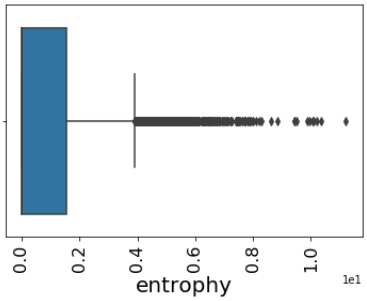


(n) Fix (Categorical Feature)

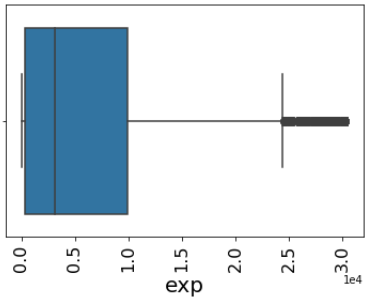
Figure B.6: Plots for numerical input features for nova



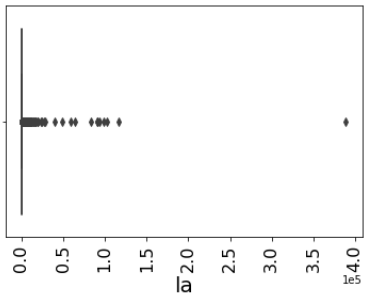
(a) age



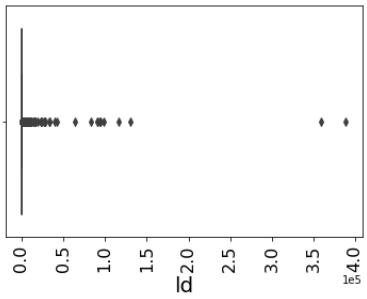
(b) entropy



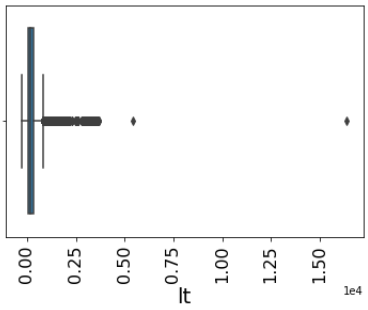
(c) exp



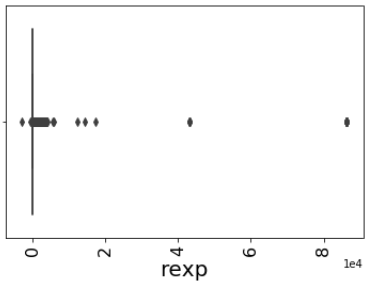
(d) la



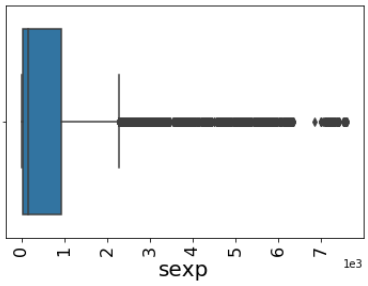
(e) ld



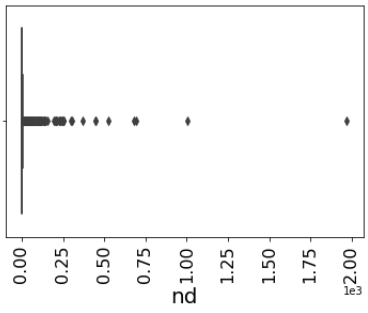
(f) lt



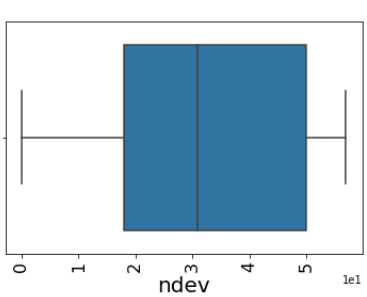
(g) rexp



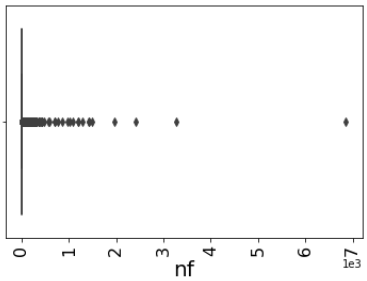
(h) sexp



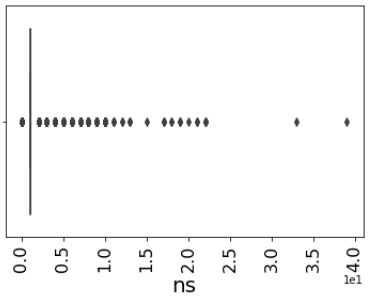
(i) nd



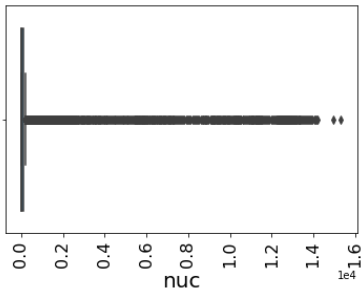
(j) ndev



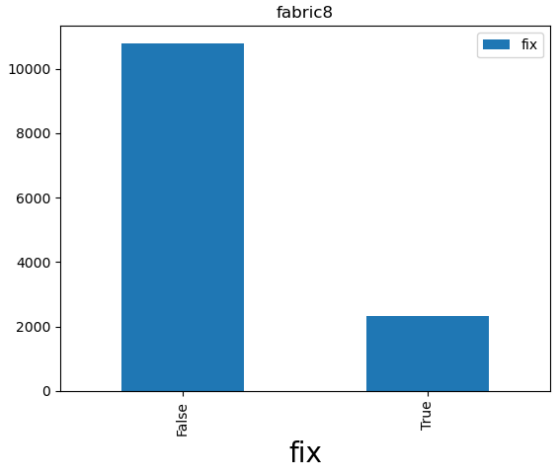
(k) nf



(l) ns

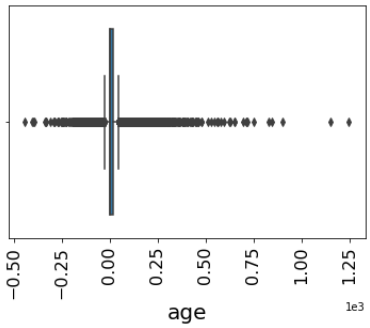


(m) nuc

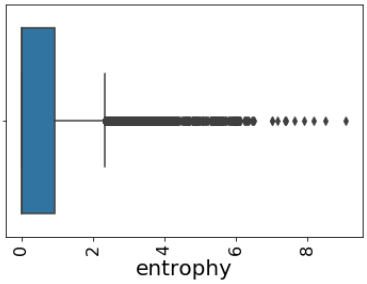


(n) Fix (Categorical Feature)

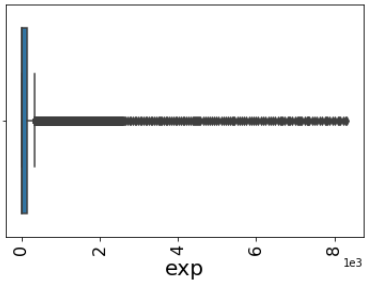
Figure B.7: Plots for numerical input features for fabric8



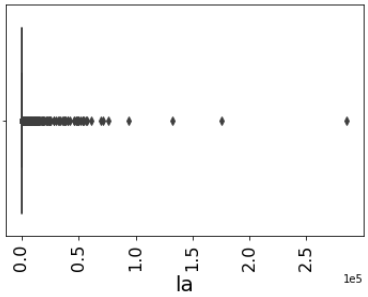
(a) age



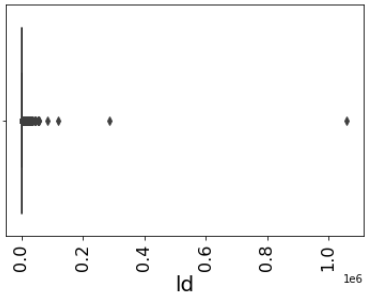
(b) entropy



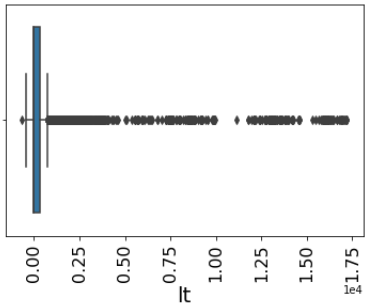
(c) exp



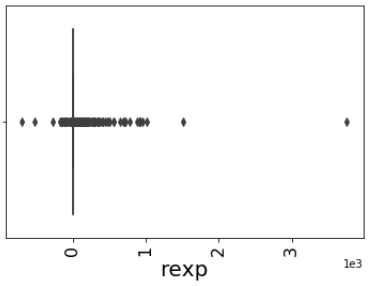
(d) la



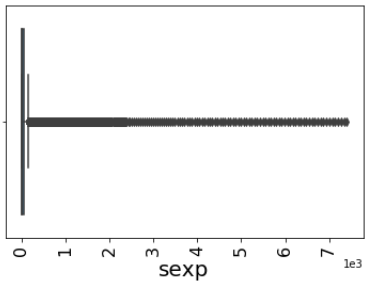
(e) ld



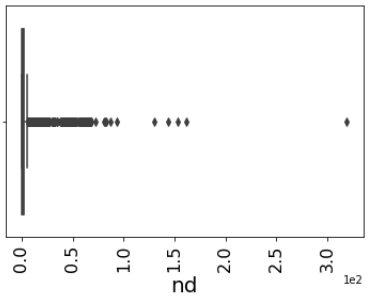
(f) lt



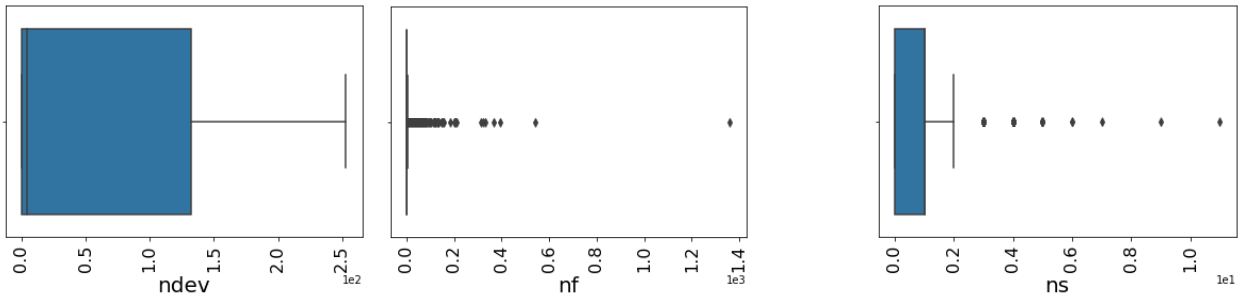
(g) rexp



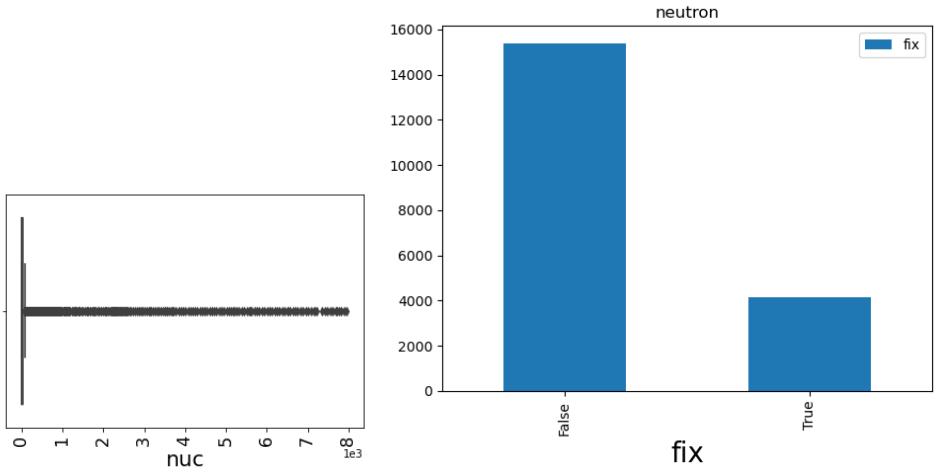
(h) sexp



(i) nd

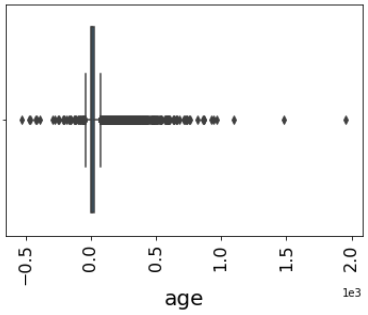


(j) ndev (k) nf (l) ns

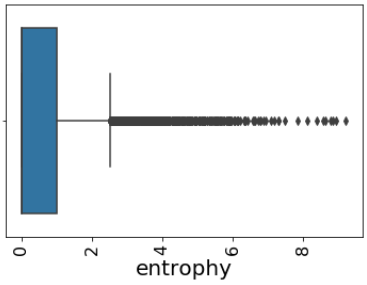


(m) nuc (n) Fix (Categorical Feature)

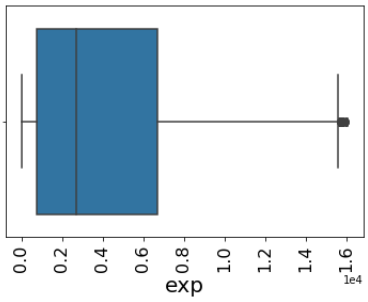
Figure B.8: Plots for numerical input features for neutron



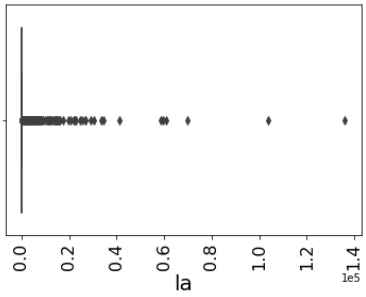
(a) age



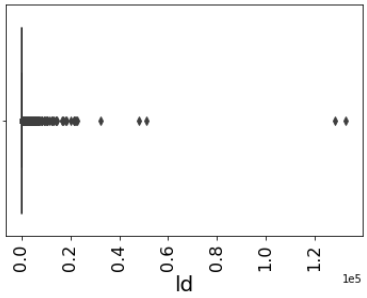
(b) entropy



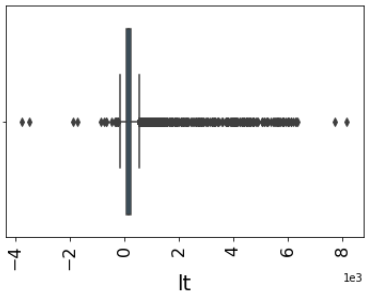
(c) exp



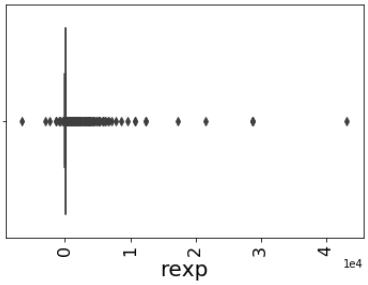
(d) la



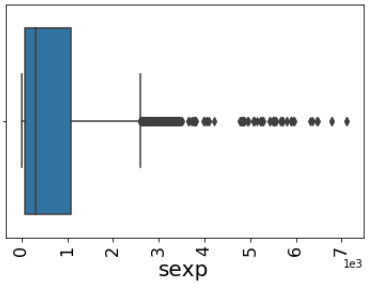
(e) ld



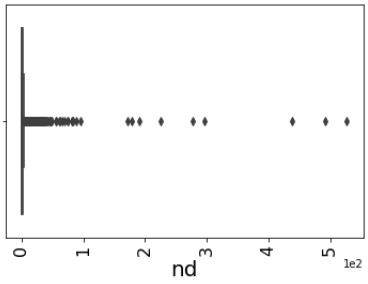
(f) lt



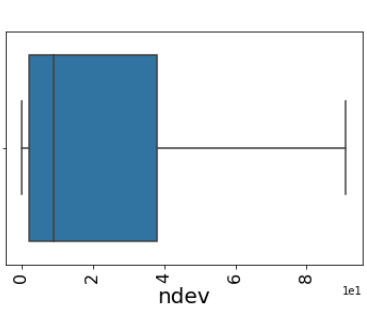
(g) rexp



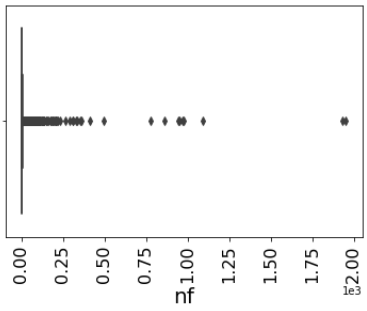
(h) sexp



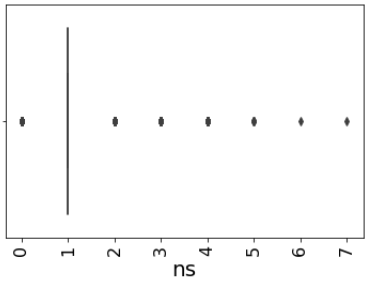
(i) nd



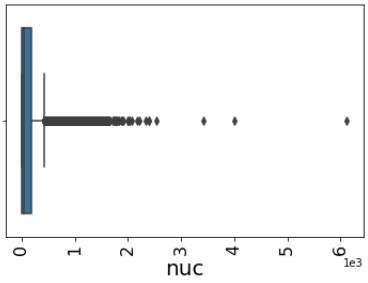
(j) ndev



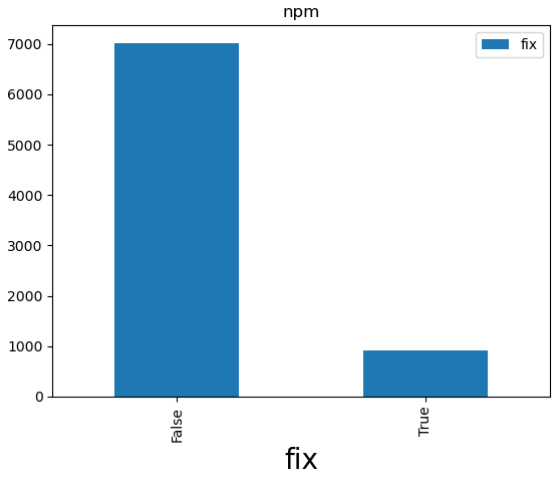
(k) nf



(l) ns

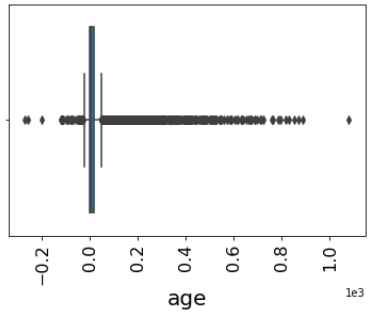


(m) nuc

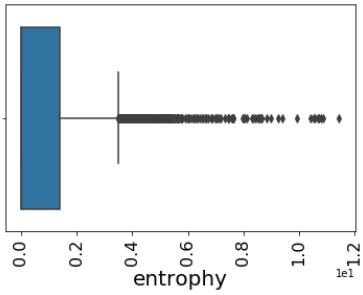


(n) Fix (Categorical Feature)

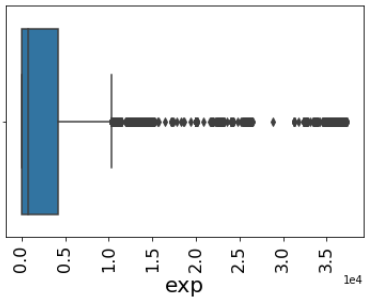
Figure B.9: Plots for numerical input features for npm



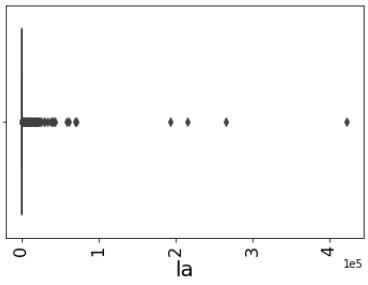
(a) age



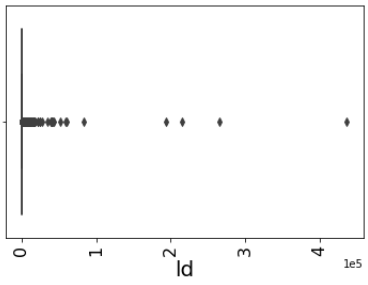
(b) entropy



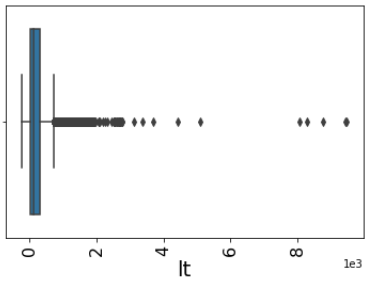
(c) exp



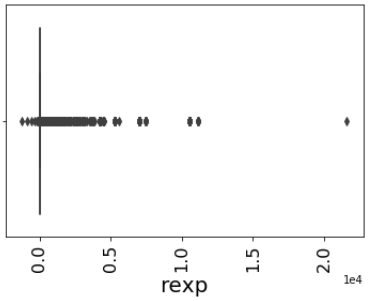
(d) la



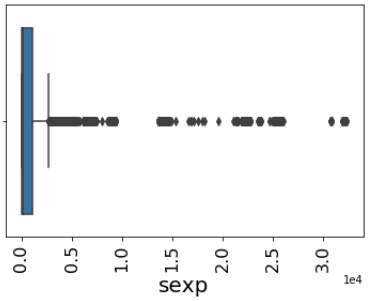
(e) ld



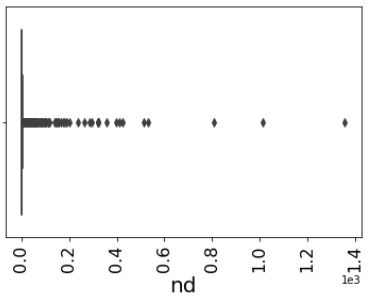
(f) lt



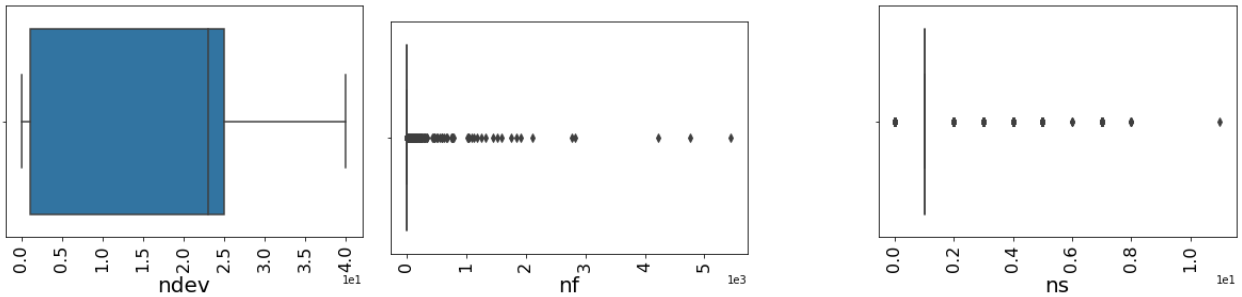
(g) rexp



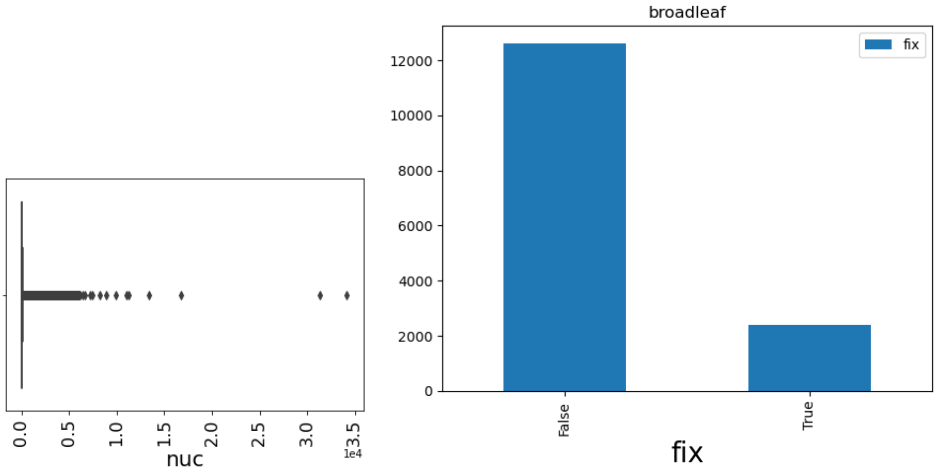
(h) sexp



(i) nd



(j) ndev (k) nf (l) ns



(m) nuc (n) Fix (Categorical Feature)

Figure B.10: Plots for numerical input features for broadleaf

Appendix Three

Supplementary Materials for Chapter Five

The contents for this appendix is based on the supplementary materials of the first and second publications (ICSE [29] and IEEE Transactions on Software Engineering [210]) listed in Chapter 1.

C.1 Tables reporting G-Mean, Recall0 and Recall1 for Chapter Five

This section provides all result tables, including tables for approaches based on OOB and ORB in Chapter 5. In addition to tables depicting the initial period, periods of sudden drop in predictive performance and stable periods, there is also a table comparing all approaches across all time steps. The tables are shown from the next page onwards, to facilitate reading.

Table C.1: Number of initial time steps of the initial phase, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB	Filter-OOB	Ensemble-OOB	OP-AIO-OOB	OP-Filter-OOB
Tomcat	2006	50	43.62(4.84)[-b]	51.95(1.25)[b]	52.5(0.71)[b]	33.67(0.63)[-b]	53.12(3.28)[b]	52.86(1.15)[b]
JGroups	1268	50	38.6(0.83)[-b]	38.29(0.84)[-b]	39.01(0.75)[-b]	16.64(0.36)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	461	50	27.3(0.12)[-b]	24.28(0.25)[-b]	38.69(0.49)[-b]	19.48(0.22)[-b]	11.15(6.13)[-b]	29.16(6.34)[-b]
Camel	3112	50	46.82(1.71)[-b]	57.75(0.74)[b]	57.07(0.82)[b]	39.42(0.52)[-b]	47.0(1.06)[-b]	46.94(0.96)[-b]
Brackets	1569	50	21.37(0.03)[-b]	64.89(1.32)[b]	66.17(0.74)[b]	46.83(0.91)[-b]	50.46(1.05)[s]	59.45(1.33)[b]
Nova	6271	50	55.42(0.41)[b]	63.15(0.68)[b]	64.39(0.34)[b]	51.66(0.29)[b]	62.55(1.16)[b]	62.88(4.43)[b]
Fabric8	795	50	27.01(0.3)[-b]	51.51(1.42)[b]	58.63(1.57)[b]	40.3(0.69)[-b]	56.57(3.12)[b]	45.52(1.08)[-b]
Neutron	917	50	44.84(5.75)[-b]	73.55(0.99)[b]	67.29(2.45)[b]	55.07(1.1)[b]	67.03(0.41)[b]	73.15(0.7)[b]
Npm	2536	50	26.92(1.1)[-b]	48.37(1.34)[-b]	45.75(1.94)[-b]	42.02(0.34)[-b]	42.89(6.01)[-b]	15.7(1.15)[-b]
BroadleafCommerce	677	50	26.37(0.26)[-b]	50.32(1.55)[*]	53.16(1.99)[b]	37.26(1.58)[-b]	40.29(2.8)[-b]	30.63(3.6)[-b]
Ranking		2	4	**1**	**1**	4	3	3

Dataset	#Time Periods	Dummy	WP-ORB	AIO-ORB	Filter-ORB	Ensemble-ORB	OP-AIO-ORB	OP-Filter-ORB
Tomcat	926	50	40.45(1.39)[-b]	45.83(1.39)[-b]	48.13(1.58)[-b]	35.32(0.46)[-b]	27.73(6.1)[-b]	32.87(8.82)[-b]
JGroups	1412	50	30.72(1.55)[-b]	31.48(1.31)[-b]	33.48(1.44)[-b]	15.14(0.79)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	900	50	25.7(0.8)[-b]	60.47(3.09)[b]	63.43(3.06)[b]	42.17(1.25)[-b]	62.77(2.92)[b]	48.32(2.29)[-b]
Camel	5111	50	49.69(1.59)[-s]	57.94(0.92)[b]	57.64(1.3)[b]	41.38(0.74)[-b]	50.32(4.15)[-s]	49.76(4.08)[-s]
Brackets	1721	50	13.5(0.32)[-b]	67.39(1.42)[b]	67.21(0.75)[b]	50.46(0.9)[b]	65.23(1.32)[b]	59.26(2.72)[b]
Nova	6271	50	52.8(1.78)[b]	66.44(0.37)[b]	65.0(0.78)[b]	51.85(0.88)[b]	34.29(14.0)[-b]	56.84(6.67)[b]
Fabric8	1613	50	29.97(0.45)[-b]	63.34(1.84)[b]	62.9(1.09)[b]	45.2(1.01)[-b]	56.59(4.16)[b]	55.11(3.05)[b]
Neutron	3304	50	71.78(1.03)[b]	75.09(1.11)[b]	74.98(1.42)[b]	71.04(1.0)[b]	63.71(13.18)[b]	70.28(4.96)[b]
Npm	1494	50	30.12(2.16)[-b]	40.07(2.86)[-b]	43.49(4.92)[-b]	43.53(1.45)[-b]	39.02(19.3)[-m]	0.0(0.0)[-b]
BroadleafCommerce	950	50	27.68(0.25)[-b]	51.82(3.62)[m]	52.41(4.96)[s]	41.69(0.85)[-b]	58.34(4.49)[b]	44.01(2.49)[-b]
Ranking		2	4	**1**	**1**	3	2	3

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. Smaller Scott-Knott.BA12 rankings are better rankings.

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

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB combined	AIO-OOB proprietary	Filter-OOB combined	OP-AIO-OOB combined	OP-AIO-OOB proprietary	OP-Filter-OOB combined
C1	284	50	15.76(4.12)[-b]	35.07(3.74)[-b]	15.97(3.9)[-b]	42.5(2.11)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	581	50	24.62(0.37)[-b]	31.03(0.43)[-b]	32.71(5.38)[-b]	46.82(0.76)[-b]	43.84(1.9)[-b]	5.62(6.26)[-b]	45.33(2.97)[-b]
C3	82	50	8.48(0.0)[-b]	23.36(0.0)[-b]	35.9(0.0)[-b]	8.62(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	24	50	7.59(0.13)[-b]	18.93(2.73)[-b]	18.63(3.19)[-b]	0.0(0.0)[-b]	20.1(0.14)[-b]	14.47(0.39)[-b]	0.0(0.0)[-b]
C5	8	50	11.62(0.0)[-b]	1.69(2.63)[-b]	31.66(4.57)[-b]	10.14(3.89)[-b]	5.16(6.01)[-b]	1.99(4.52)[-b]	39.83(0.0)[-b]
C6	20	50	3.6(0.0)[-b]	1.8(0.0)[-b]	1.8(0.0)[-b]	1.38(0.77)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C7	29	50	1.17(0.0)[-b]	42.76(2.42)[-b]	50.92(3.65)[-s]	0.0(0.0)[-b]	40.61(8.02)[-b]	23.0(10.38)[-b]	0.0(0.0)[-b]
C8	39	50	0.76(0.0)[-b]	5.87(4.91)[-b]	40.16(8.41)[-b]	0.66(0.26)[-b]	16.56(8.28)[-b]	37.57(6.39)[-b]	0.0(0.0)[-b]
C9	62	50	10.44(1.43)[-b]	15.4(6.94)[-b]	25.12(2.67)[-b]	0.55(0.0)[-b]	21.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Ranking		**1**	4	3	2	4	4	4	4

Dataset	#Time Steps	Dummy	WP-ORB	AIO-ORB combined	AIO-ORB proprietary	Filter-ORB combined	OP-AIO-ORB combined	OP-AIO-ORB proprietary	OP-Filter-ORB combined
C1	671	50	25.54(2.04)[-b]	30.77(1.43)[-b]	25.34(1.12)[-b]	35.03(2.64)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	582	50	16.67(0.51)[-b]	38.71(3.98)[-b]	39.32(3.38)[-b]	28.6(0.76)[-b]	41.93(2.97)[-b]	30.43(12.08)[-b]	40.93(5.07)[-b]
C3	396	50	38.67(0.64)[-b]	38.19(0.45)[-b]	39.31(1.26)[-b]	40.62(0.93)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	93	50	4.44(0.2)[-b]	35.18(3.62)[-b]	43.86(1.29)[-b]	23.17(4.39)[-b]	41.81(3.75)[-b]	42.81(0.97)[-b]	19.07(0.73)[-b]
C5	297	50	9.53(1.07)[-b]	23.72(4.03)[-b]	25.43(3.96)[-b]	35.72(3.56)[-b]	52.41(3.99)[s]	47.88(4.29)[-s]	54.87(2.63)[b]
C6	210	50	2.08(0.0)[-b]	31.87(0.96)[-b]	31.45(2.3)[-b]	29.12(1.21)[-b]	30.05(2.19)[-b]	11.88(0.89)[-b]	21.61(1.45)[-b]
C7	62	50	1.76(0.4)[-b]	45.34(6.83)[-b]	43.59(2.0)[-b]	14.65(1.7)[-b]	35.82(14.86)[-b]	29.86(12.14)[-b]	0.0(0.0)[-b]
C8	232	50	1.55(0.1)[-b]	48.68(5.87)[-s]	50.06(5.09)[*]	35.23(2.99)[-b]	50.43(5.74)[-s]	54.31(2.16)[b]	40.83(2.08)[-b]
C9	198	50	1.04(0.0)[-b]	29.05(2.79)[-b]	36.29(1.78)[-b]	21.21(0.02)[-b]	19.25(0.22)[-b]	1.27(4.7)[-b]	5.23(0.0)[-b]
Ranking		**1**	6	3	2	4	3	5	5

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. Smaller Scott-Knott.BA12 rankings are better rankings.

Table C.3: Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB and ORB for open source data during the drop periods

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB	Filter-OOB	Ensemble-OOB	OP-AIO-OOB	OP-Filter-OOB
Tomcat	3855	50	48.52(1.17)[-b]	58.86(0.72)[b]	60.22(1.0)[b]	52.26(0.32)[b]	36.58(6.17)[-b]	43.78(1.42)[-b]
JGroups	4353	50	43.95(1.44)[-b]	48.01(1.05)[-b]	48.08(0.89)[-b]	31.41(0.5)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	3973	50	32.98(0.7)[-b]	59.01(0.53)[b]	60.15(0.97)[b]	45.68(0.43)[-b]	52.2(5.28)[s]	34.89(4.74)[-b]
Camel	6009	50	53.15(1.68)[b]	56.17(0.98)[b]	57.0(0.92)[b]	52.4(0.41)[b]	50.16(1.49)[*]	50.67(0.69)[b]
Nova	4427	50	66.99(0.06)[b]	66.99(0.18)[b]	66.85(0.26)[b]	57.19(0.66)[b]	65.4(3.73)[b]	67.03(6.47)[b]
Fabric8	3443	50	53.79(2.18)[b]	56.34(0.76)[b]	65.57(1.04)[b]	53.98(0.4)[b]	22.94(6.32)[-b]	24.49(6.65)[-b]
Npm	1202	50	38.64(1.95)[-b]	62.97(1.47)[b]	65.37(1.25)[b]	51.13(0.62)[b]	44.21(7.42)[-s]	17.21(3.51)[-b]
BroadleafCommerce	3579	50	50.18(0.55)[s]	67.39(1.05)[b]	66.05(0.66)[b]	54.38(0.25)[b]	52.72(1.17)[b]	56.02(2.4)[b]
Ranking		3	3	2	**1**	3	4	4

Dataset	#Time Steps	Dummy	WP-ORB	AIO-ORB	Filter-ORB	Ensemble-ORB	OP-AIO-ORB	OP-Filter-ORB
Tomcat	2265	50	47.34(1.72)[-b]	56.29(1.15)[b]	56.23(1.22)[b]	51.1(0.46)[b]	25.76(8.05)[-b]	25.53(11.14)[-b]
JGroups	2066	50	52.94(1.49)[b]	53.93(1.37)[b]	55.36(1.45)[b]	34.42(0.45)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	2319	50	36.48(1.38)[-b]	64.14(0.86)[b]	65.49(1.07)[b]	47.13(0.72)[-b]	66.62(1.78)[b]	54.96(3.64)[b]
Camel	1105	50	48.98(3.85)[-m]	55.46(1.7)[b]	55.21(2.74)[b]	56.61(0.92)[b]	56.9(6.38)[b]	55.88(5.57)[b]
Nova	4408	50	67.29(0.57)[b]	67.48(0.56)[b]	66.65(0.25)[b]	61.5(0.62)[b]	20.38(18.77)[-b]	60.54(8.49)[b]
Fabric8	1473	50	55.13(5.87)[b]	57.2(1.9)[b]	63.55(0.94)[b]	55.15(1.05)[b]	51.26(6.09)[m]	44.7(4.8)[-b]
Npm	936	50	44.07(4.78)[-b]	46.78(4.55)[-b]	46.11(9.12)[-s]	50.03(0.65)[-s]	40.28(19.24)[-m]	0.0(0.0)[-b]
BroadleafCommerce	2703	50	51.72(2.03)[b]	67.03(1.11)[b]	66.98(0.98)[b]	54.38(0.65)[b]	49.33(5.1)[*]	58.01(2.88)[b]
Ranking		2	2	**1**	**1**	2	3	3

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. Smaller Scott-Knott.BA12 rankings are better rankings.

Table C.4: Total number of time steps of the performance drop periods, average G-Means, A12 effect sizes against the Dummy approach and Scott-Knott.BA12 to compare approaches using OOB and ORB for proprietary data during the drop periods

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB combined	AIO-OOB proprietary	Filter-OOB combined	OP-AIO-OOB combined	OP-AIO-OOB proprietary	OP-Filter-OOB combined
C1	452	50	32.27(4.27)[-b]	41.37(1.45)[-b]	43.4(4.45)[-b]	46.66(1.08)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C3	210	50	42.79(1.72)[-b]	47.75(1.31)[-b]	39.1(1.42)[-b]	52.17(1.3)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	249	50	33.0(0.95)[-b]	51.47(1.74)[b]	60.05(2.89)[b]	56.78(1.76)[b]	57.74(1.11)[b]	47.1(1.74)[-b]	51.35(0.0)[b]
C6	466	50	21.34(1.35)[-b]	43.64(1.49)[-b]	45.89(5.02)[-b]	44.32(1.98)[-b]	43.53(0.41)[-b]	37.47(0.18)[-b]	45.1(0.47)[-b]
C7	255	50	25.74(1.19)[-b]	41.5(5.94)[-b]	32.34(1.51)[-b]	40.34(1.02)[-b]	50.43(2.6)[-*]	36.26(0.52)[-b]	30.73(0.4)[-b]
C8	287	50	7.3(0.89)[-b]	44.65(0.96)[-b]	34.6(5.43)[-b]	40.14(1.82)[-b]	44.3(1.81)[-b]	45.49(9.63)[*]	25.57(8.7)[-b]
C9	541	50	31.75(1.77)[-b]	51.34(1.76)[b]	50.08(2.77)[s]	53.25(1.58)[b]	40.3(5.22)[-b]	34.92(3.87)[-b]	34.36(0.55)[-b]
Ranking		**1**	5	3	4	2	4	5	5

Dataset	#Time Steps	Dummy	WP-ORB	AIO-ORB combined	AIO-ORB proprietary	Filter-ORB combined	OP-AIO-ORB combined	OP-AIO-ORB proprietary	OP-Filter-ORB combined
C1	238	50	41.17(6.37)[-b]	37.26(5.59)[-b]	28.75(3.01)[-b]	47.3(6.34)[-s]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C6	64	50	35.38(0.91)[-b]	50.99(7.22)[m]	35.67(9.61)[-b]	44.61(5.36)[-b]	61.13(3.51)[b]	46.51(5.76)[-m]	42.82(5.16)[-b]
C7	85	50	31.09(4.47)[-b]	32.33(10.88)[-b]	50.09(13.02)[m]	59.45(6.92)[b]	35.64(17.93)[-s]	41.39(2.73)[-b]	0.0(0.0)[-b]
C8	80	50	20.25(1.22)[-b]	39.45(14.5)[-s]	37.97(14.69)[-s]	50.59(6.1)[b]	47.77(6.08)[-s]	32.34(8.23)[-b]	45.77(2.75)[-b]
C9	23	50	23.59(1.26)[-b]	39.88(7.47)[-b]	25.48(4.84)[-b]	20.64(1.48)[-b]	43.2(8.64)[-b]	1.64(3.75)[-b]	17.96(0.0)[-b]
Ranking		**1**	4	3	3	2	3	4	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. Smaller Scott-Knott.BA12 rankings are better rankings.

Table C.5: Number of time steps of the stable periods, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this stable periods for OOB and ORB for open source data

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB	Filter-OOB	Ensemble-OOB	OP-AIO-OOB	OP-Filter-OOB
Tomcat	13046	50	62.69(0.81)[b]	61.36(0.5)[b]	63.36(0.64)[b]	50.57(0.22)[b]	39.85(6.0)[-b]	45.14(1.49)[-b]
JGroups	12704	50	60.08(0.66)[b]	58.74(0.72)[b]	58.51(0.54)[b]	38.8(0.3)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	4316	50	64.86(1.01)[b]	65.99(0.35)[b]	66.96(0.77)[b]	56.02(0.33)[b]	55.71(8.02)[b]	45.38(8.91)[-*]
Camel	21454	50	67.33(0.56)[b]	64.48(0.67)[b]	65.05(0.62)[b]	54.61(0.35)[b]	41.5(1.52)[-b]	41.54(1.25)[-b]
Brackets	15795	50	68.27(0.12)[b]	71.44(0.48)[b]	71.38(0.42)[b]	64.25(0.54)[b]	39.05(12.78)[-b]	66.69(0.34)[b]
Nova	38291	50	79.84(0.26)[b]	82.03(0.21)[b]	81.54(0.2)[b]	68.72(0.79)[b]	73.83(16.61)[b]	75.2(0.5)[b]
Fabric8	8868	50	65.0(0.77)[b]	63.53(0.62)[b]	67.64(0.64)[b]	54.15(0.4)[b]	25.97(4.87)[-b]	28.17(4.46)[-b]
Neutron	18605	50	81.13(0.27)[b]	81.87(0.23)[b]	82.22(0.25)[b]	68.93(1.5)[b]	74.49(1.77)[b]	71.43(3.91)[b]
Npm	4182	50	59.52(1.71)[b]	64.96(0.63)[b]	66.49(0.99)[b]	55.3(0.46)[b]	29.41(2.02)[-b]	40.07(1.13)[-b]
BroadleafCommerce	10754	50	65.96(1.12)[b]	70.73(0.62)[b]	70.13(0.55)[b]	59.55(0.2)[b]	54.3(0.78)[b]	60.4(1.06)[b]
Ranking		4	2	**1**	**1**	3	5	4

Dataset	#Time Steps	Dummy	WP-ORB	AIO-ORB	Filter-ORB	Ensemble-ORB	OP-AIO-ORB	OP-Filter-ORB
Tomcat	15716	50	62.71(0.97)[b]	63.44(0.86)[b]	62.74(0.57)[b]	51.69(0.43)[b]	45.39(1.76)[-b]	37.75(2.12)[-b]
JGroups	14847	50	60.31(0.8)[b]	60.99(0.67)[b]	61.06(0.74)[b]	39.88(0.34)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	5531	50	62.36(0.94)[b]	70.48(0.7)[b]	70.75(0.41)[b]	56.27(0.61)[b]	68.57(2.2)[b]	58.38(4.57)[b]
Camel	24359	50	66.33(1.02)[b]	66.92(0.39)[b]	66.34(0.34)[b]	56.85(0.55)[b]	51.0(3.63)[b]	45.33(4.33)[-b]
Brackets	15643	50	68.79(0.53)[b]	70.26(1.01)[b]	70.07(0.83)[b]	66.28(0.79)[b]	64.33(0.62)[b]	64.42(1.56)[b]
Nova	38310	50	80.06(0.81)[b]	80.18(0.41)[b]	79.94(0.38)[b]	76.12(0.68)[b]	38.93(30.48)[*]	76.85(2.32)[b]
Fabric8	10020	50	67.41(0.81)[b]	64.72(1.01)[b]	68.72(0.59)[b]	57.74(0.6)[b]	49.22(6.71)[-s]	38.08(5.44)[-b]
Neutron	16218	50	81.09(0.63)[b]	81.87(0.39)[b]	81.83(0.51)[b]	74.93(1.23)[b]	67.06(12.37)[b]	70.65(7.2)[b]
Npm	5490	50	61.92(0.9)[b]	67.72(0.88)[b]	65.63(1.3)[b]	55.23(0.64)[b]	51.29(19.52)[b]	0.0(0.0)[-b]
BroadleafCommerce	11357	50	64.31(1.34)[b]	69.39(0.6)[b]	68.9(0.78)[b]	61.46(0.49)[b]	41.81(8.62)[-b]	49.32(2.91)[-s]
Ranking		4	2	**1**	**1**	3	4	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. Smaller Scott-Knott.BA12 rankings are better rankings.

Table C.6: Number of time steps of the stable periods, and average G-Means, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this stable periods for OOB and ORB for proprietary data

Dataset	#Time Steps	Dummy	WP-OOB	AIO-OOB combined	AIO-OOB proprietary	Filter-OOB combined	OP-AIO-OOB combined	OP-AIO-OOB proprietary	OP-Filter-OOB combined
C1	294	50	49.55(1.47)[-*]	43.65(2.33)[-b]	43.4(6.94)[-b]	45.75(2.02)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	20	50	64.59(0.23)[b]	52.86(1.75)[b]	55.58(2.72)[b]	59.42(0.62)[b]	36.49(0.93)[-b]	2.8(3.22)[-b]	37.36(1.46)[-b]
C3	125	50	54.54(1.05)[b]	54.01(0.43)[b]	33.69(2.6)[-b]	57.0(0.61)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	68	50	45.72(1.21)[-b]	53.4(1.23)[b]	56.98(4.85)[b]	27.16(8.92)[-b]	46.76(0.61)[-b]	44.62(2.6)[-b]	21.15(0.0)[-b]
C5	315	50	23.2(0.43)[-b]	46.55(1.35)[-b]	42.6(0.99)[-b]	49.93(1.71)[-s]	44.14(5.2)[-b]	49.53(2.92)[-s]	41.15(4.14)[-b]
C6	69	50	41.71(0.64)[-b]	43.05(0.65)[-b]	49.84(2.73)[-*]	41.83(3.36)[-b]	24.67(0.34)[-b]	14.76(0.07)[-b]	21.1(0.26)[-b]
C7	262	50	45.55(1.0)[-b]	52.02(5.6)[s]	52.62(2.11)[b]	52.35(2.27)[b]	57.67(1.5)[b]	52.6(1.18)[b]	40.38(0.7)[-b]
C8	472	50	26.13(0.22)[-b]	55.16(0.91)[b]	47.01(3.71)[-b]	40.38(1.65)[-b]	45.29(1.94)[-b]	48.99(5.2)[s]	41.14(2.6)[-b]
C9	91	50	49.63(1.59)[-b]	56.46(2.26)[b]	59.12(6.12)[b]	46.6(2.06)[-b]	29.03(2.98)[-b]	8.62(1.08)[-b]	14.33(2.11)[-b]
Ranking		2	3	**1**	2	3	4	5	5

Dataset	#Time Steps	Dummy	WP-ORB	AIO-ORB combined	AIO-ORB proprietary	Filter-ORB combined	OP-AIO-ORB combined	OP-AIO-ORB proprietary	OP-Filter-ORB combined
C1	121	50	48.52(2.36)[-b]	46.19(4.31)[-b]	37.82(3.52)[-b]	54.01(3.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	19	50	49.05(1.07)[-b]	61.06(1.23)[b]	47.53(6.61)[-b]	58.16(0.66)[b]	36.04(3.44)[-b]	35.0(13.46)[-b]	37.85(3.21)[-b]
C3	21	50	61.11(1.36)[b]	60.52(1.53)[b]	62.7(1.89)[b]	61.9(1.38)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	248	50	52.68(1.35)[b]	41.8(5.78)[-b]	46.85(3.7)[-b]	56.44(1.82)[b]	53.88(2.62)[b]	43.79(1.04)[-b]	66.78(4.59)[b]
C5	26	50	42.31(8.55)[-b]	56.2(1.44)[b]	56.09(1.8)[b]	54.92(3.15)[b]	57.76(7.76)[b]	47.64(4.51)[-b]	53.59(5.68)[m]
C6	281	50	51.77(0.51)[b]	53.76(2.24)[b]	52.25(3.62)[b]	51.32(1.58)[b]	62.91(2.09)[b]	49.96(2.24)[s]	54.29(1.24)[b]
C7	399	50	46.48(2.58)[-b]	47.52(8.24)[-s]	56.43(6.65)[b]	60.96(2.67)[b]	31.51(15.32)[-b]	35.19(10.53)[-b]	0.0(0.0)[-b]
C8	486	50	32.08(0.62)[-b]	45.61(11.25)[*]	42.57(13.67)[*]	48.0(2.75)[-b]	49.95(5.09)[-s]	40.18(6.55)[-b]	48.24(2.34)[-b]
C9	473	50	34.92(0.48)[-b]	38.53(2.39)[-b]	40.0(4.65)[-b]	34.07(1.49)[-b]	36.98(6.48)[-b]	2.06(4.74)[-b]	18.38(0.0)[-b]
Ranking		2	3	2	2	**1**	4	4	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the corresponding Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Dummy approach. Smaller Scott-Knott.BA12 rankings are better rankings.

Table C.7: Overall predictive performance, A12 effect sizes and Scott-Knott.BA12 statistical tests to compare learning approaches for proprietary data

Dataset	Approach	Recall0	Recall1	G-Mean
C1	Dummy	50	50	50
	WP-OOB	59.25(4.79)[b]	39.37(3.88)[-b]	32.65(1.53)[-b]
	AIO-OOB-combined	51.6(3.65)[b]	48.3(2.55)[-b]	40.28(1.53)[-b]
	AIO-OOB-proprietary	31.75(8.46)[-b]	65.14(7.2)[b]	35.84(3.45)[-b]
	Filter-OOB-combined	53.56(3.43)[b]	49.18(3.1)[-s]	45.25(1.17)[-b]
	OP-AIO-OOB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-AIO-OOB-proprietary	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-Filter-OOB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	Dummy	50	50	50
	WP-OOB	16.33(0.12)[-b]	87.15(0.0)[b]	25.95(0.37)[-b]
	AIO-OOB-combined	22.23(0.53)[-b]	76.41(0.42)[b]	31.76(0.45)[-b]
	AIO-OOB-proprietary	45.82(7.4)[-b]	49.25(10.34)[m]	33.47(5.27)[-b]
	Filter-OOB-combined	45.22(1.27)[-b]	63.81(1.77)[b]	47.24(0.74)[-b]
	OP-AIO-OOB-combined	43.5(4.07)[-b]	55.23(3.49)[b]	43.6(1.86)[-b]
	OP-AIO-OOB-proprietary	98.4(2.76)[b]	1.54(1.88)[-b]	5.53(6.16)[-b]
	OP-Filter-OOB-combined	47.57(5.6)[-s]	55.6(4.04)[b]	45.06(2.91)[-b]
C3	Dummy	50	50	50
	WP-OOB	59.78(4.95)[b]	38.31(4.5)[-b]	39.57(1.1)[-b]
	AIO-OOB-combined	41.52(0.84)[-b]	69.57(1.43)[b]	44.83(0.71)[-b]
	AIO-OOB-proprietary	26.4(2.25)[-b]	66.45(3.09)[b]	36.85(1.35)[-b]
	Filter-OOB-combined	52.59(1.18)[b]	59.66(0.99)[b]	45.05(0.82)[-b]
	OP-AIO-OOB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-AIO-OOB-proprietary	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-Filter-OOB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	Dummy	50	50	50
	WP-OOB	21.56(0.74)[-b]	81.96(0.66)[b]	33.75(0.86)[-b]
	AIO-OOB-combined	37.79(1.76)[-b]	76.64(1.34)[b]	49.57(1.45)[-s]
	AIO-OOB-proprietary	66.59(6.0)[b]	55.44(8.47)[m]	56.52(3.02)[b]
	Filter-OOB-combined	61.15(3.35)[b]	54.41(2.64)[b]	46.88(2.53)[-b]
	OP-AIO-OOB-combined	50.98(1.91)[b]	61.59(1.59)[b]	52.9(0.9)[b]
	OP-AIO-OOB-proprietary	40.49(3.43)[-b]	62.7(3.37)[b]	44.31(1.36)[-b]
	OP-Filter-OOB-combined	49.25(0.0)[-b]	61.33(0.0)[b]	41.71(0.0)[-b]
C5	Dummy	50	50	50
	WP-OOB	11.93(0.67)[-b]	85.44(1.43)[b]	22.91(0.42)[-b]
	AIO-OOB-combined	34.86(1.67)[-b]	67.71(3.08)[b]	45.44(1.35)[-b]
	AIO-OOB-proprietary	22.52(1.1)[-b]	90.05(1.31)[b]	42.33(0.98)[-b]
	Filter-OOB-combined	38.55(2.54)[-b]	67.07(4.08)[b]	48.94(1.67)[-b]
	OP-AIO-OOB-combined	76.39(3.78)[b]	26.8(5.51)[-b]	43.17(5.15)[-b]
	OP-AIO-OOB-proprietary	42.76(5.1)[-b]	67.8(4.34)[b]	48.35(2.86)[-m]
	OP-Filter-OOB-combined	23.54(6.53)[-b]	82.43(6.63)[b]	41.11(4.03)[-b]
C6	Dummy	50	50	50
	WP-OOB	10.98(0.58)[-b]	87.91(0.56)[b]	23.23(1.17)[-b]
	AIO-OOB-combined	33.15(1.82)[-b]	69.53(1.63)[b]	42.06(1.29)[-b]
	AIO-OOB-proprietary	40.29(10.26)[-b]	64.4(9.58)[b]	44.79(4.18)[-b]
	Filter-OOB-combined	35.22(2.81)[-b]	69.57(2.4)[b]	42.47(1.85)[-b]
	OP-AIO-OOB-combined	48.6(1.25)[-b]	56.66(0.81)[b]	39.62(0.37)[-b]
	OP-AIO-OOB-proprietary	70.97(0.35)[b]	30.75(0.28)[-b]	33.3(0.15)[-b]
	OP-Filter-OOB-combined	54.93(1.23)[b]	48.66(0.77)[-b]	40.49(0.42)[-b]
C7	Dummy	50	50	50
	WP-OOB	22.78(0.75)[-b]	83.52(0.51)[b]	33.94(0.92)[-b]
	AIO-OOB-combined	38.25(12.04)[-b]	70.7(7.58)[b]	46.62(5.4)[-m]
	AIO-OOB-proprietary	23.12(1.65)[-b]	93.33(1.26)[b]	43.06(1.67)[-b]
	Filter-OOB-combined	44.7(2.1)[-b]	61.98(3.79)[b]	43.96(1.28)[-b]
	OP-AIO-OOB-combined	77.52(2.22)[b]	38.87(2.4)[-b]	53.38(1.97)[b]
	OP-AIO-OOB-proprietary	34.57(1.17)[-b]	69.48(2.97)[b]	43.4(0.71)[-b]
	OP-Filter-OOB-combined	26.99(0.75)[-b]	65.72(0.78)[b]	33.73(0.52)[-b]
C8	Dummy	50	50	50
	WP-OOB	13.04(0.1)[-b]	83.1(0.01)[b]	18.12(0.43)[-b]
	AIO-OOB-combined	46.57(1.22)[-b]	60.3(1.79)[b]	48.97(0.81)[-b]
	AIO-OOB-proprietary	29.57(4.44)[-b]	74.84(4.67)[b]	42.21(4.03)[-b]
	Filter-OOB-combined	46.6(1.38)[-b]	48.41(1.65)[-b]	38.35(1.15)[-b]
	OP-AIO-OOB-combined	66.69(2.8)[b]	34.9(2.52)[-b]	43.53(1.52)[-b]
	OP-AIO-OOB-proprietary	77.63(10.75)[b]	32.16(10.48)[-b]	47.17(6.24)[*]
	OP-Filter-OOB-combined	24.94(3.74)[-b]	74.12(2.55)[b]	33.53(4.49)[-b]
C9	Dummy	50	50	50
	WP-OOB	20.64(1.34)[-b]	86.52(0.42)[b]	32.19(1.44)[-b]
	AIO-OOB-combined	44.53(2.53)[-b]	69.19(3.39)[b]	48.8(1.44)[-b]
	AIO-OOB-proprietary	59.51(4.62)[b]	52.23(7.1)[s]	49.04(2.87)[-s]
	Filter-OOB-combined	52.2(2.63)[b]	58.09(2.72)[b]	47.67(1.32)[-b]
	OP-AIO-OOB-combined	59.8(7.99)[b]	41.63(6.31)[-b]	37.1(4.46)[-b]
	OP-AIO-OOB-proprietary	93.81(5.79)[b]	13.24(5.36)[-b]	28.35(3.15)[-b]
	OP-Filter-OOB-combined	52.0(0.16)[b]	46.24(0.5)[-b]	28.67(0.68)[-b]
Ranking	Dummy	2	4	**1**
	WP-OOB	5	**1**	5
	AIO-OOB-combined	4	2	2
	AIO-OOB-proprietary	4	2	3
	Filter-OOB-combined	3	3	2
	OP-AIO-OOB-combined	**1**	5	3
	OP-AIO-OOB-proprietary	**1**	5	5
	OP-Filter-OOB-combined	2	4	5

Dataset	Approach	Recall0	Recall1	G-Mean
C1	Dummy	50	50	50
	WP-ORB	64.69(5.61)[b]	30.75(3.94)[-b]	31.85(2.57)[-b]
	AIO-ORB-combined	45.4(6.03)[-b]	51.85(6.11)[b]	34.08(2.4)[-b]
	AIO-ORB-proprietary	35.64(1.4)[-b]	57.67(1.51)[b]	27.59(1.45)[-b]
	Filter-ORB-combined	69.56(8.37)[b]	37.95(8.06)[-b]	40.1(2.02)[-b]
	OP-AIO-ORB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-AIO-ORB-proprietary	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-Filter-ORB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	Dummy	50	50	50
	WP-ORB	71.46(1.01)[b]	17.49(0.83)[-b]	17.7(0.51)[-b]
	AIO-ORB-combined	35.88(3.47)[-b]	60.89(3.72)[b]	39.41(3.86)[-b]
	AIO-ORB-proprietary	38.15(6.8)[-b]	55.99(9.41)[b]	39.58(3.32)[-b]
	Filter-ORB-combined	46.15(0.69)[-b]	43.99(0.7)[-b]	29.53(0.75)[-b]
	OP-AIO-ORB-combined	82.97(2.67)[b]	25.75(4.38)[-b]	41.74(2.98)[-b]
	OP-AIO-ORB-proprietary	87.63(8.27)[b]	15.2(9.77)[-b]	30.58(12.08)[-b]
	OP-Filter-ORB-combined	89.53(3.42)[b]	23.13(6.32)[-b]	40.83(4.99)[-b]
C3	Dummy	50	50	50
	WP-ORB	58.67(1.05)[b]	49.83(0.93)[-*]	39.8(0.65)[-b]
	AIO-ORB-combined	57.37(0.69)[b]	50.82(1.05)[b]	39.31(0.47)[-b]
	AIO-ORB-proprietary	60.62(2.22)[b]	48.21(2.01)[-b]	40.49(1.24)[-b]
	Filter-ORB-combined	63.24(2.13)[b]	47.2(1.57)[-b]	41.69(0.94)[-b]
	OP-AIO-ORB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-AIO-ORB-proprietary	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
	OP-Filter-ORB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	Dummy	50	50	50
	WP-ORB	68.92(4.34)[b]	39.78(6.02)[-b]	39.52(1.0)[-b]
	AIO-ORB-combined	29.51(7.55)[-b]	78.37(5.51)[b]	40.0(4.41)[-b]
	AIO-ORB-proprietary	36.53(3.6)[-b]	77.1(1.65)[b]	46.04(2.97)[-b]
	Filter-ORB-combined	75.46(2.74)[b]	39.43(2.26)[-b]	47.36(2.04)[-b]
	OP-AIO-ORB-combined	49.04(7.88)[-s]	63.83(10.56)[b]	50.59(2.3)[s]
	OP-AIO-ORB-proprietary	36.33(0.91)[-b]	71.6(0.54)[b]	43.52(0.84)[-b]
	OP-Filter-ORB-combined	89.73(1.63)[b]	40.73(5.87)[-b]	53.77(3.53)[b]
C5	Dummy	50	50	50
	WP-ORB	95.92(0.64)[b]	5.34(1.4)[-b]	12.17(1.62)[-b]
	AIO-ORB-combined	12.04(1.74)[-b]	91.14(0.66)[b]	26.34(3.69)[-b]
	AIO-ORB-proprietary	12.84(2.2)[-b]	90.98(2.36)[b]	27.9(3.7)[-b]
	Filter-ORB-combined	21.94(3.44)[-b]	80.9(2.82)[b]	37.26(3.41)[-b]
	OP-AIO-ORB-combined	47.65(5.97)[-s]	65.06(8.41)[b]	52.84(4.12)[m]
	OP-AIO-ORB-proprietary	35.59(10.02)[-b]	74.15(9.73)[b]	47.86(4.22)[-s]
	OP-Filter-ORB-combined	44.7(4.24)[-b]	72.84(3.68)[b]	54.77(2.56)[b]
C6	Dummy	50	50	50
	WP-ORB	63.74(0.42)[b]	42.34(0.41)[-b]	31.08(0.35)[-b]
	AIO-ORB-combined	52.93(5.44)[b]	51.18(5.37)[s]	45.16(1.91)[-b]
	AIO-ORB-proprietary	52.44(5.1)[*]	55.29(2.69)[b]	42.47(3.27)[-b]
	Filter-ORB-combined	56.47(6.7)[m]	42.3(7.2)[-m]	42.15(1.66)[-b]
	OP-AIO-ORB-combined	74.96(2.93)[b]	43.37(2.3)[-b]	50.27(1.79)[s]
	OP-AIO-ORB-proprietary	76.08(8.34)[b]	27.62(7.76)[-b]	35.15(1.62)[-b]
	OP-Filter-ORB-combined	61.99(7.35)[b]	46.38(7.38)[*]	40.6(1.39)[-b]
C7	Dummy	50	50	50
	WP-ORB	48.03(5.25)[-b]	56.78(3.82)[b]	39.01(2.53)[-b]
	AIO-ORB-combined	34.47(12.93)[-b]	77.01(9.77)[b]	44.91(7.51)[-b]
	AIO-ORB-proprietary	45.97(11.53)[s]	71.25(7.13)[b]	53.99(6.85)[m]
	Filter-ORB-combined	64.89(6.69)[b]	56.2(3.68)[b]	55.47(3.0)[b]
	OP-AIO-ORB-combined	56.03(38.06)[*]	50.6(37.75)[*]	32.64(14.9)[-b]
	OP-AIO-ORB-proprietary	35.77(18.96)[-b]	61.18(21.41)[b]	35.55(7.7)[-b]
	OP-Filter-ORB-combined	100.0(0.0)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C8	Dummy	50	50	50
	WP-ORB	44.61(0.43)[-b]	43.37(0.59)[-b]	22.02(0.47)[-b]
	AIO-ORB-combined	37.94(14.26)[-b]	70.69(10.81)[b]	45.89(8.87)[-s]
	AIO-ORB-proprietary	37.06(16.4)[-b]	72.45(12.06)[b]	44.29(10.43)[-s]
	Filter-ORB-combined	55.72(6.29)[b]	44.31(4.55)[-b]	44.55(2.25)[-b]
	OP-AIO-ORB-combined	74.41(8.76)[b]	37.11(8.19)[-b]	49.87(5.01)[-s]
	OP-AIO-ORB-proprietary	89.26(3.52)[b]	24.95(5.54)[-b]	43.5(5.18)[-b]
	OP-Filter-ORB-combined	79.73(3.35)[b]	29.25(2.94)[-b]	45.84(2.09)[-b]
C9	Dummy	50	50	50
	WP-ORB	45.65(0.25)[-b]	50.03(0.09)[b]	24.88(0.36)[-b]
	AIO-ORB-combined	39.34(2.08)[-b]	63.49(1.75)[b]	35.87(1.7)[-b]
	AIO-ORB-proprietary	44.37(5.62)[-m]	61.6(4.38)[b]	38.46(3.49)[-b]
	Filter-ORB-combined	36.93(1.02)[-b]	64.24(1.19)[b]	29.96(1.05)[-b]
	OP-AIO-ORB-combined	84.62(5.2)[b]	15.86(4.1)[-b]	32.13(4.76)[-b]
	OP-AIO-ORB-proprietary	99.9(0.31)[b]	0.29(0.8)[-b]	1.82(4.42)[-b]
	OP-Filter-ORB-combined	99.96(0.0)[b]	2.71(0.0)[-b]	14.61(0.0)[-b]
Ranking	Dummy	2	4	**1**
	WP-ORB	**1**	5	5
	AIO-ORB-combined	4	2	4
	AIO-ORB-proprietary	4	2	3
	Filter-ORB-combined	2	4	3
	OP-AIO-ORB-combined	**1**	5	3
	OP-AIO-ORB-proprietary	**1**	5	5
	OP-Filter-ORB-combined	**1**	6	4

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Dummy approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the corresponding Dummy approach. Scott-Knott.BA12 was run for all OOB- and ORB-based approaches together. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

Table C.8: Number of initial time steps of the initial phase, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data

Dataset	#Time Steps	Dummy	WP-OOB_r0	AIO-OOB_r0	Filter-OOB_r0	Ensemble-OOB_r0	OP-AIO-OOB_r0	OP-Filter-OOB_r0
Tomcat	2006	50	76.66(5.97)[b]	49.86(1.86)[*]	61.48(2.4)[b]	87.03(0.61)[b]	49.68(8.93)[-s]	50.51(4.36)[-s]
JGroups	1268	50	64.66(0.69)[b]	64.54(0.6)[b]	63.9(1.07)[b]	96.41(0.1)[b]	100.0(0.0)[b]	100.0(0.0)[b]
Spring-Integration	461	50	44.11(0.12)[-b]	82.6(0.65)[b]	89.71(0.87)[b]	93.25(0.28)[b]	94.09(1.18)[b]	93.59(1.02)[b]
Camel	3112	50	28.71(3.01)[-b]	55.13(1.48)[b]	56.56(1.34)[b]	84.71(0.53)[b]	38.99(1.72)[-b]	38.12(1.2)[-b]
Brackets	1569	50	15.37(0.04)[-b]	63.02(1.8)[b]	65.67(0.85)[b]	79.59(0.64)[b]	58.72(2.62)[b]	61.6(1.36)[b]
Nova	6271	50	40.64(0.34)[-b]	56.53(1.1)[b]	59.92(0.65)[b]	72.22(0.93)[b]	49.63(3.54)[-b]	66.27(4.6)[b]
Fabric8	795	50	20.81(0.24)[-b]	82.77(1.63)[b]	74.36(2.21)[b]	75.59(1.81)[b]	48.5(5.7)[-*]	72.31(3.69)[b]
Neutron	917	50	31.13(4.78)[-b]	76.39(2.09)[b]	83.57(2.48)[b]	78.45(1.24)[b]	55.59(0.31)[b]	67.5(1.98)[b]
Npm	2536	50	13.73(1.44)[-b]	44.8(4.75)[-b]	37.43(4.23)[-b]	66.7(0.68)[b]	45.28(12.29)[-*]	14.74(0.97)[-b]
BroadleafCommerce	677	50	28.24(0.2)[-b]	44.06(2.5)[-b]	52.74(3.63)[b]	81.64(0.44)[b]	38.71(7.36)[-b]	97.11(0.84)[b]
Ranking		5	6	3	2	**1**	4	2

Dataset	#Time Steps	Dummy	WP-ORB_r0	AIO-ORB_r0	Filter-ORB_r0	Ensemble-ORB_r0	OP-AIO-ORB_r0	OP-Filter-ORB_r0
Tomcat	926	50	44.26(2.49)[-b]	39.42(2.27)[-b]	42.81(2.36)[-b]	85.91(0.24)[b]	14.05(3.88)[-b]	33.08(6.91)[-b]
JGroups	1412	50	60.91(1.0)[b]	61.11(1.04)[b]	62.73(1.86)[b]	96.21(0.18)[b]	100.0(0.0)[b]	100.0(0.0)[b]
Spring-Integration	900	50	60.2(1.08)[b]	51.5(4.09)[b]	54.75(5.34)[b]	78.73(1.67)[b]	64.58(2.43)[b]	65.41(3.4)[b]
Camel	5111	50	51.18(2.63)[b]	48.79(1.86)[-b]	47.72(2.3)[-b]	81.14(0.78)[b]	49.09(10.16)[-b]	47.19(10.24)[-s]
Brackets	1721	50	90.26(0.58)[b]	56.53(3.0)[b]	57.46(1.31)[b]	66.97(2.0)[b]	60.73(1.59)[b]	69.53(3.63)[b]
Nova	6271	50	59.67(2.78)[b]	54.18(1.34)[b]	52.18(1.44)[b]	78.64(1.01)[b]	63.48(33.59)[m]	71.91(13.46)[b]
Fabric8	1613	50	68.33(1.0)[b]	64.01(1.22)[b]	63.02(2.22)[b]	63.35(1.33)[b]	50.32(6.14)[-s]	67.42(8.41)[b]
Neutron	3304	50	80.65(1.69)[b]	78.67(2.11)[b]	78.83(4.22)[b]	75.2(1.11)[b]	72.37(16.77)[b]	75.19(9.22)[b]
Npm	1494	50	34.45(1.91)[-b]	27.68(5.38)[-b]	35.37(9.1)[-b]	70.43(2.04)[b]	71.36(24.54)[b]	100.0(0.0)[b]
BroadleafCommerce	950	50	66.31(0.51)[b]	48.84(2.41)[-m]	49.12(2.88)[-s]	81.0(0.95)[b]	61.85(3.31)[b]	73.71(3.41)[b]
Ranking		5	3	4	4	**1**	3	2

Table C.9: Number of initial time steps of the initial phase, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data

Dataset	#Time Steps	Dummy	WP-OOB_r1	AIO-OOB_r1	Filter-OOB_r1	Ensemble-OOB_r1	OP-AIO-OOB_r1	OP-Filter-OOB_r1
Tomcat	2006	50	29.86(7.28)[-b]	68.53(1.27)[b]	53.51(3.0)[b]	13.92(0.61)[-b]	68.46(12.11)[b]	67.95(4.63)[b]
JGroups	1268	50	33.4(1.01)[-b]	33.2(1.08)[-b]	34.69(1.18)[-b]	3.47(0.14)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	461	50	46.42(0.12)[-b]	17.44(0.21)[-b]	23.35(0.85)[-b]	5.84(0.18)[-b]	4.25(2.27)[-b]	16.06(3.65)[-b]
Camel	3112	50	86.83(2.57)[b]	66.32(0.85)[b]	63.64(1.3)[b]	18.93(0.63)[-b]	73.69(1.78)[b]	75.82(0.71)[b]
Brackets	1569	50	90.16(0.02)[b]	72.24(4.03)[b]	72.41(1.55)[b]	30.66(0.95)[-b]	57.13(2.92)[b]	65.82(2.26)[b]
Nova	6271	50	89.01(0.2)[b]	77.39(0.59)[b]	74.26(0.65)[b]	38.58(0.76)[-b]	84.05(5.58)[b]	63.43(8.28)[b]
Fabric8	795	50	82.23(0.0)[b]	35.07(2.2)[-b]	52.17(3.01)[b]	23.28(1.07)[-b]	79.12(4.46)[b]	38.91(2.74)[-b]
Neutron	917	50	87.54(1.87)[b]	71.63(3.0)[b]	55.22(4.71)[b]	39.72(2.15)[-b]	82.38(0.61)[b]	81.75(1.46)[b]
Npm	2536	50	83.75(1.66)[b]	64.47(5.55)[b]	66.9(3.34)[b]	31.45(0.61)[-b]	54.4(13.11)[*]	82.11(1.26)[b]
BroadleafCommerce	677	50	68.1(0.27)[b]	68.31(1.68)[b]	61.71(3.15)[b]	19.0(1.74)[-b]	57.57(6.4)[b]	15.62(3.59)[-b]
Ranking		3	**1**	2	2	4	2	2

Dataset	#Time Steps	Dummy	WP-ORB_r1	AIO-ORB_r1	Filter-ORB_r1	Ensemble-ORB_r1	OP-AIO-ORB_r1	OP-Filter-ORB_r1
Tomcat	926	50	62.76(3.14)[b]	69.21(2.15)[b]	69.17(1.55)[b]	15.28(0.42)[-b]	89.31(3.92)[b]	71.27(3.18)[b]
JGroups	1412	50	33.59(1.41)[-b]	34.15(1.75)[-b]	36.92(2.56)[-b]	3.45(0.22)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	900	50	33.67(0.71)[-b]	75.53(2.76)[b]	76.83(4.92)[b]	23.52(1.65)[-b]	65.55(4.19)[b]	49.04(3.92)[-s]
Camel	5111	50	64.0(2.48)[b]	73.87(1.39)[b]	74.77(1.38)[b]	22.62(0.92)[-b]	60.05(5.27)[b]	61.62(7.17)[b]
Brackets	1721	50	12.52(0.64)[-b]	83.0(4.49)[b]	81.15(1.65)[b]	41.78(1.67)[-b]	72.56(2.48)[b]	56.79(6.54)[b]
Nova	6271	50	67.77(2.12)[b]	82.87(1.83)[b]	82.68(0.96)[b]	34.71(1.46)[-b]	48.24(29.93)[-s]	50.51(15.16)[-s]
Fabric8	1613	50	29.88(0.81)[-b]	66.59(4.18)[b]	66.66(2.39)[b]	38.18(1.22)[-b]	72.86(5.75)[b]	52.85(5.62)[b]
Neutron	3304	50	66.62(3.15)[b]	73.45(3.72)[b]	73.2(5.88)[b]	68.09(1.99)[b]	60.24(11.34)[b]	69.18(18.12)[b]
Npm	1494	50	64.79(1.01)[b]	74.2(6.94)[b]	66.41(8.14)[b]	29.39(2.14)[-b]	38.91(27.87)[-s]	0.0(0.0)[-b]
BroadleafCommerce	950	50	32.16(0.49)[-b]	66.7(5.17)[b]	65.56(6.86)[b]	22.82(1.17)[-b]	59.82(6.63)[b]	40.34(3.41)[-b]
Ranking		3	3	**1**	**1**	4	2	3

Table C.10: Number of time steps of the drop periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data

Dataset	#Time Steps	Dummy	WP-OOB_r0	AIO-OOB_r0	Filter-OOB_r0	Ensemble-OOB_r0	OP-AIO-OOB_r0	OP-Filter-OOB_r0
Tomcat	3855	50	31.23(2.14)[-b]	56.4(1.5)[b]	55.7(1.87)[b]	52.88(0.77)[b]	16.43(5.44)[-b]	24.05(2.03)[-b]
JGroups	4353	50	35.87(2.17)[-b]	44.61(2.27)[-b]	45.33(1.87)[-b]	87.01(0.74)[b]	100.0(0.0)[b]	100.0(0.0)[b]
Spring-Integration	3973	50	55.56(0.93)[b]	60.18(0.97)[b]	66.39(1.19)[b]	75.7(0.67)[b]	39.74(8.09)[-b]	14.66(3.63)[-b]
Camel	6009	50	35.64(2.26)[-b]	38.98(1.58)[-b]	40.43(1.45)[-b]	58.66(0.75)[b]	33.15(3.18)[-b]	34.51(1.2)[-b]
Nova	4427	50	50.2(0.15)[b]	50.37(0.32)[b]	50.08(0.42)[m]	57.99(2.9)[b]	51.22(6.41)[b]	60.05(8.38)[b]
Fabric8	3443	50	39.92(4.78)[-b]	44.8(1.56)[-b]	57.77(3.01)[b]	46.04(1.47)[-b]	7.38(2.8)[-b]	9.57(5.02)[-b]
Npm	1202	50	16.96(1.59)[-b]	61.28(3.82)[b]	65.92(2.88)[b]	43.91(1.65)[-b]	29.65(11.83)[-b]	3.59(1.47)[-b]
BroadleafCommerce	3579	50	30.83(0.89)[-b]	63.03(2.05)[b]	61.69(1.63)[b]	52.65(1.29)[b]	35.69(2.06)[-b]	48.59(7.28)[*]
Ranking		3	4	2	**1**	**1**	4	4

Dataset	#Time Steps	Dummy	WP-ORB_r0	AIO-ORB_r0	Filter-ORB_r0	Ensemble-ORB_r0	OP-AIO-ORB_r0	OP-Filter-ORB_r0
Tomcat	2265	50	36.98(2.68)[-b]	51.19(1.39)[b]	50.41(1.46)[s]	63.44(1.1)[b]	11.32(7.33)[-b]	11.85(8.18)[-b]
JGroups	2066	50	54.18(2.44)[b]	56.29(2.23)[b]	58.59(2.13)[b]	86.4(0.84)[b]	100.0(0.0)[b]	100.0(0.0)[b]
Spring-Integration	2319	50	80.73(1.08)[b]	67.42(1.13)[b]	69.26(1.31)[b]	78.78(0.64)[b]	75.17(2.31)[b]	85.25(10.16)[b]
Camel	1105	50	31.65(4.85)[-b]	39.12(2.46)[-b]	39.62(3.65)[-b]	49.84(2.02)[*]	62.66(14.5)[b]	69.42(15.09)[b]
Nova	4408	50	50.13(1.16)[-*]	51.79(1.58)[b]	49.12(0.5)[-b]	75.65(0.92)[b]	62.33(45.16)[m]	71.99(14.35)[b]
Fabric8	1473	50	42.44(9.12)[-b]	46.4(2.69)[-b]	56.46(1.83)[b]	44.15(2.17)[-b]	68.9(6.57)[b]	76.77(22.73)[b]
Npm	936	50	24.56(6.1)[-b]	25.46(5.22)[-b]	29.76(14.42)[-b]	51.03(1.82)[s]	63.73(30.54)[s]	100.0(0.0)[b]
BroadleafCommerce	2703	50	36.45(3.67)[-b]	61.89(2.39)[b]	63.22(1.42)[b]	60.94(1.41)[b]	84.17(4.51)[b]	73.09(4.61)[b]
Ranking		**1**	**1**	**1**	**1**	**1**	**1**	**1**

Table C.11: Number of time steps of the drop periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data

Dataset	#Time Steps	Dummy	WP-OOB_r1	AIO-OOB_r1	Filter-OOB_r1	Ensemble-OOB_r1	OP-AIO-OOB_r1	OP-Filter-OOB_r1
Tomcat	3855	50	79.18(1.59)[b]	66.49(1.43)[b]	68.96(1.55)[b]	55.09(0.89)[b]	88.44(3.35)[b]	83.28(1.76)[b]
JGroups	4353	50	66.56(1.72)[b]	62.87(1.54)[b]	61.74(1.41)[b]	12.58(0.49)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	3973	50	46.64(0.61)[-b]	61.5(1.37)[b]	58.25(1.56)[b]	29.61(0.68)[-b]	76.14(6.23)[b]	92.76(1.73)[b]
Camel	6009	50	82.26(0.96)[b]	83.98(0.81)[b]	83.43(0.82)[b]	50.67(0.66)[b]	79.0(2.42)[b]	79.26(0.95)[b]
Nova	4427	50	89.62(0.25)[b]	89.38(0.36)[b]	89.56(0.28)[b]	56.78(2.24)[b]	85.2(7.7)[b]	77.97(13.94)[b]
Fabric8	3443	50	75.01(3.84)[b]	75.04(1.69)[b]	75.48(2.09)[b]	64.82(1.62)[b]	98.38(0.9)[b]	93.65(4.43)[b]
Npm	1202	50	91.4(0.63)[b]	65.49(3.48)[b]	65.63(3.93)[b]	60.13(1.23)[b]	72.95(9.92)[b]	95.92(0.84)[b]
BroadleafCommerce	3579	50	83.84(0.72)[b]	73.57(1.44)[b]	71.92(1.27)[b]	59.12(1.27)[b]	81.22(2.79)[b]	68.43(6.13)[b]
Ranking		3	**1**	2	2	3	2	**1**

Dataset	#Time Steps	Dummy	WP-ORB_r1	AIO-ORB_r1	Filter-ORB_r1	Ensemble-ORB_r1	OP-AIO-ORB_r1	OP-Filter-ORB_r1
Tomcat	2265	50	68.09(1.91)[b]	64.83(1.63)[b]	65.49(1.56)[b]	47.3(1.24)[-b]	79.75(7.14)[b]	78.84(6.49)[b]
JGroups	2066	50	52.87(1.89)[b]	52.96(2.08)[b]	53.38(2.15)[b]	14.83(0.42)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	2319	50	23.79(1.23)[-b]	61.78(1.91)[b]	62.65(2.08)[b]	30.52(0.82)[-b]	59.67(4.55)[b]	36.54(6.56)[-b]
Camel	1105	50	79.67(2.64)[b]	80.45(1.59)[b]	78.7(1.8)[b]	64.43(1.78)[b]	53.3(5.85)[b]	48.44(13.95)[s]
Nova	4408	50	90.62(0.92)[b]	88.25(1.32)[b]	90.76(0.42)[b]	50.09(1.31)[s]	43.35(42.41)[-m]	54.78(18.64)[s]
Fabric8	1473	50	74.95(3.71)[b]	72.6(2.28)[b]	72.23(1.47)[b]	69.46(1.63)[b]	46.74(9.2)[-m]	34.42(20.9)[-b]
Npm	936	50	84.78(3.52)[b]	89.18(2.12)[b]	81.1(9.66)[b]	49.68(1.19)[-s]	47.6(34.37)[-s]	0.0(0.0)[-b]
BroadleafCommerce	2703	50	77.47(2.05)[b]	73.22(1.72)[b]	71.44(1.93)[b]	52.63(1.41)[b]	31.66(5.93)[-b]	47.8(3.95)[-b]
Ranking		**1**	**1**	**1**	**1**	**1**	**1**	**1**

Table C.12: Number of time steps of the stable periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data

Dataset	#Time Steps	Dummy	WP-OOB_r0	AIO-OOB_r0	Filter-OOB_r0	Ensemble-OOB_r0	OP-AIO-OOB_r0	OP-Filter-OOB_r0
Tomcat	13046	50	63.41(2.5)[b]	64.18(1.87)[b]	71.17(1.47)[b]	63.42(0.91)[b]	19.51(6.07)[-b]	25.14(2.25)[-b]
JGroups	12704	50	64.27(1.94)[b]	71.36(1.86)[b]	73.2(1.63)[b]	81.77(0.41)[b]	100.0(0.0)[b]	100.0(0.0)[b]
Spring-Integration	4316	50	68.09(3.04)[b]	68.21(0.87)[b]	70.84(0.7)[b]	68.54(0.79)[b]	49.06(12.12)[s]	26.51(9.06)[-b]
Camel	21454	50	65.71(1.45)[b]	57.89(1.33)[b]	59.6(0.95)[b]	50.03(1.23)[s]	21.71(2.47)[-b]	20.51(1.36)[-b]
Brackets	15795	50	52.52(0.31)[b]	65.71(1.15)[b]	69.5(0.91)[b]	63.43(0.83)[b]	28.61(10.71)[-b]	48.21(1.36)[-b]
Nova	38291	50	74.79(0.3)[b]	74.53(0.39)[b]	75.12(0.39)[b]	78.54(0.75)[b]	61.52(15.06)[b]	73.25(0.91)[b]
Fabric8	8868	50	57.35(2.36)[b]	57.82(1.12)[b]	62.45(1.86)[b]	47.53(1.28)[-b]	8.86(2.35)[-b]	12.59(3.59)[-b]
Neutron	18605	50	71.95(0.73)[b]	73.08(0.43)[b]	74.46(0.43)[b]	78.59(1.09)[b]	70.95(1.0)[b]	76.44(3.18)[b]
Npm	4182	50	56.86(4.13)[b]	59.0(1.14)[b]	63.26(1.66)[b]	49.61(1.35)[-s]	79.69(1.73)[b]	22.34(1.15)[-b]
BroadleafCommerce	10754	50	69.53(2.14)[b]	70.16(1.62)[b]	70.23(1.07)[b]	63.82(0.9)[b]	45.69(1.31)[-b]	53.39(4.63)[b]
Ranking		3	2	2	**1**	2	3	3

Dataset	#Time Steps	Dummy	WP-ORB_r0	AIO-ORB_r0	Filter-ORB_r0	Ensemble-ORB_r0	OP-AIO-ORB_r0	OP-Filter-ORB_r0
Tomcat	15716	50	62.33(2.08)[b]	64.86(1.38)[b]	63.69(1.06)[b]	60.08(1.4)[b]	53.99(5.5)[b]	55.9(4.88)[b]
JGroups	14847	50	62.18(0.76)[b]	63.83(1.0)[b]	63.7(1.18)[b]	82.16(0.37)[b]	100.0(0.0)[b]	100.0(0.0)[b]
Spring-Integration	5531	50	70.14(1.53)[b]	70.15(0.82)[b]	70.53(0.78)[b]	72.19(0.67)[b]	78.67(2.39)[b]	82.84(8.52)[b]
Camel	24359	50	62.63(1.3)[b]	62.2(0.74)[b]	61.72(0.56)[b]	57.85(1.03)[b]	62.17(9.55)[b]	52.39(11.16)[*]
Brackets	15643	50	58.12(0.78)[b]	68.04(1.09)[b]	67.31(1.02)[b]	61.12(1.1)[b]	51.29(1.6)[b]	47.6(2.14)[-b]
Nova	38310	50	79.42(1.86)[b]	79.99(1.02)[b]	78.8(0.13)[b]	76.58(0.96)[b]	51.45(39.22)[s]	69.0(2.4)[b]
Fabric8	10020	50	62.97(1.73)[b]	57.94(1.52)[b]	65.44(0.57)[b]	48.2(1.32)[-b]	76.29(6.63)[b]	78.59(19.32)[b]
Neutron	16218	50	80.44(1.88)[b]	74.66(0.65)[b]	73.03(1.07)[b]	77.38(1.86)[b]	69.93(18.31)[b]	74.16(9.09)[b]
Npm	5490	50	65.67(1.21)[b]	61.62(1.05)[b]	59.2(1.95)[b]	47.85(1.65)[-b]	75.57(13.52)[b]	100.0(0.0)[b]
BroadleafCommerce	11357	50	64.33(3.56)[b]	69.45(0.82)[b]	69.17(0.71)[b]	60.09(1.43)[b]	86.2(4.61)[b]	77.91(5.57)[b]
Ranking		4	2	2	2	3	**1**	**1**

Table C.13: Number of time steps of the stable periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Opensource data

Dataset	#Time Steps	Dummy	WP-OOB_r1	AIO-OOB_r1	Filter-OOB_r1	Ensemble-OOB_r1	OP-AIO-OOB_r1	OP-Filter-OOB_r1
Tomcat	13046	50	64.16(1.84)[b]	62.22(1.63)[b]	58.42(1.22)[b]	43.98(0.91)[-b]	87.77(3.7)[b]	84.37(1.98)[b]
JGroups	12704	50	58.97(1.89)[b]	51.37(1.91)[b]	49.8(1.54)[-s]	22.18(0.43)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	4316	50	65.82(2.5)[b]	66.35(1.04)[b]	65.44(1.26)[b]	48.34(0.98)[-b]	73.27(10.71)[b]	95.21(0.76)[b]
Camel	21454	50	70.71(1.21)[b]	74.15(0.87)[b]	73.27(0.84)[b]	61.68(0.99)[b]	85.22(3.59)[b]	89.01(0.92)[b]
Brackets	15795	50	89.55(0.26)[b]	80.42(1.4)[b]	75.74(1.38)[b]	66.01(1.25)[b]	92.2(2.7)[b]	92.68(1.92)[b]
Nova	38291	50	86.3(0.67)[b]	90.73(0.7)[b]	89.09(0.48)[b]	60.33(1.88)[b]	94.92(1.2)[b]	78.04(1.82)[b]
Fabric8	8868	50	75.41(1.83)[b]	72.29(1.62)[b]	74.27(1.66)[b]	63.53(1.57)[b]	97.89(1.3)[b]	89.65(5.59)[b]
Neutron	18605	50	92.02(0.63)[b]	92.36(0.51)[b]	91.32(0.57)[b]	60.77(3.51)[b]	81.57(3.07)[b]	70.63(9.17)[b]
Npm	4182	50	66.38(2.76)[b]	72.18(0.89)[b]	70.47(1.6)[b]	61.9(1.56)[b]	16.7(2.57)[-b]	76.91(0.6)[b]
BroadleafCommerce	10754	50	65.26(2.92)[b]	71.86(1.92)[b]	70.63(1.18)[b]	57.95(1.13)[b]	69.83(2.73)[b]	72.2(5.26)[b]
Ranking		3	**1**	**1**	**1**	2	**1**	**1**

Dataset	#Time Steps	Dummy	WP-ORB_r1	AIO-ORB_r1	Filter-ORB_r1	Ensemble-ORB_r1	OP-AIO-ORB_r1	OP-Filter-ORB_r1
Tomcat	15716	50	64.07(1.41)[b]	62.82(0.83)[b]	62.65(0.89)[b]	49.63(1.18)[-s]	48.3(4.64)[-m]	38.64(3.47)[-b]
JGroups	14847	50	59.72(1.38)[b]	59.33(1.03)[b]	59.68(1.33)[b]	22.42(0.49)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Spring-Integration	5531	50	57.05(1.85)[b]	71.73(1.11)[b]	71.86(0.91)[b]	46.63(0.84)[-b]	61.54(5.01)[b]	43.48(9.61)[-b]
Camel	24359	50	70.95(1.2)[b]	72.6(0.93)[b]	72.0(0.57)[b]	58.01(1.27)[b]	45.11(8.54)[-b]	43.31(12.28)[-s]
Brackets	15643	50	83.71(1.53)[b]	74.05(1.74)[b]	74.72(1.49)[b]	72.15(1.31)[b]	85.52(2.18)[b]	88.95(4.3)[b]
Nova	38310	50	81.5(3.3)[b]	81.08(1.46)[b]	81.96(0.69)[b]	76.05(2.24)[b]	72.83(24.19)[b]	87.05(7.02)[b]
Fabric8	10020	50	72.89(1.25)[b]	75.11(1.35)[b]	72.57(1.16)[b]	69.55(1.36)[b]	36.16(10.58)[-b]	27.81(22.51)[-b]
Neutron	16218	50	82.15(1.91)[b]	90.39(0.75)[b]	92.2(0.74)[b]	72.82(3.94)[b]	67.64(7.47)[b]	70.68(22.39)[b]
Npm	5490	50	59.76(1.78)[b]	75.55(1.43)[b]	73.79(2.62)[b]	64.48(1.41)[b]	45.0(23.09)[s]	0.0(0.0)[-b]
BroadleafCommerce	11357	50	66.01(3.91)[b]	69.67(1.07)[b]	68.97(1.56)[b]	63.72(1.74)[b]	22.65(7.75)[-b]	32.59(4.78)[-b]
Ranking		4	2	**1**	**1**	3	4	5

Table C.14: Number of initial time steps of the initial phase, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data

Dataset	#Time Steps	Dummy	WP-OOB_r0	AIO-OOB combined_r0	AIO-OOB proprietary_r0	Filter-OOB combined_r0	OP-AIO-OOB combined_r0	OP-AIO-OOB proprietary_r0	OP-Filter-OOB combined_r0
C1	284	50	8.88(3.47)[-b]	73.04(4.3)[b]	8.22(2.68)[-b]	70.29(4.65)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C2	581	50	15.39(0.12)[-b]	21.93(0.52)[-b]	45.97(7.61)[-b]	45.29(1.3)[-b]	44.24(4.15)[-b]	98.34(2.85)[b]	48.42(5.71)[s]
C3	82	50	96.32(0.0)[b]	81.86(0.0)[b]	33.22(0.0)[-b]	96.9(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C4	24	50	91.82(0.91)[b]	81.24(0.45)[b]	90.17(10.07)[b]	100.0(0.0)[b]	81.0(0.37)[b]	92.34(2.85)[b]	100.0(0.0)[b]
C5	8	50	87.37(0.0)[b]	100.0(0.0)[b]	59.16(9.56)[b]	92.0(6.19)[b]	100.0(0.0)[b]	100.0(0.0)[b]	68.66(0.0)[b]
C6	20	50	91.76(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C7	29	50	100.0(0.0)[b]	82.15(4.53)[b]	64.66(4.4)[b]	100.0(0.0)[b]	79.19(0.64)[b]	94.6(5.13)[b]	100.0(0.0)[b]
C8	39	50	100.0(0.0)[b]	100.0(0.0)[b]	86.75(6.83)[b]	100.0(0.0)[b]	99.15(1.02)[b]	87.98(3.62)[b]	100.0(0.0)[b]
C9	62	50	86.42(2.18)[b]	99.8(0.12)[b]	97.87(2.06)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
Ranking		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Dataset	#Time Steps	Dummy	WP-ORB_r0	AIO-ORB combined_r0	AIO-ORB proprietary_r0	Filter-ORB combined_r0	OP-AIO-ORB combined_r0	OP-AIO-ORB proprietary_r0	OP-Filter-ORB combined_r0
C1	671	50	80.46(3.18)[b]	57.56(4.17)[b]	48.18(1.51)[-b]	83.32(2.87)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C2	582	50	72.19(0.99)[b]	35.51(3.57)[-b]	38.38(7.0)[-b]	46.17(0.7)[-b]	82.54(2.74)[b]	87.6(8.27)[b]	89.23(3.54)[b]
C3	396	50	58.74(1.01)[b]	57.67(0.67)[b]	60.56(2.2)[b]	63.26(1.94)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C4	93	50	99.8(0.03)[b]	52.43(3.65)[s]	64.64(2.18)[b]	99.77(0.7)[b]	65.36(5.61)[b]	68.91(0.74)[b]	97.48(0.37)[b]
C5	297	50	96.65(0.51)[b]	9.29(1.88)[-b]	10.18(2.27)[-b]	20.07(3.33)[-b]	46.86(5.94)[-m]	35.64(10.12)[-b]	44.37(4.11)[-b]
C6	210	50	99.74(0.0)[b]	66.1(1.25)[b]	79.56(5.66)[b]	69.83(0.59)[b]	84.04(2.45)[b]	95.09(0.43)[b]	91.91(0.63)[b]
C7	62	50	98.68(0.2)[b]	62.36(7.41)[b]	61.35(3.72)[b]	96.15(0.99)[b]	57.67(29.59)[*]	73.09(22.51)[b]	100.0(0.0)[b]
C8	232	50	99.31(0.09)[b]	49.48(13.63)[-s]	54.53(9.78)[m]	79.1(5.3)[b]	65.88(7.72)[b]	78.09(1.74)[b]	79.09(3.19)[b]
C9	198	50	99.52(0.0)[b]	87.47(1.51)[b]	91.55(1.11)[b]	88.18(0.04)[b]	99.9(0.03)[b]	99.84(0.6)[b]	100.0(0.0)[b]
Ranking		4	**1**	3	2	**1**	**1**	**1**	**1**

Table C.15: Number of initial time steps of the initial phase, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data

Dataset	#Time Steps	Dummy	WP-OOB_r1	AIO-OOB combined_r1	AIO-OOB proprietary_r1	Filter-OOB combined_r1	OP-AIO-OOB combined_r1	OP-AIO-OOB proprietary_r1	OP-Filter-OOB combined_r1
C1	284	50	77.86(1.91)[b]	23.38(3.7)[-b]	78.88(1.3)[b]	31.5(5.25)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	581	50	86.87(0.0)[b]	75.92(0.43)[b]	48.36(10.42)[s]	63.19(1.79)[b]	55.04(3.5)[b]	1.59(1.93)[-b]	55.4(4.09)[b]
C3	82	50	4.2(0.0)[-b]	21.43(0.0)[-b]	70.61(0.0)[b]	4.2(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	24	50	4.07(0.0)[-b]	13.33(2.96)[-b]	10.57(5.23)[-b]	0.0(0.0)[-b]	14.63(0.0)[-b]	6.82(0.0)[-b]	0.0(0.0)[-b]
C5	8	50	8.25(0.0)[-b]	0.77(1.19)[-b]	40.69(7.15)[-b]	6.49(2.79)[-b]	2.47(2.87)[-b]	0.95(2.16)[-b]	42.5(0.0)[-b]
C6	20	50	2.02(0.0)[-b]	0.65(0.0)[-b]	0.65(0.0)[-b]	0.5(0.28)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C7	29	50	0.4(0.0)[-b]	33.96(3.36)[-b]	61.03(9.37)[b]	0.0(0.0)[-b]	32.62(7.1)[-b]	17.21(10.02)[-b]	0.0(0.0)[-b]
C8	39	50	0.23(0.0)[-b]	1.98(1.69)[-b]	29.03(13.19)[-b]	0.2(0.08)[-b]	7.09(4.37)[-b]	24.31(9.13)[-b]	0.0(0.0)[-b]
C9	62	50	6.84(1.14)[-b]	7.08(4.15)[-b]	13.61(1.72)[-b]	0.19(0.0)[-b]	8.91(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
Ranking		**1**	3	3	2	3	3	3	3

Dataset	#Time Steps	Dummy	WP-ORB_r1	AIO-ORB combined_r1	AIO-ORB proprietary_r1	Filter-ORB combined_r1	OP-AIO-ORB combined_r1	OP-AIO-ORB proprietary_r1	OP-Filter-ORB combined_r1
C1	671	50	14.15(2.64)[-b]	36.95(3.74)[-b]	41.14(1.7)[-b]	23.79(4.41)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	582	50	16.46(0.81)[-b]	60.29(3.77)[b]	55.39(9.46)[b]	42.99(0.69)[-b]	26.15(4.43)[-b]	15.15(9.76)[-b]	23.4(6.47)[-b]
C3	396	50	49.01(0.88)[-b]	49.76(1.01)[-*]	47.38(1.9)[-b]	46.45(1.4)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	93	50	1.79(0.15)[-b]	49.26(5.9)[s]	48.15(0.53)[-b]	8.05(3.1)[-b]	45.95(7.39)[-s]	41.94(0.96)[-b]	11.08(0.96)[-b]
C5	297	50	3.93(0.87)[-b]	92.74(0.54)[b]	92.54(2.47)[b]	81.8(2.73)[b]	65.37(9.0)[b]	74.68(9.81)[b]	73.72(3.75)[b]
C6	210	50	0.67(0.0)[-b]	26.33(2.51)[-b]	21.52(3.79)[-b]	19.53(1.79)[-b]	20.69(2.99)[-b]	3.02(0.64)[-b]	8.7(0.74)[-b]
C7	62	50	0.61(0.15)[-b]	42.65(13.22)[-m]	39.4(5.5)[-b]	7.28(1.36)[-b]	52.9(34.22)[*]	31.78(27.81)[-b]	0.0(0.0)[-b]
C8	232	50	0.35(0.06)[-b]	64.09(9.64)[b]	61.67(8.08)[b]	21.64(4.81)[-b]	46.37(5.66)[-b]	43.14(3.28)[-b]	29.06(3.05)[-b]
C9	198	50	0.21(0.0)[-b]	19.27(2.23)[-b]	20.31(1.71)[-b]	15.84(0.0)[-b]	5.31(0.08)[-b]	0.33(1.25)[-b]	0.87(0.0)[-b]
Ranking		**1**	4	**1**	**1**	2	2	3	4

Table C.16: Number of time steps of the drop periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data

Dataset	Time Steps	Dummy	WP-OOB_r0	AIO-OOB combined_r0	AIO-OOB proprietary_r0	Filter-OOB combined_r0	OP-AIO-OOB combined_r0	OP-AIO-OOB proprietary_r0	OP-Filter-OOB combined_r0
C1	452	50	84.77(6.19)[b]	26.11(2.22)[-b]	42.66(13.83)[-s]	32.12(2.29)[-b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C3	210	50	50.1(7.97)[-s]	27.54(1.59)[-b]	29.44(3.41)[-b]	34.69(1.83)[-b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C4	249	50	11.97(0.65)[-b]	30.74(2.16)[-b]	63.51(6.07)[b]	51.73(4.37)[*]	47.24(2.17)[-b]	32.49(3.42)[-b]	39.19(0.0)[-b]
C6	466	50	5.15(0.64)[-b]	26.33(2.09)[-b]	34.66(11.62)[-b]	28.3(3.19)[-b]	45.26(1.32)[-b]	69.05(0.41)[b]	52.66(1.28)[b]
C7	255	50	7.14(0.65)[-b]	23.99(9.19)[-b]	10.95(1.03)[-b]	21.37(1.38)[-b]	83.72(2.23)[b]	16.52(0.61)[-b]	13.71(0.14)[-b]
C8	287	50	0.63(0.17)[-b]	29.88(1.99)[-b]	14.91(4.91)[-b]	23.19(2.19)[-b]	40.46(5.0)[-b]	78.4(12.78)[b]	8.73(5.57)[-b]
C9	541	50	11.22(1.27)[-b]	34.4(3.19)[-b]	53.35(5.06)[b]	42.36(3.28)[-b]	51.07(9.71)[s]	92.54(7.04)[b]	41.65(0.21)[-b]
Ranking		**1**	3	3	2	2	**1**	**1**	**1**

Dataset	#Time Steps	Dummy	WP-ORB_r0	AIO-ORB combined_r0	AIO-ORB proprietary_r0	Filter-ORB combined_r0	OP-AIO-ORB combined_r0	OP-AIO-ORB proprietary_r0	OP-Filter-ORB combined_r0
C1	238	50	28.63(13.15)[-b]	18.76(9.38)[-b]	9.59(1.98)[-b]	39.79(21.63)[-m]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C6	64	50	15.33(0.84)[-b]	43.57(15.27)[-s]	17.24(14.47)[-b]	35.03(13.43)[-b]	70.55(11.14)[b]	48.53(18.28)[s]	27.19(15.64)[-b]
C7	85	50	11.98(7.33)[-b]	15.04(14.72)[-b]	38.83(18.82)[*]	52.99(13.53)[b]	65.2(35.14)[*]	24.48(10.46)[-b]	100.0(0.0)[b]
C8	80	50	5.22(0.63)[-b]	28.9(21.24)[-b]	26.31(21.4)[-b]	46.81(14.55)[-b]	73.35(15.25)[b]	94.11(6.34)[b]	64.39(8.61)[b]
C9	23	50	6.55(0.76)[-b]	19.86(7.08)[-b]	7.77(3.32)[-b]	4.68(0.68)[-b]	70.41(10.03)[b]	99.92(0.22)[b]	100.0(0.0)[b]
Ranking		2	6	4	5	3	**1**	**1**	**1**

Table C.17: Number of time steps of the drop periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data

Dataset	#Time Steps	Dummy	WP-OOB_r1	AIO-OOB combined_r1	AIO-OOB proprietary_r1	Filter-OOB combined_r1	OP-AIO-OOB combined_r1	OP-AIO-OOB proprietary_r1	OP-Filter-OOB combined_r1
C1	452	50	14.17(5.18)[-b]	75.04(1.66)[b]	55.83(12.28)[b]	71.1(2.06)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C3	210	50	38.31(7.4)[-b]	84.22(2.58)[b]	53.73(4.45)[b]	79.31(1.67)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	249	50	92.64(0.8)[b]	86.92(1.29)[b]	60.85(9.01)[b]	67.4(3.03)[b]	71.16(2.01)[b]	71.82(3.15)[b]	74.73(0.0)[b]
C6	466	50	93.91(0.63)[b]	76.17(1.87)[b]	69.32(10.57)[b]	76.44(2.69)[b]	61.56(0.87)[b]	33.49(0.32)[-b]	52.8(0.83)[b]
C7	255	50	95.62(0.4)[b]	78.93(6.05)[b]	96.88(0.44)[b]	79.55(2.64)[b]	30.7(3.52)[-b]	81.1(2.69)[b]	74.01(1.18)[b]
C8	287	50	96.02(0.01)[b]	67.92(2.38)[b]	84.97(4.07)[b]	71.01(1.43)[b]	50.75(3.4)[*]	29.82(14.73)[-b]	88.28(3.54)[b]
C9	541	50	96.6(0.32)[b]	79.08(3.72)[b]	56.0(7.53)[b]	69.08(3.31)[b]	48.92(7.65)[-s]	16.19(6.51)[-b]	56.0(0.61)[b]
Ranking		4	**1**	**1**	3	2	5	5	4

Dataset	#Time Steps	Dummy	WP-ORB_r1	AIO-ORB combined_r1	AIO-ORB proprietary_r1	Filter-ORB combined_r1	OP-AIO-ORB combined_r1	OP-AIO-ORB proprietary_r1	OP-Filter-ORB combined_r1
C1	238	50	66.05(9.28)[b]	81.33(10.05)[b]	89.84(1.73)[b]	66.29(16.21)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C6	64	50	82.03(0.92)[b]	64.93(12.54)[b]	84.23(6.62)[b]	62.55(14.31)[b]	54.05(6.53)[b]	51.0(15.78)[-s]	76.68(14.92)[b]
C7	85	50	86.32(6.53)[b]	87.8(9.7)[b]	74.54(11.65)[b]	69.47(7.09)[b]	42.91(36.14)[*]	77.92(14.99)[b]	0.0(0.0)[-b]
C8	80	50	79.23(0.73)[b]	72.81(17.24)[b]	76.26(17.36)[b]	58.39(10.8)[b]	33.68(12.15)[-b]	12.23(7.25)[-b]	33.15(4.47)[-b]
C9	23	50	85.4(0.1)[b]	83.72(3.25)[b]	88.1(4.28)[b]	91.63(1.04)[b]	28.74(8.68)[-b]	0.16(0.38)[-b]	3.23(0.0)[-b]
Ranking		4	2	2	**1**	3	5	5	5

Table C.18: Number of time steps of the stable periods, and average Recall0, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data

Dataset	#Time Steps	Dummy	WP-OOB_r0	AIO-OOB combined_r0	AIO-OOB proprietary_r0	Filter-OOB combined_r0	OP-AIO-OOB combined_r0	OP-AIO-OOB proprietary_r0	OP-Filter-OOB combined_r0
C1	294	50	68.67(6.6)[b]	70.1(8.12)[b]	37.69(12.32)[-b]	70.35(7.36)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C2	20	50	43.79(0.31)[-b]	30.97(1.95)[-b]	41.47(3.28)[-b]	43.23(0.92)[-b]	21.96(2.01)[-b]	99.98(0.05)[b]	22.88(2.55)[-b]
C3	125	50	52.06(3.31)[m]	38.54(0.51)[-b]	16.82(2.45)[-b]	53.6(0.98)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C4	68	50	31.88(1.5)[-b]	48.23(1.18)[-b]	69.56(7.66)[b]	81.94(1.26)[b]	54.09(1.83)[b]	51.46(4.05)[*]	68.18(0.0)[b]
C5	315	50	10.01(0.68)[-b]	33.2(1.71)[-b]	21.59(1.02)[-b]	37.2(2.6)[-b]	75.79(3.88)[b]	41.3(5.23)[-b]	22.39(6.7)[-b]
C6	69	50	26.94(0.59)[-b]	59.83(0.9)[b]	60.97(6.52)[b]	63.14(1.68)[b]	56.29(1.15)[b]	75.51(0.09)[b]	57.25(1.23)[b]
C7	262	50	29.44(1.12)[-b]	47.27(16.27)[-s]	30.36(2.47)[-b]	61.28(3.15)[b]	71.3(2.59)[b]	45.49(1.51)[-b]	31.83(1.42)[-b]
C8	472	50	13.41(0.09)[-b]	52.31(1.41)[b]	33.75(5.39)[-b]	56.43(1.92)[b]	79.96(2.43)[b]	76.31(10.7)[b]	28.6(3.5)[-b]
C9	91	50	31.84(2.71)[-b]	67.15(1.64)[b]	70.01(7.7)[b]	78.09(1.55)[b]	84.37(3.2)[b]	97.16(2.58)[b]	80.82(0.0)[b]
Ranking		2	4	2	3	**1**	**1**	**1**	**1**

Dataset	#Time Steps	Dummy	WP-ORB_r0	AIO-ORB combined_r0	AIO-ORB proprietary_r0	Filter-ORB combined_r0	OP-AIO-ORB combined_r0	OP-AIO-ORB proprietary_r0	OP-Filter-ORB combined_r0
C1	121	50	48.22(13.97)[-m]	30.34(12.18)[-b]	17.33(2.88)[-b]	51.83(15.36)[-*]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C2	19	50	49.13(2.37)[-m]	47.11(1.46)[-b]	31.07(7.39)[-b]	45.37(0.7)[-b]	96.24(2.41)[b]	88.57(12.86)[b]	98.83(0.49)[b]
C3	21	50	57.23(2.36)[b]	51.87(1.99)[b]	61.63(3.36)[b]	62.94(6.05)[b]	100.0(0.0)[b]	100.0(0.0)[b]	100.0(0.0)[b]
C4	248	50	57.35(5.96)[b]	20.91(9.31)[-b]	26.0(4.32)[-b]	66.35(3.78)[b]	42.92(9.0)[-b]	24.12(1.07)[-b]	86.82(2.22)[b]
C5	26	50	87.66(3.65)[b]	43.52(1.95)[-b]	43.15(2.6)[-b]	43.25(6.74)[-b]	56.71(16.81)[s]	35.02(10.21)[-b]	48.57(17.23)[-m]
C6	281	50	47.86(0.68)[-b]	45.21(8.02)[-b]	40.18(6.55)[-b]	51.36(10.57)[m]	69.18(4.7)[b]	68.15(12.22)[b]	47.56(10.78)[-b]
C7	399	50	47.84(5.89)[-m]	34.28(14.53)[-b]	45.1(11.62)[-*]	62.56(6.54)[b]	53.82(40.33)[*]	32.37(20.88)[-b]	100.0(0.0)[b]
C8	486	50	24.99(0.64)[-b]	33.92(17.42)[-b]	30.49(21.0)[-b]	46.02(8.36)[-m]	78.66(10.29)[b]	93.79(4.3)[b]	82.56(4.77)[b]
C9	473	50	25.0(0.33)[-b]	20.13(2.46)[-b]	26.41(7.98)[-b]	17.05(1.48)[-b]	78.91(7.13)[b]	99.92(0.2)[b]	99.94(0.0)[b]
Ranking		2	2	3	3	2	**1**	**1**	**1**

Table C.19: Number of time steps of the stable periods, and average Recall1, A12 effect sizes and Scott-Knott.BA12 to compare learning approaches on this initial phase for OOB and ORB for Proprietary data

Dataset	**#Time Steps**	**Dummy**	**WP-OOB_r1**	**AIO-OOB combined_r1**	**AIO-OOB proprietary_r1**	**Filter-OOB combined_r1**	**OP-AIO-OOB combined_r1**	**OP-AIO-OOB proprietary_r1**	**OP-Filter-OOB combined_r1**
C1	294	50	40.94(5.53)[-b]	31.25(5.18)[-b]	66.19(9.31)[b]	32.55(4.31)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	20	50	95.41(0.0)[b]	90.63(0.34)[b]	75.08(8.04)[b]	81.76(1.55)[b]	60.95(3.34)[b]	0.18(0.24)[-b]	61.4(3.04)[b]
C3	125	50	60.68(2.77)[b]	76.54(0.57)[b]	85.09(4.0)[b]	63.03(0.57)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	68	50	70.33(0.48)[b]	61.33(2.18)[b]	51.46(10.89)[-s]	26.06(3.45)[-b]	43.13(0.67)[-b]	49.03(5.56)[-s]	33.88(0.0)[-b]
C5	315	50	87.4(1.47)[b]	69.41(3.14)[b]	91.31(1.18)[b]	68.6(4.17)[b]	27.42(5.61)[-b]	69.5(4.43)[b]	83.45(6.8)[b]
C6	69	50	72.3(0.29)[b]	44.65(0.54)[-b]	49.66(6.76)[*]	43.24(2.6)[-b]	39.98(0.67)[-b]	21.17(0.19)[-b]	34.78(0.67)[-b]
C7	262	50	80.94(0.76)[b]	66.75(10.32)[b]	93.45(1.45)[b]	51.74(5.52)[s]	47.51(1.64)[-b]	63.96(3.6)[b]	64.92(0.49)[b]
C8	472	50	82.1(0.01)[b]	60.49(2.11)[b]	72.47(5.79)[b]	38.65(2.13)[-b]	27.55(2.61)[-b]	34.23(9.42)[-b]	71.64(2.57)[b]
C9	91	50	80.9(0.96)[b]	52.67(3.64)[b]	56.11(12.84)[m]	32.22(2.12)[-b]	20.54(2.67)[-b]	4.74(2.18)[-b]	19.72(0.54)[-b]
Ranking		3	**1**	2	**1**	3	5	5	4
Dataset	**#Time Steps**	**Dummy**	**WP-ORB_r1**	**AIO-ORB combined_r1**	**AIO-ORB proprietary_r1**	**Filter-ORB combined_r1**	**OP-AIO-ORB combined_r1**	**OP-AIO-ORB proprietary_r1**	**OP-Filter-ORB combined_r1**
C1	121	50	53.31(10.56)[m]	76.43(12.42)[b]	86.06(2.21)[b]	60.7(13.86)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C2	19	50	49.07(1.59)[-b]	79.24(2.8)[b]	74.37(8.71)[b]	74.63(1.04)[b]	13.63(2.74)[-b]	16.78(11.42)[-b]	14.6(2.42)[-b]
C3	21	50	65.32(2.08)[b]	70.67(2.53)[b]	63.99(4.56)[b]	61.4(5.67)[b]	0.0(0.0)[-b]	0.0(0.0)[-b]	0.0(0.0)[-b]
C4	248	50	54.03(8.28)[m]	89.28(6.21)[b]	87.96(2.23)[b]	51.19(2.72)[s]	70.53(12.32)[b]	82.72(0.47)[b]	51.85(7.74)[s]
C5	26	50	21.41(7.72)[-b]	72.84(2.94)[b]	73.21(3.27)[b]	70.62(5.21)[b]	61.45(8.44)[b]	68.01(10.35)[b]	62.72(9.68)[b]
C6	281	50	64.44(0.61)[b]	66.63(8.06)[b]	73.93(4.33)[b]	54.71(11.81)[-s]	57.88(3.96)[b]	40.69(11.55)[-b]	67.63(11.18)[b]
C7	399	50	59.21(4.05)[b]	80.05(10.58)[b]	75.5(7.1)[b]	60.97(3.76)[b]	51.88(38.94)[*]	62.18(22.32)[b]	0.0(0.0)[-b]
C8	486	50	58.0(0.87)[b]	73.5(12.77)[b]	76.96(15.2)[b]	52.81(5.95)[s]	33.26(9.93)[-b]	18.36(6.82)[-b]	28.7(3.47)[-b]
C9	473	50	69.17(0.12)[b]	81.02(1.72)[b]	77.59(6.14)[b]	83.16(1.7)[b]	19.65(5.55)[-b]	0.29(0.68)[-b]	3.45(0.0)[-b]
Ranking		4	3	**1**	**1**	2	5	5	5

C.2 Tables reporting FPR, FNR, Balanced Accuracy and Informedness for open source and proprietary data for Chapter Five including overall, drop, stable and initial periods

Table C.20: FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-OOB	41.74	36.41	60.925	21.85
tomcat	AIO-OOB	38.92	36.24	62.420	24.84
tomcat	Filter-OOB	33.01	39.95	63.520	27.04
tomcat	Ensemble-OOB	36.22	56.94	53.420	6.84
tomcat	OfflineCP-AIO-OOB	77.91	14.13	53.980	7.96
tomcat	OfflineCP-Filter-OOB	72.39	17.59	55.010	10.02
JGroups	WP-OOB	42.45	40.99	58.280	16.56
JGroups	AIO-OOB	35.46	47.15	58.695	17.39
JGroups	Filter-OOB	34.06	48.41	58.765	17.53
JGroups	Ensemble-OOB	15.97	81.39	51.320	2.64
JGroups	OfflineCP-AIO-OOB	0.00	100.00	50.000	0.00
JGroups	OfflineCP-Filter-OOB	0.00	100.00	50.000	0.00
spring-integration	WP-OOB	38.86	43.90	58.620	17.24
spring-integration	AIO-OOB	34.67	38.42	63.455	26.91
spring-integration	Filter-OOB	30.18	40.04	64.890	29.78
spring-integration	Ensemble-OOB	26.90	62.40	55.350	10.70
spring-integration	OfflineCP-AIO-OOB	52.80	29.06	59.070	18.14
spring-integration	OfflineCP-Filter-OOB	75.33	10.07	57.300	14.60
camel	WP-OOB	43.96	25.37	65.335	30.67
camel	AIO-OOB	46.11	24.71	64.590	29.18
camel	Filter-OOB	44.48	25.71	64.905	29.81
camel	Ensemble-OOB	44.74	44.83	55.215	10.43
camel	OfflineCP-AIO-OOB	74.28	17.17	54.275	8.55

Continued on next page

Table C.20: FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
camel	OfflineCP-Filter-OOB	74.94	14.24	55.410	10.82
brackets	WP-OOB	50.83	10.39	69.390	38.78
brackets	AIO-OOB	34.52	20.32	72.580	45.16
brackets	Filter-OOB	30.84	24.55	72.305	44.61
brackets	Ensemble-OOB	35.11	37.18	63.855	27.71
brackets	OfflineCP-AIO-OOB	68.66	10.96	60.190	20.38
brackets	OfflineCP-Filter-OOB	50.58	9.74	69.840	39.68
nova	WP-OOB	31.80	13.05	77.575	55.15
nova	AIO-OOB	29.96	11.09	79.475	58.95
nova	Filter-OOB	29.08	12.76	79.080	58.16
nova	Ensemble-OOB	24.12	42.77	66.555	33.11
nova	OfflineCP-AIO-OOB	40.92	7.34	75.870	51.74
nova	OfflineCP-Filter-OOB	28.83	23.83	73.670	47.34
fabric8	WP-OOB	49.44	24.28	63.140	26.28
fabric8	AIO-OOB	44.09	29.24	63.335	26.67
fabric8	Filter-OOB	38.06	26.75	67.595	35.19
fabric8	Ensemble-OOB	51.15	38.57	55.140	10.28
fabric8	OfflineCP-AIO-OOB	89.12	3.12	53.880	7.76
fabric8	OfflineCP-Filter-OOB	84.58	12.37	51.525	3.05
neutron	WP-OOB	29.97	8.18	80.925	61.85
neutron	AIO-OOB	26.76	8.61	82.315	64.63
neutron	Filter-OOB	25.10	10.37	82.265	64.53
neutron	Ensemble-OOB	21.41	40.22	69.185	38.37
neutron	OfflineCP-AIO-OOB	29.76	18.39	75.925	51.85
neutron	OfflineCP-Filter-OOB	23.97	28.84	73.595	47.19
npm	WP-OOB	63.00	24.26	56.370	12.74
npm	AIO-OOB	45.20	31.30	61.750	23.50
npm	Filter-OOB	44.60	31.40	62.000	24.00
npm	Ensemble-OOB	45.78	48.11	53.055	6.11
npm	OfflineCP-AIO-OOB	38.92	62.69	49.195	-1.61
npm	OfflineCP-Filter-OOB	82.94	18.53	49.265	-1.47

Continued on next page

Table C.20: FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
BroadleafCommerce	WP-OOB	41.55	30.17	64.140	28.28
BroadleafCommerce	AIO-OOB	32.71	27.89	69.700	39.40
BroadleafCommerce	Filter-OOB	32.59	29.46	68.975	37.95
BroadleafCommerce	Ensemble-OOB	38.04	43.52	59.220	18.44
BroadleafCommerce	OfflineCP-AIO-OOB	57.00	28.00	57.500	15.00
BroadleafCommerce	OfflineCP-Filter-OOB	45.78	31.25	61.485	22.97

Table C.21: FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-ORB	41.59	35.50	61.455	22.91
tomcat	AIO-ORB	38.02	36.62	62.680	25.36
tomcat	Filter-ORB	38.92	36.69	62.195	24.39
tomcat	Ensemble-ORB	38.25	52.33	54.710	9.42
tomcat	OfflineCP-AIO-ORB	53.07	45.92	50.505	1.01
tomcat	OfflineCP-Filter-ORB	50.49	54.95	47.280	-5.44
JGroups	WP-ORB	38.81	43.06	59.065	18.13
JGroups	AIO-ORB	37.22	43.32	59.730	19.46
JGroups	Filter-ORB	36.94	42.78	60.140	20.28
JGroups	Ensemble-ORB	16.27	79.89	51.920	3.84
JGroups	OfflineCP-AIO-ORB	0.00	100.00	50.000	0.00
JGroups	OfflineCP-Filter-ORB	0.00	100.00	50.000	0.00
spring-integration	WP-ORB	28.07	54.16	58.885	17.77
spring-integration	AIO-ORB	32.48	30.51	68.505	37.01
spring-integration	Filter-ORB	31.42	30.06	69.260	38.52
spring-integration	Ensemble-ORB	25.39	60.01	57.300	14.60
spring-integration	OfflineCP-AIO-ORB	23.70	38.54	68.880	37.76
spring-integration	OfflineCP-Filter-ORB	18.31	57.78	61.955	23.91
camel	WP-ORB	40.40	29.89	64.855	29.71
camel	AIO-ORB	40.86	26.89	66.125	32.25

Continued on next page

Table C.21: FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
camel	Filter-ORB	41.42	27.29	65.645	31.29
camel	Ensemble-ORB	38.54	47.67	56.895	13.79
camel	OfflineCP-AIO-ORB	39.99	52.09	53.960	7.92
camel	OfflineCP-Filter-ORB	47.86	53.44	49.350	-1.30
brackets	WP-ORB	38.69	23.34	68.985	37.97
brackets	AIO-ORB	33.10	25.06	70.920	41.84
brackets	Filter-ORB	33.66	24.64	70.850	41.70
brackets	Ensemble-ORB	38.29	30.86	65.425	30.85
brackets	OfflineCP-AIO-ORB	47.77	15.76	68.235	36.47
brackets	OfflineCP-Filter-ORB	50.22	14.24	67.770	35.54
nova	WP-ORB	25.74	19.43	77.415	54.83
nova	AIO-ORB	25.84	18.04	78.060	56.12
nova	Filter-ORB	27.28	17.15	77.785	55.57
nova	Ensemble-ORB	23.24	31.57	72.595	45.19
nova	OfflineCP-AIO-ORB	46.03	32.97	60.500	21.00
nova	OfflineCP-Filter-ORB	30.35	20.53	74.560	49.12
fabric8	WP-ORB	38.67	32.17	64.580	29.16
fabric9	AIO-ORB	42.61	26.22	65.585	31.17
fabric8	Filter-ORB	35.86	28.19	67.975	35.95
fabric8	Ensemble-ORB	50.39	34.31	57.650	15.30
fabric8	OfflineCP-AIO-ORB	27.73	58.13	57.070	14.14
fabric8	OfflineCP-Filter-ORB	22.99	68.36	54.325	8.65
neutron	WP-ORB	19.52	20.48	80.000	60.00
neutron	AIO-ORB	24.66	12.47	81.435	62.87
neutron	Filter-ORB	25.98	11.01	81.505	63.01
neutron	Ensemble-ORB	22.98	27.97	74.525	49.05
neutron	OfflineCP-AIO-ORB	29.65	33.60	68.375	36.75
neutron	OfflineCP-Filter-ORB	25.66	29.57	72.385	44.77
npm	WP-ORB	45.07	36.33	59.300	18.60
npm	AIO-ORB	49.05	23.09	63.930	27.86
npm	Filter-ORB	48.77	26.73	62.250	24.50

Continued on next page

Table C.21: FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
npm	Ensemble-ORB	47.51	43.88	54.305	8.61
npm	OfflineCP-AIO-ORB	26.62	55.84	58.770	17.54
npm	OfflineCP-Filter-ORB	0.00	100.00	50.000	0.00
BroadleafCommerce	WP-ORB	40.56	34.06	62.690	25.38
BroadleafCommerce	AIO-ORB	33.21	29.87	68.460	36.92
BroadleafCommerce	Filter-ORB	33.16	30.80	68.020	36.04
BroadleafCommerce	Ensemble-ORB	38.43	40.86	60.355	20.71
BroadleafCommerce	OfflineCP-AIO-ORB	15.71	73.37	55.460	10.92
BroadleafCommerce	OfflineCP-Filter-ORB	23.22	64.18	56.300	12.60

Table C.22: FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-OOB	23.34	70.14	53.260	6.52
JGroups	WP-OOB	35.34	66.60	49.030	-1.94
spring-integration	WP-OOB	55.89	53.58	45.265	-9.47
camel	WP-OOB	71.29	13.17	57.770	15.54
brackets	WP-OOB	84.63	9.84	52.765	5.53
nova	WP-OOB	59.36	10.99	64.825	29.65
fabric8	WP-OOB	79.19	17.77	51.520	3.04
neutron	WP-OOB	68.87	12.46	59.335	18.67
npm	WP-OOB	86.27	16.25	48.740	-2.52
BroadleafCommerce	WP-OOB	71.76	31.90	48.170	-3.66
tomcat	AIO-OOB	50.14	31.47	59.195	18.39
JGroups	AIO-OOB	35.46	66.80	48.870	-2.26
spring-integration	AIO-OOB	17.40	82.56	50.020	0.04
camel	AIO-OOB	44.87	33.68	60.725	21.45
brackets	AIO-OOB	36.98	27.76	67.630	35.26
nova	AIO-OOB	43.47	22.61	66.960	33.92
fabric8	AIO-OOB	17.23	64.93	58.920	17.84

Continued on next page

Table C.22: FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
neutron	AIO-OOB	23.61	28.37	74.010	48.02
npm	AIO-OOB	55.20	35.53	54.635	9.27
BroadleafCommerce	AIO-OOB	55.94	31.69	56.185	12.37
tomcat	Filter-OOB	38.52	46.49	57.495	14.99
JGroups	Filter-OOB	36.10	65.31	49.295	-1.41
spring-integration	Filter-OOB	10.29	76.65	56.530	13.06
camel	Filter-OOB	43.44	36.36	60.100	20.20
brackets	Filter-OOB	34.33	27.59	69.040	38.08
nova	Filter-OOB	40.08	25.74	67.090	34.18
fabric8	Filter-OOB	25.64	47.83	63.265	26.53
neutron	Filter-OOB	16.43	44.78	69.395	38.79
npm	Filter-OOB	62.57	33.10	52.165	4.33
BroadleafCommerce	Filter-OOB	47.26	38.29	57.225	14.45
tomcat	Ensemble-OOB	12.97	86.08	50.475	0.95
JGroups	Ensemble-OOB	3.59	96.53	49.940	-0.12
spring-integration	Ensemble-OOB	6.75	94.16	49.545	-0.91
camel	Ensemble-OOB	15.29	81.07	51.820	3.64
brackets	Ensemble-OOB	20.41	69.34	55.125	10.25
nova	Ensemble-OOB	27.78	61.42	55.400	10.80
fabric8	Ensemble-OOB	24.41	76.72	49.435	-1.13
neutron	Ensemble-OOB	21.55	60.28	59.085	18.17
npm	Ensemble-OOB	33.30	68.55	49.075	-1.85
BroadleafCommerce	Ensemble-OOB	18.36	81.00	50.320	0.64
tomcat	OP-AIO-OOB	50.32	31.54	59.070	18.14
JGroups	OP-AIO-OOB	0.00	100.00	50.000	0.00
spring-integration	OP-AIO-OOB	5.91	95.75	49.170	-1.66
camel	OP-AIO-OOB	61.01	26.31	56.340	12.68
brackets	OP-AIO-OOB	41.28	42.87	57.925	15.85
nova	OP-AIO-OOB	50.37	15.95	66.840	33.68
fabric8	OP-AIO-OOB	51.50	20.88	63.810	27.62
neutron	OP-AIO-OOB	44.41	17.62	68.985	37.97

Continued on next page

Table C.22: FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
npm	OP-AIO-OOB	54.72	45.60	49.840	-0.32
BroadleafCommerce	OP-AIO-OOB	61.29	42.43	48.140	-3.72
tomcat	OP-Filter-OOB	49.49	32.05	59.230	18.46
JGroups	OP-Filter-OOB	0.00	100.00	50.000	0.00
spring-integration	OP-Filter-OOB	6.41	83.94	54.825	9.65
camel	OP-Filter-OOB	61.88	24.18	56.970	13.94
brackets	OP-Filter-OOB	38.40	34.18	63.710	27.42
nova	OP-Filter-OOB	33.73	36.57	64.850	29.70
fabric8	OP-Filter-OOB	27.69	61.09	55.610	11.22
neutron	OP-Filter-OOB	32.50	18.25	74.625	49.25
npm	OP-Filter-OOB	85.26	17.89	48.425	-3.15
BroadleafCommerce	OP-Filter-OOB	2.89	84.38	56.365	12.73

Table C.23: FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-ORB	55.74	37.24	53.510	7.02
JGroups	WP-ORB	39.09	66.41	47.250	-5.50
spring-integration	WP-ORB	39.80	66.33	46.935	-6.13
camel	WP-ORB	48.82	36.00	57.590	15.18
brackets	WP-ORB	9.74	87.48	51.390	2.78
nova	WP-ORB	40.33	32.23	63.720	27.44
fabric8	WP-ORB	31.67	70.12	49.105	-1.79
neutron	WP-ORB	19.35	33.38	73.635	47.27
npm	WP-ORB	65.55	35.21	49.620	-0.76
BroadleafCommerce	WP-ORB	33.69	67.84	49.235	-1.53
tomcat	AIO-ORB	60.58	30.79	54.315	8.63
JGroups	AIO-ORB	38.89	65.85	47.630	-4.74
spring-integration	AIO-ORB	48.50	24.47	63.515	27.03
camel	AIO-ORB	51.21	26.13	61.330	22.66

Continued on next page

Table C.23: FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
brackets	AIO-ORB	43.47	17.00	69.765	39.53
nova	AIO-ORB	45.82	17.13	68.525	37.05
fabric8	AIO-ORB	35.99	33.41	65.300	30.60
neutron	AIO-ORB	21.33	26.55	76.060	52.12
npm	AIO-ORB	72.32	25.80	50.940	1.88
BroadleafCommerce	AIO-ORB	51.16	33.30	57.770	15.54
tomcat	Filter-ORB	57.19	30.83	55.990	11.98
JGroups	Filter-ORB	37.27	63.08	49.825	-0.35
spring-integration	Filter-ORB	45.25	23.17	65.790	31.58
camel	Filter-ORB	52.28	25.23	61.245	22.49
brackets	Filter-ORB	42.54	18.85	69.305	38.61
nova	Filter-ORB	47.82	17.32	67.430	34.86
fabric8	Filter-ORB	36.98	33.34	64.840	29.68
neutron	Filter-ORB	21.17	26.80	76.015	52.03
npm	Filter-ORB	64.63	33.59	50.890	1.78
BroadleafCommerce	Filter-ORB	50.88	34.44	57.340	14.68
tomcat	Ensemble-ORB	14.09	84.72	50.595	1.19
JGroups	Ensemble-ORB	3.79	96.55	49.830	-0.34
spring-integration	Ensemble-ORB	21.27	76.48	51.125	2.25
camel	Ensemble-ORB	18.86	77.38	51.880	3.76
brackets	Ensemble-ORB	33.03	58.22	54.375	8.75
nova	Ensemble-ORB	21.36	65.29	56.675	13.35
fabric8	Ensemble-ORB	36.65	61.82	50.765	1.53
neutron	Ensemble-ORB	24.80	31.91	71.645	43.29
npm	Ensemble-ORB	29.57	70.61	49.910	-0.18
BroadleafCommerce	Ensemble-ORB	19.00	77.18	51.910	3.82
tomcat	OP-AIO-ORB	85.95	10.69	51.680	3.36
JGroups	OP-AIO-ORB	0.00	100.00	50.000	0.00
spring-integration	OP-AIO-ORB	35.42	34.45	65.065	30.13
camel	OP-AIO-ORB	50.91	39.95	54.570	9.14
brackets	OP-AIO-ORB	39.27	27.44	66.645	33.29

Continued on next page

Table C.23: FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
nova	OP-AIO-ORB	36.52	51.76	55.860	11.72
fabric8	OP-AIO-ORB	49.68	27.14	61.590	23.18
neutron	OP-AIO-ORB	27.63	39.76	66.305	32.61
npm	OP-AIO-ORB	28.64	61.09	55.135	10.27
BroadleafCommerce	OP-AIO-ORB	38.15	40.18	60.835	21.67
tomcat	OP-Filter-ORB	66.92	28.73	52.175	4.35
JGroups	OP-Filter-ORB	0.00	100.00	50.000	0.00
spring-integration	OP-Filter-ORB	34.59	50.96	57.225	14.45
camel	OP-Filter-ORB	52.81	38.38	54.405	8.81
brackets	OP-Filter-ORB	30.47	43.21	63.160	26.32
nova	OP-Filter-ORB	28.09	49.49	61.210	22.42
fabric8	OP-Filter-ORB	32.58	47.15	60.135	20.27
neutron	OP-Filter-ORB	24.81	30.82	72.185	44.37
npm	OP-Filter-ORB	0.00	100.00	50.000	0.00
BroadleafCommerce	OP-Filter-ORB	26.29	59.66	57.025	14.05

Table C.24: FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-OOB	68.77	20.82	55.205	10.41
JGroups	WP-OOB	64.13	33.44	51.215	2.43
spring-integration	WP-OOB	44.44	53.36	51.100	2.20
camel	WP-OOB	64.36	17.74	58.950	17.90
nova	WP-OOB	49.80	10.38	69.910	39.82
fabric8	WP-OOB	60.08	24.99	57.465	14.93
npm	WP-OOB	83.04	8.60	54.180	8.36
BroadleafCommerce	WP-OOB	69.17	16.16	57.335	14.67
tomcat	AIO-OOB	43.60	33.51	61.445	22.89
JGroups	AIO-OOB	55.39	37.13	53.740	7.48
spring-integration	AIO-OOB	39.82	38.50	60.840	21.68

Continued on next page

Table C.24: FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
camel	AIO-OOB	61.02	16.02	61.480	22.96
nova	AIO-OOB	49.63	10.62	69.875	39.75
fabric8	AIO-OOB	55.20	24.96	59.920	19.84
npm	AIO-OOB	38.72	34.51	63.385	26.77
BroadleafCommerce	AIO-OOB	36.97	26.43	68.300	36.60
tomcat	Filter-OOB	44.30	31.04	62.330	24.66
JGroups	Filter-OOB	54.67	38.26	53.535	7.07
spring-integration	Filter-OOB	33.61	41.75	62.320	24.64
camel	Filter-OOB	59.57	16.57	61.930	23.86
nova	Filter-OOB	49.92	10.44	69.820	39.64
fabric8	Filter-OOB	42.23	24.52	66.625	33.25
npm	Filter-OOB	34.08	34.37	65.775	31.55
BroadleafCommerce	Filter-OOB	38.31	28.08	66.805	33.61
tomcat	Ensemble-OOB	47.12	44.91	53.985	7.97
JGroups	Ensemble-OOB	12.99	87.42	49.795	-0.41
spring-integration	Ensemble-OOB	24.30	70.39	52.655	5.31
camel	Ensemble-OOB	41.34	49.33	54.665	9.33
nova	Ensemble-OOB	42.01	43.22	57.385	14.77
fabric8	Ensemble-OOB	53.96	35.18	55.430	10.86
npm	Ensemble-OOB	56.09	39.87	52.020	4.04
BroadleafCommerce	Ensemble-OOB	47.35	40.88	55.885	11.77
tomcat	OP-AIO-OOB	83.57	11.56	52.435	4.87
JGroups	OP-AIO-OOB	0.00	100.00	50.000	0.00
spring-integration	OP-AIO-OOB	60.26	23.86	57.940	15.88
camel	OP-AIO-OOB	66.85	21.00	56.075	12.15
nova	OP-AIO-OOB	48.78	14.80	68.210	36.42
fabric8	OP-AIO-OOB	92.62	1.62	52.880	5.76
npm	OP-AIO-OOB	70.35	27.05	51.300	2.60
BroadleafCommerce	OP-AIO-OOB	64.31	18.78	58.455	16.91
tomcat	OP-Filter-OOB	75.95	16.72	53.665	7.33
JGroups	OP-Filter-OOB	0.00	100.00	50.000	0.00

Continued on next page

Table C.24: FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
spring-integration	OP-Filter-OOB	85.34	7.24	53.710	7.42
camel	OP-Filter-OOB	65.49	20.74	56.885	13.77
nova	OP-Filter-OOB	39.95	22.03	69.010	38.02
fabric8	OP-Filter-OOB	90.43	6.35	51.610	3.22
npm	OP-Filter-OOB	96.41	4.08	49.755	-0.49
BroadleafCommerce	OP-Filter-OOB	51.41	31.57	58.510	17.02

Table C.25: FPR, FNR, BalancedAccuracy, Informedness for Drift period using ORB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-ORB	63.02	31.91	52.535	5.07
JGroups	WP-ORB	45.82	47.13	53.525	7.05
spring-integration	WP-ORB	19.27	76.21	52.260	4.52
camel	WP-ORB	68.35	20.33	55.660	11.32
nova	WP-ORB	49.87	9.38	70.375	40.75
fabric8	WP-ORB	57.56	25.05	58.695	17.39
npm	WP-ORB	75.44	15.22	54.670	9.34
BroadleafCommerce	WP-ORB	63.55	22.53	56.960	13.92
tomcat	AIO-ORB	48.81	35.17	58.010	16.02
JGroups	AIO-ORB	43.71	47.04	54.625	9.25
spring-integration	AIO-ORB	32.58	38.22	64.600	29.20
camel	AIO-ORB	60.88	19.55	59.785	19.57
nova	AIO-ORB	48.21	11.75	70.020	40.04
fabric8	AIO-ORB	53.60	27.40	59.500	19.00
npm	AIO-ORB	74.54	10.82	57.320	14.64
BroadleafCommerce	AIO-ORB	38.11	26.78	67.555	35.11
tomcat	Filter-ORB	49.59	34.51	57.950	15.90
JGroups	Filter-ORB	41.41	46.62	55.985	11.97
spring-integration	Filter-ORB	30.74	37.35	65.955	31.91
camel	Filter-ORB	60.38	21.30	59.160	18.32

Continued on next page

Table C.25: FPR, FNR, BalancedAccuracy, Informedness for Drift period using ORB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
nova	Filter-ORB	50.88	9.24	69.940	39.88
fabric8	Filter-ORB	43.54	27.77	64.345	28.69
npm	Filter-ORB	70.24	18.90	55.430	10.86
BroadleafCommerce	Filter-ORB	36.78	28.56	67.330	34.66
tomcat	Ensemble-ORB	36.56	52.70	55.370	10.74
JGroups	Ensemble-ORB	13.60	85.17	50.615	1.23
spring-integration	Ensemble-ORB	21.22	69.48	54.650	9.30
camel	Ensemble-ORB	50.16	35.57	57.135	14.27
nova	Ensemble-ORB	24.35	49.91	62.870	25.74
fabric8	Ensemble-ORB	55.85	30.54	56.805	13.61
npm	Ensemble-ORB	48.97	50.32	50.355	0.71
BroadleafCommerce	Ensemble-ORB	39.06	47.37	56.785	13.57
tomcat	OP-AIO-ORB	88.68	20.25	45.535	-8.93
JGroups	OP-AIO-ORB	0.00	100.00	50.000	0.00
spring-integration	OP-AIO-ORB	24.83	40.33	67.420	34.84
camel	OP-AIO-ORB	37.34	46.70	57.980	15.96
nova	OP-AIO-ORB	37.67	56.65	52.840	5.68
fabric8	OP-AIO-ORB	31.10	53.26	57.820	15.64
npm	OP-AIO-ORB	36.27	52.40	55.665	11.33
BroadleafCommerce	OP-AIO-ORB	15.83	68.34	57.915	15.83
tomcat	OP-Filter-ORB	88.15	21.16	45.345	-9.31
JGroups	OP-Filter-ORB	0.00	100.00	50.000	0.00
spring-integration	OP-Filter-ORB	14.75	63.46	60.895	21.79
camel	OP-Filter-ORB	30.58	51.56	58.930	17.86
nova	OP-Filter-ORB	28.01	45.22	63.385	26.77
fabric8	OP-Filter-ORB	23.23	65.58	55.595	11.19
npm	OP-Filter-ORB	0.00	100.00	50.000	0.00
BroadleafCommerce	OP-Filter-ORB	26.91	52.20	60.445	20.89

Table C.26: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-OOB	36.59	35.84	63.785	27.57
JGroups	WP-OOB	35.73	41.03	61.620	23.24
spring-integration	WP-OOB	31.91	34.18	66.955	33.91
camel	WP-OOB	34.29	29.29	68.210	36.42
brackets	WP-OOB	47.48	10.45	71.035	42.07
nova	WP-OOB	25.21	13.70	80.545	61.09
fabric8	WP-OOB	42.65	24.59	66.380	32.76
neutron	WP-OOB	28.05	7.98	81.985	63.97
npm	WP-OOB	43.14	33.62	61.620	23.24
BroadleafCommerce	WP-OOB	30.47	34.74	67.395	34.79
tomcat	AIO-OOB	35.82	37.78	63.200	26.40
JGroups	AIO-OOB	28.64	48.63	61.365	22.73
spring-integration	AIO-OOB	31.79	33.65	67.280	34.56
camel	AIO-OOB	42.11	25.85	66.020	32.04
brackets	AIO-OOB	34.29	19.58	73.065	46.13
nova	AIO-OOB	25.47	9.27	82.630	65.26
fabric8	AIO-OOB	42.18	27.71	65.055	30.11
neutron	AIO-OOB	26.92	7.64	82.720	65.44
npm	AIO-OOB	41.00	27.82	65.590	31.18
BroadleafCommerce	AIO-OOB	29.84	28.14	71.010	42.02
tomcat	Filter-OOB	28.83	41.58	64.795	29.59
JGroups	Filter-OOB	26.80	50.20	61.500	23.00
spring-integration	Filter-OOB	29.16	34.56	68.140	36.28
camel	Filter-OOB	40.40	26.73	66.435	32.87
brackets	Filter-OOB	30.50	24.26	72.620	45.24
nova	Filter-OOB	24.88	10.91	82.105	64.21
fabric8	Filter-OOB	37.55	25.73	68.360	36.72
neutron	Filter-OOB	25.54	8.68	82.890	65.78
npm	Filter-OOB	36.74	29.53	66.865	33.73
BroadleafCommerce	Filter-OOB	29.77	29.37	70.430	40.86
tomcat	Ensemble-OOB	36.58	56.02	53.700	7.40

Continued on next page

Table C.26: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using OOB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
JGroups	Ensemble-OOB	18.23	77.82	51.975	3.95
spring-integration	Ensemble-OOB	31.46	51.66	58.440	16.88
camel	Ensemble-OOB	49.97	38.32	55.855	11.71
brackets	Ensemble-OOB	36.57	33.99	64.720	29.44
nova	Ensemble-OOB	21.46	39.67	69.435	38.87
fabric8	Ensemble-OOB	52.47	36.47	55.530	11.06
neutron	Ensemble-OOB	21.41	39.23	69.680	39.36
npm	Ensemble-OOB	50.39	38.10	55.755	11.51
BroadleafCommerce	Ensemble-OOB	36.18	42.05	60.885	21.77
tomcat	OP-AIO-OOB	80.49	12.23	53.640	7.28
JGroups	OP-AIO-OOB	0.00	100.00	50.000	0.00
spring-integration	OP-AIO-OOB	50.94	26.73	61.165	22.33
camel	OP-AIO-OOB	78.29	14.78	53.465	6.93
brackets	OP-AIO-OOB	71.39	7.80	60.405	20.81
nova	OP-AIO-OOB	38.48	5.08	78.220	56.44
fabric8	OP-AIO-OOB	91.14	2.11	53.375	6.75
neutron	OP-AIO-OOB	29.05	18.43	76.260	52.52
npm	OP-AIO-OOB	20.31	83.30	48.195	-3.61
BroadleafCommerce	OP-AIO-OOB	54.31	30.17	57.760	15.52
tomcat	OP-Filter-OOB	74.86	15.63	54.755	9.51
JGroups	OP-Filter-OOB	0.00	100.00	50.000	0.00
spring-integration	OP-Filter-OOB	73.49	4.79	60.860	21.72
camel	OP-Filter-OOB	79.49	10.99	54.760	9.52
brackets	OP-Filter-OOB	51.79	7.32	70.445	40.89
nova	OP-Filter-OOB	26.75	21.96	75.645	51.29
fabric8	OP-Filter-OOB	87.41	10.35	51.120	2.24
neutron	OP-Filter-OOB	23.56	29.37	73.535	47.07
npm	OP-Filter-OOB	77.66	23.09	49.625	-0.75
BroadleafCommerce	OP-Filter-OOB	46.61	27.80	62.795	25.59

Table C.27: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using ORB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
tomcat	WP-ORB	37.67	35.93	63.200	26.40
JGroups	WP-ORB	37.82	40.28	60.950	21.90
spring-integration	WP-ORB	29.86	42.95	63.595	27.19
camel	WP-ORB	37.37	29.05	66.790	33.58
brackets	WP-ORB	41.88	16.29	70.915	41.83
nova	WP-ORB	20.58	18.50	80.460	60.92
fabric8	WP-ORB	37.03	27.11	67.930	35.86
neutron	WP-ORB	19.56	17.85	81.295	62.59
npm	WP-ORB	34.33	40.24	62.715	25.43
BroadleafCommerce	WP-ORB	35.67	33.99	65.170	30.34
tomcat	AIO-ORB	35.14	37.18	63.840	27.68
JGroups	AIO-ORB	36.17	40.67	61.580	23.16
spring-integration	AIO-ORB	29.85	28.27	70.940	41.88
camel	AIO-ORB	37.80	27.40	67.400	34.80
brackets	AIO-ORB	31.96	25.95	71.045	42.09
nova	AIO-ORB	20.01	18.92	80.535	61.07
fabric8	AIO-ORB	42.06	24.89	66.525	33.05
neutron	AIO-ORB	25.34	9.61	82.525	65.05
npm	AIO-ORB	38.38	24.45	68.585	37.17
BroadleafCommerce	AIO-ORB	30.55	30.33	69.560	39.12
tomcat	Filter-ORB	36.31	37.35	63.170	26.34
JGroups	Filter-ORB	36.30	40.32	61.690	23.38
spring-integration	Filter-ORB	29.47	28.14	71.195	42.39
camel	Filter-ORB	38.28	28.00	66.860	33.72
brackets	Filter-ORB	32.69	25.28	71.015	42.03
nova	Filter-ORB	21.20	18.04	80.380	60.76
fabric8	Filter-ORB	34.56	27.43	69.005	38.01
neutron	Filter-ORB	26.97	7.80	82.615	65.23
npm	Filter-ORB	40.80	26.21	66.495	32.99
BroadleafCommerce	Filter-ORB	30.83	31.03	69.070	38.14
tomcat	Ensemble-ORB	39.92	50.37	54.855	9.71

Continued on next page

Table C.27: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using ORB for Opensource data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
JGroups	Ensemble-ORB	17.84	77.58	52.290	4.58
spring-integration	Ensemble-ORB	27.81	53.37	59.410	18.82
camel	Ensemble-ORB	42.15	41.99	57.930	15.86
brackets	Ensemble-ORB	38.88	27.85	66.635	33.27
nova	Ensemble-ORB	23.42	23.95	76.315	52.63
fabric8	Ensemble-ORB	51.80	30.45	58.875	17.75
neutron	Ensemble-ORB	22.62	27.18	75.100	50.20
npm	Ensemble-ORB	52.15	35.52	56.165	12.33
BroadleafCommerce	Ensemble-ORB	39.91	36.28	61.905	23.81
tomcat	OP-AIO-ORB	46.01	51.70	51.145	2.29
JGroups	OP-AIO-ORB	0.00	100.00	50.000	0.00
spring-integration	OP-AIO-ORB	21.33	38.46	70.105	40.21
camel	OP-AIO-ORB	37.83	54.89	53.640	7.28
brackets	OP-AIO-ORB	48.71	14.48	68.405	36.81
nova	OP-AIO-ORB	48.55	27.17	62.140	24.28
fabric8	OP-AIO-ORB	23.71	63.84	56.225	12.45
neutron	OP-AIO-ORB	30.07	32.36	68.785	37.57
npm	OP-AIO-ORB	24.43	55.00	60.285	20.57
BroadleafCommerce	OP-AIO-ORB	13.80	77.35	54.425	8.85
tomcat	OP-Filter-ORB	44.10	61.36	47.270	-5.46
JGroups	OP-Filter-ORB	0.00	100.00	50.000	0.00
spring-integration	OP-Filter-ORB	17.16	56.52	63.160	26.32
camel	OP-Filter-ORB	47.61	56.69	47.850	-4.30
brackets	OP-Filter-ORB	52.40	11.05	68.275	36.55
nova	OP-Filter-ORB	31.00	12.95	78.025	56.05
fabric8	OP-Filter-ORB	21.41	72.19	53.200	6.40
neutron	OP-Filter-ORB	25.84	29.32	72.420	44.84
npm	OP-Filter-ORB	0.00	100.00	50.000	0.00
BroadleafCommerce	OP-Filter-ORB	22.09	67.41	55.250	10.50

Table C.28: FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-OOB	40.75	60.62	49.315	-1.37
c1	AIO-OOB-combined	48.39	51.70	49.955	-0.09
c1	AIO-OOB-proprietary	68.25	34.86	48.445	-3.11
c1	Filter-OOB-combined	46.44	50.82	51.370	2.74
c1	OfflineCP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c1	OfflineCP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c1	OfflineCP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c2	WP-OOB	83.66	12.85	51.745	3.49
c2	AIO-OOB-combined	77.76	23.58	49.330	-1.34
c2	AIO-OOB-proprietary	54.18	50.74	47.540	-4.92
c2	Filter-OOB-combined	54.78	36.19	54.515	9.03
c2	OfflineCP-AIO-OOB-combined	56.49	44.76	49.375	-1.25
c2	OfflineCP-AIO-OOB-proprietary	1.60	98.46	49.970	-0.06
c2	OfflineCP-Filter-OOB-combined	52.43	44.39	51.590	3.18
c3	WP-OOB	40.22	61.68	49.050	-1.90
c3	AIO-OOB-combined	58.48	30.43	55.545	11.09
c3	AIO-OOB-proprietary	73.59	33.55	46.430	-7.14
c3	Filter-OOB-combined	47.40	40.33	56.135	12.27
c3	OfflineCP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c3	OfflineCP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c3	OfflineCP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c4	WP-OOB	78.43	18.04	51.765	3.53
c4	AIO-OOB-combined	62.21	23.35	57.220	14.44
c4	AIO-OOB-proprietary	33.40	44.55	61.025	22.05
c4	Filter-OOB-combined	38.84	45.58	57.790	15.58
c4	OfflineCP-AIO-OOB-combined	49.01	38.40	56.295	12.59
c4	OfflineCP-AIO-OOB-proprietary	59.51	37.30	51.595	3.19
c4	OfflineCP-Filter-OOB-combined	50.75	38.67	55.290	10.58
c5	WP-OOB	88.06	14.56	48.690	-2.62
c5	AIO-OOB-combined	65.14	32.28	51.290	2.58
c5	AIO-OOB-proprietary	77.47	9.94	56.295	12.59

Continued on next page

Table C.28: FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c5	Filter-OOB-combined	61.44	32.93	52.815	5.63
c5	OfflineCP-AIO-OOB-combined	23.61	73.19	51.600	3.20
c5	OfflineCP-AIO-OOB-proprietary	57.24	32.19	55.285	10.57
c5	OfflineCP-Filter-OOB-combined	76.46	17.56	52.990	5.98
c6	WP-OOB	89.02	12.08	49.450	-1.10
c6	AIO-OOB-combined	66.84	30.46	51.350	2.70
c6	AIO-OOB-proprietary	59.71	35.60	52.345	4.69
c6	Filter-OOB-combined	64.78	30.42	52.400	4.80
c6	OfflineCP-AIO-OOB-combined	51.39	43.34	52.635	5.27
c6	OfflineCP-AIO-OOB-proprietary	29.03	69.25	50.860	1.72
c6	OfflineCP-Filter-OOB-combined	45.06	51.33	51.805	3.61
c7	WP-OOB	77.22	16.47	53.155	6.31
c7	AIO-OOB-combined	61.74	29.30	54.480	8.96
c7	AIO-OOB-proprietary	76.88	6.67	58.225	16.45
c7	Filter-OOB-combined	55.30	38.01	53.345	6.69
c7	OfflineCP-AIO-OOB-combined	22.48	61.13	58.195	16.39
c7	OfflineCP-AIO-OOB-proprietary	65.43	30.51	52.030	4.06
c7	OfflineCP-Filter-OOB-combined	73.01	34.28	46.355	-7.29
c8	WP-OOB	86.95	16.89	48.080	-3.84
c8	AIO-OOB-combined	53.42	39.69	53.445	6.89
c8	AIO-OOB-proprietary	70.43	25.15	52.210	4.42
c8	Filter-OOB-combined	53.39	51.58	47.515	-4.97
c8	OfflineCP-AIO-OOB-combined	33.30	65.10	50.800	1.60
c8	OfflineCP-AIO-OOB-proprietary	22.36	67.84	54.900	9.80
c8	OfflineCP-Filter-OOB-combined	75.05	25.87	49.540	-0.92
c9	WP-OOB	79.36	13.47	53.585	7.17
c9	AIO-OOB-combined	55.46	30.81	56.865	13.73
c9	AIO-OOB-proprietary	40.48	47.77	55.875	11.75
c9	Filter-OOB-combined	47.80	41.91	55.145	10.29
c9	OfflineCP-AIO-OOB-combined	40.19	58.37	50.720	1.44
c9	OfflineCP-AIO-OOB-proprietary	6.18	86.75	53.535	7.07

Continued on next page

Table C.28: FPR, FNR, BalancedAccuracy, Informedness for overall period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c9	OfflineCP-Filter-OOB-combined	48.00	53.76	49.120	-1.76

Table C.29: FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-ORB	35.30	69.25	47.725	-4.55
c1	AIO-ORB-combined	54.60	48.15	48.625	-2.75
c1	AIO-ORB-proprietary	64.35	42.33	46.660	-6.68
c1	Filter-ORB-combined	30.43	62.05	53.760	7.52
c1	OfflineCP-AIO-ORB-combined	0.00	100.00	50.000	0.00
c1	OfflineCP-AIO-ORB-proprietary	0.00	100.00	50.000	0.00
c1	OfflineCP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c2	WP-ORB	28.53	82.50	44.485	-11.03
c2	AIO-ORB-combined	64.12	39.11	48.385	-3.23
c2	AIO-ORB-proprietary	61.84	44.01	47.075	-5.85
c2	Filter-ORB-combined	53.85	56.01	45.070	-9.86
c2	OfflineCP-AIO-ORB-combined	17.02	74.24	54.370	8.74
c2	OfflineCP-AIO-ORB-proprietary	12.36	84.80	51.420	2.84
c2	OfflineCP-Filter-ORB-combined	10.47	76.87	56.330	12.66
c3	WP-ORB	41.33	50.17	54.250	8.50
c3	AIO-ORB-combined	42.62	49.18	54.100	8.20
c3	AIO-ORB-proprietary	39.38	51.78	54.420	8.84
c3	Filter-ORB-combined	36.76	52.79	55.225	10.45
c3	OfflineCP-AIO-ORB-combined	0.00	100.00	50.000	0.00
c3	OfflineCP-AIO-ORB-proprietary	0.00	100.00	50.000	0.00
c3	OfflineCP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c4	WP-ORB	31.07	60.22	54.355	8.71
c4	AIO-ORB-combined	70.49	21.63	53.940	7.88
c4	AIO-ORB-proprietary	63.46	22.90	56.820	13.64
c4	Filter-ORB-combined	24.53	60.57	57.450	14.90

Continued on next page

Table C.29: FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c4	OfflineCP-AIO-ORB-combined	50.96	36.17	56.435	12.87
c4	OfflineCP-AIO-ORB-proprietary	63.66	28.39	53.975	7.95
c4	OfflineCP-Filter-ORB-combined	10.26	59.26	65.240	30.48
c5	WP-ORB	4.07	94.66	50.635	1.27
c5	AIO-ORB-combined	87.95	8.86	51.595	3.19
c5	AIO-ORB-proprietary	87.16	9.01	51.915	3.83
c5	Filter-ORB-combined	78.06	19.09	51.425	2.85
c5	OfflineCP-AIO-ORB-combined	52.34	34.94	56.360	12.72
c5	OfflineCP-AIO-ORB-proprietary	64.40	25.85	54.875	9.75
c5	OfflineCP-Filter-ORB-combined	55.29	27.16	58.775	17.55
c6	WP-ORB	36.25	57.65	53.050	6.10
c6	AIO-ORB-combined	47.07	48.81	52.060	4.12
c6	AIO-ORB-proprietary	47.56	44.71	53.865	7.73
c6	Filter-ORB-combined	43.53	57.70	49.385	-1.23
c6	OfflineCP-AIO-ORB-combined	25.03	56.63	59.170	18.34
c6	OfflineCP-AIO-ORB-proprietary	23.92	72.37	51.855	3.71
c6	OfflineCP-Filter-ORB-combined	38.00	53.62	54.190	8.38
c7	WP-ORB	51.97	43.22	52.405	4.81
c7	AIO-ORB-combined	65.52	22.98	55.750	11.50
c7	AIO-ORB-proprietary	54.03	28.74	58.615	17.23
c7	Filter-ORB-combined	35.11	43.80	60.545	21.09
c7	OfflineCP-AIO-ORB-combined	43.97	49.40	53.315	6.63
c7	OfflineCP-AIO-ORB-proprietary	64.23	38.82	48.475	-3.05
c7	OfflineCP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c8	WP-ORB	55.38	56.63	43.995	-12.01
c8	AIO-ORB-combined	62.06	29.30	54.320	8.64
c8	AIO-ORB-proprietary	62.94	27.55	54.755	9.51
c8	Filter-ORB-combined	44.28	55.69	50.015	0.03
c8	OfflineCP-AIO-ORB-combined	25.58	62.88	55.770	11.54
c8	OfflineCP-AIO-ORB-proprietary	10.74	75.05	57.105	14.21
c8	OfflineCP-Filter-ORB-combined	20.27	70.74	54.495	8.99

Continued on next page

Table C.29: FPR, FNR, BalancedAccuracy, Informedness for overall period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c9	WP-ORB	54.34	49.96	47.850	-4.30
c9	AIO-ORB-combined	60.66	36.50	51.420	2.84
c9	AIO-ORB-proprietary	55.62	38.40	52.990	5.98
c9	Filter-ORB-combined	63.06	35.76	50.590	1.18
c9	OfflineCP-AIO-ORB-combined	15.38	84.14	50.240	0.48
c9	OfflineCP-AIO-ORB-proprietary	0.10	99.70	50.100	0.20
c9	OfflineCP-Filter-ORB-combined	0.04	97.29	51.335	2.67

Table C.30: FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-OOB	91.12	22.14	43.370	-13.26
c2	WP-OOB	84.61	13.13	51.130	2.26
c3	WP-OOB	3.68	95.80	50.260	0.52
c4	WP-OOB	8.18	95.93	47.945	-4.11
c5	WP-OOB	12.63	91.75	47.810	-4.38
c6	WP-OOB	8.24	97.98	46.890	-6.22
c7	WP-OOB	0.00	99.60	50.200	0.40
c8	WP-OOB	0.00	99.77	50.115	0.23
c9	WP-OOB	13.58	93.16	46.630	-6.74
c1	AIO-OOB-combined	26.96	76.62	48.210	-3.58
c2	AIO-OOB-combined	78.07	24.08	48.925	-2.15
c3	AIO-OOB-combined	18.14	78.57	51.645	3.29
c4	AIO-OOB-combined	18.76	86.67	47.285	-5.43
c5	AIO-OOB-combined	0.00	99.23	50.385	0.77
c6	AIO-OOB-combined	0.00	99.35	50.325	0.65
c7	AIO-OOB-combined	17.85	66.04	58.055	16.11
c8	AIO-OOB-combined	0.00	98.02	50.990	1.98
c9	AIO-OOB-combined	0.20	92.92	53.440	6.88
c1	AIO-OOB-proprietary	91.78	21.12	43.550	-12.90

Continued on next page

Table C.30: FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c2	AIO-OOB-proprietary	54.03	51.64	47.165	-5.67
c3	AIO-OOB-proprietary	66.78	29.39	51.915	3.83
c4	AIO-OOB-proprietary	9.83	89.43	50.370	0.74
c5	AIO-OOB-proprietary	40.84	59.31	49.925	-0.15
c6	AIO-OOB-proprietary	0.00	99.35	50.325	0.65
c7	AIO-OOB-proprietary	35.34	38.97	62.845	25.69
c8	AIO-OOB-proprietary	13.25	70.97	57.890	15.78
c9	AIO-OOB-proprietary	2.13	86.39	55.740	11.48
c1	Filter-OOB-combined	29.71	68.50	50.895	1.79
c2	Filter-OOB-combined	54.71	36.81	54.240	8.48
c3	Filter-OOB-combined	3.10	95.80	50.550	1.10
c4	Filter-OOB-combined	0.00	100.00	50.000	0.00
c5	Filter-OOB-combined	8.00	93.51	49.245	-1.51
c6	Filter-OOB-combined	0.00	99.50	50.250	0.50
c7	Filter-OOB-combined	0.00	100.00	50.000	0.00
c8	Filter-OOB-combined	0.00	99.80	50.100	0.20
c9	Filter-OOB-combined	0.00	99.81	50.095	0.19
c1	OP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c2	OP-AIO-OOB-combined	55.76	44.96	49.640	-0.72
c3	OP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c4	OP-AIO-OOB-combined	19.00	85.37	47.815	-4.37
c5	OP-AIO-OOB-combined	0.00	97.53	51.235	2.47
c6	OP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c7	OP-AIO-OOB-combined	20.81	67.38	55.905	11.81
c8	OP-AIO-OOB-combined	0.85	92.91	53.120	6.24
c9	OP-AIO-OOB-combined	0.00	91.09	54.455	8.91
c1	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c2	OP-AIO-OOB-proprietary	1.66	98.41	49.965	-0.07
c3	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c4	OP-AIO-OOB-proprietary	7.66	93.18	49.580	-0.84
c5	OP-AIO-OOB-proprietary	0.00	99.05	50.475	0.95

Continued on next page

Table C.30: FPR, FNR, BalancedAccuracy, Informedness for Initial period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c6	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c7	OP-AIO-OOB-proprietary	5.40	82.79	55.905	11.81
c8	OP-AIO-OOB-proprietary	12.02	75.69	56.145	12.29
c9	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c1	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c2	OP-Filter-OOB-combined	51.58	44.60	51.910	3.82
c3	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c4	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c5	OP-Filter-OOB-combined	31.34	57.50	55.580	11.16
c6	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c7	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c8	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c9	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00

Table C.31: FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-ORB	19.54	85.85	47.305	-5.39
c2	WP-ORB	27.81	83.54	44.325	-11.35
c3	WP-ORB	41.26	50.99	53.875	7.75
c4	WP-ORB	0.20	98.21	50.795	1.59
c5	WP-ORB	3.35	96.07	50.290	0.58
c6	WP-ORB	0.26	99.33	50.205	0.41
c7	WP-ORB	1.32	99.39	49.645	-0.71
c8	WP-ORB	0.69	99.65	49.830	-0.34
c9	WP-ORB	0.48	99.79	49.865	-0.27
c1	AIO-ORB-combined	42.44	63.05	47.255	-5.49
c2	AIO-ORB-combined	64.49	39.71	47.900	-4.20
c3	AIO-ORB-combined	42.33	50.24	53.715	7.43
c4	AIO-ORB-combined	47.57	50.74	50.845	1.69

Continued on next page

Table C.31: FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c5	AIO-ORB-combined	90.71	7.26	51.015	2.03
c6	AIO-ORB-combined	33.90	73.67	46.215	-7.57
c7	AIO-ORB-combined	37.64	57.35	52.505	5.01
c8	AIO-ORB-combined	50.52	35.91	56.785	13.57
c9	AIO-ORB-combined	12.53	80.73	53.370	6.74
c1	AIO-ORB-proprietary	51.82	58.86	44.660	-10.68
c2	AIO-ORB-proprietary	61.62	44.61	46.885	-6.23
c3	AIO-ORB-proprietary	39.44	52.62	53.970	7.94
c4	AIO-ORB-proprietary	35.36	51.85	56.395	12.79
c5	AIO-ORB-proprietary	89.82	7.46	51.360	2.72
c6	AIO-ORB-proprietary	20.44	78.48	50.540	1.08
c7	AIO-ORB-proprietary	38.65	60.60	50.375	0.75
c8	AIO-ORB-proprietary	45.47	38.33	58.100	16.20
c9	AIO-ORB-proprietary	8.45	79.69	55.930	11.86
c1	Filter-ORB-combined	16.68	76.21	53.555	7.11
c2	Filter-ORB-combined	53.83	57.01	44.580	-10.84
c3	Filter-ORB-combined	36.74	53.55	54.855	9.71
c4	Filter-ORB-combined	0.23	91.95	53.910	7.82
c5	Filter-ORB-combined	79.93	18.20	50.935	1.87
c6	Filter-ORB-combined	30.17	80.47	44.680	-10.64
c7	Filter-ORB-combined	3.85	92.72	51.715	3.43
c8	Filter-ORB-combined	20.90	78.36	50.370	0.74
c9	Filter-ORB-combined	11.82	84.16	52.010	4.02
c1	OP-AIO-ORB-combined	0.00	100.00	50.000	0.00
c2	OP-AIO-ORB-combined	17.46	73.85	54.345	8.69
c3	OP-AIO-ORB-combined	0.00	100.00	50.000	0.00
c4	OP-AIO-ORB-combined	34.64	54.05	55.655	11.31
c5	OP-AIO-ORB-combined	53.14	34.63	56.115	12.23
c6	OP-AIO-ORB-combined	15.96	79.31	52.365	4.73
c7	OP-AIO-ORB-combined	42.33	47.10	55.285	10.57
c8	OP-AIO-ORB-combined	34.12	53.63	56.125	12.25

Continued on next page

Table C.31: FPR, FNR, BalancedAccuracy, Informedness for Initial period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c9	OP-AIO-ORB-combined	0.10	94.69	52.605	5.21
c1	OP-AIO-ORB-proprietary	0.00	100.00	50.000	0.00
c2	OP-AIO-ORB-proprietary	12.40	84.85	51.375	2.75
c3	OP-AIO-ORB-proprietary	0.00	100.00	50.000	0.00
c4	OP-AIO-ORB-proprietary	31.09	58.06	55.425	10.85
c5	OP-AIO-ORB-proprietary	64.36	25.32	55.160	10.32
c6	OP-AIO-ORB-proprietary	4.91	96.98	49.055	-1.89
c7	OP-AIO-ORB-proprietary	26.91	68.22	52.435	4.87
c8	OP-AIO-ORB-proprietary	21.91	56.86	60.615	21.23
c9	OP-AIO-ORB-proprietary	0.16	99.67	50.085	0.17
c1	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c2	OP-Filter-ORB-combined	10.77	76.60	56.315	12.63
c3	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c4	OP-Filter-ORB-combined	2.52	88.92	54.280	8.56
c5	OP-Filter-ORB-combined	55.63	26.28	59.045	18.09
c6	OP-Filter-ORB-combined	8.09	91.30	50.305	0.61
c7	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c8	OP-Filter-ORB-combined	20.91	70.94	54.075	8.15
c9	OP-Filter-ORB-combined	0.00	99.13	50.435	0.87

Table C.32: FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-OOB	15.23	85.83	49.470	-1.06
c3	WP-OOB	49.90	61.69	44.205	-11.59
c4	WP-OOB	88.03	7.36	52.305	4.61
c6	WP-OOB	94.85	6.09	49.530	-0.94
c7	WP-OOB	92.86	4.38	51.380	2.76
c8	WP-OOB	99.37	3.98	48.325	-3.35
c9	WP-OOB	88.78	3.40	53.910	7.82

Continued on next page

Table C.32: FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	AIO-OOB-combined	73.89	24.96	50.575	1.15
c3	AIO-OOB-combined	72.46	15.78	55.880	11.76
c4	AIO-OOB-combined	69.26	13.08	58.830	17.66
c6	AIO-OOB-combined	73.67	23.83	51.250	2.50
c7	AIO-OOB-combined	76.01	21.07	51.460	2.92
c8	AIO-OOB-combined	70.12	32.08	48.900	-2.20
c9	AIO-OOB-combined	65.60	20.92	56.740	13.48
c1	AIO-OOB-proprietary	57.34	44.17	49.245	-1.51
c3	AIO-OOB-proprietary	70.56	46.27	41.585	-16.83
c4	AIO-OOB-proprietary	36.49	39.15	62.180	24.36
c6	AIO-OOB-proprietary	65.34	30.68	51.990	3.98
c7	AIO-OOB-proprietary	89.05	3.12	53.915	7.83
c8	AIO-OOB-proprietary	85.09	15.03	49.940	-0.12
c9	AIO-OOB-proprietary	46.65	44.00	54.675	9.35
c1	Filter-OOB-combined	67.88	28.90	51.610	3.22
c3	Filter-OOB-combined	65.31	20.69	57.000	14.00
c4	Filter-OOB-combined	48.27	32.60	59.565	19.13
c6	Filter-OOB-combined	71.70	23.56	52.370	4.74
c7	Filter-OOB-combined	78.63	20.45	50.460	0.92
c8	Filter-OOB-combined	76.81	28.99	47.100	-5.80
c9	Filter-OOB-combined	57.64	30.92	55.720	11.44
c1	OP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c3	OP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c4	OP-AIO-OOB-combined	52.76	28.84	59.200	18.40
c6	OP-AIO-OOB-combined	54.74	38.44	53.410	6.82
c7	OP-AIO-OOB-combined	16.28	69.30	57.210	14.42
c8	OP-AIO-OOB-combined	59.54	49.25	45.605	-8.79
c9	OP-AIO-OOB-combined	48.93	51.08	49.995	-0.01
c1	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c3	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c4	OP-AIO-OOB-proprietary	67.51	28.18	52.155	4.31

Continued on next page

Table C.32: FPR, FNR, BalancedAccuracy, Informedness for Drift period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c6	OP-AIO-OOB-proprietary	30.95	66.51	51.270	2.54
c7	OP-AIO-OOB-proprietary	83.48	18.90	48.810	-2.38
c8	OP-AIO-OOB-proprietary	21.60	70.18	54.110	8.22
c9	OP-AIO-OOB-proprietary	7.46	83.81	54.365	8.73
c1	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c3	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c4	OP-Filter-OOB-combined	60.81	25.27	56.960	13.92
c6	OP-Filter-OOB-combined	47.34	47.20	52.730	5.46
c7	OP-Filter-OOB-combined	86.29	25.99	43.860	-12.28
c8	OP-Filter-OOB-combined	91.27	11.72	48.505	-2.99
c9	OP-Filter-OOB-combined	58.35	44.00	48.825	-2.35

Table C.33: FPR, FNR, BalancedAccuracy, Informedness for Drift period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-ORB	71.37	33.95	47.340	-5.32
c6	WP-ORB	84.67	17.97	48.680	-2.64
c7	WP-ORB	88.02	13.68	49.150	-1.70
c8	WP-ORB	94.78	20.77	42.225	-15.55
c9	WP-ORB	93.45	14.60	45.975	-8.05
c1	AIO-ORB-combined	81.24	18.67	50.045	0.09
c6	AIO-ORB-combined	56.43	35.07	54.250	8.50
c7	AIO-ORB-combined	84.96	12.20	51.420	2.84
c8	AIO-ORB-combined	71.10	27.19	50.855	1.71
c9	AIO-ORB-combined	80.14	16.28	51.790	3.58
c1	AIO-ORB-proprietary	90.41	10.16	49.715	-0.57
c6	AIO-ORB-proprietary	82.76	15.77	50.735	1.47
c7	AIO-ORB-proprietary	61.17	25.46	56.685	13.37
c8	AIO-ORB-proprietary	73.69	23.74	51.285	2.57
c9	AIO-ORB-proprietary	92.23	11.90	47.935	-4.13

Continued on next page

Table C.33: FPR, FNR, BalancedAccuracy, Informedness for Drift period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	Filter-ORB-combined	60.21	33.71	53.040	6.08
c6	Filter-ORB-combined	64.97	37.45	48.790	-2.42
c7	Filter-ORB-combined	47.01	30.53	61.230	22.46
c8	Filter-ORB-combined	53.19	41.61	52.600	5.20
c9	Filter-ORB-combined	95.32	8.37	48.155	-3.69
c1	OP-AIO-ORB-combined	0.00	100.00	50.000	0.00
c6	OP-AIO-ORB-combined	29.45	45.95	62.300	24.60
c7	OP-AIO-ORB-combined	34.80	57.09	54.055	8.11
c8	OP-AIO-ORB-combined	26.65	66.32	53.515	7.03
c9	OP-AIO-ORB-combined	29.59	71.26	49.575	-0.85
c1	OP-AIO-ORB-proprietary	0.00	100.00	50.000	0.00
c6	OP-AIO-ORB-proprietary	51.47	49.00	49.765	-0.47
c7	OP-AIO-ORB-proprietary	75.52	22.08	51.200	2.40
c8	OP-AIO-ORB-proprietary	5.89	87.77	53.170	6.34
c9	OP-AIO-ORB-proprietary	0.08	99.84	50.040	0.08
c1	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c6	OP-Filter-ORB-combined	72.81	23.32	51.935	3.87
c7	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c8	OP-Filter-ORB-combined	35.61	66.85	48.770	-2.46
c9	OP-Filter-ORB-combined	0.00	96.77	51.615	3.23

Table C.34: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-OOB	31.33	59.06	54.805	9.61
c2	WP-OOB	56.21	4.59	69.600	39.20
c3	WP-OOB	47.94	39.32	56.370	12.74
c4	WP-OOB	68.12	29.67	51.105	2.21
c5	WP-OOB	89.99	12.60	48.705	-2.59
c6	WP-OOB	73.06	27.70	49.620	-0.76

Continued on next page

Table C.34: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c7	WP-OOB	70.56	19.06	55.190	10.38
c8	WP-OOB	86.59	17.90	47.755	-4.49
c9	WP-OOB	68.16	19.10	56.370	12.74
c1	AIO-OOB-combined	29.90	68.75	50.675	1.35
c2	AIO-OOB-combined	69.03	9.37	60.800	21.60
c3	AIO-OOB-combined	61.46	23.46	57.540	15.08
c4	AIO-OOB-combined	51.77	38.67	54.780	9.56
c5	AIO-OOB-combined	66.80	30.59	51.305	2.61
c6	AIO-OOB-combined	40.17	55.35	52.240	4.48
c7	AIO-OOB-combined	52.73	33.25	57.010	14.02
c8	AIO-OOB-combined	47.69	39.51	56.400	12.80
c9	AIO-OOB-combined	32.85	47.33	59.910	19.82
c1	AIO-OOB-proprietary	62.31	33.81	51.940	3.88
c2	AIO-OOB-proprietary	58.53	24.92	58.275	16.55
c3	AIO-OOB-proprietary	83.18	14.91	50.955	1.91
c4	AIO-OOB-proprietary	30.44	48.54	60.510	21.02
c5	AIO-OOB-proprietary	78.41	8.69	56.450	12.90
c6	AIO-OOB-proprietary	39.03	50.34	55.315	10.63
c7	AIO-OOB-proprietary	69.64	6.55	61.905	23.81
c8	AIO-OOB-proprietary	66.25	27.53	53.110	6.22
c9	AIO-OOB-proprietary	29.99	43.89	63.060	26.12
c1	Filter-OOB-combined	29.65	67.45	51.450	2.90
c2	Filter-OOB-combined	56.77	18.24	62.495	24.99
c3	Filter-OOB-combined	46.40	36.97	58.315	16.63
c4	Filter-OOB-combined	18.06	73.94	54.000	8.00
c5	Filter-OOB-combined	62.80	31.40	52.900	5.80
c6	Filter-OOB-combined	36.86	56.76	53.190	6.38
c7	Filter-OOB-combined	38.72	48.26	56.510	13.02
c8	Filter-OOB-combined	43.57	61.35	47.540	-4.92
c9	Filter-OOB-combined	21.91	67.78	55.155	10.31
c1	OP-AIO-OOB-combined	0.00	100.00	50.000	0.00

Continued on next page

Table C.34: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using OOB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c2	OP-AIO-OOB-combined	78.04	39.05	41.455	-17.09
c3	OP-AIO-OOB-combined	0.00	100.00	50.000	0.00
c4	OP-AIO-OOB-combined	45.91	56.87	48.610	-2.78
c5	OP-AIO-OOB-combined	24.21	72.58	51.605	3.21
c6	OP-AIO-OOB-combined	43.71	60.02	48.135	-3.73
c7	OP-AIO-OOB-combined	28.70	52.49	59.405	18.81
c8	OP-AIO-OOB-combined	20.04	72.45	53.755	7.51
c9	OP-AIO-OOB-combined	15.63	79.46	52.455	4.91
c1	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c2	OP-AIO-OOB-proprietary	0.02	99.82	50.080	0.16
c3	OP-AIO-OOB-proprietary	0.00	100.00	50.000	0.00
c4	OP-AIO-OOB-proprietary	48.54	50.97	50.245	0.49
c5	OP-AIO-OOB-proprietary	58.70	30.50	55.400	10.80
c6	OP-AIO-OOB-proprietary	24.49	78.83	48.340	-3.32
c7	OP-AIO-OOB-proprietary	54.51	36.04	54.725	9.45
c8	OP-AIO-OOB-proprietary	23.69	65.77	55.270	10.54
c9	OP-AIO-OOB-proprietary	2.84	95.26	50.950	1.90
c1	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c2	OP-Filter-OOB-combined	77.12	38.60	42.140	-15.72
c3	OP-Filter-OOB-combined	0.00	100.00	50.000	0.00
c4	OP-Filter-OOB-combined	31.82	66.12	51.030	2.06
c5	OP-Filter-OOB-combined	77.61	16.55	52.920	5.84
c6	OP-Filter-OOB-combined	42.75	65.22	46.015	-7.97
c7	OP-Filter-OOB-combined	68.17	35.08	48.375	-3.25
c8	OP-Filter-OOB-combined	71.40	28.36	50.120	0.24
c9	OP-Filter-OOB-combined	19.18	80.28	50.270	0.54

Table C.35: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c1	WP-ORB	51.78	46.69	50.765	1.53
c2	WP-ORB	50.87	50.93	49.100	-1.80
c3	WP-ORB	42.77	34.68	61.275	22.55
c4	WP-ORB	42.65	45.97	55.690	11.38
c5	WP-ORB	12.34	78.59	54.535	9.07
c6	WP-ORB	52.14	35.56	56.150	12.30
c7	WP-ORB	52.16	40.79	53.525	7.05
c8	WP-ORB	75.01	42.00	41.495	-17.01
c9	WP-ORB	75.00	30.83	47.085	-5.83
c1	AIO-ORB-combined	69.66	23.57	53.385	6.77
c2	AIO-ORB-combined	52.89	20.76	63.175	26.35
c3	AIO-ORB-combined	48.13	29.33	61.270	22.54
c4	AIO-ORB-combined	79.09	10.72	55.095	10.19
c5	AIO-ORB-combined	56.48	27.16	58.180	16.36
c6	AIO-ORB-combined	54.79	33.37	55.920	11.84
c7	AIO-ORB-combined	65.72	19.95	57.165	14.33
c8	AIO-ORB-combined	66.08	26.50	53.710	7.42
c9	AIO-ORB-combined	79.87	18.98	50.575	1.15
c1	AIO-ORB-proprietary	82.67	13.94	51.695	3.39
c2	AIO-ORB-proprietary	68.93	25.63	52.720	5.44
c3	AIO-ORB-proprietary	38.37	36.01	62.810	25.62
c4	AIO-ORB-proprietary	74.00	12.04	56.980	13.96
c5	AIO-ORB-proprietary	56.85	26.79	58.180	16.36
c6	AIO-ORB-proprietary	59.82	26.07	57.055	14.11
c7	AIO-ORB-proprietary	54.90	24.50	60.300	20.60
c8	AIO-ORB-proprietary	69.51	23.04	53.725	7.45
c9	AIO-ORB-proprietary	73.59	22.41	52.000	4.00
c1	Filter-ORB-combined	48.17	39.30	56.265	12.53
c2	Filter-ORB-combined	54.63	25.37	60.000	20.00
c3	Filter-ORB-combined	37.06	38.60	62.170	24.34
c4	Filter-ORB-combined	33.65	48.81	58.770	17.54

Continued on next page

Table C.35: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c5	Filter-ORB-combined	56.75	29.38	56.935	13.87
c6	Filter-ORB-combined	48.64	45.29	53.035	6.07
c7	Filter-ORB-combined	37.44	39.03	61.765	23.53
c8	Filter-ORB-combined	53.98	47.19	49.415	-1.17
c9	Filter-ORB-combined	82.95	16.84	50.105	0.21
c1	OP-AIO-ORB-combined	0.00	100.00	50.000	0.00
c2	OP-AIO-ORB-combined	3.76	86.37	54.935	9.87
c3	OP-AIO-ORB-combined	0.00	100.00	50.000	0.00
c4	OP-AIO-ORB-combined	57.08	29.47	56.725	13.45
c5	OP-AIO-ORB-combined	43.29	38.55	59.080	18.16
c6	OP-AIO-ORB-combined	30.82	42.12	63.530	27.06
c7	OP-AIO-ORB-combined	46.18	48.12	52.850	5.70
c8	OP-AIO-ORB-combined	21.34	66.74	55.960	11.92
c9	OP-AIO-ORB-combined	21.09	80.35	49.280	-1.44
c1	OP-AIO-ORB-proprietary	0.00	100.00	50.000	0.00
c2	OP-AIO-ORB-proprietary	11.43	83.22	52.675	5.35
c3	OP-AIO-ORB-proprietary	0.00	100.00	50.000	0.00
c4	OP-AIO-ORB-proprietary	75.88	17.28	53.420	6.84
c5	OP-AIO-ORB-proprietary	64.98	31.99	51.515	3.03
c6	OP-AIO-ORB-proprietary	31.85	59.31	54.420	8.84
c7	OP-AIO-ORB-proprietary	67.63	37.82	47.275	-5.45
c8	OP-AIO-ORB-proprietary	6.21	81.64	56.075	12.15
c9	OP-AIO-ORB-proprietary	0.08	99.71	50.105	0.21
c1	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c2	OP-Filter-ORB-combined	1.17	85.40	56.715	13.43
c3	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c4	OP-Filter-ORB-combined	13.18	48.15	69.335	38.67
c5	OP-Filter-ORB-combined	51.43	37.28	55.645	11.29
c6	OP-Filter-ORB-combined	52.44	32.37	57.595	15.19
c7	OP-Filter-ORB-combined	0.00	100.00	50.000	0.00
c8	OP-Filter-ORB-combined	17.44	71.30	55.630	11.26

Continued on next page

Table C.35: FPR, FNR, BalancedAccuracy, Informedness for NonDrift period using ORB for Proprietary data

Dataset	Approach	FPR	FNR	BalancedAcc	Informedness
c9	OP-Filter-ORB-combined	0.06	96.55	51.695	3.39

C.3 Plots of Predictive Performance Through Time

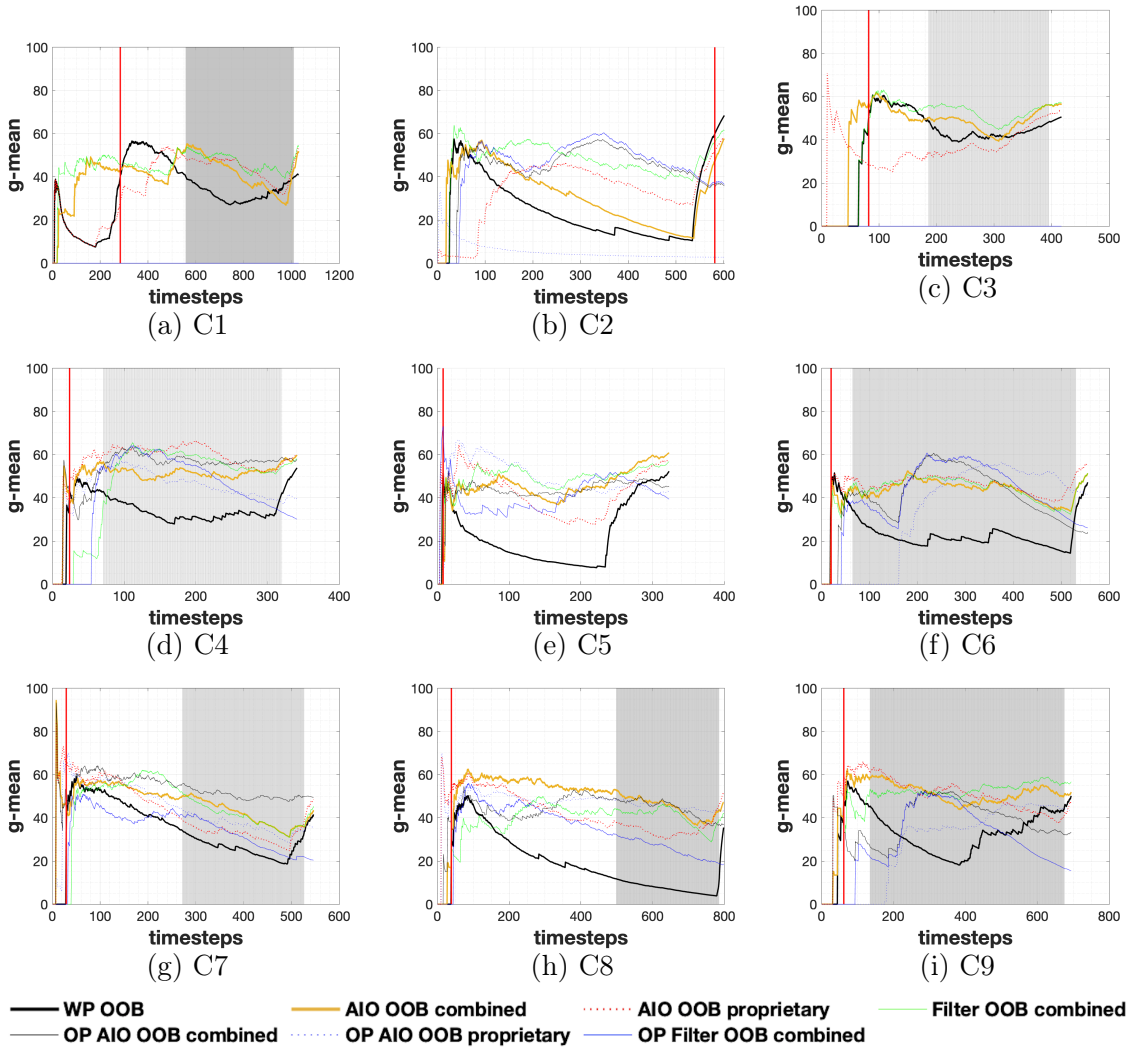


Figure C.1: G-Mean for all datasets through time using OOB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods.

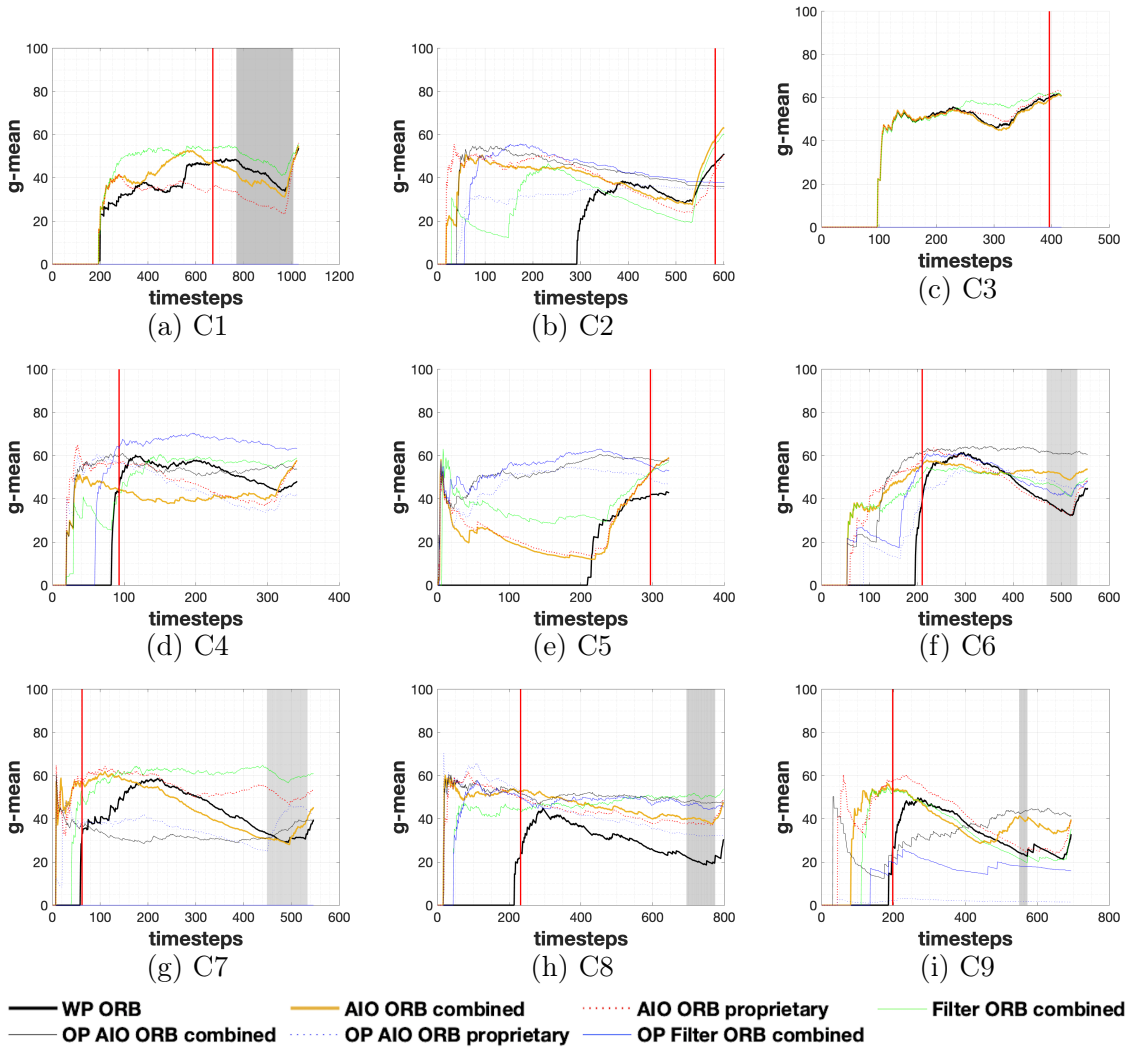
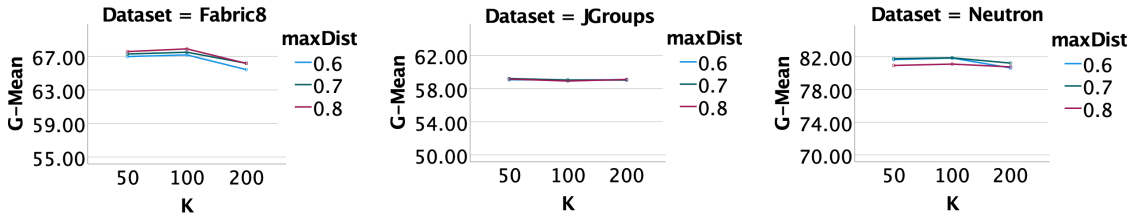


Figure C.2: G-Mean for all datasets through time using ORB. The vertical red bar indicates the last time step of the initial phase of the project. The periods highlighted in grey background are the sudden drop periods.

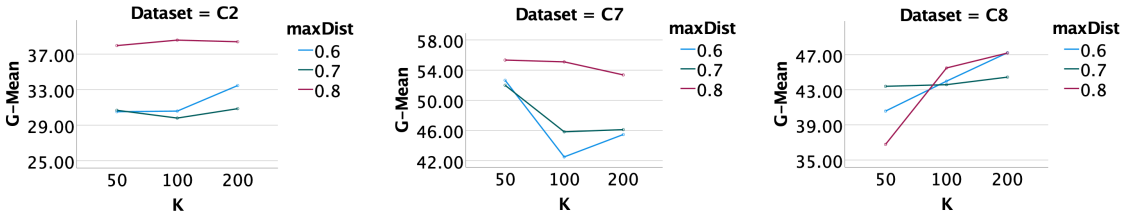
C.4 Hyper-parameter Sensitivity Analysis

This section shows the tables and plots of results corresponding to the analysis of sensitivity to hyper-parameters when using Filtering-ORB in the chapter 5.



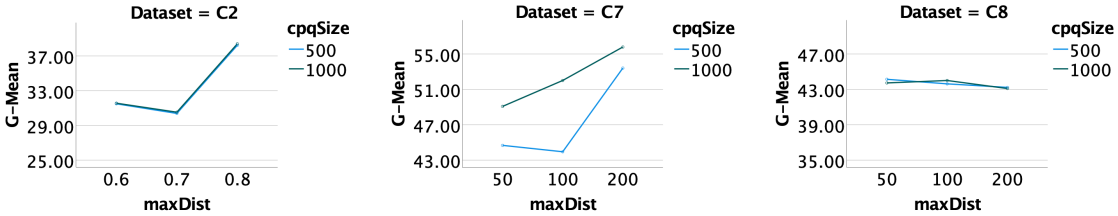
(a) Effect of $K{*}maxDist$ with Fabric8 (b) Effect of $K * maxDist$ with JGroups (c) Effect of $K{*}maxDist$ with Neutron

Figure C.3: Plots of Marginal Means for Filtering-ORB's Factors K, $maxDist$ and $dataset$ (Open source data)



(a) Effect of $K{*}maxDist$ with C2 (b) Effect of $K * maxDist$ with C7 (c) Effect of $K{*}maxDist$ with C8

Figure C.4: Plots of Marginal Means for Filtering-ORB's Factors K, $maxDist$ and $dataset$ (Proprietary data)



(a) Effect of $maxDist * cpqSize$ with C2 (b) Effect of $maxDist * cpqSize$ with C7 (c) Effect of $maxDist * cpqSize$ with C8

Figure C.5: Plots of Marginal Means for Filtering-ORB's Factors $cpqSize$, $maxDist$ and $dataset$ (Proprietary data)

Table C.36: ANOVA and Effect Size Results for Filtering-ORB (Opensource Data)

Within Subject Test for Filtering-ORB (Opensource Data)					
K	554.041	1.847	300.027	1139.471	0.929
K * Dataset	417.928	4	104.482	429.767	0.908
maxDist * Dataset	212.243	4	53.061	245.435	0.849
windowSize*K	175.995	3.447	51.062	179.843	0.674
windowSize	89.059	2	44.529	151.658	0.635
maxDist	50.859	2	25.43	117.626	0.575
windowSize * K * Dataset	100.475	8	12.559	51.336	0.541
windowSize * Dataset	56.363	4	14.091	47.99	0.525
K * maxDist * Dataset	39.91	8	4.989	22.385	0.34
K * maxDist	23.317	4	5.829	26.156	0.231
windowSize * maxDist	23.299	4	5.825	23.286	0.211
windowSize * maxDist * Dataset	17.442	8	2.18	8.716	0.167
windowSize * K * maxDist	31.295	8	3.912	15.318	0.15
cpqSize * Dataset	2.593	2	1.296	5.246	0.108
K * cpqSize * Dataset	3.871	4	0.968	3.102	0.067
windowSize * K * maxDist * Dataset	12.364	16	0.773	3.026	0.065
maxDist * cpqSize	1.977	2	0.989	3.697	0.041
windowSize * K * cpqSize	2.5	3.543	0.706	2.872	0.032
windowSize * maxDist * cpqSize * Dataset	2.82	8	0.352	1.428	0.032
K * maxDist * cpqSize * Dataset	2.598	8	0.325	1.328	0.03
windowSize * K * maxDist * cpqSize * Dataset	3.545	16	0.222	0.886	0.02
windowSize * cpqSize * Dataset	0.806	4	0.202	0.763	0.017
windowSize * K * maxDist * cpqSize	2.651	6.754	0.392	1.325	0.015
K * cpqSize	0.748	2	0.374	1.199	0.014
maxDist * cpqSize * Dataset	0.518	4	0.129	0.484	0.011
windowSize * K * cpqSize * Dataset	0.524	8	0.066	0.301	0.007
windowSize * maxDist * cpqSize	0.574	4	0.143	0.581	0.007
windowSize * cpqSize	0.194	2	0.097	0.367	0.004
cpqSize	0.064	1	0.064	0.257	0.003
K * maxDist * cpqSize	0.285	4	0.071	0.291	0.003
Between Subject Test with ORB					
Dataset	412679.309	2	206339.654	918130.525	1

Table C.37: ANOVA Analysis of Hyperparameters for Filtering-ORB (Proprietary Data)

Within Subject Test with ORB					
Factor/Int.	SS	DF	MS	F	η_p^2
maxDist	23566.525	1.747	13493.132	1355.966	0.938
maxDist*dataset	16407.829	4	4101.957	472.035	0.913
K*dataset	20278.404	4	5069.601	428.2	0.905
cpqSize*dataset	6741.644	2	3370.822	390.947	0.897
cpqSize	3519.707	1	3519.707	408.214	0.819
K*maxDist	6321.242	4	1580.31	170.739	0.655
K*maxDist*dataset	4267.894	8	533.487	57.639	0.562
maxDist*cpqSize*dataset	1409.002	4	352.25	40.993	0.477
maxDist*cpqSize	946.645	2	473.322	55.083	0.38
K	1285.156	2	642.578	54.275	0.376
K*cpqSize*dataset	203.465	4	50.866	4.622	0.093
K*maxDist*cpqSize	250.999	4	62.75	7.113	0.073
Between Subject Test with ORB					
dataset	229263.263	2	114631.632	9383.673	0.995

Appendix Four

Supplementary materials for Chapter Six

D.1 Hyper-parameter values investigated for hyper-parameter tuning in Chapter Six

Table D.1 lists the hyper-parameter values investigated for hyper-parameter tuning in Chapter 6.

Table D.1: List of hyper-parameter values explored in hyper-parameter tuning using random search

Classifier/Base-Learner	**Investigated Hyper-parameters values and probability distribution**
ORB	orb-waiting-time [90,180] uniform
	orb-ma-window-size [50,200] uniform
	orb-th [0.3,0.5] uniform
	orb-l0 [1.0,20.0] loguniform
	orb-l1 [1.0,20.0] loguniform
	orb-m [1.1,2.718281828459045] uniform
	orb-decay-factor [0.9,0.999] loguniform
	orb-n [3,7] uniform
BORB	borb-waiting-time [90,180] uniform
	borb-ma-window-size [50,200] uniform
	borb-th [0.3,0.5] uniform
	borb-l0 [1.0,20.0] loguniform
	borb-l1 [1.0,20.0] loguniform
	borb-m [1.1,2.718281828459045] uniform
	borb-pull-request-size [50,500] uniform
	borb-sample-size [1000,4000] uniform
Base Learners	oht-n-estimators [10,40] uniform
	oht-grace-period [100,500] uniform
	oht-split-criterion {'gini','info_gain', 'hellinger'} uniform
	oht-split-confidence [1e-07,0.5] loguniform
	oht-tie-threshold [0.05,0.5] uniform
	oht-no-preprune {0,1} uniform
	oht-leaf-prediction {'nba', 'nb', 'mc'} uniform
	ihf-n-estimators [10,30] uniform
	ihf-grace-period [100,500] uniform
	ihf-split-criterion {'gini','info_gain', 'hellinger'} uniform
	ihf-split-confidence [1e-07,0.5] loguniform
	ihf-tie-threshold [0.05,0.5] uniform
	ihf-no-preprune {0,1} uniform
	ihf-leaf-prediction {'nba', 'nb', 'mc'} uniform
	lr-alpha [0.01,1.0] loguniform
	lr-n-epochs [10,80] uniform
	lr-batch-size [128,512] loguniform
	lr-log-transformation {0,1} uniform
	mlp-learning-rate [0.0001,0.01] loguniform
	mlp-n-hidden-layers [1,3] uniform
	mlp-hidden-layers-size [5,15] uniform
	mlp-dropout-input-layer [0.1,0.3] uniform
	mlp-dropout-hidden-layer [0.3,0.5] uniform
	mlp-n-epochs [10,80] uniform
	mlp-batch-size [128,512] loguniform
	mlp-log-transformation {0,1} uniform
	nb-n-updates [10,80] uniform
	irf-n-estimators [20,100] uniform
	irf-criterion {'gini',entropy} uniform
	irf-min-samples-leaf [100,300] uniform
	irf-max-features [3,7] uniform

D.2 Online approach and offline approach comparison

Online learning techniques have the ability to continuously adapt to new data over time, which sets them apart from offline learning techniques [17]. However, during the early stages of training, online models may exhibit lower predictive performance due to a lack of sufficient training data [9]. As online models become trained with more data, their predictive performance improves gradually, and eventually reaches a level that is comparable to that of offline learning models.

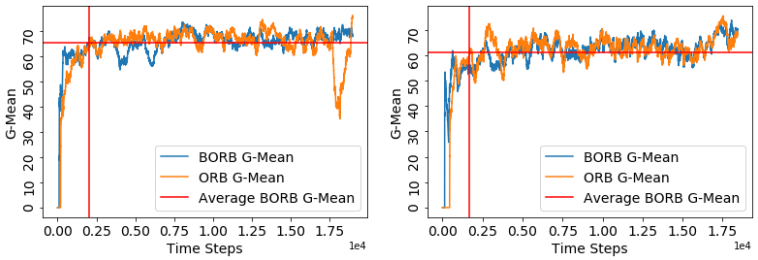
Predictive performance of the online approach (ORB) and the offline approach (BORB) on base learners Hoeffding Tree (OHT) and Iterative Hoeffding Forest (IHF), respectively have been compared using WP data. OHT was chosen as the preferred online base learner based on the results of the experiment conducted in Chapter six, where it was found to be the best-performing base learner. On the other hand, IHF was selected as it is an offline iterative ensemble approach that is based on the Hoeffding Tree, making it more comparable to OHT (Chapter 2 section 2.1). To investigate the amount of data required for online approach to reach to a level of performance which is acceptable close to an offline technique, we consider the number of data points required by ORB-OHT to reach to the average G-Mean of BORB-IHF. Table D.2 reports the number of data points required by online teqnique (ORB-OHT) to reach the average G-Mean of offline teqnique (BORB-IHF) for each datasets.

Table D.2: Data points required by ORB-OHT to reach average g-mean of BORB-IHF for WP data

Dataset	**Data Points required to reach average**
Brackets	1797
Broadleaf	1530
Camel	929
Fabric8	2075
Jgroups	1691
Neutron	859
Nova	6263
Npm	683
Spring_integration	869
Tomcat	2024

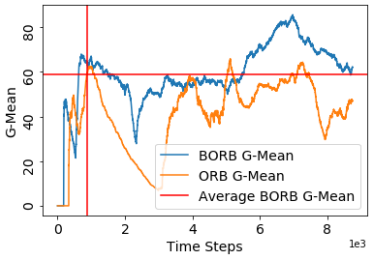
From Table D.2, except for Nova, number of data points required to reach to the average G-Mean by BORB-IHF was less than or equal to 2075. ORB-OHT required minimum 683 (Npm) and maximum 6263 (Nova) data points to reach the average G-Mean achieved by BORB-IHF. For Nova, it is understandable that it required more data points to reach to the average because it has the largest number of total commits (48989), and the average G-Mean of BORB-IHF for Nova was also higher (76.47) compared to other datasets.

Fig. D.1 shows the plots for predictive performance of ORB-OHT and BORB-IHF for open source datasets. In the plots, the horizontal red line shows the average G-Mean achieved by BORB, and the vertical red line shows the time period for ORB-OHT to reach the average G-Mean of BORB-IHF for WP data. From the plots, it can be seen that at the early stage, for some datasets (Fig. D.1a,D.1b, D.1c, D.1e, D.1g, D.1j), predictive performance of ORB-OHT was slightly lower compared to BORB-IHF. However, performance of ORB-OHT gets improved quickly (within 2000 time steps for these datasets).

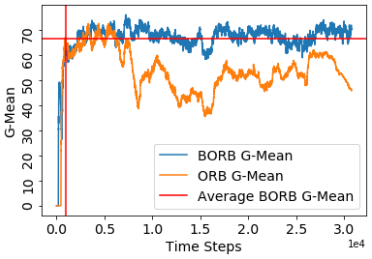


(a) Tomcat

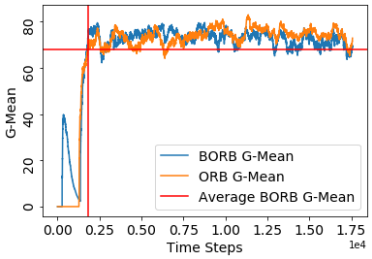
(b) JGroups



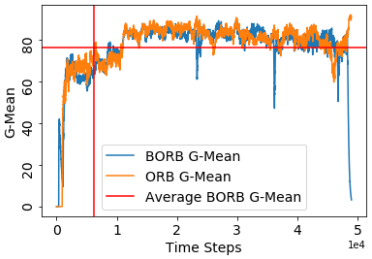
(c) Spring Integration



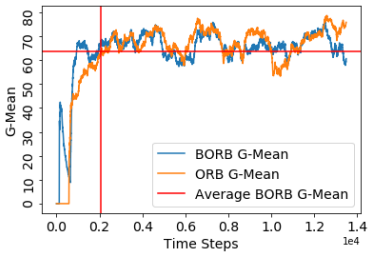
(d) Camel



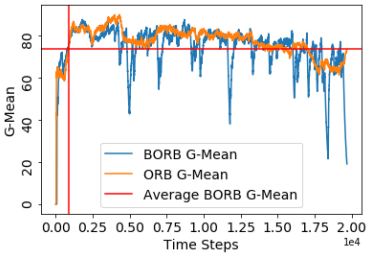
(e) Brackets



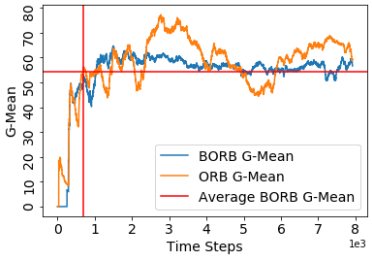
(f) Nova



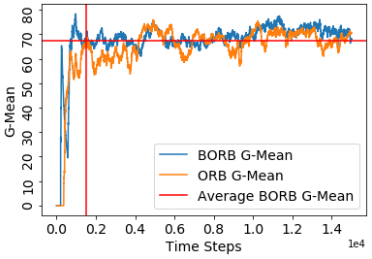
(g) Fabric8



(h) Neutron



(i) Npm



(j) BroadleafCommerce

Figure D.1: G-Mean for Online learning (ORB-OHT) and Offline learning (BORB-IHF) approach

D.3 Recall0, Recall1, FPR and FNR for Chapter Six

Table D.3: Average Recall0 for ORB with WP data using different base learners

dataset	orb_lr	orb_mlp	orb_nb	orb_oht
tomcat	39.18	48.16	44.26	66.51
JGroups	57.55	54.31	58.08	65.63
spring_integration	51.59	67.16	65.83	78.67
camel	51.67	57.09	47.26	87.89
brackets	54.09	61.37	57.43	71.67
nova	57.57	42.97	72.02	76.02
fabric8	69.41	59.5	53.79	77.68
neutron	50.65	55.57	28.09	81.97
npm	54.23	50.42	41.42	65.51
broadleaf	43.37	55.05	59.83	72.14

Table D.4: Average Recall1 for ORB with WP data using different base learners

dataset	orb_lr	orb_mlp	orb_nb	orb_oht
tomcat	60.65	52.36	67.86	64.3
JGroups	43.47	43.63	57.83	59.38
spring_integration	46.83	30.55	49.13	31.87
camel	48.4	43.61	67.63	35.52
brackets	45.73	39.79	78.57	72.93
nova	42.63	64.35	75.37	82.7
fabric8	30.3	40.34	68.82	55.85
neutron	50.7	46.31	84.79	73.39
npm	49.7	51.27	65.38	56.28
broadleaf	57.33	43.63	60.21	60.52

Table D.5: Average Recall0 for BORB with WP data using different base learners

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	72.4	71.4	68.52	71.76	57.71
JGroups	64.15	73.6	58.78	67.55	68.95
spring_integration	66.86	66.8	63.75	71.56	50.74
camel	74.54	67.49	62.02	69.86	65.61
brackets	58.04	70.08	62.06	70.03	67.52
nova	76.33	82.21	57.04	73.27	53.47
fabric8	66.05	70.71	59.61	69.08	70.23
neutron	62.28	53.89	62.26	70.75	50.09
npm	59.11	62.26	59.79	67.23	61.53
broadleaf	73.26	70.04	69.73	68.46	64.41

Table D.6: Average Recall1 for BORB with WP data using different base learners

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	59.95	69.19	71.6	68.32	59.04
JGroups	59.56	55.72	72.93	66	49.62
spring_integration	56.08	71.44	77.06	66.82	60.76
camel	60.55	74.19	80.52	72.32	52.57
brackets	85.65	80.57	89.12	82.17	64.54
nova	79.68	75.75	92	85.18	84.76
fabric8	64.21	67.62	77.93	73.95	55.29
neutron	90.23	95.23	93.47	86.08	77.96
npm	54.46	67.52	76.01	66.27	50.44
broadleaf	64.71	73.83	75.77	76.14	60.45

Table D.7: Average Recall0 for ORB with CP data using different base learners

dataset	orb_lr	orb_mlp	orb_nb	orb_oht
tomcat	49.11	52.84	58.98	68.41
JGroups	54.23	54.21	58.44	65
spring_integration	52.95	57.65	66.63	67.68
camel	43.94	52.55	67.12	63.6
brackets	50.34	52.59	55.58	77.35
nova	48.71	54.1	60.59	77.32
fabric8	54.99	58.44	61.91	70.86
neutron	56.02	60.41	67.34	83.55
npm	52.57	53.41	53.04	62.05
broadleaf	62.63	59.8	69.74	70.44

Table D.8: Average Recall1 for ORB with CP data using different base learners

dataset	orb_lr	orb_mlp	orb_nb	orb_oht
tomcat	51.35	44.98	50.09	65.86
JGroups	45.62	43.92	54.35	60.1
spring_integration	47.99	42.07	51.64	74.3
camel	56.65	44.07	50.12	75.44
brackets	49.72	47.93	80.9	74.89
nova	51.75	45.94	89.75	83.09
fabric8	47.18	41.83	57.58	68.23
neutron	44.81	45.36	94.84	80.67
npm	46.44	44.13	59.85	70.06
broadleaf	39	43.35	48.59	73.03

Table D.9: Average Recall0 for BORB with CP data using different base learners

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	62.38	65.16	65.26	73.71	56.95
JGroups	64.11	73.67	58.68	67.57	68.72
spring_integration	67.26	67.02	66.86	75.61	59.35
camel	57.42	63.09	62.95	71.58	57.89
brackets	63.81	67.53	69.84	76.29	63.04
nova	72.05	58.66	58.63	76.63	48.54
fabric8	69.11	61.98	68.2	70.36	62.74
neutron	58.47	62.38	62.26	75.01	67.63
npm	58.68	60.96	60.96	67.96	62.04
broadleaf	74.68	61.61	71.22	70.18	63.97

Table D.10: Average Recall1 for BORB with CP data using different base learners

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	69.55	73.43	73.44	64.42	54.03
JGroups	59.05	56.01	72.35	65.54	46.39
spring_integration	74.23	78.57	78.79	69.79	58.85
camel	77.55	78.99	78.07	69.9	58.17
brackets	83.28	87.92	86.17	78.97	77.41
nova	81.97	92.05	91.82	83.5	87.77
fabric8	66.97	74.58	66.06	63.98	59.26
neutron	93.36	93.86	93.73	88.69	86.52
npm	70.49	75.71	76.47	69.6	57.41
broadleaf	63.95	81.08	73.94	72.48	65.45

Table D.11: FPR for BORB-WP

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	27.6	28.6	31.48	28.24	42.29
JGroups	35.85	26.4	41.22	32.45	31.05
spring_integration	33.14	33.2	36.25	28.44	49.26
camel	25.46	32.51	37.98	30.14	34.39
brackets	41.96	29.92	37.94	29.97	32.48
nova	23.67	17.79	42.96	26.73	46.53
fabric8	33.95	29.29	40.39	30.92	29.77
neutron	37.72	46.11	37.74	29.25	49.91
npm	40.89	37.74	40.21	32.77	38.47
broadleaf	26.74	29.96	30.27	31.54	35.59

Table D.12: FPR for BORB-CP

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	37.62	34.84	34.74	26.29	43.05
JGroups	35.89	26.33	41.32	32.43	31.28
spring_integration	32.74	32.98	33.14	24.39	40.65
camel	42.58	36.91	37.05	28.42	42.11
brackets	36.19	32.47	30.16	23.71	36.96
nova	27.95	41.34	41.37	23.37	51.46
fabric8	30.89	38.02	31.8	29.64	37.26
neutron	41.53	37.62	37.74	24.99	32.37
npm	41.32	39.04	39.04	32.04	37.96
broadleaf	25.32	38.39	28.78	29.82	36.03

Table D.13: FPR for ORB-WP

dataset	orb_lr	orb_mlp	orb_nb	orb_oht
tomcat	60.82	51.84	55.74	33.49
JGroups	42.45	45.69	41.92	34.37
spring_integration	48.41	32.84	34.17	21.33
camel	48.33	42.91	52.74	12.11
brackets	45.91	38.63	42.57	28.33
nova	42.43	57.03	27.98	23.98
fabric8	30.59	40.5	46.21	22.32
neutron	49.35	44.43	71.91	18.03
npm	45.77	49.58	58.58	34.49
broadleaf	56.63	44.95	40.17	27.86

Table D.14: FPR for ORB-CP

dataset	orb_lr	orb_mlp	orb_nb	orb_oht
tomcat	50.89	47.16	41.02	31.59
JGroups	45.77	45.79	41.56	35
spring_integration	47.05	42.35	33.37	32.32
camel	56.06	47.45	32.88	36.4
brackets	49.66	47.41	44.42	22.65
nova	51.29	45.9	39.41	22.68
fabric8	45.01	41.56	38.09	29.14
neutron	43.98	39.59	32.66	16.45
npm	47.43	46.59	46.96	37.95
broadleaf	37.37	40.2	30.26	29.56

Table D.15: FNR for BORB-WP

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	40.05	30.81	28.4	31.68	40.96
JGroups	40.44	44.28	27.07	34	50.38
spring_integration	43.92	28.56	22.94	33.18	39.24
camel	39.45	25.81	19.48	27.68	47.43
brackets	14.35	19.43	10.88	17.83	35.46
nova	20.32	24.25	8	14.82	15.24
fabric8	35.79	32.38	22.07	26.05	44.71
neutron	9.77	4.77	6.53	13.92	22.04
npm	45.54	32.48	23.99	33.73	49.56
broadleaf	35.29	26.17	24.23	23.86	39.55

Table D.16: FNR for BORB-CP

dataset	borb_ihf	borb_irf	borb_lr	borb_mlp	borb_nb
tomcat	30.45	26.57	26.56	35.58	45.97
JGroups	40.95	43.99	27.65	34.46	53.61
spring_integration	25.77	21.43	21.21	30.21	41.15
camel	22.45	21.01	21.93	30.1	41.83
brackets	16.72	12.08	13.83	21.03	22.59
nova	18.03	7.95	8.18	16.5	12.23
fabric8	33.03	25.42	33.94	36.02	40.74
neutron	6.64	6.14	6.27	11.31	13.48
npm	29.51	24.29	23.53	30.4	42.59
broadleaf	36.05	18.92	26.06	27.52	34.55

Table D.17: FNR for ORB-WP

dataset	**orb_lr**	**orb_mlp**	**orb_nb**	**orb_oht**
tomcat	39.35	47.64	32.14	35.7
JGroups	56.53	56.37	42.17	40.62
spring_integration	53.17	69.45	50.87	68.13
camel	51.6	56.39	32.37	64.48
brackets	54.27	60.21	21.43	27.07
nova	57.37	35.65	24.63	17.3
fabric8	69.7	59.66	31.18	44.15
neutron	49.3	53.69	15.21	26.61
npm	50.3	48.73	34.62	43.72
broadleaf	42.67	56.37	39.79	39.48

Table D.18: FNR for ORB-CP

dataset	**orb_lr**	**orb_mlp**	**orb_nb**	**orb_oht**
tomcat	48.65	55.02	49.91	34.14
JGroups	54.38	56.08	45.65	39.9
spring_integration	52.01	57.93	48.36	25.7
camel	43.35	55.93	49.88	24.56
brackets	50.28	52.07	19.1	25.11
nova	48.25	54.06	10.25	16.91
fabric8	52.82	58.17	42.42	31.77
neutron	55.19	54.64	5.16	19.33
npm	53.56	55.87	40.15	29.94
broadleaf	61	56.65	51.41	26.97

Appendix Five

Supplementary materials for Chapter Seven

E.1 Recall0, Recall1, FPR and FNR for Chapter Seven

Table E.1: Recall0, Recall1, FPR and FNR for Hyperon and Grid search

Dataset	**Approach**	**R0**	**R1**	**FPR**	**FNR**
Tomcat	AIO-Hyperon	81.26	64.81	18.74	35.19
	AIO-Grid	70.23	55.25	29.77	44.75
Jgroups	AIO-Hyperon	82.4	54.56	17.6	45.44
	AIO-Grid	69.09	51.17	30.91	48.83
Spring-integration	AIO-Hyperon	81.08	71.82	18.92	28.18
	AIO-Grid	71.91	63.51	28.09	36.49
Camel	AIO-Hyperon	78.96	70.55	21.04	29.45
	AIO-Grid	67.56	65.35	32.44	34.65
Brackets	AIO-Hyperon	81.02	82.31	18.98	17.69
	AIO-Grid	72.5	75.6	27.5	24.4
Nova	AIO-Hyperon	80.44	90.34	19.56	9.66
	AIO-Grid	74.56	84.53	25.44	15.47
Fabric8	AIO-Hyperon	79.8	75.26	20.2	24.74
	AIO-Grid	71.09	67.61	28.91	32.39
Neutron	AIO-Hyperon	86.55	91.83	13.45	8.17
	AIO-Grid	80.76	86.3	19.24	13.7
Npm	AIO-Hyperon	76.36	68.97	23.64	31.03
	AIO-Grid	62.42	66.51	37.58	33.49
Broadleafcommerce	AIO-Hyperon	81.55	72.84	18.45	27.16
	AIO-Grid	72.33	69.05	27.67	30.95

References

[1] Mrinal Singh Rawat and Sanjay Kumar Dubey. "Software defect prediction models for quality improvement: a literature study". In: *International Journal of Computer Science Issues (IJCSI)* 9.5 (2012), p. 288.

[2] Ning Nan and Donald E Harter. "Impact of budget and schedule pressure on software development cycle time and effort". In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 624–637.

[3] Mahesh Kumar Thota, Francis H Shajin, P Rajesh, et al. "Survey on software defect prediction techniques". In: *International Journal of Applied Science and Engineering* 17.4 (2020), pp. 331–344.

[4] Jaechang Nam. "Survey on software defect prediction". In: *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep* (2014).

[5] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. "Progress on approaches to software defect prediction". In: *Iet Software* 12.3 (2018), pp. 161–175.

[6] Yasutaka Kamei et al. "A large-scale empirical study of just-in-time quality assurance". In: *IEEE TSE* (2013). ISSN: 00985589. DOI: 10.1109/TSE.2012.70.

[7] Yasutaka Kamei et al. "A large-scale empirical study of just-in-time quality assurance". In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 757–773.

[8] Shane McIntosh and Yasutaka Kamei. "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction". In: *IEEE Transactions on Software Engineering* 44.5 (2017), pp. 412–428.

[9] Yasutaka Kamei et al. "Studying just-in-time defect prediction using cross-project models". In: *Empirical Software Engineering* 21.5 (2016), pp. 2072–2106.

[10] Ming Tan et al. "Online defect prediction for imbalanced data". In: *ICSE*. Vol. 2. 2015, pp. 99–108.

[11] Ying Ma et al. "Transfer learning for cross-company software defect prediction". In: *Information and Software Technology* 54.3 (2012), pp. 248–256.

[12] Yasutaka Kamei et al. "Studying just-in-time defect prediction using cross-project models". In: *EMSE* 21.5 (2016), pp. 2072–2106. ISSN: 15737616. DOI: 10.1007/s10664-015-9400-x.

[13] Xiang Chen et al. "MULTI: Multi-objective effort-aware just-in-time software defect prediction". In: *IST* 93 (2018), pp. 1–13.

[14] Chakkrit Tantithamthavorn et al. "Automated parameter optimization of classification techniques for defect prediction models". In: *Proceedings of the 38th international conference on software engineering*. 2016, pp. 321–332.

[15] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. "Tunability: importance of hyperparameters of machine learning algorithms". In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 1934–1965.

[16] Leandro L Minku. "A novel online supervised hyperparameter tuning procedure applied to cross-company software effort estimation". In: *Empirical Software Engineering* 24.5 (2019), pp. 3153–3204.

[17] Gregory Ditzler et al. "Learning in nonstationary environments: A survey". In: *IEEE CIM* 10.4 (2015), pp. 12–25.

[18] George G Cabral et al. "Class imbalance evolution and verification latency in just-in-time software defect prediction". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 666–676.

[19] George Gomes Cabral and Leandro L Minku. "Towards Reliable Online Just-in-time Software Defect Prediction". In: *IEEE Transactions on Software Engineering* (2022).

[20] Shane McIntosh and Yasutaka Kamei. "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction". In: *IEEE TSE* 44.5 (2018), pp. 412–428.

[21] David G Kleinbaum et al. *Logistic regression*. Springer, 2002.

[22] Matt W Gardner and SR Dorling. "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences". In: *Atmospheric environment* 32.14-15 (1998), pp. 2627–2636.

[23] Leo Breiman et al. *Classification and regression trees*. 1st. CRC Press, 1984, p. 358. ISBN: 9781351460491. DOI: 10.1201/9781315139470.

[24] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[25] Weiwei Li et al. "Effort-Aware Semi-Supervised Just-in-Time Defect Prediction". In: *Information and Software Technology* (2020), p. 106364.

[26] Burak Turhan et al. "On the relative value of cross-company and within-company data for defect prediction". In: *EMSE* 14 (5 2009).

[27] Pedro Domingos and Geoff Hulten. "Mining high-speed data streams". In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2000, pp. 71–80.

[28] Richard Brendon Kirkby. "Improving hoeffding trees". PhD thesis. The University of Waikato, 2007.

[29] Sadia Tabassum et al. "An investigation of cross-project learning in online just-in-time software defect prediction". In: *ICSE*. 2020, pp. 554–565.

[30] Sadia Tabassum, Leandro L. Minku, and Danyi Feng. "Cross-Project Online Just-In-Time Software Defect Prediction". In: *IEEE Transactions on Software Engineering* (2022 (in press)). DOI: 10.1109/TSE.2022.3150153.

[31] Andy M Connor, Jacqui Finlay, and Russel Pears. "Mining developer communication data streams". In: *arXiv preprint arXiv:1407.6104* (2014).

[32] Bakshi Rohit Prasad and Sonali Agarwal. "Critical parameter analysis of Vertical Hoeffding Tree for optimized performance using SAMOA". In: *International Journal of Machine Learning and Cybernetics* 8 (2017), pp. 1389–1402.

[33] Jan N Van Rijn and Frank Hutter. "Hyperparameter importance across datasets". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 2367–2376.

[34] Thitiya Trithipkaiwanpon and Unchalisa Taetragool. "Sensitivity Analysis of Random Forest Hyperparameters". In: *2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. IEEE. 2021, pp. 1163–1167.

[35] Andrew Y Ng. "Feature selection, L 1 vs. L 2 regularization, and rotational invariance". In: *Proceedings of the twenty-first international conference on Machine learning*. 2004, p. 78.

[36] Hui Zou and Trevor Hastie. "Regularization and variable selection via the elastic net". In: *Journal of the royal statistical society: series B (statistical methodology)* 67.2 (2005), pp. 301–320.

[37] Thanh Ngoc Tran, Dang Thi Phuc, et al. "Grid search of multilayer perceptron based on the walk-forward validation methodology". In: *International Journal of Electrical and Computer Engineering* 11.2 (2021), p. 1742.

[38] Sajjad Nematzadeh et al. "Tuning hyperparameters of machine learning algorithms and deep neural networks using metaheuristics: A bioinformatics study on biomedical and biological cases". In: *Computational biology and chemistry* 97 (2022), p. 107619.

[39] Faseeha Matloob et al. "Software defect prediction using ensemble learning: A systematic literature review". In: *IEEE Access* (2021).

[40] Santosh S Rathore and Sandeep Kumar. "An empirical study of ensemble techniques for software fault prediction". In: *Applied Intelligence* 51.6 (2021), pp. 3615–3644.

[41] Shamsul Huda et al. "An ensemble oversampling model for class imbalance problem in software defect prediction". In: *IEEE access* 6 (2018), pp. 24184–24195.

[42] Tin Kam Ho. "Multiple classifier combination: Lessons and next steps". In: *Hybrid methods in pattern recognition*. World Scientific, 2002, pp. 171–198.

[43] Thomas G Dietterich. "Ensemble methods in machine learning". In: *International workshop on multiple classifier systems*. Springer. 2000, pp. 1–15.

[44] Leo Breiman. "Bagging predictors". In: *Machine learning* 24.2 (1996), pp. 123–140.

[45] Yoav Freund and Robert E Schapire. "Schapire R: Experiments with a new boosting algorithm". In: *in: Thirteenth International Conference on ML*. Citeseer. 1996.

[46] Nikunj C Oza and Stuart J Russell. "Online bagging and boosting". In: *International Workshop on Artificial Intelligence and Statistics*. PMLR. 2001, pp. 229–236.

[47] Xinli Yang et al. "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction". In: *Information and Software Technology* 87 (2017), pp. 206–220.

[48] Shuo Wang, Leandro L Minku, and Xin Yao. "A learning framework for online class imbalance learning". In: *2013 IEEE Symposium on Computational Intelligence and Ensemble Learning (CIEL)*. IEEE. 2013, pp. 36–45.

[49] Shuo Wang, Leandro L Minku, and Xin Yao. "Resampling-based ensemble methods for online class imbalance learning". In: *IEEE Transactions on Knowledge and Data Engineering* 27.5 (2014), pp. 1356–1368.

[50] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. "On evaluating stream learning algorithms". In: *Machine learning* 90.3 (2013), pp. 317–346.

[51] George G Cabral et al. "Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction". In: *ICSE*. 2019, pp. 666–676.

[52] Shuo Wang, Leandro L Minku, and Xin Yao. "A systematic study of online class imbalance learning with concept drift". In: *IEEE TNNLS* 29.10 (2018), pp. 4802–4821.

[53] Qinbao Song, Yuchen Guo, and Martin Shepperd. "A comprehensive investigation of the role of imbalanced learning for software defect prediction". In: *IEEE Transactions on Software Engineering* 45.12 (2018), pp. 1253–1269.

[54] Qiuming Zhu. "On the performance of Matthews correlation coefficient (MCC) for imbalanced dataset". In: *Pattern Recognition Letters* 136 (2020), pp. 71–80.

[55] Amalia Luque et al. "The impact of class imbalance in classification performance metrics based on the binary confusion matrix". In: *Pattern Recognition* 91 (2019), pp. 216–231.

[56] Nikolaos Mittas and Lefteris Angelis. "Ranking and clustering software cost estimation models through a multiple comparisons algorithm". In: *IEEE TSE* 39.4 (2012), pp. 537–551.

[57] Tim Menzies et al. “Negative results for software effort estimation”. In: *EMSE* 22.5 (2017), pp. 2658–2683.

[58] J. Demšar. “Statistical Comparisons of Classifiers over Multiple Data Sets”. In: *JMLR* 7 (2006), pp. 1–30.

[59] András Vargha and Harold D Delaney. “A critique and improvement of the CL common language effect size statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[60] Iker Gondra. “Applying machine learning to software fault-proneness prediction”. In: *Journal of Systems and Software* 81.2 (2008), pp. 186–195.

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

[62] Tracy Hall et al. “A systematic literature review on fault prediction performance in software engineering”. In: *IEEE TSE* 38.6 (2012), pp. 1276–1304.

[63] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. “Revisiting the impact of classification techniques on the performance of defect prediction models”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 789–800.

[64] Lin Chen et al. “Tackling class overlap and imbalance problems in software defect prediction”. In: *Software Quality Journal* 26.1 (2018), pp. 97–125.

[65] Tiejian Wang et al. “Multiple kernel ensemble learning for software defect prediction”. In: *ASE* 23.4 (2016), pp. 569–590.

[66] Hamoud Aljamaan and Amal Alazba. “Software defect prediction using tree-based ensembles”. In: *Proceedings of the 16th ACM international conference on predictive models and data analytics in software engineering*. 2020, pp. 1–10.

[67] Shuo Wang and Xin Yao. "Using class imbalance learning for software defect prediction". In: *IEEE Transactions on Reliability* 62.2 (2013), pp. 434–443.

[68] Shuo Wang, Huanhuan Chen, and Xin Yao. "Negative correlation learning for classification ensembles". In: *The 2010 international joint conference on neural networks (IJCNN)*. IEEE. 2010, pp. 1–8.

[69] Issam H Laradji, Mohammad Alshayeb, and Lahouari Ghouti. "Software defect prediction using ensemble learning on selected features". In: *Information and Software Technology* 58 (2015), pp. 388–402.

[70] Shulong Liu et al. "FECAR: A feature selection framework for software defect prediction". In: *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE. 2014, pp. 426–435.

[71] Wangshu Liu et al. "Fecs: A cluster based feature selection method for software fault prediction with noises". In: *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 2. IEEE. 2015, pp. 276–281.

[72] Zhou Xu et al. "MICHAC: Defect prediction via feature selection based on maximal information coefficient with hierarchical agglomerative clustering". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 370–381.

[73] Kwabena Ebo Bennin, Jacky W Keung, and Akito Monden. "On the relative value of data resampling approaches for software defect prediction". In: *Empirical Software Engineering* 24.2 (2019), pp. 602–636.

[74] Zhi-Wu Zhang, Xiao-Yuan Jing, and Tie-Jian Wang. "Label propagation based semi-supervised learning for software defect prediction". In: *Automated Software Engineering* 24.1 (2017), pp. 47–69.

[75] Huihua Lu, Bojan Cukic, and Mark Culp. "Software defect prediction using semi-supervised learning with dimension reduction". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, pp. 314–317.

[76] Golnoush Abaei, Ali Selamat, and Hamido Fujita. "An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction". In: *Knowledge-Based Systems* 74 (2015), pp. 28–39.

[77] Naeem Seliya and Taghi M Khoshgoftaar. "Software quality analysis of unlabeled program modules with semisupervised clustering". In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 37.2 (2007), pp. 201–211.

[78] Tiejian Wang et al. "Non-negative sparse-based SemiBoost for software defect prediction". In: *Software Testing, Verification and Reliability* 26.7 (2016), pp. 498–515.

[79] Jaechang Nam and Sunghun Kim. "Clami: Defect prediction on unlabeled datasets (t)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 452–463.

[80] Partha S Bishnu and Vandana Bhattacherjee. "Software fault prediction using quad tree-based k-means clustering algorithm". In: *IEEE Transactions on knowledge and data engineering* 24.6 (2011), pp. 1146–1150.

[81] Shi Zhong, Taghi M Khoshgoftaar, and Naeem Seliya. "Analyzing software measurement data with clustering techniques". In: *IEEE Intelligent Systems* 19.2 (2004), pp. 20–27.

[82] Burak Turhan et al. "On the relative value of cross-company and within-company data for defect prediction". In: *EMSE* 14.5 (2009), pp. 540–578.

[83] Fayola Peters, Tim Menzies, and Andrian Marcus. "Better cross company defect prediction". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 409–418.

[84] Feng Zhang, Iman Keivanloo, and Ying Zou. "Data transformation in cross-project defect prediction". In: *Empirical Software Engineering* 22.6 (2017), pp. 3186–3218.

[85] Zhimin He et al. "An investigation on the feasibility of cross-project defect prediction". In: *ASE* 19.2 (2012), pp. 167–199.

[86] Fayola Peters et al. "Balancing privacy and utility in cross-company defect prediction". In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1054–1068.

[87] Thomas Zimmermann et al. "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process". In: *FSE*. 2009, pp. 91–100.

[88] Zhimin He et al. "Learning from open-source projects: An empirical study on defect prediction". In: *ESEM*. 2013, pp. 45–54.

[89] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. "Transfer defect learning". In: *ICSE*. 2013, pp. 382–391.

[90] Chao Liu et al. "A two-phase transfer learning model for cross-project defect prediction". In: *Information and Software Technology* 107 (2019), pp. 125–136.

[91] Rahul Krishna and Tim Menzies. "Bellwethers: A baseline method for transfer learning". In: *IEEE Transactions on Software Engineering* 45.11 (2018), pp. 1081–1105.

[92] Zhongbin Sun et al. "CFPS: Collaborative filtering based source projects selection for cross-project defect prediction". In: *Applied Soft Computing* 99 (2021), p. 106940.

[93] Steffen Herbold. "Training data selection for cross-project defect prediction". In: *Proceedings of the 9th international conference on predictive models in software engineering*. 2013, pp. 1–10.

[94] Xiao-Yuan Jing et al. "An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems". In: *IEEE TSE* 43.4 (2016), pp. 321–339.

[95] Duksan Ryu, Okjoo Choi, and Jongmoon Baik. "Value-cognitive boosting with a support vector machine for cross-project defect prediction". In: *EMSE* 21.1 (2016), pp. 43–71.

[96] Duksan Ryu, Jong-In Jang, and Jongmoon Baik. "A transfer cost-sensitive boosting approach for cross-project defect prediction". In: *SQJ* 25.1 (2017), pp. 235–272.

[97] Lin Chen et al. "Negative samples reduction in cross-company software defects prediction". In: *Information and Software Technology* 62 (2015), pp. 67–77.

[98] Kwabena Ebo Bennin et al. "An empirical study on the effectiveness of data resampling approaches for cross-project software defect prediction". In: *IET Software* 16.2 (2022), pp. 185–199.

[99] Jun Zheng. "Cost-sensitive boosting neural networks for software defect prediction". In: *Expert Systems with Applications* 37.6 (2010), pp. 4537–4543.

[100] Gerardo Canfora et al. "Multi-objective cross-project defect prediction". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 252–261.

[101] Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. "Cross-project defect prediction models: L'union fait la force". In: *CSMR-WCRE*. IEEE. 2014, pp. 164–173.

[102] Cong Jin. "Cross-project software defect prediction based on domain adaptation learning and optimization". In: *Expert Systems with Applications* 171 (2021), p. 114637.

[103] Audris Mockus and David M Weiss. "Predicting risk of software changes". In: *Bell Labs Technical Journal* 5.2 (2000), pp. 169–180.

[104] Sunghun Kim, E James Whitehead, and Yi Zhang. "Classifying software changes: Clean or buggy?" In: *IEEE Transactions on software engineering* 34.2 (2008), pp. 181–196.

[105] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *ACM sigsoft software engineering notes* 30.4 (2005), pp. 1–5.

[106] Jon Eyolfson, Lin Tan, and Patrick Lam. "Do time of day and developer experience affect commit bugginess?" In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. 2011, pp. 153–162.

[107] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. "Fine-grained just-in-time defect prediction". In: *Journal of Systems and Software* 150 (2019), pp. 22–36.

[108] Charu C Aggarwal et al. *Data mining: the textbook*. Vol. 1. Springer, 2015.

[109] Xinli Yang et al. "Deep learning for just-in-time defect prediction". In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE. 2015, pp. 17–26.

[110] Yibiao Yang et al. "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models". In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 2016, pp. 157–168.

[111] Wei Fu and Tim Menzies. "Revisiting unsupervised learning for defect prediction". In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 72–83.

[112] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. "Studying high impact fix-inducing changes". In: *Empirical Software Engineering* 21.2 (2016), pp. 605–641.

[113] Feng Zhang et al. “Towards building a universal defect prediction model”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 182–191.

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

[115] Christoffer Rosen, Ben Grawi, and Emad Shihab. “Commit guru: analytics and risk prediction of software commits”. In: *FSE*. 2015, pp. 966–969.

[116] Thong Hoang et al. “DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 34–45.

[117] Kun Zhu et al. “Within-Project and Cross-Project Just-In-Time Defect Prediction Based on Denoising AutoEncoder and Convolutional Neural Network”. In: *IET Software* (2020).

[118] Liyan Song and Leandro L Minku. “A Procedure to Continuously Evaluate Predictive Performance of Just-In-Time Software Defect Prediction Models During Software Development”. In: *IEEE Transactions on Software Engineering* (2022).

[119] Li Yang and Abdallah Shami. “On hyperparameter optimization of machine learning algorithms: Theory and practice”. In: *Neurocomputing* 415 (2020), pp. 295–316.

[120] A Günes Koru and Hongfang Liu. “An investigation of the effect of module size on defect prediction using static measures”. In: *Proceedings of the 2005 workshop on Predictor models in software engineering*. 2005, pp. 1–5.

[121] Chakkrit Tantithamthavorn et al. “The impact of automated parameter optimization on defect prediction models”. In: *IEEE Transactions on Software Engineering* 45.7 (2018), pp. 683–711.

[122] Thilo Mende and Rainer Koschke. "Revisiting the evaluation of defect prediction models". In: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*. 2009, pp. 1–10.

[123] Matthias Feurer and Frank Hutter. "Hyperparameter Optimization". In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Cham: Springer International Publishing, 2019, pp. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

[124] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212 (2021), p. 106622.

[125] Quanming Yao et al. "Taking human out of learning applications: A survey on automated machine learning". In: *arXiv preprint arXiv:1810.13306* (2018).

[126] Matthias Feurer and Frank Hutter. "Hyperparameter optimization". In: *Automated machine learning: Methods, systems, challenges* (2019), pp. 3–33.

[127] Hugo Jair Escalante. "Automated Machine Learning–a brief review at the end of the early years". In: *arXiv preprint arXiv:2008.08516* (2020).

[128] Anh Truong et al. "Towards automated machine learning: Evaluation and comparison of AutoML approaches and tools". In: *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)*. IEEE. 2019, pp. 1471–1479.

[129] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. "Explorekit: Automatic feature generation and selection". In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE. 2016, pp. 979–984.

[130] James Max Kanter and Kalyan Veeramachaneni. "Deep feature synthesis: Towards automating data science endeavors". In: *2015 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE. 2015, pp. 1–10.

[131] Binh Tran, Bing Xue, and Mengjie Zhang. "Genetic programming for feature construction and selection in classification on high-dimensional data". In: *Memetic Computing* 8 (2016), pp. 3–15.

[132] Lars Kotthoff et al. "Auto-WEKA: Automatic model selection and hyperparameter optimization in WEKA". In: *Automated machine learning: methods, systems, challenges* (2019), pp. 81–95.

[133] James Bergstra et al. "Hyperopt: a python library for model selection and hyperparameter optimization". In: *Computational Science & Discovery* 8.1 (2015), p. 014008.

[134] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn". In: *ICML workshop on AutoML*. Vol. 9. Citeseer Austin, TX. 2014, p. 50.

[135] Matthias Feurer et al. "Efficient and robust automated machine learning". In: *Advances in neural information processing systems* 28 (2015).

[136] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012).

[137] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems* 25 (2012).

[138] Kevin Swersky, Jasper Snoek, and Ryan P Adams. "Multi-task bayesian optimization". In: *Advances in neural information processing systems* 26 (2013).

[139] James Bergstra et al. "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems* 24 (2011).

[140] James Bergstra, Daniel Yamins, and David Cox. "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures". In: *International conference on machine learning*. PMLR. 2013, pp. 115–123.

[141] Dougal Maclaurin, David Duvenaud, and Ryan Adams. "Gradient-based hyperparameter optimization through reversible learning". In: *International conference on machine learning*. PMLR. 2015, pp. 2113–2122.

[142] MohammadNoor Injadat et al. "Systematic ensemble model selection approach for educational data mining". In: *Knowledge-Based Systems* 200 (2020), p. 105992.

[143] Petro Liashchynskyi and Pavlo Liashchynskyi. "Grid search, random search, genetic algorithm: A big comparison for NAS". In: *arXiv preprint arXiv:1912.06059* (2019).

[144] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[145] Fabian Pedregosa. "Hyperparameter optimization with approximate gradient". In: *International conference on machine learning*. PMLR. 2016, pp. 737–746.

[146] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. "BOA: The Bayesian optimization algorithm". In: *Proceedings of the genetic and evolutionary computation conference GECCO-99*. Vol. 1. Citeseer. 1999, pp. 525–532.

[147] Jia Wu et al. "Hyperparameter optimization for machine learning models based on Bayesian optimization". In: *Journal of Electronic Science and Technology* 17.1 (2019), pp. 26–40.

[148] Matthias Seeger. "Gaussian processes for machine learning". In: *International journal of neural systems* 14.02 (2004), pp. 69–106.

[149] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *International conference on learning and intelligent optimization*. Springer. 2011, pp. 507–523.

[150] Nikolaus Hansen and Andreas Ostermeier. "Completely derandomized self-adaptation in evolution strategies". In: *Evolutionary computation* 9.2 (2001), pp. 159–195.

[151] Masahiro Nomura et al. “Warm starting cma-es for hyperparameter optimization”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 10. 2021, pp. 9188–9196.

[152] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. “Differential evolution: A survey of the state-of-the-art”. In: *IEEE transactions on evolutionary computation* 15.1 (2010), pp. 4–31.

[153] Rawaa Dawoud Al-Dabbagh et al. “Algorithmic design issues in adaptive differential evolution schemes: Review and taxonomy”. In: *Swarm and Evolutionary Computation* 43 (2018), pp. 284–311.

[154] Radwa Elshawi, Mohamed Maher, and Sherif Sakr. “Automated machine learning: State-of-the-art and open challenges”. In: *arXiv preprint arXiv:1906.02287* (2019).

[155] Chris Thornton et al. “Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms”. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2013, pp. 847–855.

[156] Leonardo Villalobos-Arias and Christian Quesada-López. “Comparative study of random search hyper-parameter tuning for software effort estimation”. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2021, pp. 21–29.

[157] Fred Glover. “Tabu search: A tutorial”. In: *Interfaces* 20.4 (1990), pp. 74–94.

[158] Anna Corazza et al. “How effective is tabu search to configure support vector regression for effort estimation?” In: *Proceedings of the 6th international conference on predictive models in software engineering*. 2010, pp. 1–10.

[159] Gilles Lebrun et al. “Tabu search model selection for SVM”. In: *International journal of neural systems* 18.01 (2008), pp. 19–31.

[160] Anna Corazza et al. “Using tabu search to configure support vector regression for effort estimation”. In: *Empirical Software Engineering* 18.3 (2013), pp. 506–546.

[161] Fred Glover and Manuel Laguna. “Tabu search”. In: *Handbook of combinatorial optimization*. Springer, 1998, pp. 2093–2229.

[162] Baozhen Yao et al. “A support vector machine with the tabu search algorithm for freeway incident detection”. In: *International Journal of Applied Mathematics and Computer Science* 24.2 (2014), pp. 397–404.

[163] Ananto Setyo Wicaksono and Ahmad Afif Supianto. “Hyper parameter optimization using genetic algorithm on machine learning methods for online news popularity prediction”. In: *International Journal of Advanced Computer Science and Applications* 9.12 (2018).

[164] Adriano LI Oliveira et al. “GA-based method for feature selection and parameters optimization for machine learning regression applied to software effort estimation”. In: *information and Software Technology* 52.11 (2010), pp. 1155–1166.

[165] Iwan Syarif, Adam Prugel-Bennett, and Gary Wills. “SVM parameter optimization using grid search and genetic algorithm to improve classification performance”. In: *TELKOMNIKA (Telecommunication Computing Electronics and Control)* 14.4 (2016), pp. 1502–1509.

[166] Tianpei Xia et al. “Why software effort estimation needs SBSE”. In: *arXiv preprint arXiv:1804.00626* (2018).

[167] Wei Fu and Tim Menzies. “Easy over hard: A case study on deep learning”. In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 49–60.

[168] Sofia Fernandes, Hadi Fanaee-T, and Joao Gama. "The initialization and parameter setting problem in tensor decomposition-based link prediction". In: *2017 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE. 2017, pp. 99–108.

[169] Robertas Damaševičius. "Optimization of SVM parameters for recognition of regulatory DNA sequences". In: *Top* 18.2 (2010), pp. 339–353.

[170] Gilles Cohen, Patrick Ruch, and Mélanie Hilario. "Model Selection for Support Vector Classifiers via Direct Simplex Search." In: *FLAIRS conference*. 2005, pp. 431–435.

[171] John A Nelder and Roger Mead. "A simplex method for function minimization". In: *The computer journal* 7.4 (1965), pp. 308–313.

[172] Kevin Jamieson and Ameet Talwalkar. "Non-stochastic best arm identification and hyperparameter optimization". In: *Artificial intelligence and statistics*. PMLR. 2016, pp. 240–248.

[173] Lisha Li et al. "Hyperband: A novel bandit-based approach to hyperparameter optimization". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6765–6816.

[174] Wei Fu, Tim Menzies, and Xipeng Shen. "Tuning for software analytics: Is it really necessary?" In: *Information and Software Technology* 76 (2016), pp. 135–146.

[175] Meetesh Nevendra and Pradeep Singh. "Empirical investigation of hyperparameter optimization for software defect count prediction". In: *Expert Systems with Applications* 191 (2022), p. 116217.

[176] Amal Alazba and Hamoud Aljamaan. "Software Defect Prediction Using Stacking Generalization of Optimized Tree-Based Ensembles". In: *Applied Sciences* 12.9 (2022), p. 4577.

[177] Chun Shan et al. “Software defect prediction model based on LLE and SVM”. In: (2014).

[178] Rizal Broer Bahaweres et al. “Software Defect Prediction Using Neural Network Based SMOTE”. In: *2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI)*. IEEE. 2020, pp. 71–76.

[179] Amritanshu Agrawal and Tim Menzies. “Is" Better Data" Better Than" Better Data Miners"?” In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 1050–1061.

[180] Max Kuhn. “Caret: classification and regression training”. In: *Astrophysics Source Code Library* (2015), ascl–1505.

[181] Cong Pan et al. “An improved CNN model for within-project software defect prediction”. In: *Applied Sciences* 9.10 (2019), p. 2138.

[182] Ke Li et al. “Understanding the automated parameter optimization on transfer learning for cross-project defect prediction: an empirical study”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 566–577.

[183] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. “Hyperparameter optimization to improve bug prediction accuracy”. In: *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE. 2017, pp. 33–38.

[184] Faiza Khan et al. “Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction”. In: *IEEE Access* 8 (2020), pp. 20954–20964.

[185] Xiang Chen et al. “Large-scale empirical studies on effort-aware security vulnerability prediction methods”. In: *IEEE Transactions on Reliability* 69.1 (2019), pp. 70–87.

[186] Xingguang Yang et al. "A differential evolution-based approach for effort-aware just-in-time software defect prediction". In: *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*. 2020, pp. 13–16.

[187] Pasquale Ardimento et al. "Temporal Convolutional Networks for Just-in-Time Software Defect Prediction." In: *ICSOFT*. 2020, pp. 384–393.

[188] Tinu Theckel Joy et al. "Hyperparameter tuning for big data using Bayesian optimisation". In: *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE. 2016, pp. 2574–2579.

[189] Chen Lin et al. "Online hyper-parameter learning for auto-augmentation strategy". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 6579–6588.

[190] Luca Franceschi et al. "Forward and reverse gradient-based hyperparameter optimization". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1165–1173.

[191] Isah A Lawal and Salihu A Abdulkarim. "Adaptive SVM for data stream classification". In: *South African Computer Journal* 29.1 (2017), pp. 27–42.

[192] Philip Pfaffe et al. "Online-autotuning in the presence of algorithmic choice". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, pp. 1379–1388.

[193] Hongyuan Zhan et al. "Efficient online hyperparameter optimization for kernel ridge regression with applications to traffic time series prediction". In: *arXiv preprint arXiv:1811.00620* (2018).

[194] Liyan Song, Leandro L Minku, and Xin Yao. "The impact of parameter tuning on software effort estimation using learning machines". In: *Proceedings of the 9th international conference on predictive models in software engineering*. 2013, pp. 1–10.

[195] Bruno Veloso et al. "Hyperparameter self-tuning for data streams". In: *Information Fusion* 76 (2021), pp. 75–86.

[196] Arie Ben-David. "About the relationship between ROC curves and Cohen's kappa". In: *Engineering Applications of Artificial Intelligence* 21.6 (2008), pp. 874–882.

[197] Tracy Hall et al. "Some code smells have a significant but small effect on faults". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23.4 (2014), pp. 1–39.

[198] Yuanrui Fan et al. "The impact of mislabeled changes by szz on just-in-time defect prediction". In: *IEEE Transactions on Software Engineering* 47.8 (2019), pp. 1559–1586.

[199] Steffen Herbold et al. "Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection". In: *Empirical Software Engineering* 27.2 (2022), p. 42.

[200] Jinping Liu et al. "Code churn: A neglected metric in effort-aware just-in-time defect prediction". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2017, pp. 11–19.

[201] Xingguang Yang et al. "Local versus global models for just-in-time software defect prediction". In: *Scientific Programming* 2019 (2019).

[202] Jonggu Kang et al. "HASPO: Harmony Search-Based Parameter Optimization for Just-in-Time Software Defect Prediction in Maritime Software". In: *Applied Sciences* 11.5 (2021), p. 2002.

[203] Xingguang Yang et al. "DEJIT: a differential evolution algorithm for effort-aware just-in-time software defect prediction". In: *International Journal of Software Engineering and Knowledge Engineering* 31.03 (2021), pp. 289–310.

[204] Xingguang Yang et al. "An Empirical Study on Progressive Sampling for Just-in-Time Software Defect Prediction." In: *QuASoQ@ APSEC*. 2019, pp. 12–18.

[205] Nachiappan Nagappan and Thomas Ball. "Use of relative code churn measures to predict system defect density". In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 284–292.

[206] Todd L Graves et al. "Predicting fault incidence using software change history". In: *IEEE Transactions on software engineering* 26.7 (2000), pp. 653–661.

[207] Philip J Guo et al. "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 495–504.

[208] S.W. Flint, J. Chauhan, and R. Dyer. "Escaping the Time Pit: Pitfalls and Guidelines for Using Time-Based Git Data". In: *MSR*. 2021, pp. 85–96.

[209] Zaheed Mahmood et al. "What is the impact of imbalance on software defect prediction performance?" In: *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2015, pp. 1–4.

[210] Sadia Tabassum, Leandro L Minku, and Danyi Feng. "Cross-Project Online Just-In-Time Software Defect Prediction". In: *IEEE Transactions on Software Engineering* (2022).

[211] Pedro Domingos and Geoff Hulten. "Mining High-speed Data Streams". In: *ACM SIGKDD KDD*. 2000, pp. 71–80.

[212] M. Duarte. *detecta: A Python module to detect events in data*. https://github.com/demotu/detecta. 2020.

[213] V.N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 2000.

[214] R. Moser, W. Pedrycz, and Succi G. "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction". In: *ICSE*. 2008.

[215] Y. Fan et al. "The Impact of Mislabelled Changes by SZZ on Just-in-Time Defect Prediction". In: *IEEE TSE* 47 (8 2021).

[216] Dinaldo Andrade PESSOA. "Batch algorithms and fixed prediction rates for online Just-In-Time Software Defect Prediction". MA thesis. Universidade Federal de Pernambuco, 2021.

[217] Rafael G Mantovani et al. "Effectiveness of random search in SVM hyper-parameter tuning". In: *2015 International Joint Conference on Neural Networks (IJCNN)*. Ieee. 2015, pp. 1–8.

[218] George G Cabral and Leandro L Minku. "Towards Reliable Online Just-in-time Software Defect Prediction". In: *IEEE Transactions on Software Engineering* (2022), pp. 1–1.

[219] Yubin Qu et al. "Impact of hyper parameter optimization for cross-project software defect prediction". In: *International Journal of Performability Engineering* 14.6 (2018), p. 1291.

[220] Mikhail Zhabitsky and Evgeniya Zhabitskaya. "Asynchronous differential evolution with adaptive correlation matrix". In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 2013, pp. 455–462.

[221] Evgeniya Zhabitskaya and Mikhail Zhabitsky. "Asynchronous differential evolution". In: *International Conference on Mathematical Modeling and Computational Physics*. Springer. 2011, pp. 328–333.

[222] Rainer Storn and Kenneth Price. "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces". In: *Journal of global optimization* 11.4 (1997), pp. 341–359.

[223] Frank Vavak and Terence C Fogarty. "Comparison of steady state and generational genetic algorithms for use in nonstationary environments". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. IEEE. 1996, pp. 192–195.

[224] Mikhail Zhabitsky. "Comparison of the asynchronous differential evolution and jade minimization algorithms". In: *EPJ Web of Conferences*. Vol. 108. EDP Sciences. 2016, p. 02048.

[225] Jingqiao Zhang and Arthur C Sanderson. "JADE: adaptive differential evolution with optional external archive". In: *IEEE Transactions on evolutionary computation* 13.5 (2009), pp. 945–958.

[226] Frank Wilcoxon. "Individual comparisons by ranking methods". In: *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[227] Sunghun Kim et al. "Dealing with noise in defect prediction". In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 481–490.