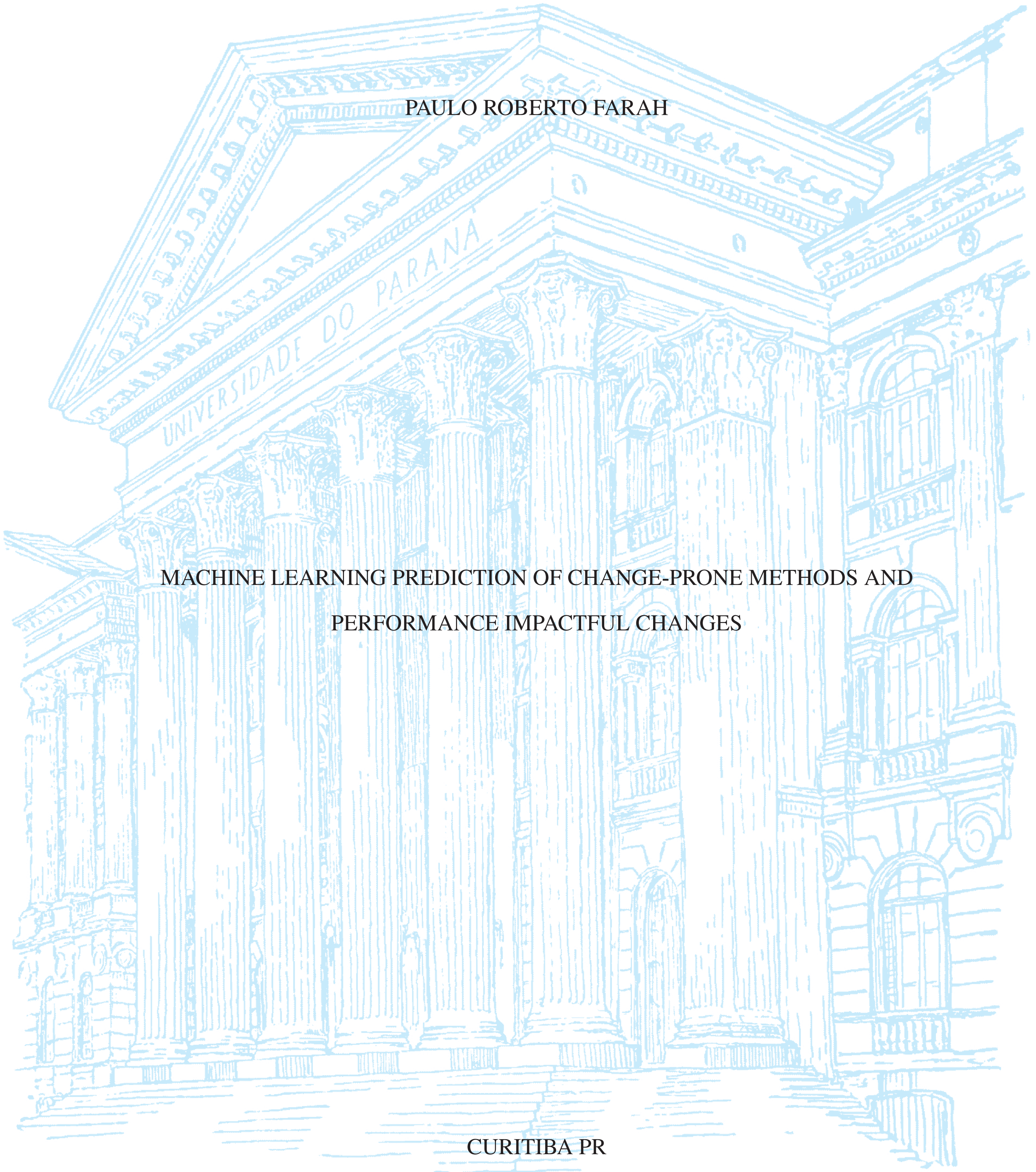
UNIVERSIDADE FEDERAL DO PARANÁ



PAULO ROBERTO FARAH

MACHINE LEARNING PREDICTION OF CHANGE-PRONE METHODS AND PERFORMANCE IMPACTFUL CHANGES

CURITIBA PR

2023

PAULO ROBERTO FARAH

MACHINE LEARNING PREDICTION OF CHANGE-PRONE METHODS AND PERFORMANCE IMPACTFUL CHANGES

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof.ª Dr.ª Silvia Regina Vergilio.

CURITIBA PR

2023

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Farah, Paulo Roberto
Machine learning prediction of change-prone methods and performance impactful changes / Paulo Roberto Farah. – Curitiba, 2023.
1 recurso on-line : PDF.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Silvia Regina Vergilio

1. Aprendizagem de máquina. 2. Software – Desenvolvimento. 3. Software – Desempenho. 4. Teoria da previsão. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Vergilio, Silvia Regina. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA -
40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **PAULO ROBERTO FARAH** intitulada: **MACHINE LEARNING PREDICTION OF CHANGE-PRONE METHODS AND PERFORMANCE IMPACTFUL CHANGES**, sob orientação da Profa. Dra. SILVIA REGINA VERGILIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.
A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 26 de Julho de 2023.

Assinatura Eletrônica
27/07/2023 14:19:49.0
SILVIA REGINA VERGILIO
Presidente da Banca Examinadora

Assinatura Eletrônica
27/07/2023 14:25:32.0
EDUARDO JAQUES SPINOSA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
28/07/2023 10:41:03.0
PLÍNIO DE SÁ LEITÃO JÚNIOR
Avaliador Externo (UNIVERSIDADE FEDERAL DE GOIÁS)

Assinatura Eletrônica
27/07/2023 13:40:50.0
WESLEY KLEWERTON GUEZ ASSUNÇÃO
Avaliador Externo (NORTH-CAROLINA UNIVERSITY)

Rua Cel. Francisco H. dos Santos, 100 - Centro Politécnico da UFPR - CURITIBA - Paraná - Brasil
CEP 81531-980 - Tel: (41) 3361-3101 - E-mail: ppginf@inf.ufpr.br
Documento assinado eletronicamente de acordo com o disposto na legislação federal Decreto 8539 de 08 de outubro de 2015.
Gerado e autenticado pelo SIGA-UFPR, com a seguinte identificação única: 300252
Para autenticar este documento/assinatura, acesse https://siga.ufpr.br/siga/visitante/autenticacaoassinaturas.jsp e insira o codigo 300252

À minha esposa Elizandra e ao meu filho Eduardo.

ACKNOWLEDGEMENTS

This Thesis would not have been possible without the support of several people for which I am very grateful. First, I would like to thank my advisor Dr. Silvia Regina Vergilio for giving me the opportunity to work with her. She was a wonderful advisor and role model of an exceptional professional through the successful and challenging moments of my Ph.D. She guided me with confidence and patience, welcoming me in such a way that I felt part of her research group from the beginning. With her deep knowledge, she provided me with many experiences to learn about software engineering research very deeply and made my passion for this area grow even more. Thus, she is my greatest inspiration as a researcher and I hope that we can work together in partnership in many opportunities to come.

During this period, I could have learned from other incredible friends which I also want to thank. Researchers Wesley Assunção, Thelma Colanzi, Thainá Mariani, Willian Mendonça, and Jackson Lima from UFPR and Professor Alessandro Garcia, Juliana Pereira, and Anderson Oliveira from PUC-Rio. I am also grateful to Professors Luiz Eduardo Oliveira and André Guedes for the excellent classes and to all my friends from the C-Bio GrES group who enriched my knowledge during this period, I have learned a lot from you.

My eternal gratitude to my wife Elizandra and my son Eduardo for always supporting me with love, encouragement, and unconditional assistance, even in the most difficult moments, to fulfill a dream. It certainly made me stronger knowing that they would be by my side encouraging and supporting me with all their energy and love.

I also thank my parents Paulo (*in memoriam*) and Marli and my brothers Diogo and Daniela. I always carry with me the special moments we spent together that bring me joy and pride for having shared life with them for so many years. I also thank my friends who, unfortunately, are far away but I always have them in my heart and I hope to be able to get closer to them again soon.

Lastly, I would like to thank the two educational institutions that made possible the realization of this doctorate. First, to UFPR which offered the Post-graduation Program in Informatics with excellence, and second, to UDESC for offering me all the support to attend the Ph.D. Program in a full-time period, thank you very much!

RESUMO

Mudanças em software podem ocorrer devido a vários motivos, por exemplo, atendimento de demandas de clientes, melhoria de qualidade, correções de falhas, mudanças de tecnologia, entre outros. Assim, alterações no código-fonte são inevitáveis durante a evolução do software e, à medida que evoluem, tornam-se maiores e mais complexos. Como consequência, os desenvolvedores devem gerenciar essas mudanças em sistemas muito grandes e devem escolher qual parte do código receberá mais atenção para a próxima versão. Muitas vezes, eles têm que escolher entre centenas de classes e milhares de métodos e a escolha certa pode impactar positivamente na produtividade da equipe, na alocação de recursos e na qualidade do software. A predição de mudança de software é uma alternativa para identificar, nos estágios iniciais do desenvolvimento, elementos de código propensos a modificações. Os trabalhos mais recentes mostram que experimentos que usam diferentes conjuntos de métricas de software em modelos de aprendizado de máquina provaram ser os melhores preditores classes propensas à mudanças. No entanto, classes com muitos métodos podem ter um escopo muito amplo. Nesses casos, a predição de métodos, ao invés da predição de classes, conduz melhor o desenvolvedor para encontrar o local da mudança com mais rapidez e precisão. Além disso, outra limitação da predição de mudança de software é não informar o propósito da mudança a ser realizada e apenas indicar sua localização no código. Com o amplo escopo das classes, é mais difícil classificar o propósito da mudança porque as classes agrupam uma grande variedade de características do código. O escopo menor dos métodos em relação às classes pode ajudar o modelo a classificar o propósitos da predição de propensão à mudança. Diante disso, esta tese explora o uso de modelos de aprendizado de máquina para predizer métodos propensos a mudanças em geral, e mudanças de métodos com impacto no desempenho. Esses objetivos visam avaliar a eficácia dos modelos em predizer códigos propensos à mudanças em uma unidade de código com escopo menor e especializar a predição para mudanças relacionadas ao desempenho de software de forma eficaz. Para avaliar a viabilidade da nossa solução de aprendizado de máquina, foram realizados dois experimentos. Um foi conduzido usando sete sistemas diferentes, com mais de 1 milhão de métodos, para avaliar a predição de métodos propensos à mudanças. O outro utilizou cinco sistemas para avaliar a predição de mudanças de métodos com impacto no desempenho, com mais de 21 mil métodos. Expandimos ambos os experimentos para usar um método de janela deslizante para remodelar os dados em um formato com dependência temporal e avaliamos a importância das variáveis usadas para a predição. Os resultados mostram que o algoritmo Random Forest com métricas estruturais e baseadas em evolução obteve os melhores resultados, com um valor de ROC AUC médio de 0,82 para a predição de métodos propensos à mudanças e 0,77 para mudanças de métodos com impacto no desempenho. O primeiro supera o estado da arte para o mesmo problema e o último é o primeiro resultado disponível para o problema. Concluímos também que o método da janela deslizante provou ser eficaz em melhorar a predição de métodos propensos à mudanças, mas não para mudanças de métodos com impacto no desempenho. Os resultados apresentados podem ajudar os desenvolvedores a focar melhor nas partes propensas a mudanças corretas para minimizar o número de mudanças futuras e orientar as equipes na distribuição de recursos, reduzindo esforços e custos.

Palavras-chave: Propensão à mudanças. Desempenho de software. Aprendizagem de máquina.

ABSTRACT

Changes in software can occur due to many reasons, for instance, accommodation of customers' demands, quality improvement, fault corrections, changes in technology, and others. Thus, source code changes are inevitable during software evolution and, as they evolve, they become larger and more complex. As a consequence, developers must manage these changes in very large software systems and must choose which part of the code will receive more attention for the next version. Many times, they have to choose from hundreds of classes and thousands of methods and the right choice can positively impact the team's productivity, the allocation of resources, and the quality of the software. Software change prediction is an option to detect, in the early stages of development, code elements that are prone to change. More recent work shows that experiments using different sets of software metrics in machine learning models have proven to be the best predictors of change-prone classes. However, classes can have a very wide scope when they contain too many methods. In these cases, predicting methods are better to drive the developer to find the change location faster and with more accuracy. Beyond that, another limitation of software change prediction is not classifying the purpose of the change-prone element and only indicating the local. With the wide scope of classes, is difficult to classify the purpose of a change because classes group a big variety of code characteristics. The smaller scope of the methods can help the model to classify the purposes of change-proneness prediction. In light of this, this thesis explores the use of machine learning models to predict change-prone methods in general and to predict performance impactful method changes. These goals aim to assess the capacity of the models in predicting change-prone code in a lower scope of code unit and to specialize the prediction to changes related to performance effectively. For assessing the feasibility of our machine learning solution, two experiments were performed. One experiment was conducted using seven different systems, with more than 1 million methods, to asses the prediction of change-prone methods. The other used five systems to evaluate the prediction of performance impactful method changes, with more than 21 thousand methods. We expanded both experiments to use a sliding window approach to reshape the data in a temporal dependent format and evaluated the importance of the features used for the predictions. The results show that the Random Forest algorithm with structural and evolution-based metrics as features obtained the best results with a mean ROC AUC score of 0.82 for the prediction of change-prone methods and 0.77 for performance impactful changes. The former surpasses the state of the art for the first problem and the latter is the first available result for the second problem. We also conclude that the sliding window approach proved to be effective in improving the prediction of change-prone methods, but not for performance impactful method changes. The results herein presented can help developers better focus on the correct change-prone parts to minimize the number of future changes and guide the teams in resource distributions, reducing future efforts and costs.

Keywords: Change-proneness. Software performance. Machine learning.

LIST OF FIGURES

1.1 An illustrative example of performance impactful changes. 20

3.1 Overview of the methodology 34

4.1 Statistical analysis of the performance of the algorithms 46
4.2 Comparing ROC AUC values of Resampling Techniques 47
4.3 Statistical analysis of the algorithms' performance in each system 48
4.4 Statistical analysis regarding the performance of the feature sets 49
4.5 Statistical analysis regarding the performance of the feature set 50
4.6 Comparing both approaches per indicator from the results of the best models. . . 53
4.7 Comparing both approaches per indicator with StrEvo feature set 54
4.8 Statistical analysis comparing both approaches considering results of StrEvo set . 55

5.1 Statistical analysis of the performance of the algorithms 63
5.2 Statistical analysis of the algorithms' performance in each system 64
5.3 Statistical analysis regarding the performance of the feature set 65
5.4 Window size statistical analysis for ROC AUC score on all feature sets 66
5.5 Window size statistical analysis of ROC AUC score on each feature set 66
5.6 Comparing both approaches per indicator 68
5.7 Statistical analysis comparing both approaches considering results of all feature sets 69

C.1 PerfoRT architecture. 106
C.2 PerfoRT automated workflow. 107
C.3 Sample of methods resource usage from Apache Commons BCEL. 110
C.4 CPU usage for Apache Commons BCEL system. 110
C.5 Test cases monitored from PLSETestCase.java. 112
C.6 Cumulative execution time (in seconds) of methods called by the test case testRemoveLocalVariables() from generic.MethodGenTestCase. 113

LIST OF TABLES

2.1 Software metrics used for software change prediction. 24
2.2 Generated by Spread-LaTeX 27
2.3 Prediction algorithms used for software change prediction. 29

3.1 Change-prone method dataset details 36
3.2 Performance impactful method changes dataset details 36
3.3 Example of how metrics are collected by release. 37
3.4 Dataset with the traditional approach 40
3.5 Representation of Table 3.4 reshaped with $S = 2$ 40
3.6 Hyperparameters used in performance impactful method changes models training 41

4.1 Selected Models According to AUC Values 45
4.2 Number of times each algorithm reaches the best value for each indicator considering all systems 45
4.3 Resample techniques with the best performance 46
4.4 Number of times each feature set reaches the best value for each indicator considering all systems 48
4.5 Window statistical Post hoc analysis for ROC AUC indicator of the feature sets presented in Figure 4.5 51
4.6 Results from the best obtained models for each algorithm and feature set 52
4.7 Comparing both approaches 56
4.8 Ranks with the most important features 57

5.1 Performance impactful method changes selected models according to AUC ROC score. 61
5.2 Number of times each algorithm reaches the best value for each indicator considering all systems 62
5.3 Resample techniques with the best performance for PIC. 62
5.4 Number of times each set of features reaches the best value for each indicator considering all systems 64
5.5 Results from the best obtained models for each algorithm and set of features . . . 67
5.6 Number of times each algorithm reaches the best value for each indicator considering all systems 68
5.7 Ranks with the most important features 70
5.8 Comparison Between Change-Prone Method Prediction and Performance Impactful Method Changes. 72

A.1 Previous work that employed supervised learning methods to predict code change-proneness. 89
B.1 Structural metrics.. 102
B.2 Software evolution-based metrics (Source: Elish and Al-Khiaty, 2013 [39]). . . . 104
C.1 Process performance metrics collected by PerfoRT using gopsutil. 108
C.2 Process performance metrics collected by PerfoRT using JFR. 109
C.3 Statistics by commit. 111

LIST OF ACRONYMS

Acc	Accuracy
ACDF	Aggregated Change Density Frequency
ADA	Adaptive Synthetic
ANN	Artificial Neural Network
ATAF	Aggregated Change Size Normalized by Frequency of Change
AUC	Area Under Curve
BOM	Birth Of Method
CBO	Coupling Between Objects
CHD	CHange Density
CHO	Change Occurred
CK	Chidamber & Kemerer object-oriented metrics suite
CPCP	Change-Prone Class Prediction
CPMP	Change-Prone Method Prediction
CSB	Changes Since Birth
CSBS	Changes Since Birth Normalized by Size
CSV	Comma-Separated Values file
DL	Deep Learning
DRS	Data Resident Set
DT	Decision Tree
ENN	Edited Nearest Neighbours
Evo	Evolution-base metrics
F1	F1-Score
FCH	First Change
FGC	Fine-Grained Changes
FRCH	Frequency of Changes
FS	Feature Set
GRU	Gated Recurrent Unit
HWM	High Water Mark
JaCoCo	Java Code Coverage tool
JFR	JDK Flight Recorder
JVM	Java Virtual Machine
KNN	K-Nearest Neighbors
LCH	Last Change
LCA	Last Change Ammount
LCD	Last Change Density

LDA	Linear Discriminant Analysis
LOC	Lines Of Code
LR	Logistic Regression
LSTM	Long Short Term Memory
M	Metric
Max	Maximum value
MDI	Mean Decrease in Impurity
Min	Minimum value
ML	Machine Learning
MLP	Multilayer Perceptron
NB	Naive Bayes
NFR	Non-Functional Requirement
NLP	Natural Language Processing
NN	Neural Network
NOC	Number of Children
OO	Object Oriented
PCA	Principal Component Analysis
PCI	Performance Impactful Change
QMOOD	Quality Model for Object Oriented Design
R	Release
RF	Random Forest
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
ROC AUC	Area Under the Receiver Operating Characteristic Curve
ROS	Random Over-Sampler
RQ	Research Question
RS	Resample technique
RSS	Resident Set Size
RUS	Random Under-Sampling
S	Window size
Sen	Sensitivity
SLOC	Source Lines Of Code
SMOTE	Synthetic Minority Oversampling Technique
Std	Standard deviation
Str	Structural metrics
StrEvo	Structural and Evolution-base metrics
SHAP	SHapley Additive exPlanations
Slw	Sliding Window

SVM	Support Vector Machine
SW	Sliding Window
T	Total
TACH	Total Amount of Changes
TD	Technical Debt
TL	TomekLinks
Trad	Traditional
TS	Time Serie
VMS	Virtual Memory Size
WFR	Weighted Frequency of Changes
WMC	Weight Method Class
X	Independent variables or input features
Y	Dependent variables or target values

CONTENTS

1 INTRODUCTION 16
1.1 RESEARCH PROBLEMS AND MOTIVATIONS 17
1.2 RESEARCH GOALS 19
1.3 THESIS OUTLINE 21
2 RELATED WORK 22
2.1 SOFTWARE CHANGE PREDICTION 22
2.1.1 Change Definition Approaches 23
2.1.2 Independent Prediction Variables 24
2.1.3 Existing approaches for change-prone code prediction 28
2.2 SOFTWARE PERFORMANCE PREDICTION 31
2.3 CONCLUDING REMARKS 31
3 STUDY DESCRIPTION 32
3.1 RESEARCH QUESTIONS 32
3.2 METHODOLOGY OVERVIEW 33
3.3 SELECT VARIABLES 33
3.3.1 Independent Variables 35
3.3.2 Dependent Variables 35
3.4 COLLECT DATA 35
3.4.1 Select Target Systems 36
3.4.2 Collect Static Metrics 37
3.4.3 Collect Dynamic Metrics 37
3.4.4 Prepare Data. 38
3.5 PREPROCESS DATASET 38
3.5.1 Transform 39
3.5.2 Resample 39
3.5.3 Build SW Representation 39
3.5.4 Analyze Features 40
3.6 BUILD ML MODELS AND PREDICT CHANGES 41
3.7 CONCLUDING REMARKS 42
4 CHANGE-PRONE METHOD PREDICTION 44
4.1 RQ1 - WHAT IS THE EFFECTIVENESS OF THE MACHINE LEARNING TECHNIQUES TO PREDICT CHANGE-PRONE METHODS? 44
4.2 RQ2 - WHICH FEATURE SETS YIELD THE MOST ACCURATE PREDICTION OF CHANGE-PRONE METHODS? 47

4.3 RQ3 - HOW DO DIFFERENT WINDOW SIZES IN THE SLIDING WINDOW APPROACH AFFECT THE PREDICTION PERFORMANCE? 48
4.4 RQ4 - HOW DOES THE EFFECTIVENESS OF MODELS OBTAINED WITH THE SLIDING WINDOW APPROACH COMPARE TO THE MODELS OBTAINED WITH A TRADITIONAL REPRESENTATION? 50
4.5 RQ5 - WHICH FEATURES ARE THE MOST IMPORTANT FOR CHANGE-PRONE METHODS PREDICTION? 56
4.6 DISCUSSION. 56
4.6.1 Prediction of change-prone methods 57
4.6.2 Choice of the algorithm 58
4.6.3 Choice of sliding window size 58
4.6.4 The reshaping of data to a temporal dependent format 58
4.6.5 Kinds of models adopting different feature sets 58
4.6.6 Features importance. 59
4.7 THREATS TO VALIDITY 59
4.7.1 Construct validity 59
4.7.2 Internal validity 59
4.7.3 Conclusion Validity 60
4.7.4 External validity. 60
4.8 CONCLUDING REMARKS 60
5 PERFORMANCE IMPACTFUL CHANGES PREDICTION 61
5.1 RQ1 - WHAT IS THE EFFECTIVENESS OF THE MACHINE LEARNING TECHNIQUES TO PREDICT PERFORMANCE IMPACTFUL METHOD CHANGES? 61
5.2 RQ2 - WHICH FEATURE SETS YIELD THE MOST ACCURATE PREDICTION OF PERFORMANCE IMPACTFUL METHOD CHANGES? 63
5.3 RQ3 - HOW DO DIFFERENT WINDOW SIZES IN THE SLIDING WINDOW APPROACH AFFECT THE PREDICTION PERFORMANCE? 65
5.4 RQ4 - HOW DOES THE PREDICTIVE EFFECTIVENESS OF MODELS OBTAINED WITH THE SLIDING WINDOW APPROACH COMPARE TO THE MODELS OBTAINED WITH A TRADITIONAL REPRESENTATION? . 67
5.5 RQ5 - WHICH FEATURES ARE THE MOST IMPORTANT FOR PERFORMANCE IMPACTFUL METHOD CHANGES PREDICTION? 69
5.6 DISCUSSION. 71
5.6.1 Predicting performance impactful method changes. 71
5.6.2 Results of the sliding window approach 71
5.6.3 Features importance. 72
5.6.4 Comparison Between Change-Prone Method Prediction and Performance Impactful Method Changes. 72

5.7 THREATS TO VALIDITY 72
5.7.1 Construct validity 73
5.7.2 Internal validity 73
5.7.3 Conclusion Validity 73
5.7.4 External validity. 74
5.8 CONCLUDING REMARKS 74
6 FINAL REMARKS AND FUTURE WORK 75
6.1 REVISITING OUR CONTRIBUTIONS 75
6.2 PUBLICATIONS 76
6.3 FUTURE WORK 77
REFERENCES 78
APPENDIX A – RELATED WORK 89
APPENDIX B – SOFTWARE METRICS 102
APPENDIX C – PERFORT: A TOOL FOR SOFTWARE PERFORMANCE REGRESSION 105
C.1 ARCHITECTURE 105
C.2 WORKFLOW 106
C.3 METRICS 106
C.4 USAGE EXAMPLE 108
C.5 CONCLUDING REMARKS 111

1 INTRODUCTION

The adoption of interactive and incremental software development by the software industry encompasses the generation of newer software versions and releases of the software over time. As the systems evolve from one version/release to another, they become larger and more complex. This makes software maintenance one of the most expensive and arduous tasks in the software life cycle [76, 120].

Changes in the software can occur due to many reasons, for instance, accommodation of new functionalities demanded by customers and stakeholders, quality improvement, fault corrections, changes in technology, etc. The more changes developers apply to the software system the more complex the system is likely to become [23]. This may lead to an erosion in the original design and, consequently, a reduction in the overall software quality. Thus, managing and controlling changes is critical [105], and change management of software is a fundamental activity in order to plan and implement evolution and maintenance.

As changes are unavoidable and occur often, it is very costly and resource consuming to monitor equally all the parts of the code [39]. Hence, efficient ways to control them are required and the industry has adopted change-proneness prediction models to identify regions of the code that are most likely to change in subsequent versions of the software [97].

We observed in the literature a large number of works that are mainly concerned with functional correction and fault prediction [67, 92], as well as refactoring and smell prediction [8, 72, 124]. They specialize in a specific type of change and do not focus on other types that may impact, for example, non-functional properties such as security or performance. Thus, the focus of software change prediction is predicting change in general, regardless of the type. Systematic reviews of works in this area show the predominance of the adoption of *Machine Learning (ML)* algorithms to predict classes prone to change [105]. They use as predictors (independent variables) different sets of metrics; structural-based models are the most common, but these metrics are frequently used in combination with other metrics such as evolution-based metrics [3, 20, 21, 23, 39, 103, 104], and intensity of smells or anti-patterns [24, 64, 123].

Although much research exists about software change prediction, there are some topics that can be more explored. First, the literature focusing on change-prone methods is scarce, despite the researchers highlighting the importance of considering the usage of metrics at the method level [105]. This allows focusing on a low level of granularity and more specific parts of the code, reducing the effort to identify change-prone parts of the code, mainly in classes with a large number of methods. Moreover, classes contain many characteristics of different software attributes and some software behaviors are more granular and better captured at the method level. For example, how can we examine software performance for a class? Performance is inherent to the method scope, as well as other non-functional characteristics, such as vulnerabilities, which may be more related to fine-grained source changes.

There is another lack of research evaluating models to predict performance impactful changes, that is, predicting code elements that, when changed, can cause a significant impact on system performance. Performance is an important non-functional property of software quality little studied by researchers of software change prediction. It can directly impact multiple aspects like user experience, reliability, security, and business revenue, among others. Software applications that perform poorly can cause frustration for users and result in lost business for companies. Moreover, software systems failures can be very often due to performance issues rather than functional bugs [27].

One of the most common performance issues during the software life cycle is performance regression, such as response time degradation and increased resource utilization. Performance issues are behaviors of software elements where the response time for output is longer than the expected time to execute them [60]. They are common in software systems, and improving them is part of the development process. However, some empirical studies that analyzed performance problems concluded that these problems take longer to get reported and fixed than functional problems [56, 62, 143]. Predicting performance impactful changes is an innovative proposal to help developers deal with performance issues. First, they will be aware that a recommended change-prone method may impact performance and can focus on solutions to deal with that. Also, they can choose to maintain change-prone methods if they want to improve the performance of the software. At last, managers can allocate more performance-savvy developers to change these methods.

Additionally, when changes are made to a software application, they often have dependencies on prior changes or may impact subsequent modifications. In fact, the changes occur over time as a sequence, and the learning instances are not independent. Most of the existing approaches do not take into account the temporal dependency between the changes.

In summary, the limitations previously contextualized that drive this thesis include: the incipient research on software change prediction at the method level, the scarcity of research on predicting performance impactful changes, and the procedural limitation of the existing works of considering the software change history. Thus, we expose in more detail the research problems and the motivations of our work as follows.

1.1 RESEARCH PROBLEMS AND MOTIVATIONS

We introduced the context of software change prediction by presenting its importance for software development, as well as, some limitations of the existing approaches. We can observe that software change prediction is a challenging problem, because it encompasses different aspects and dimensions, as pointed out by many works, such as: i) to assess the capability of different predictors, used (or not) in a combined way, as well as the investigation of new ones, capturing other software evolution aspects; ii) to investigate the use of feature selection/dimensionality reduction techniques; iii) to deal with real change-prone datasets, which are naturally imbalanced; iv) to take into account temporal validation; v) to perform cross-validation between projects and transfer learning; and vi) to build and make available a benchmark for the research community [105]. As a consequence, related work presents some limitations, and there is still room for improvement. Some of the aforesaid gaps and limitations constitute two research problems investigated by this thesis. We present the research problems and the motivations with examples as follows.

> Research problem 1: The class scope of change-prone code prediction is too wide to accurately predict the local in big classes containing many methods and to classify the purpose of the changes.

Motivation: The class is definitely the most studied code unit in software change prediction research [105], while there is incipient research on methods. However, in some situations, the method scope is much better to be used. Change-prone methods prediction allows focusing on a low level of granularity and on more specific regions of the code, reducing effort mainly in classes with a large number of methods.

Grund et al. [53] surveyed 30 developers from industry and 12 from academia and found that 90% are most interested in method-level, which also leads to the class, instead of file-level granularity. LaToza and Myers [84] surveyed 179 professional software developers at Microsoft and asked them to list hard-to-answer questions about code. The responses were like "Where was this variable last changed?" when debugging, "When, how, by whom, and why was this code changed or inserted?" when they want to find the code's creation in history to understand its context and motivation, and finally "How has it changed over time?" when they want to know the entire history of a block of code, rather than its most recent change. These findings motivate the need for research that can track change history at a more fine-grained level, focusing on specific program elements, such as methods.

Consider one example with the class `org.apache.bcel.verifier.structurals.InstConstraintVisitor` of the Apache `commons-bcel` project[1]. It is listed in 13 releases, contains 180 methods and 1481 lines of code in the release `rel/commons-bcel-6.4.1(bebe70d)`. Observing the modifications from this release and the release `rel/commons-bcel-6.5.0(a9c13ed)`, only one modification in this class was performed, in the method `public void visitINVOKEDDYNAMIC(final INVOKEDDYNAMIC)`. If we had identified this class as change-prone by applying the traditional class-level prediction approach, all of the 180 methods would have to be inspected, even if only one (or few) method(s) changed between releases.

Moreover, classes contain many different characteristics of software attributes and some software behaviors are more granular and better captured at the method level. For example, software performance makes more sense in the method scope, as well as other non-functional characteristics, such as robustness and vulnerabilities, which may be more related to fine-grained source changes. Although the class is the main modular unit in object-oriented programming, it can contain a significant number of code parts with different purposes. Some methods may cause more impact on performance, while others are more close to security, and so on. So, the class scope of software change prediction is a very large scope to localize the part of the code that should change in big classes and to choose what to do. On the other hand, the method level reduces the code size to be analyzed and may help developers define maintenance goals more easily. Discussing the purposes of changes lead us to define the second research problem of this thesis, presented as follows.

> Research problem 2: Software performance is handled reactively and developers can not know if a change in a change-prone method has a tendency to impact performance.

Motivation: Software performance is a crucial factor in determining the quality of software applications. Predicting software performance impactful changes during the development process can help developers identify potential performance issues early and take action to improve the software quality. Predicting which changes in methods have an impact on performance during software evolution allows developers to prioritize and allocate resources efficiently.

To illustrate how performance impactful method changes occur in practice, we present another example in Figure 1.1. It shows the source code of the class `org.apache.commons.csv.CSVFormat` of the project Apache `commons-csv`[2]. The modifications in the release `rel/commons-csv-1.9.0`, from the commit `bfdcad2` to `8e25a2b`, on method `printWithEscapes` affected significantly the performance. The

[1]https://github.com/apache/commons-bcel

[2]https://github.com/apache/commons-csv

change in the example modifies the code by adding two loops. Firstly, inside the method `isDelimiter`, marked in yellow in the figure. This method contains a loop that iterates over the array `charSeq` from 1 to `delimLength`. The other element is the for loop which also iterates from 1 to `delimLength` variable over the same array, `charSeq`. As these loops are inside a while command, it is possible to observe that these loops changed the time complexity from linear to quadratic and that is the cause of the impact on performance.

Many times developers are not aware of the impact of changes on performance and the code runs in production that may cause significant performance degradation [119]. Maintenance is done reactively where the systems are developed without any information if the code change could significantly impact performance. In this case, significantly means that after changes, it will present a variation in performance metrics between the two versions of the modified codes with statistical significance. Models to predict performance impactful changes can be used to create tools that notify the developer about this impact.

In this thesis, we call attention to the limitations of existing software change prediction research. First, the use of class scope in software change prediction. Then, the lack of research on predicting performance impactful changes. Next, do not take into account the temporal dependency between instances of changed code elements. Motivated by these facts, we argue that supervised machine learning models, in conjunction with the Sliding Window approach [36] to represent the temporal dependence between instances of methods, can be harnessed to overcome these limitations and empirically evaluate the solution performance. Thus, we present the goals of this thesis as follows.

1.2 RESEARCH GOALS

Given the context, motivations, and open research problems presented in the last section, this work has as a general goal to investigate the use of machine learning approaches to predict change-prone methods and performance impactful method changes. To reach such a goal, we adopted a common methodology for both problems. Through a literature review, we identified existing techniques and methodologies employed in this area. We also investigated the factors influencing the change-proneness of code methods and their relationship with software performance.

We designed and empirically evaluated the performance of four ML algorithms often used by other works [105]: Random Forest, Decision Tree, Logistic Regression, and Multi-Layer Perceptron, using as predictors three sets of metrics: structural, evolution-based metrics, and a combination of both. Furthermore, we evaluated the adoption of the sliding window approach to reshape data in a temporal dependent format to improve the results. With these objectives in mind, we conducted empirical studies on real-world software systems to assess the accuracy and applicability of the prediction models. Our study encompasses seven open-source Java projects. We chose Java because of its support of existing code analysis tools and ease of replication

In summary, the contributions of this thesis related to our two research problems are: i) to develop the concept that the class scope is too wide to predict software changes, as well as the implementation of machine learning algorithms and models for the prediction of change-prone Java methods; ii) to introduce the concept and the implementation of machine learning algorithms and models for the prediction of performance impactful method changes; iii) to provide empirical evidence on the best algorithms and models; iv) to evaluate the effects of the Sliding Window approach to capture temporal dependency representation; v) to perform a study exploiting the feature importance, with the aim of finding the subset of predictors most relevant; The findings herein presented can help developers to effectively monitor changes through software evolution at a low level of granularity and to monitor changes that may impact the performance of the system.

```java
// rel/commons-csv-1.9.0 (bfdcad2)
private void printWithEscapes(final CharSequence value,
final Appendable out) throws IOException {
(...)
while (pos < end) {
      char c = value.charAt(pos);
      if (c == CR || c == LF || c == delim || c == escape) {
         // write out segment up until this char
         if (pos > start) {
            out.append(value, start, pos);
         }
         if (c == LF) {
            c = 'n';
         } else if (c == CR) {
            c = 'r';
         }
         out.append(escape);
         out.append(c);
         start = pos + 1;
```

```java
// rel/commons-csv-1.9.0 (8e25a2b)
 private void printWithEscapes(final CharSequence charSeq,
 final Appendable appendable) throws IOException {
 (...)
   while (pos < end) {
      char c = charSeq.charAt(pos);
      final boolean isDelimiterStart = isDelimiter(c, charSeq,
                pos, delim, delimLength);}
      if (c == CR || c == LF || c == escape || isDelimiterStart) {
         if (pos > start) {
            appendable.append(charSeq, start, pos);
         }
         if (c == LF) {
            c = 'n';
         } else if (c == CR) {
            c = 'r';
         }
         appendable.append(escape);
         appendable.append(c);
         if (isDelimiterStart) {
            for (int i = 1; i < delimLength; i++)
            {  pos++;
               c = charSeq.charAt(pos);
               appendable.append(escape);
               appendable.append(c);
            }
         }
         start = pos + 1;
```

Figure 1.1: An illustrative example of performance impactful changes.

This contributes to reducing poor software quality, additional costs regarding error correction, and non-compliance with non-functional requirements. Moreover, they can help developers to choose an ML algorithm and features. They can be used by researchers to build recommendation systems that adopt change-proneness as a quality indicator to warn developers in order to plan preventive maintenance operations, such as refactoring, peer-code review, and testing.

At last, we provide supplementary material, including the source code of the tested algorithms, all data, instructions, and scripts to reproduce the experiments, and additional details for the results [42].

1.3 THESIS OUTLINE

The remainder of this thesis is structured into seven chapters. Chapter 2 introduces the software change prediction problem by describing its main characteristics and predictors used in the literature, and reviews related work. Chapter 3 presents the methodology adopted to reach the goals herein discussed, illustrates an overview of all steps of this research, the datasets definition, collection, and preprocessing, how we implemented the Sliding Window approach, and details about the machine learning models. Chapter 4 contains the experimental evaluation results obtained for the prediction of change-prone methods. Chapter 5 reports the experimental evaluation results obtained for the prediction of performance impactful method changes. Chapter 6 concludes the work and presents future research directions. Finally, Appendix A contains a list of publications related to change-prone prediction with ML algorithms, published between 2017 and 2022, Appendix B describes the structural and evolution-based metrics, and Appendix C presents the tool PerfoRT that was used to measure software performance.

2 RELATED WORK

This chapter provides a review of the existing research related to our thesis in the fields of software change prediction and software performance prediction. It summarizes existing approaches on how to measure changes, addresses feature predictors, summarizes the main ML models that are used, and work on software performance prediction. The examination of these elements is important to establish a foundation for choosing the proper representation of changes, selecting algorithms and features, and defining methods and tools to guide this research.

The chapter is organized in the following structure. First, Section 2.1 contains research about software change prediction. Next, Section 2.2 presents related studies on software performance prediction. At last, Section 2.3 concludes the chapter.

2.1 SOFTWARE CHANGE PREDICTION

Software undergoes changes continuously during its lifecycle because of several reasons. Bug fixing, the addition of new requirements and features, adaptation to a new environment, enhancements in security and performance, or refactoring are some examples of different causes of those changes. However, changes performed for a specific reason can impact the overall software quality, improving or worsening multiple attributes such as maintainability, performance, testability, robustness, and usability, among others. Managing code changes effectively is essential for maintaining software quality.

One strategy for managing code changes is change-proneness prediction. This strategy intends to identify parts of the source code that are more likely to change in the next release of the software in order to perform a maintenance activity [78, 85]. It indicates a degree of change across various versions of software [51] that is used to predict which code part probably will be modified by the developer. As it is very costly and resource-consuming to monitory equally all the parts of the code, identifying which parts of code are prone to change during software evolution allows developers to prevent maintainability issues [23], such as refactoring, peer-code reviews, to prioritize tasks and to allocate resources efficiently [22, 39].

Changes can occur in different parts of the code, including declarations of classes, interfaces, and methods, or in code body parts such as loops, control structures, statements, and comments. To analyze code changes, it is necessary to delimit an observation unit. These boundaries can be considered at version, package, file [25, 142], class, method, or component unit. In the literature, existing change-proneness prediction approaches are focused mainly on system classes to delimit code prone to change. Malhotra and Khanna [105] reviewed the literature on software change prediction. They evaluated predictors, features, prediction methods, performance measures, and other characteristics of 38 primary works from January 2000 to June 2019. We complemented the review of the existing studies with a forward snowballing procedure to update the systematic literature review of Malhotra and Khanna [105]. We used Google Scholar to find publications that cited three software change prediction studies: Malhotra and Khanna [105], Godara et al. [52] and Catolino et al. [24]. We also searched in the scientific digital libraries IEEE Xplore Digital Library[1], ACM Digital Library[2], Elsevier ScienceDirect[3]

[1]https://ieeexplore.ieee.org/
[2]http://dl.acm.org/
[3]https://www.sciencedirect.com/

and SpringerLink[4]. Appendix A contains a list of publications on software change prediction using ML learning methods, published in the period between 2017 and 2022. The following sections summarize the results of this search. They are organized into how the studies considered a change occurs, which independent variables were used as input predictors, and which prediction algorithms were used.

2.1.1 Change Definition Approaches

There are different approaches to define if a class (or any other part of the code) changed in existing research. Two types that are most frequently used can be highlighted, and described as follows. The first one is based on the number of *Source Lines of Code* (SLOC). It compares the SLOC between current and previous releases and considers that a class changed if the values are different. This approach is defined as "SLOC-based" by Malhotra and Khanna [105]. This is the most frequent approach and can be observed in many publications [9, 64, 68, 75, 90, 106, 109, 144]. Some works use a SLOC-based intensity equation, composed of added, deleted, and changed lines of the elements [5, 63, 90, 107]. The equation is formed by $change = SLOC_{added} + SLOC_{deleted} + 2 * SLOC_{changed}$.

Certain other studies [16, 41, 110] consider structural changes in the code elements, in which they extract changes comparing the abstract syntax tree between two versions. Examples of the changes include the addition or deletion of classes from a package, the addition or deletion of attributes from a class, scope modification (public, protected, and private), or types, among others. This approach usually uses the *ChangeDistiller* tool. A description of the tool and the complete list of operations are presented in [46, 48]. This approach is known as *Fine-Grained Changes* (FGC) in the literature. Some works [24, 128] adopted the number of structural changes higher than the median of the distribution of the number of changes experienced by all the classes of the system. In these cases, the median of changes is calculated with the number of changes of all classes of the release and the classes with the number of changes higher than this median are considered changed.

There are many other change definition approaches found in the literature. Given a change in a class, Tsantalis et al. [135] evaluate the probability that each class of the system will be affected when new functionality is added or when existing functionality is modified. By change, they consider that given a change in one of the affecting classes, the affected classes should be updated, in order for the software to operate correctly. This behavior is known as *ripple effect*. Di Penta et al. [35] adopt a "Design Motif Identification Multi-layered Approach" (DeMIMA) to identify changes in design patterns. Khomn et al. [70] define that changes to and faults in a class are related to the class participating in anti-patterns. The work does not consider releases where classes changed due to a very small number of changes (less than 10). Another approach [65] considers changes in basic control structures as a sequence, branch, iterations, embedded components, and concurrency with a framework called CognitiveC to identify these changes in software components.

As discussed above, different approaches to defining code change have been used in literature. Analyzing the two main approaches we can observe that both have pros and cons. The "SLOC-based" approach identifies that a class changed even if only a few SLOCs have been changed. This ensures that all changes will be detected. For example, in a refactoring that renames a method, only one line is modified. On the other hand, this approach has the disadvantage to have a huge quantity of varied types of changes, including even trivial changes, and does not capture the details about the semantics of changes [50]. The FGC approach identifies

[4]https://link.springer.com/

Table 2.1: Software metrics used for software change prediction.

Metric Category	Description	Studies
Static metrics		
Structural	These metrics are software internal quality metrics, which depict structural attributes of Object-Oriented elements such as size, coupling, complexity, cohesion, etc. These structural metrics include suites such as CK [28], QMOOD [12], among others.	All
Evolution-based	These metrics quantify the change history of software evolution, from one release to another, e.g., Birth of a class, amount of changes, frequency of changes, change density, and others.	[3, 20, 21, 23, 39, 103, 104]
Developer-related	Entropy of changes introduced by a developer in a time period, number of developers employed on a specific segment in a specific time, structural and semantic scattering of developers in a specific time period.	[20, 21, 22, 23]
Code Smells	These metrics are related to poor Object-Oriented design structures, which can cause source code degradation as code smells [47], antipatterns [17] and identity disharmonies [83].	[64, 123]
Others	Less used metrics based on the word vector method, the dependency graph of the software, and bugs collected in issue tracking repositories.	[5, 50, 145]
Dynamic metrics		
Events tracing	These are metrics collected during runtime that trace software events, including methods execution time, number of method calls, execution flow, etc.	[51]

which kind of code structure was changed (loop, control structure, statement) and can provide more detailed information about the changes on the level of statement and declaration changes. The drawback of the FGC approach is that it is more complex to implement and can let some small changes not be considered.

2.1.2 Independent Prediction Variables

There are different types of independent variables used to predict code change-proneness in literature. Table 2.1 contains a brief description of static and dynamic metrics adopted by previous work. The categories of static metrics are: structural, evolution-based, network, developer-related, word vector, code smells, and bugs. Dynamic metrics contain only software event tracing and are costly to be obtained. Because of this, the related works found in the literature that use static metrics, like software quality metrics, code smells and anti-patterns, are described as follows.

2.1.2.1 Software Metrics

Recent works have been demonstrating that source code metrics can be used as independent variables to identify change-prone classes. This section shows related work that found which

metrics have a strong correlation with change-proneness. Various change-proneness models have used different categories of metrics for localizing and predicting the changes, but the set of structural metrics stands out for being used in all models [24, 52, 70, 90, 93, 105, 135].

Structural metrics, or internal quality metrics, can be observed in class, method, or component scopes and can include several sets of metrics. In class level scope, *Chidamber-Kemerer* (CK) [28] metrics suite, *Quality Model for Object-Oriented Design* (QMOOD) [12] and *Lorenz and Kidd* metric suite [89] are important sets of metrics. *McCabe* [111] and *Halstead* [32] metrics can be used for method scope. McCabe uses a set of four software metrics: cyclomatic complexity, essential complexity, design complexity, and the number of *Lines Of Code* (LOC). Halstead uses three sets: base measures, derived measures, and LOC measures. The logic of Halstead metrics is that code that is hard to read is more likely to be changed. Component scope includes attributes of a software component like portability, reusability, usability, suitability, and complexity in the context of a Component-Based Software System.

Lu et al. [90] employed statistical meta-analysis techniques to investigate the relationships between 62 structural metrics and change-proneness based on 102 Java systems. The investigated metrics cover four metric dimensions, including 7 size metrics, 18 cohesion metrics, 20 coupling metrics, and 17 inheritance metrics. They concluded that size metrics exhibit moderate or almost moderate ability in discriminating between change-prone and not change-prone classes. Coupling and cohesion metrics generally have a lower predictive ability compared to size metrics and inheritance metrics have a poor ability to discriminate between change-prone and not change-prone classes.

Zhou et al. [144] investigated the confounding effect of three structural metrics LOC, *Number of Methods Implemented In a Class* (NMINP) and the *Number of the Parameters* of these methods (NumPara) on the associations between cohesion, coupling, and inheritance metrics and change-proneness. The concept of confounding effect refers to an apparent association between independent and dependent variables that is thought to be the result of a third variable. The third variable that distorts the true association between the independent and dependent variables is called a confounder. The research has found that class size has a strong confounding effect on the associations between OO metrics and change-proneness and suggests that considers a confounding variable when validating OO metrics on fault-proneness and they aren't good predictors. They considered cohesion, coupling, and inheritance metrics to be more suitable for predicting change-prone classes, finding a subset of them that should be used in the context of change prediction models. At the same time, they also showed that the number of lines of code is not a good predictor. We can observe that their results diverge from the work of Lu et al. [90], mentioned before.

In addition to structural metrics, another commonly used set includes evolution-based metrics. Evolution-based metrics describe process metrics between changes in software classes that are indicators for change-proneness. Elish and Al-Khiaty [39] measured different dimensions than product metrics and complement them. Also, they are significantly correlated with class change-proneness and the combination of both sets of metrics improved the accuracy of software change prediction. The evolution-based metrics involve class creation and changes, frequency of changes, the time interval between changes, and the percentage of lines of code changed.

Developer-related metrics are factors in which how developers apply changes in the source code to capture development process complexity. Catolino et al. [22] adapted three developer-related models from fault prediction context to change-prone class, called *Basic Code Change Model* (BCCM) [57], *Developer Changes Based Model* (DCBM) [34], and *Developer Model* (DM) [15]. In the evaluation, the models are compared with existing change-prediction models based on evolution and code metrics. Results indicate that DCBM tends to perform

better than the others, achieving the best scores with values of accuracy=78%, precision=61%, recall=70%, F-Measure=66%, and ROC AUC=69%. Compared with existing approaches the DCBM model had an overall F-Measure 15% higher than the Code Metrics model and 8% higher than the Evolution Metrics model. This set of metrics contains structural and semantic scatterings which calculates the distance between pairs of modified classes and textual similarities between pairs of classes, respectively.

Tsantalis et al. [135] studied the relation of addition and modification of functionality and change-proneness when a change in one module that would need a change in any other module, known as *ripple effect*. They categorized their work in three axes i.e. dependency, inheritance, and reference. The results indicated that their approach improved the prediction accuracy over a simple model that relies only on past data. They argue that structural metrics have been proven to be statistically insignificant predictors, except for class size.

Godara et al. [51] proposed some new metrics, mainly to trace internal events of the software, i.e., their work incorporates both static and dynamic features (class dependency and frequency) for prediction of change-prone classes in object-oriented software. The metrics are execution time, runtime information (i.e. trace events), class dependency, frequency, and popularity. Execution time is measured by the time of method execution. Runtime information is measured by which methods are executed during runtime and the count of times each method is run. Class dependency measures classes affected by ripple effect i.e., if a class is affected by changes, it will affect another class also. Frequency can be defined as the number of times a method is called by another method. Popularity is obtained from the frequency metrics, it indicates how popular a method is in terms of calling and being called. For evaluating and comparing the proposed change prediction model they have used the multivariate logistic regression algorithm. The artificial Bee Colony algorithm was implemented for data classification. Results indicated that behavioral dependency and frequency have a significant effect in predicting change. Also, the proposed prediction model using the Artificial Bee Colony Algorithm showed better results than with Logistic Regression, with AUC up to 0.925 for the former and 0.750 for the latter. However, their work is limited because they evaluated only two healthcare management systems as datasets.

Kumar et al. [79] performed an empirical study of 11 feature selection techniques to identify the suitable set of source code metrics, out of 21 metrics collected with the Understand tool, for change-proneness prediction. The results indicate that the selected subset of source code metrics, selected by feature selection techniques, had a better prediction performance than the models that used all metrics. They concluded that a large number of input features not only increases model building time but they also decrease the accuracy of the prediction. Hence, it is crucial to select a suitable set of features for developing the model. All works are summarized in Table 2.2.

2.1.2.2 Code Smells and Anti-Patterns

Code smells [47], anti-patterns [17] and identity disharmonies [83] represent poor and flawed software design and implementation. Researches show that they have a strong association with code change-proneness and some of them are presented as follows.

Khomh et al. [69] investigated how 29 types of code smell impacted class change-proneness in releases of Azureus and Eclipse projects. They analyzed if classes with code smells are more change-prone than the classes without smells and the impact of specific smells on change-proneness. Their results showed that classes with code smells are more prone to change than classes without smells. For Azureus, the p-values are always significative with a high effect size, indicating that for all the analyzed releases change-prone classes are those with a higher

Table 2.2: Generated by Spread-LaTeX

Authors	Metrics	Algorithms	Indicators	#Systems
Lu et al. [90]	Structural	random-effect model	pooled ROC AUC, Sensitivity	109
Zhou et al. [144]	Structural	linear regression	descriptive analysis, spearman correlation, Confounding Effect Analysis	1
Elish and Al-Khiaty [39]	Structural and Evolution-based	PCA, Bivariate correlation analysis, Multivariate logistic regression	Correct classification rate average	2
Catolino et al. [22]	Structural, Evolution-based and Developer-related	ADTree, MLP, Decision Table Majority, Logistic Regression, SVM and Naive Bayes	Accuracy, precision, recall, F-Measure, AUC-ROC	10
Tsantalis et al. [135]	Inheritance, Reference and Dependency metrics	probabilistic analysis and logistic regression	change probability, accuracy, sensitivity, false positive ratio, false negative ratio, others	2
Godara et al. [51]	Structural, execution time, frequency, run time information, popularity class dependency	Logistic Regression, Artificial Bee Colony	Sensitivity, Specificity and ROC Curve	2
Kumar et al. [79]	Structural	18 learning algorithms and 3 ensemble techniques	Accuracy (%) and F-Measure	10

number of smells. For Eclipse, the results are significant with a small effect size, except for one release, where differences were not significant. Smell NotAbstract had a significant impact on change-proneness in more than 75% for Azureus releases, and HasChildren, MessageChainsClass, and NotComplex smells for Eclipse using Logistic Regression also had a high correlation.

Khomh et al. [70] also studied the relation between the presence of 13 anti-patterns and class change-proneness. The selected anti-patterns were AntiSingleton, Blob, CDSBP, ComplexClass, LargeClass, LazyClass, LongMethod, LPL, MessageChain, RPB, SpaghettiCode, SG, and SwissArmyKnife. Results show that classes that are part of anti-patterns are more change-prone than classes that do not participate. MessageChain correlated with change-proneness in more than 90% for the four software applications evaluated. LongMethod presented a percentage more than 90% in three of the four applications. Other anti-patterns also correlated in more than 75% for some applications were AntiSingleton, ComplexClass, LazyClass, LPL, and RPB.

Kaur and Jain [64] evaluated ten code smells as predictors for change-prone class prediction. Code smells investigated were: feature envy (F_E), long method (L_M), God class (GOD), empty catch block (ECB), unprotected main program (UMP), dummy handler (DH), nested try statement (NTS), careless cleanup (CC), exceptions thrown from finally block (ETFFB) and overlogging (OL). The results indicated that God class and long method smells are better predictors of change-proneness.

Pritam et al. [123] empirically validated the capability of the use of code smell for software change prediction with 14 open-source projects, including large support systems like ERP. The results indicated that code smell can predict class change-proneness with a probability superior to 70%. Catolino et al. [24] studied if the adoption of code smell-related information improves the change prediction model's performance. They explored the intensity index metric to capture the severity of code smells. They evaluated its contribution in conjunction with the state-of-the-art change prediction models based on product, process, and developer-related metrics and compared it with a previous work that used anti-pattern metrics. The results show

that the prediction performance of the intensity-including models is statistically better than the baselines and the intensity is a better predictor than anti-pattern metrics in up to 20%.

2.1.3 Existing approaches for change-prone code prediction

A wide variety of approaches have been used in previous studies to deal with software change prediction. We can classify the evolution of the main approaches applied over time in statistical, conventional ML, meta-heuristics, ensemble algorithms, and others including hybrids and unsupervised learning algorithms. One early effort started using DL algorithms. Table 2.3 summarizes ML techniques and others applied by previous work and the studies classified by approach are described as follows.

In an initial research stage, regression techniques, such as Linear Regression, Logistic Regression, and Linear Discriminant Analysis (LDA) stand out for being used in many works [39, 80, 135, 144]. Some of these traditional statistical algorithms have still been used in more recent studies to be compared with newer techniques [68, 104, 112]. Studies shifted from traditional statistical methods to conventional ML algorithms in the second stage. Malhotra et al. [98] compared the use between ML algorithms and statistical algorithms to predict class change-proneness. ML demonstrated better results than statistical algorithms. Authors start comparing multiple algorithms such as Naïve Bayes, SVM, MLP, Decision Trees algorithms, and Random Forest ensemble algorithm [22, 63, 81, 112]. Kumar et al. [78] also applied a transfer learning technique to the problem. After that, researchers started to apply ensemble algorithms like bagging, best-in-training, majority voting, and others to predict software changes [20, 68, 78, 80, 96, 101, 123]. We can also observe some works using meta-heuristics [101, 104], hybridized techniques [104] and unsupervised learning algorithms [86, 141].

Temporal dependency is the endeavor of extracting meaningful summary and statistical information from points arranged in chronological order. It is done to diagnose past behavior as well as to predict future behavior [118]. In general, temporal dependence is not directly supported by machine learning models. So, this kind of data must be restructured in an appropriate layout for supervised learning input. Data are sequenced in time windows which are sets of data with a fixed length called window size.

Melo et al. [112] adopted GRU and RNN DL algorithms and compared them with conventional ML algorithms considering and not considering temporal dependency. They adopted two proposed temporal dependency approaches called Concatenated and Recurrent to consider temporal dependency between code instances. To evaluate the approach, they used four applications and window sizes of 2, 3, and 4. The results indicated that the utilization of time dependency improved the performance of both proposed approaches in all evaluated datasets by up to 23.6% than the state-of-art approach. They used four systems as datasets and the ROC AUC results using DL and concurrent time dependency approach ranged from 0.654 up to 0.918. The authors describe the network configuration but do not mention which parameters were tested. Parameters like the number of hidden layers, number of neurons in each hidden layer, epochs, and others can have a significant impact on the accuracy of the network. Another limitation is that they did not use a validation approach to improve the model generalization. Moreover, the generated model is only based on structural metrics. Predicting software changes with time dependency consideration is scarce in the literature.

In a previous work [130], we evaluated the sliding window approach in machine learning models to predict change-prone classes. In this way, the proposed approach better captures the class change history and avoids the prediction that is impacted only by the current metrics values. The approach was evaluated with four ML algorithms by using structural, evolutionary, and smell-based metrics. The use of the *SW* approach improved significantly the prediction

Table 2.3: Prediction algorithms used for software change prediction

Category	Techniques	Studies
Statistical	Statistical algorithms that include, for example, regression techniques and Linear Discriminant Analysis.	[2, 21, 39, 41, 54, 68, 78, 80, 96, 98, 100, 106, 109, 113, 114, 128, 129, 135]
Conventional ML	Conventional supervised learning algorithms including various categories such as Decision Trees, ANN, among others	[1, 2, 11, 21, 49, 50, 65, 68, 71, 78, 80, 81, 86, 95, 96, 97, 98, 100, 101, 104, 109, 114, 123]
Meta-heuristics	Heuristic algorithms including several categories as search-based and bio-inspired algorithms, e.g., Artificial Bee Colony, Particle Swarm Optimization and Gene Expression Programming.	[10, 11, 65, 66, 71, 87, 99, 100, 102, 103, 108]
Unsupervised Learning	Algorithms that use unlabeled data to clustering, association and dimensionality reduction, like k-means, PCA and CLAMI+.	[40, 86, 141]
Ensemble	Algorithms that combine multiple learning algorithms e.g., bagging, boosting, voting, etc.	[1, 2, 3, 11, 21, 40, 65, 68, 78, 79, 80, 96, 100, 101, 102, 104, 109, 113, 114, 123, 145]
Deep Learning	Neural network based algorithms with multiple layers. The algorithm used was GRU.	[113]

performance, in comparison with traditional approaches that consider independent learning instances. Random Forest reached the best performance and no improvements were observed by using smell-based models.

2.1.3.1 Method-level Approaches

The works described as follows predict changes in the method level of granularity. Most of them associate changes by logical rules between source and target methods and classes of the changed versions. ROSE [146] reads a version archive, groups the changes into transactions, and mines the transactions for rules that describe implications between the software entities. When the user changes some entity, the ROSE client queries the rule set for applicable rules and makes appropriate suggestions for further changes. However, the performance presented is low, the best result obtained for methods was precision with 0.28 and recall 0.39 in the open-source compiler GCC.

Logical Structural Diff (LSdiff) [73] infer systematic changes and report exceptions that deviate from these systematic changes. LSdiff abstracts a program as code elements (e.g., methods and fields) and their structural dependencies (e.g., field accesses and overriding). LSdiff infers logical rules to discover and represent systematic structural differences. Any differences not represented by the inferred rules are retained as logical facts. Their results identified 75% of structural differences as systematic changes.

Bantelay et al. [13] presented an approach to creating predictions of evolutionary couplings using 4 years of commit histories and interactions to predict changes in files and methods [13]. They employed association rules mining technique, similar to ROSE. Precision and recall metrics were used to measure the performance of these models. They found that the commit model predicts interactions with a higher precision than the interaction model at both file and method levels, the interaction model can predict commits with a higher recall than the commit model at the file level and the combination does not outperform individual models in terms of precision and recall. They achieved a precision of less than 0.25 and a recall of less than 0.35 for the interactions and a precision of 0.4 and a recall of 0.6 for commits in the method level.

Sultana et al. [134] designed and conducted experiments with supervised learning techniques SVM and logistic regression to classify Java code as vulnerable or non-vulnerable for classes and methods. method-level metrics showed ROC AUC between 76.8% and 84.7% for SVM and 84.8% and 89.2 for logistic regression.

Some studies only investigate the relationship between different metrics and changes, without generating a prediction model. For instance, the study of Aman et al. [7] concludes that Java methods having longer named local variables and comments are more change-prone. Some works learn systematic edits from one or more manually provided training examples and build generalized patterns from them [38, 115, 127]. They search the code base for locations to apply these patterns to and present the code of the applied patterns as recommendations to developers. The patterns are created using Abstract Syntax Trees (AST), sets of code changes with annotations as inputs, which are manually provided by the developers. Kitsu et al. [74] propose a mechanism to detect program changes from code edit history that was recorded in past development sessions. They claim that developers want to grasp program changes in the scopes of classes, methods, and fields in object-oriented programs. Thus, the code changes are described based on types of C programs. By extracting small differences and aggregating them, their mechanism detects program changes and provides them to maintainers. They detect changes based on the heuristic rules, restore versions of source code from edit operations done in the past, and analyze class information extracted from the restored source code. These works have a different goal from ours. They automatically suggest changes to the developers in the current version of the system. Differently, our work considers methods that have changed in past releases and predicts change-prone code.

The study of Burhandenny et al. [19] shows that ML prediction models based on complexity measures can be improved by considering a metric that captures the abnormal density of comments. Our work differs from this work because we adopt a different methodology that compares different kinds of algorithms and models. We adopt a broader set of features composed of evolution-based and structural metrics. The metrics in this set capture comments and different kinds of complexity. We also performed a feature importance analysis and provide additional analysis and findings.

In a recent work [44] we empirically assess the performance of four machine learning algorithms for change-prone method prediction in seven open-source software projects. We derived and compared models obtained with three sets of independent variables (features): a set composed of structural metrics, a second set composed of evolution-based metrics, and a third that includes a combination of both kinds of metrics. The results show that Random Forest presents the best general performance, independently of the used indicator and set of features. The model composed of both sets of metrics outperforms the other two. Two features based on the frequency of changes that happened in the evolutionary history of the method are pointed out as the most important for our problem.

2.2 SOFTWARE PERFORMANCE PREDICTION

Performance is an important non-functional property of software quality that can directly impact different aspects like user experience, reliability, and security, among others. Moreover, software systems failures can be very often due to performance issues rather than functional bugs [27]. One of the most common performance issues is performance regression, such as response time degradation and increased resource utilization. Performance regression identifies such performance issues by comparing successive versions of a software project using existing tests.

The use of existing tests to compare performance between commits may prevent early-stage degradation and fix bugs in subsequent versions before they reach the users [59]. However, existing research is dedicated to identify performance regression after the fact, i.e., after the system is built and deployed in the field or dedicated performance testing environments [27].

Corrêa et al. [31] used software internal quality metrics with Neural Networks to estimate physical properties cycles, power, and energy for embedded systems. These metrics are related to the performance usage metrics of IoT hardware resources. These hardware metrics are indirectly associated with the execution time used in our work. They compare with multivariate regression predictions and the results demonstrate that even in cases where the data exceeds the regression confidence limits, the Neural Network was able to accurately predict. Vieira et al. [137] analyzed software metrics extracted from class diagrams and classes from source code and used regression analysis with five ML algorithms to predict energy consumption and estimate NFRs for embedded applications. The results were highly accurate, with only 1.66% errors.

2.3 CONCLUDING REMARKS

This chapter presented important aspects like definitions, sets of metrics, algorithms, and evaluation approaches for software change prediction and software performance prediction found in related work. In regards to software change prediction, a combination of structural and evolution-based metrics, relying on ensemble algorithms such as Random Forest is the model most used in the state of the art to predict software changes. Moreover, the best results of the empirical evaluations of ROC AUC are 0.76, considering common evaluation criteria [105].

Studies do not show a consensus on the use of structural metrics, despite practically all software change prediction research using them. Tsantalis et al. [135] concluded that structural metrics are insignificant features, except class size. Lu et al. [90] analyzed the highest number of systems in literature and confirmed that size structural metrics have moderate capacity. On the other hand, Zhou et al. [144] indicate that size metrics present a strong confound effect and are not good predictors. Besides the divergences, Kumar et al. [79] show the importance of choosing the right features to improve accuracy and build a more generic model.

It was observed that there are few works on the prediction of change-prone methods, mostly by associating logical rules between source and target changed methods, having the best result with identifying 75% of the structural differences as systematic changes with LSdiff [72]. Furthermore, machine learning algorithms are used in the majority of works with the class level of granularity in literature. Worth highlighting that most studies do not [113] employed any temporal dependency approach and Sultana [134] reached the metric ROC AUC up to 89.2% predicting security impactful method changes with logistic regression. These results inspired us in defining our methodology to predict change-prone methods, presented in Chapter 3. Hence, as far as we are aware, this research is the first effort on predicting performance impactful method changes.

3 STUDY DESCRIPTION

As mentioned in Chapter 2, there is a predominance of class scope in software change prediction research, the works consider that the instances are all independent, and models to predict changes that impact performance were not found. To fulfill the gaps in predicting software changes at the method level and performance impactful changes and evaluate the temporal dependency between instances, two studies were conducted in this work. One uses *Fine-Grained Changes* (FGC) to predict change-prone methods in the releases of seven target systems and the other uses the execution time of methods to predict performance impactful method changes on five target systems. The models use three feature sets arranged in two formats: traditional and with a temporal dependency between the training instances. The traditional is the typical format used in supervised ML models with a set of feature vectors that represent samples and are divided into a feature matrix and corresponding target. The temporal dependent format uses the sliding window (SW) approach. The sliding window approach involves dividing the historical data into consecutive windows of fixed sizes, allowing the prediction model to capture temporal patterns and changes over time. The details are explained as follows.

This chapter is organized into seven sections. Section 3.1 presents our research questions. Next, Section 3.2 shows a comprehensive overview of the methodology of this thesis. Section 3.3 defines the variables we choose to create the datasets. Section 3.4 explains how the data is collected and the dataset is created. Section 3.5 explains how we preprocessed the data and Section 3.6 describes how the models are built and evaluated. Section 3.7 contains the concluding remarks of this chapter.

3.1 RESEARCH QUESTIONS

Before proceeding, we recap that our goals are to explore the machine learning models for the prediction of change-prone methods in general and the prediction of performance impactful method changes with and without data reshaped to a temporal dependent format. Considering our goals, we designed five research questions (RQs) to investigate the two problems we are studying. First, we want to know in RQ1 what is the power prediction of the models with a dataset in the traditional format. Next, RQ2 evaluates the effectiveness of the feature sets in the prediction. RQ3 is concerned with comparing three window sizes to evaluate their influence on the effectiveness of the prediction models of this research. RQ4 compares the prediction of traditional and temporal dependent dataset formats. RQ5 regards evaluating the predictive power of the features to find the most important features for the machine learning models of this research. All of these RQs were designed to reach our goals.

- RQ1 - **What is the effectiveness of the machine learning techniques?**

  This research question aims to assess and compare the performance of various supervised machine learning techniques in predicting change-prone methods specified in our goals. It is motivated by the hypothesis for a potential similarity between change-prone class and method predictions. By examining the predictive ability of the algorithms: decision tree, logistic regression, neural networks, and random forest, we explore the effectiveness and reliability of these techniques in accurately identifying change-prone methods and performance impactful method changes. The results are evaluated with accuracy, F1-score, Sensitivity, and ROC AUC score of the models (see Section 3.6).

- RQ2 - **Which feature sets yield the most accurate prediction?** This research question aims to explore the effectiveness of two feature sets and their combination in predicting the objects of this research. Such an analysis is important to define how these feature sets affect the results of prediction models and which one is better to be used.
- RQ3 - **How do different window sizes in the sliding window approach affect the predictive ability of the models proposed to reach our goals?** This research question aims to analyze the influence of various window sizes in the sliding window approach on the accuracy of predicting the models designed to reach the two goals of this thesis. We want to explore how the choice of window size affects the prediction accuracy of change-prone methods and determine the window size that maximizes the prediction accuracy.
- RQ4 - **How does the predictive effectiveness of models obtained with the Sliding Window approach compare to the models obtained with a traditional representation?** In this RQ, we intend to assess and compare the performance of the models with the sliding window data representation based on a temporal dependency with respect to the algorithms' performance using a traditional machine learning prediction format.
- RQ5 - **Which features are the most important?** In addition to analyzing different feature sets and algorithms, we investigate the individual importance of each metric. To answer this RQ, we apply two techniques to measure the feature importance: Information Gain [117] and Mean Decrease in Impurity (MDI) [88].

3.2 METHODOLOGY OVERVIEW

An overview of the methodology employed to develop the prediction models used in the experiments is shown in Figure 3.1. In the initial phase, we studied previous work and identified pertinent change-proneness and software performance indicators to establish the dependent and independent variables. To choose the change-proneness variables, we followed the works of Catolino et al. [24], Elish et al. [39], Malhotra and Khanna [105] and selected the structural, and the evolution-based metrics. The chosen software performance metric was the execution time of methods, based on Reichelt and Kühne [125]. After the selection of the metrics in the step of Select Variables, we created the datasets by mining public repositories of GitHub and collected the metrics using static and dynamic analysis tools (Step Collect Data). In step Preprocess Dataset, we first applied some procedures to normalize, balance, reshape to sliding window representation, and analyze feature importance. At the end of this step, we prepared four datasets for each target system. Two datasets were created for the prediction of change-prone methods, one traditional and the other reshaped with the SW approach. The other two datasets were created for performance impactful method change prediction, also with traditional and SW formats. In the fourth step, Build ML Models, we used the resulting datasets to train and validate the ML models. In the last step, Evaluate ML Models, we evaluated the predictive ability of the ML algorithms by using prediction performance indicators. Details about how these steps were conducted are presented next.

3.3 SELECT VARIABLES

This work uses supervised learning to predict change-prone methods and performance impactful method changes and we need to choose which features will be employed to obtain independent

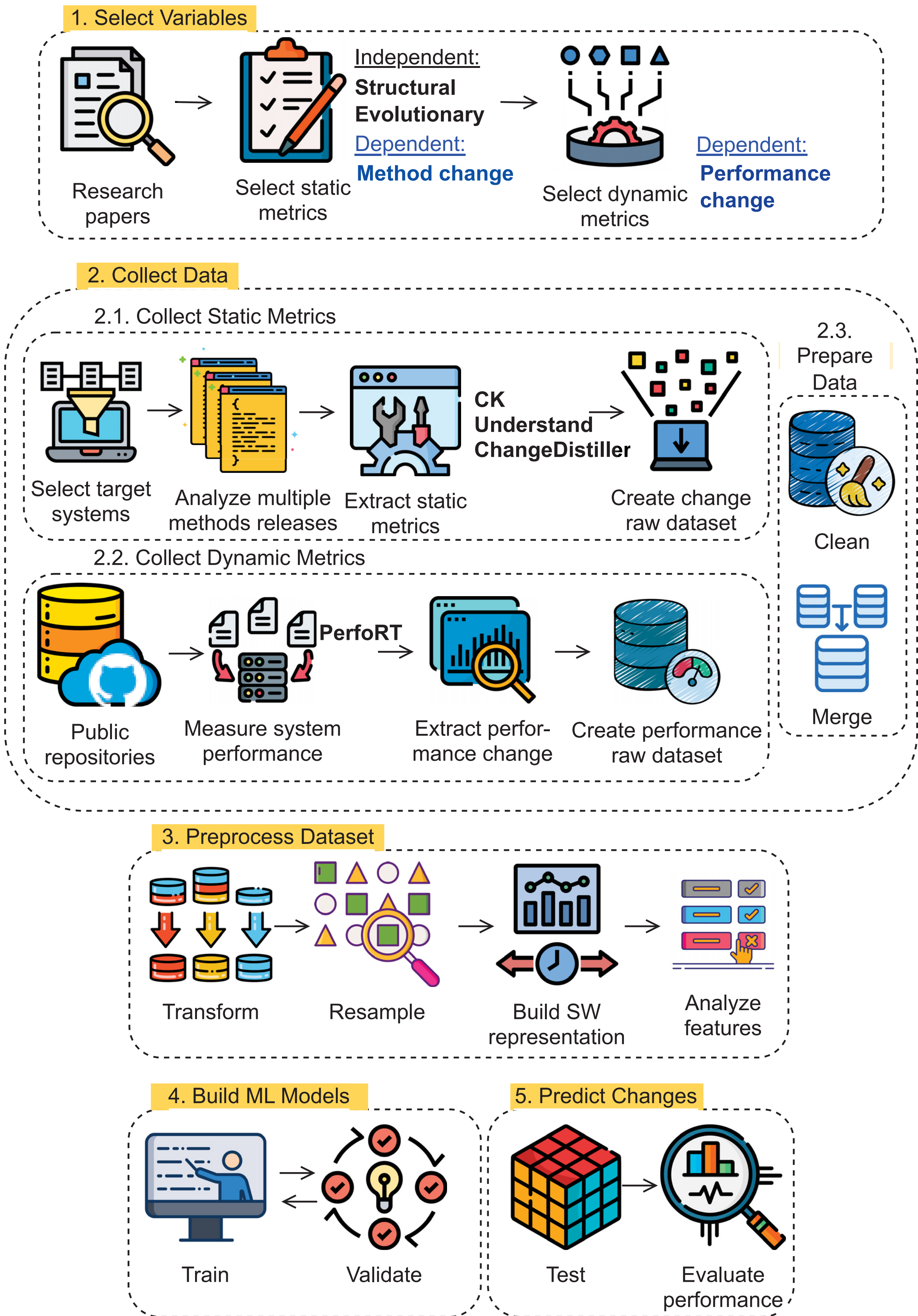


Figure 3.1: Overview of the methodology

and dependent variables (X and Y pairs) for training, validation, and evaluation of the models. To this purpose, we performed a review of previous software change prediction works to identify the features to be used as independent and dependent variables. This section describes them.

3.3.1 Independent Variables

As reported in Chapter 2, previous work indicates that structural, and evolution-based are the most used features to represent software change prediction. Following these findings, these two categories of code features are used as the independent variables in this project. The description of the metrics is available in Appendix B. We use 79 structural metrics related to cohesion, coupling, complexity, inheritance, and size, which depict attributes of *Object-Oriented (OO)* elements. These metrics include suites such as CK [28], and QMOOD [12], among others. Evolution-based metrics are related to the software evolution from one release to another, e.g., the birth of a method, amount of changes, frequency of changes, change density, and so on. It is worth mentioning that the changes captured by the evolution-based metrics are related to a simple SLOC change between the two versions.

3.3.2 Dependent Variables

We have two different dependent variables: will_change and performance_change. Will_change represents whether a method changed or not between two subsequent releases. Releases are deployable software iterations packaged and made available for a wider audience to download and use[1]. The releases are self-declared by the developers through tags in their projects. To decide whether a method has changed in a given context we considered the *Fine-Grained Changes* (FGC) approach [24, 128]. A method changed if, in a period of time t, the number of structural changes of this method is higher than the median of the distribution of the number of changes experienced by all the methods of the system.

Software performance change represents if a method change between two subsequent releases impacted the execution time with a statistically significant difference. For software performance impactful method change, we do not consider FGC approach. Instead of it, we consider methods that suffered any change. It was necessary because changes that impact method performance are a subgroup of all method changes and consequently occur less frequently. The initial results with FGC showed an insufficient amount to predict the changes. Thus, to improve the number of occurrences we decided on this alternative. We used PerfoRT, described in Appendix C, to collect the execution time of each method. We assume that software performance metrics are random variables, as each measurement of execution time may result in different values. Thus, we measured methods in at least five executions and used Student's t-test to identify performance changes between subsequent releases to compare the distributions of the measurements, as used by other works [4, 6, 82].

3.4 COLLECT DATA

In this section, we present the steps to collect the data and create the datasets. First, we describe how the target systems were selected. Next, how we cloned the repositories of these systems, collected static and dynamic metrics as independent and dependent variables, and set values to label the dependent variables. At last, we specify how we prepared the raw datasets to be used by the predicting models.

[1]https://docs.github.com/en/github/administering-a-repository/releasing-projects-on-github/about-releases

3.4.1 Select Target Systems

The criteria to select systems used in our research are described as follows. We searched in repositories of GitHub for open-source projects with at least 10 releases, developed primarily in Java. The choice of Java increases replicability and has more support from external tools, especially for extracting resources from source code. Thus, an advanced search was performed in the GitHub search engine, selecting active systems, developed in Java, created before 2018, and updated in 2022. The results were sorted by popularity, based on the highest number of stars. Moreover, since some tools work on compiled code to compute metrics values, selected systems need to be compilable, i.e., not producing any errors during compiling.

We used two sets of systems, one for the change-prone methods prediction and the other for the performance impactful method changes prediction. For change-prone methods prediction, we collected source code files from seven open-source software systems. The selected systems and their main characteristics are described in Table 3.1. In total, the dataset of the change-prone methods has more than one million method instances analyzed, from 25530 distinct method definitions derived from 3125 class definitions, across 364 releases of the seven systems. For software performance impactful changes prediction, we used five target systems listed in Table 3.2. We collected 21421 instances of 1909 distinct methods implemented in 382 different classes in 40 releases of the five target systems. The number of releases collected to the systems related to software performance is smaller because measuring the performance metrics takes too long to execute.

Table 3.1: Change-prone method dataset details

Application	Releases	Classes	Methods	Instances
`commons-bcel`	32	397	3970	113768
`commons-csv`	28	43	682	17861
`commons-io`	62	300	1826	70459
`easymock`	18	195	1356	43318
`openfire`	162	719	7753	579173
`pdfbox`	10	953	6399	102691
`wro4j`	52	518	3354	75630
Total	364	3125	25340	1002901

Table 3.2: Performance impactful method changes dataset details

Application	Releases	Classes	Methods	Instances
`commons-bcel`	6	71	144	2508
`commons-csv`	6	31	486	4190
`easymock`	5	136	730	9332
`jgit`	7	89	292	2337
`openfire`	16	55	257	3054
Total	40	382	1909	21421

3.4.2 Collect Static Metrics

In this section, we describe how static metrics are collected to be used as independent and dependent variables. First, we collected the independent variables. We cloned the *git* version control repositories of the target systems and used Pydriller [132] to select commits tagged as releases and extracted their hashes. Then, we traversed commits from the most recent to the oldest previous commit and run a set of tools to extract static metrics in each release. In order to obtain data related to static metrics, we use the tools Understand [2], and CK [3]. These tools are static code analyzers that collect structural metrics of classes and methods. To extract methods from classes and evolution-based metrics from methods, we developed a tool. This tool and the collected data are available in our replication package [42].

The values of the independent variables were computed considering the release before the one where the dependent variable is computed, i.e., we computed the independent variables in the release R_i, while the change-proneness was computed between R_i and R_{i+1}; in this way, we avoid biases due to the computation of the change-proneness in the same periods as the independent ones. Table 3.3 illustrates how the data is collected. In release R_0, three metrics values (M_1, M_2, and M_3) are extracted for a method. But at this time we do not have information about changes. In release R_1, the metrics values for the same method are calculated again, and at this time it is possible to set a Boolean value for the dependent variable Change; 1 represents the method changed using the FGC approach, and 0 otherwise.

Table 3.3: Example of how metrics are collected by release

R	M_1	M_2	M_3	Change
0	1	10	100	
1	2	20	200	1
2	3	30	300	1
3	4	40	400	0
4	5	50	500	1
5	2	20	400	0

Next, for each pair of releases, represented by their respective commits (c_i, c_{i+1}) of t, we ran the *ChangeDistiller* tool [46] to identify changes. This tool implements a differencing algorithm that generates and compares the syntax tree between two versions of a project, ignoring white space-related differences and documentation-related updates.

Therefore, a method is considered as change-prone if it underwent a number of changes higher than the median of the distribution of the number of changes experienced by all the methods of the application. In this case, we define the dependent variable method change equals 1 and 0 otherwise in the change raw dataset.

3.4.3 Collect Dynamic Metrics

Dynamic metrics are more difficult to collect due to many reasons. First, it is necessary to establish a strategy to run the systems. A deployed application can be used to measure the dynamic metrics. It implies that you have access to the environment to install the monitors and save the metrics while real users are using the system. Another alternative is to install the system with the monitors in a laboratory environment and use a load generator to simulate users

[2] https://www.scitools.com/
[3] https://github.com/mauricioaniche/ck

accessing the system. In this strategy is necessary to characterize the usage pattern of the users to plan the load [138]. A third option is to run regression tests of the system to execute parts of the code covered by the test cases [125]. In the scope of this research, this is the most appropriate strategy as we want to measure the execution time of the methods and we are not worried about the behavior of the users specifically. Running regression tests in many releases is an arduous task. It implies that the system compiles, executes, runs tests, and monitors the performance without errors in all releases. Additionally, software performance is unpredictable and the results can vary in each execution. Thus, we must run the tests many times repeatedly and apply a statistical analysis to generate more reliable results. Another factor that makes the collection of dynamic metrics difficult is the fact that it tends to overload hardware, OS, and application resources. Measurements must be re-performed in these situations.

After selecting the target systems, we measure the dynamic metrics using PerfoRT. As detailed in Appendix C, it clones the systems and runs their existing JUnit test cases on the releases to be assessed. We assume that software performance metrics are random variables, as each measurement of execution may result in different values. Thus, we measure 5 iterations of execution for each existing method called by test cases of the project. Many methods are called multiple times by the code of their own project during the project execution. Thus, each method is measured at least five times but can be executed more times. During the execution of the methods called by test cases, PerfoRT measures all metrics described in Section C.3. To create the software performance raw dataset, first, we extracted the following dynamic metrics from the measured data: system, commit hash, class name, method name, method own duration, and committer date. The method's own duration is the execution time of the method without including the execution time of other methods called by him. A method call stack was used to collect these execution times. Next, to identify software performance changes we compare the method's execution time difference between subsequent releases with Student's t-test.We consider the difference to be statistically significant if the p-value < 0.05, and not otherwise. Then, we define in the software performance raw dataset the dependent variable performance change to 1 if the difference is significant and 0 if not.

3.4.4 Prepare Data

After collecting variables, we performed the data cleaning and merging. Cleaning implies identifying and discarding methods without values in one of the feature sets. For example, if the CK tool collected data from one method but other tools, e.g. Understand, did not retrieve any data, this method was discarded. Next, empty values were filled with zero and identical rows were removed. Duplicate rows are feature sets that refer to the same real-world entity, where each value in each column for that row appears identically to the same column values [61]. Duplicate data rows could be useless to the modeling process, if not dangerously misleading during model evaluation. For example, it is possible for a duplicate row or rows to appear in both train and test datasets, and any evaluation of the model on these rows will be correct and result in an optimistically biased estimate of performance on unseen data [18]. To finish the data preparation, we organized data by merging all the collected metrics to unify the dataset. The features obtained with each respective tool were merged based on the key fields: release hash, and class and method names.

3.5 PREPROCESS DATASET

This section describes how we transform the collected raw data to fit and evaluate machine learning algorithms. In addition, we explain how we reshape the data to format it with temporal

dependency between instances using the Sliding Window approach. At last, we set out how feature analysis is employed to evaluate the most important features of the prediction models.

3.5.1 Transform

The datasets were divided into training (70% of the instances), and testing (30% of the instances) sets with the command `train_test_split` of scikit-learn. These percentages are approximately 2/3 of the data to be used for training and validation and 1/3 for testing. To ensure the training set is on the same scale, it was normalized using the *min-max* technique, where the data are scaled with values ranging from 0 to 1. Applying data preparation techniques before splitting data for model evaluation can result in an incorrect estimate of a model's prediction result. For example, normalizing is necessary to calculate the smallest and largest values. If this information is used to build the model during the training data, it creates a bias in training that should not have. Fair training cannot consider these values and only values from the training itself. Thus, we applied these preparation techniques only to the training data.

3.5.2 Resample

By analyzing the data distribution, we found a high imbalance in relation to the dependent variable. Training unbalanced models can make the models tend to predict the class that has the highest frequency, generating erroneous results. To mitigate this, we applied to the training datasets the following resample techniques: a) three oversampling techniques: Synthetic Minority Oversampling Technique (SMOTE) [26], Random Over-Sampler (ROS) [14] and Adaptive Synthetic (ADA) [58]; and b) three undersampling techniques: Random Under-Sampling (RUS), Edited Nearest Neighbours (ENN), and TomekLinks (TL). We used the implementation provided by scikit-learn [122]. At this point, we described all the steps we used to create the datasets with the traditional format used to be used by machine learning models of this research.

3.5.3 Build SW Representation

In general, temporal dependent data are not directly supported by machine learning models. So, the data must be restructured in an appropriate layout for supervised learning input. As this representation is part of our goals, we use the Sliding Window (*SW*) approach [37] to reshape the data to capture the temporal dependency between the instances. In this approach, data are sequenced in time windows which are sets of data with fixed lengths called window sizes. It transforms the sequential dataset in a format that integrates into a single sample multiple prior time steps as input features (X) to predict future time changes. As a result, a dataset is built from the traditional dataset containing the values of the metrics represented in a chronological way by employing the *SW* approach. This concept allows the ML algorithm to use a window of size S, containing data of the previous S releases, as the source of information. In this way, the analysis is not limited to a fixed release and each instance of the dataset may contain the total or partial history of the method changes. The SW approach works as follows. First, we select metric values from the versions of a method for all releases, if the number of releases of the method is less than S, the method is dropped. Then we select metrics of the S first releases to create the first instance of the dataset. In the next iteration, we take the interval of S releases by starting from the second analyzed release, creating the second instance of the dataset. This process is repeated until the last release, always following the order of the chronological dataset.

To better illustrate the SW representation we use Tables 3.4 and 3.5. The first one illustrates the representation adopted by the traditional dataset format for the data collected

(Table 3.3). The model would predict 1 in case the method should undergo a change, or 0 otherwise. In this structure, each instance represents a version of the method in a given release of the project, and each instance contains the value of the metrics M as the independent variables. Table 3.5 shows the new dataset built by reshaping the original one with the sliding window approach and $S = 2$. Instance 0 contains metrics values M_1, M_2, and M_3 from Releases R_0 and R_1 in order to predict the dependent variable $changed$ from Release R_2, so one step ahead. The process is repeated by shifting the two boxes simultaneously over the samples, one step at a time, creating new rows until the window reaches the end of the table. In summary, the main difference between the traditional and SW approaches is that the traditional considers the metrics values of a single release, and in the second, the metrics values of the S previous releases are used into a single structure. The role of the window is to say how many releases will be aggregated at a time, that is, the size of the history of the metrics over this period.

The sliding window approach requires that observed data exists in the number of releases to sufficiently complete the window size to capture historical patterns. If it does not exist in a number of sequenced releases as the window size, the data is discarded. Thus, in this work, we compared three sizes of the window with values 2, 3, and 4 with the Kruskal-Wallis statistical test of ROC-AUC for datasets of structural, structural and evolution-based, only evolution-based feature sets. These values were chosen because they do not need a very large number of method changes in a row. First, we evaluated these three window sizes, and the results are presented in Sections 4.3 and 5.3. After evaluating the performance of these values, we selected the window size with the best result to compare the prediction performance of models that use the SW approach and traditional representation.

Table 3.4: Dataset with the traditional approach

R	M_1	M_2	M_3	Change
0	1	10	100	1
1	2	20	200	1
2	3	30	300	0
3	4	40	400	1
4	5	50	500	0
5	2	20	400	1

Table 3.5: Representation of Table 3.4 reshaped with $S = 2$

	R_i			R_{i+1}			R_{i+2}
R	M_1	M_2	M_3	M_1	M_2	M_3	Change
0	1	10	100	2	20	200	0
1	2	20	200	3	30	300	1
2	3	30	300	4	40	400	0
3	4	40	400	5	50	500	1

3.5.4 Analyze Features

We used Information Gain [117] and Mean Decrease in Impurity (MDI) [88] in the feature importance analysis, which provided valuable insights into the relevance and impact of different features, described in Section 3.3.1, to predict the target variable. Information Gain quantifies the amount of information provided by a feature to predict the target variable, with higher values

indicating greater predictive power. The results revealed that certain features exhibited higher Information Gain scores, indicating their importance in capturing the underlying patterns related to change-prone methods and performance impactful method changes. Similarly, the MDI approach, commonly used in ensemble algorithms like random forests, measures the contribution of each feature in reducing the impurity within the decision trees. Features with higher MDI values were identified as having greater importance in the prediction process. By leveraging both Information Gain and MDI, we gained a comprehensive understanding of feature importance, allowing us to identify the most influential features for predictions of change-prone methods and performance impactful method changes.

3.6 BUILD ML MODELS AND PREDICT CHANGES

To build the ML models, we employed the algorithms widely used in the literature [105]: Decision Tree (DT); Logistic Regression (LR); Multi Layer Perceptron (MLP), and Random Forest (RF). We used the algorithms implemented by the scikit-learn tool [122]. Default parameters were used for change-prone method prediction and we search the hyperparameters described in Table 3.6 for performance impactful method changes prediction.

Table 3.6: Hyperparameters used in performance impactful method changes models training

Algorithm	Parameter	Values
Decision Tree	max_depth	1 to 10
MLP	hidden_layer_sizes	(1,), (2,), (5,), (10,), (50,), (100,), (50, 50)
	activation	relu and tanh
	solver	adam and sgd
	learning_rate	constant and adaptive
Random Forest	n_estimators	100, 200, ..., 2000
	max_depth	10, 20, ..., 110

We generate three feature sets in order to evaluate the role of the different kinds of metrics used, these sets are named: i) *Str*, composed of only structural metrics, ii) *Evo*, composed of evolution-based metrics, and iii) *StrEvo*, including both, structural and evolution-based metrics. The training set is used to train the algorithms and measure the performance of the generated models, identifying, among the trained models, the models with the best results. To tune the model to increase the reliability and accuracy of the results, we performed a stratified 10-fold cross-validation [133], which consists of dividing the training data into distinct subsets and using part of this subset for training and the rest for validation (i.e., 9 folds for training and 1 fold for validation). The cross-validation was implemented with the scikit-learn tool [122].

In summary, for both problems, we used four algorithms, three feature sets, the original dataset without a resample technique plus six resample techniques, the traditional dataset format, and the SW format with three window sizes. Also, we applied the change-prone methods models in seven systems and the performance impactful method changes models in five systems. Thus, the number of models is calculated by: 4 algorithms x 3 feature sets x 7 resample techniques, multiplied by 7 systems x 4 dataset formats for change-prone methods prediction, and by 5 systems x 4 dataset formats. In the end, we generated a number of 2352 models for change-prone methods prediction and 1680 models for performance impactful method changes prediction, totaling 4032 models. All the raw data, scripts used, and the values of the indicators mentioned above for these models are available in our supplementary material [42].

Test data are used for a final evaluation of the trained models. The models are evaluated on samples that were not used to build or fine-tune the model to provide unbiased prediction performance results. The following indicators are used to measure the performance of the predictors. Consider that TP, TN, FP, and FN refer to the number of true positives, true negatives, false positives, and false negatives.

- **F1-Score**(3.1): is a weighted average of precision and recall. It gives equal importance to both precision and recall by considering their harmonic mean [139]. Precision is the ability of the classifier not to label as positive a sample that is negative. The precision is the ratio $TP/(TP+FP)$. Recall is intuitively the ability of the classifier to find all the positive samples. The recall is the ratio $TP/(TP+FN)$ [122].

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \tag{3.1}$$

- **Accuracy**(3.2): is the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined [116].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.2}$$

- **Sensitivity**(3.3): fraction of the study population that is actually positive [116].

$$Sensitivity = \frac{TP}{TP + FN} \tag{3.3}$$

- **ROC AUC**: Receiver operating characteristic (ROC) Area Under Curve. The ROC curve shows the trade-off between the true positive (TP) rate and the FP rate. The area under the ROC curve is a measure that evaluates the performance of the binary classifier in terms of TP and FP rates. An area close to 1 indicates a high-performance model and an area about 0.5 indicates a low-performance model. An increase in recall (TP rate) also results in an increase in FP rate. Therefore, the ROC measure helps to track the optimum point where the metric performs well with respect to the TP and FP rates [134]. ROC AUC is calculated as the Area Under the $Sensitivity$(TPR)-$(1 - Specificity)$(FPR) Curve.

ROC AUC is the main performance indicator used in our work to compare all models and choose the models with the best results. It considers equally positive and negative classes. We selected this indicator because it is more adequate to compare classification models, especially with unbalanced data because it cares about true negatives as much as true positives [55]. The other indicators are used to complement the analysis.

3.7 CONCLUDING REMARKS

This chapter described how this study is structured and its main implementation aspects. We defined the research questions and presented the steps of the methodology to select independent and dependent variables, collect static and dynamic data, create and preprocess datasets, build ML models, and predict change-prone methods and performance impactful method changes. The methodology applies to the two problems studied in this thesis that will be conducted and detailed in the next chapters. The research questions seek to assess which is the best algorithm

and feature set, and which is the best approach, traditional or based on the SW. In addition, a study on the importance of the features used is carried out. In the next chapters, the results for both problems are presented and analyzed: change-prone methods prediction (Chapter 4) and performance impactful method changes prediction (Chapter 5).

4 CHANGE-PRONE METHOD PREDICTION

The previous chapter provided a comprehensive description of this study, including the research questions, methodology, data collection process, and ML models building and evaluation. In this chapter, we present the outcomes of our investigation to verify whether change-prone methods can be predicted by ML models with the aim of reducing the scope and increasing the localization accuracy of software change prediction. We argue that it is an improvement to predict changes in very large classes and to create conditions to classify the purpose of the changes to be made. The next five sections report the obtained results of the experiments designed to answer the first research question. Then, in Section 4.6 we analyze the results and discuss the findings. Section 4.7 contains the threats to the validity of this study. At last, Section 4.8 presents the conclusions of the chapter.

4.1 RQ1 - WHAT IS THE EFFECTIVENESS OF THE MACHINE LEARNING TECHNIQUES TO PREDICT CHANGE-PRONE METHODS?

With the aim of exploring the effectiveness of ML models to predict change-prone methods, we compare ML models build with the algorithms Decision Tree, Logistic Regression, MLP, and Random Forest with the feature sets Str, StrEvo, and Evo with the traditional datasets containing all extracted instances and with datasets modified by resample techniques.

To answer this RQ we refer to Table 4.1, which contains, for each algorithm, the combination of the seven systems with the three feature sets and the results of F1-Score (F1), Accuracy (Acc), sensitivity (Sen), and ROC AUC of the models that have the best ROC AUC values. Column RS shows the resample technique employed by the models. The resample technique shown in the RS column is the one that achieved the best ROC-AUC for that system, feature set and algorithm, comparing all seven evaluated techniques.

The values highlighted in bold are the results with the highest and lowest values, for each indicator, considering the models of the table. The F1-Score values range from 0.41 to 0.96, Accuracy from 0.27 to 0.93, sensitivity from 0.38 to 0.94, and ROC AUC from 0.51 to 0.89. We can observe that the indicators values vary according to algorithms, feature sets, and systems. Thus, we conducted our analysis considering these three aspects.

Thus, we derived Table 4.2 that summarizes the number of times each algorithm reaches the best result (or equivalent to the best one) considering all systems. Adding up the results of all the indicators, we can see in the columns of totals (T) of the row Total that, in a general case, Random Forest performed better, independently of the feature set. MLP presents the second best performance, followed by Decision Tree and Logistic Regression. Considering the results of the indicators individually, Random Forest shows a greater amount than the other algorithms in ROC AUC and Sensitivity, with respective values of 13 and 14. It is also observed that Random Forest ties or has a smaller amount than MLP and Logistic Regression for F1-Score and Accuracy, but with a small difference. Evaluating only ROC AUC score with Evo feature set, Random Forest, and Decision Tree tie in 6 cases. The Evo feature set also performed better, except for MLP which tied with the total of 11 best models to each of the three feature sets. MLP has a balanced number of cases for each of the three feature sets and does not seem to be impacted by the feature set used.

Table 4.1: Selected Models According to AUC Values

	Decision Tree					**Logistic Regression**					**MLP**					**Random Forest**				
Features	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS
commons-bcel																				
Str	0.79	0.66	0.75	0.71	RUS	0.82	0.70	0.63	0.67	ENN	0.79	0.66	0.79	0.72	RUS	0.81	0.69	0.73	0.71	RUS
StrEvo	0.88	0.80	0.83	0.81	RUS	0.89	0.81	0.80	0.80	ENN	0.90	0.82	0.85	0.83	RUS	0.90	0.82	0.86	0.84	RUS
Evo	0.90	0.81	0.74	0.78	ROS	0.92	0.85	0.65	0.75	RUS	0.90	0.82	0.73	0.78	ROS	0.90	0.81	0.74	0.78	ROS
commons-csv																				
Str	0.85	0.75	0.80	0.77	RUS	0.89	0.80	0.69	0.75	ADA	0.91	0.84	0.75	0.79	ROS	0.86	0.76	0.83	0.80	RUS
StrEvo	0.88	0.78	0.82	0.80	RUS	0.89	0.81	0.67	0.74	ROS	**0.96**	0.92	0.74	0.83	ROS	0.88	0.79	0.85	0.82	RUS
Evo	0.84	0.73	0.67	0.70	RUS	0.88	0.79	**0.38**	0.59	RUS	0.83	0.72	0.62	0.67	ROS	0.84	0.73	0.67	0.70	RUS
commons-io																				
Str	0.81	0.69	0.63	0.66	RUS	0.83	0.71	0.63	0.67	ROS	0.85	0.74	0.66	0.70	ROS	0.79	0.66	0.75	0.71	RUS
StrEvo	0.96	**0.93**	0.66	0.80	ENN	0.87	0.78	0.70	0.74	SMOTE	0.95	0.90	0.77	0.84	ROS	0.88	0.79	0.83	0.81	RUS
Evo	0.91	0.84	0.66	0.75	ROS	0.90	0.83	0.51	0.67	RUS	0.91	0.84	0.64	0.74	ADA	0.91	0.84	0.66	0.75	ROS
easymock																				
Str	0.94	0.89	0.66	0.78	ROS	0.83	0.71	0.66	0.69	ADA	0.92	0.85	0.79	0.82	ROS	0.94	0.88	0.70	0.79	ROS
StrEvo	0.85	0.75	0.78	0.76	RUS	0.83	0.72	0.66	0.69	SMOTE	0.95	0.90	0.83	0.87	ROS	0.96	0.92	0.63	0.78	ROS
Evo	0.42	0.27	**0.94**	0.60	ENN	0.44	0.29	0.74	**0.51**	ENN	**0.41**	**0.27**	0.93	0.60	SMOTE	0.42	**0.27**	**0.94**	0.60	ENN
openfire																				
Str	0.86	0.76	0.74	0.75	RUS	0.86	0.76	0.65	0.71	ENN	0.91	0.84	0.68	0.76	ROS	0.86	0.76	0.80	0.78	RUS
StrEvo	0.86	0.76	0.74	0.75	RUS	0.86	0.76	0.66	0.71	ADA	0.93	0.87	0.68	0.78	ROS	0.87	0.77	0.81	0.79	RUS
Evo	0.50	0.34	0.80	0.57	ADA	0.71	0.55	0.46	**0.51**	ROS	0.49	0.33	0.80	0.56	ROS	0.50	0.34	0.80	0.57	NONE
pdfbox																				
Str	0.90	0.83	0.90	0.86	ROS	0.80	0.69	0.58	0.64	ENN	0.87	0.79	0.77	0.78	ROS	0.90	0.83	0.90	0.86	ROS
StrEvo	0.94	0.90	0.88	**0.89**	ROS	0.86	0.77	0.71	0.74	ROS	0.89	0.82	0.85	0.84	ROS	0.94	0.90	0.87	**0.89**	ROS
Evo	0.92	0.86	0.50	0.70	RUS	0.92	0.86	0.50	0.70	RUS	0.92	0.86	0.50	0.70	RUS	0.92	0.86	0.50	0.70	RUS
wro4j																				
Str	0.75	0.61	0.63	0.62	RUS	0.85	0.75	0.60	0.68	SMOTE	0.80	0.67	0.70	0.69	RUS	0.79	0.66	0.66	0.66	RUS
StrEvo	0.77	0.64	0.66	0.65	RUS	0.84	0.74	0.62	0.68	SMOTE	0.81	0.69	0.71	0.70	RUS	0.81	0.68	0.67	0.67	RUS
Evo	0.84	0.73	0.46	0.60	ENN	0.77	0.63	0.49	0.56	ADA	0.71	0.56	0.64	0.60	SMOTE	0.85	0.74	0.45	0.60	ENN

Legend: Multi-Layer Perceptron (MLP); structural model (Str); structural and evolution model (StrEvo); evolution-based model (Evo); F1-score (F1); accuracy (Acc); sensitivity (Sen); Resample technique (RS);

Evo feature set obtained the most number of best models for almost all indicators in the four algorithms, except for F1-Score and Accuracy in MLP on which Evo had a smaller number than the others.

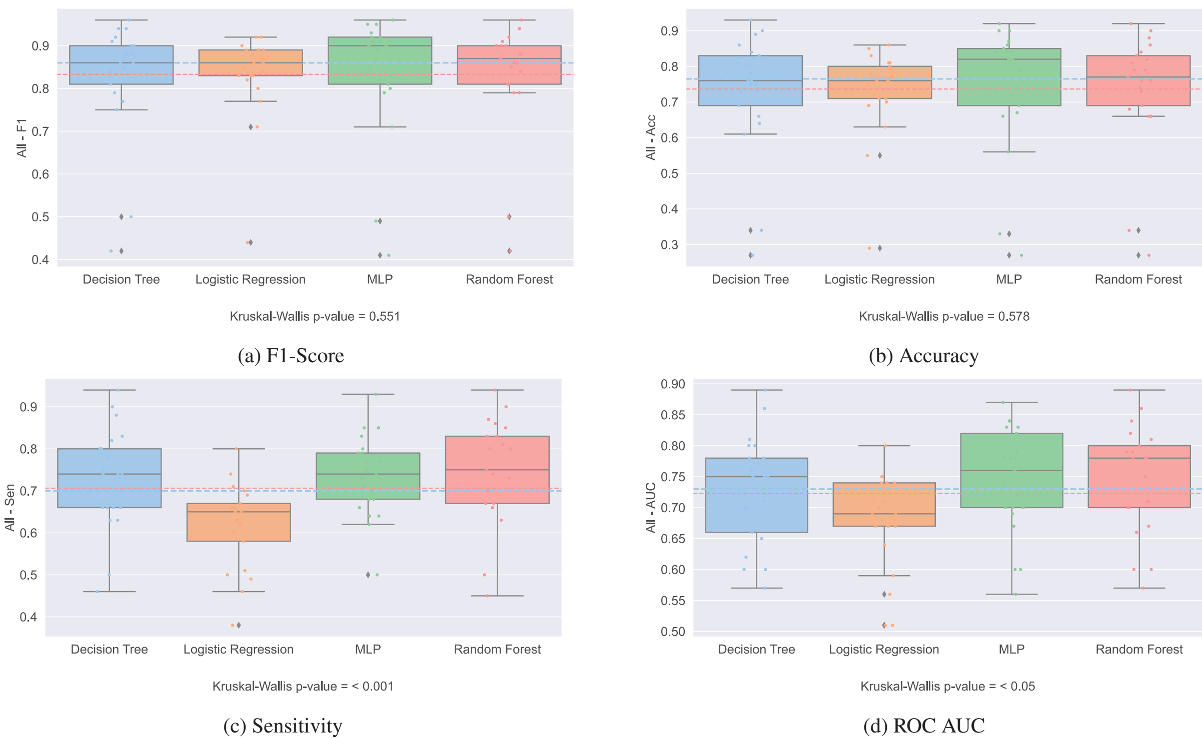
It is worth highlighting the performance of Random Forest regarding the values of sensitivity and ROC AUC. ROC AUC is considered a robust indicator, even in imbalanced datasets. High values of sensitivity imply a high value of true positive and a lower value of false negative, meaning that methods that really will change were identified correctly and that the developer will not waste time and effort monitory false positive or true negative cases. The Decision Tree algorithm presents good performance for these indicators too, but only when the set of evolution-based features is used.

Table 4.2: Number of times each algorithm reaches the best value for each indicator considering all systems

	Decision Tree				**Logistic Regression**				**MLP**				**Random Forest**			
Indicator	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T
F1-Score	2	2	2	6	2	1	5	**8**	3	3	2	**8**	2	3	3	**8**
Accuracy	2	2	2	6	2	1	5	**8**	3	3	2	**8**	1	3	3	7
Sensitivity	1	1	6	8	0	0	1	1	3	2	3	8	4	4	6	**14**
ROC AUC	0	1	6	7	0	0	1	1	2	3	4	9	4	3	6	**13**
Total	5	6	16	27	4	2	12	18	**11**	11	11	33	**11**	**13**	**18**	42

Legend: structural model (Str); structural and evolution model (StrEvo); evolution-based model (Evo); Total (T).

To corroborate this analysis, we employed the Kruskal-Wallis test and calculated the p-value considering models of Table 4.1. Figure 4.1 shows the obtained results per indicator. MLP has the best results in F1-Score and Accuracy, and Random Forest in sensitivity and ROC AUC, on average. For F1-Score and Accuracy, the algorithms do not present statistical differences, with p-values equal to 0.551 and 0.578, respectively. For ROC AUC and sensitivity, we found differences with statistical significance between Logistic Regression and the other three algorithms with p-value less or equal to 0.05. In relation to the other algorithms, except Logistic Regression, they did not show statistical differences. We applied a post-hoc analysis and the



(a) F1-Score (b) Accuracy

(c) Sensitivity (d) ROC AUC

Figure 4.1: Statistical analysis of the performance of the algorithms

details are available in our supplementary material [42]. The red dashed line indicates the mean and the blue the median.

The adoption of resampling techniques generates positive results. Table 4.3 was derived from Table 4.1 to show the number of resampling techniques used per algorithm. We can observe in Table 4.3 that only one of the best models selected for our analysis was not obtained using resampling. The random undersampling techniques, ROS and RUS, appear together in 61 out of 84 (72.6%) best results (7 systems x 3 feature sets x 4 algorithms). RUS appears 33 times in around 40% of the cases while ROS in 28 times, around 32%. TL does not appear in the best results and NONE appeared only once.

To corroborate this result and better compare the resampling techniques, we refer to Figure 4.2, created by using all the 588 models we generated. This figure shows the ROC AUC values of the models obtained with the resampling techniques and without any one of them (NONE). We can see that RUS has the best performance followed by ROS. The other resampling techniques achieved similar or worse results than the results without the use of resampling, on average. Similar results are obtained for the other indicators.

Table 4.3: Resample techniques with the best performance

Alg	**NONE**	**ADA**	**ROS**	**SMOTE**	**RUS**	**ENN**	**TL**
DT	0	1	5	0	12	3	0
LR	0	4	4	4	4	5	0
MLP	0	1	13	2	5	0	0
RF	1	0	6	0	12	2	0
Total	1	6	28	6	33	10	0

Figure 4.3 presents the results obtained by applying the Kruskal-Wallis statistical test using the models of Table 4.1 in order to analyze the performance of the algorithms in each

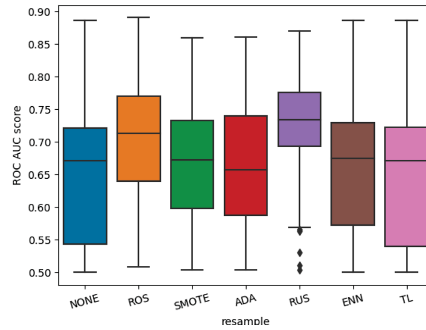


Figure 4.2: Comparing ROC AUC values of Resampling Techniques

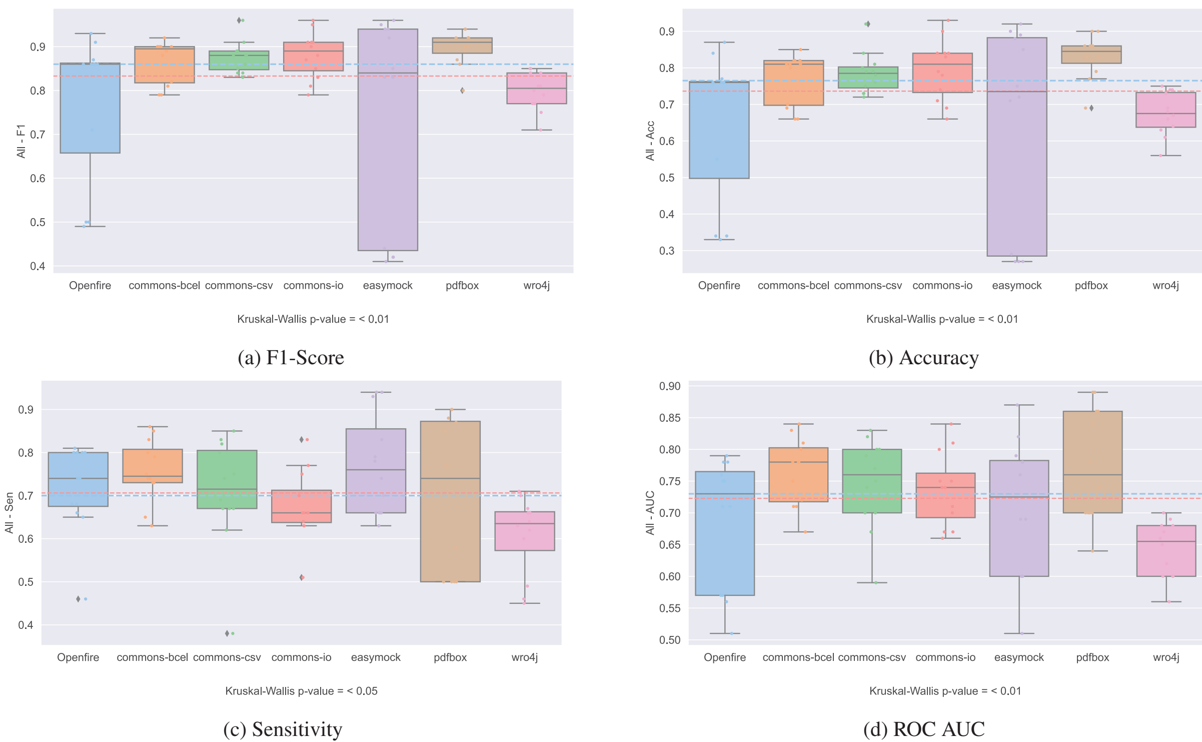
system according to each indicator. The results show prediction differences between the systems in the four indicators. The red dashed line indicates the mean and the blue the median values. The best F1-Score and Accuracy mean results are obtained for `pdfbox`, while the results of the system `commons-io` reach the maximum value; the highest mean and maximum values of sensitivity are obtained for `easymock`; for `commons-bcel` the best mean value of ROC AUC, while the maximum value for `pdfbox`. The worst values for all indicators were obtained for `wro4j`.

Comparing the results of the systems, we can observe that `commons-bcel` had the best ROC AUC result on average, while wro4J lowest result among the others. However, considering a model with good results on all indicators, like the MLP algorithm with the StrEvo feature set, ROC AUC values range from 0.70 to 0.87, with an average of 0.81, and the other indicators presented similar or higher results. Despite the difference in the results of the systems, to predict a change-prone method with 81% chance to hit on average, we can say that the results are satisfactory to help developers localize the next methods to work on, especially in cases where classes have a large number of methods.

Response to RQ1: Observing the ML techniques to predict change-prone methods, the model built with the MLP algorithm and the feature set composed of structural and evolution-based features obtained ROC AUC scores mean value of 0.82 for the target systems. There were other models that had similar results. The analysis indicates that Random Forest presents the best general performance, independently of the used indicator and feature set, and is followed by the MLP algorithm. Both algorithms present high ROC AUC and sensitivity values. On the other hand, the Logistic Regression algorithm obtained the worst performance. The random resample techniques RUS and ROS got the best performance among the others, improving the results in most cases.

4.2 RQ2 - WHICH FEATURE SETS YIELD THE MOST ACCURATE PREDICTION OF CHANGE-PRONE METHODS?

To answer this RQ, our analysis compares the models obtained with the different feature sets used as independent variables. From Table 4.1 we derived Table 4.4. This table presents the number of times each kind of model presents the best values for each indicator, considering all the systems. In the first row of the table, we see that the structural model presents the best (or equivalent to the best) F1-Score value only for one system, the evolution-based model for 2, and



(a) F1-Score

(b) Accuracy

(c) Sensitivity

(d) ROC AUC

Figure 4.3: Statistical analysis of the algorithms' performance in each system

the model that combines both kinds of features for 5. This last model clearly achieved the best result in all indicators.

By using the models of Table 4.1 and applying Kruskall-Wallis test to the results of the different feature sets, we obtained Figure 4.4. The p-value is equal to or less than 0.05 for all indicators, which means a statistical difference between the sets. The set using both kinds of metrics is the best model with statistical significance, followed by the set with structural metrics and the worst set is the one that contains only evolution-based metrics.

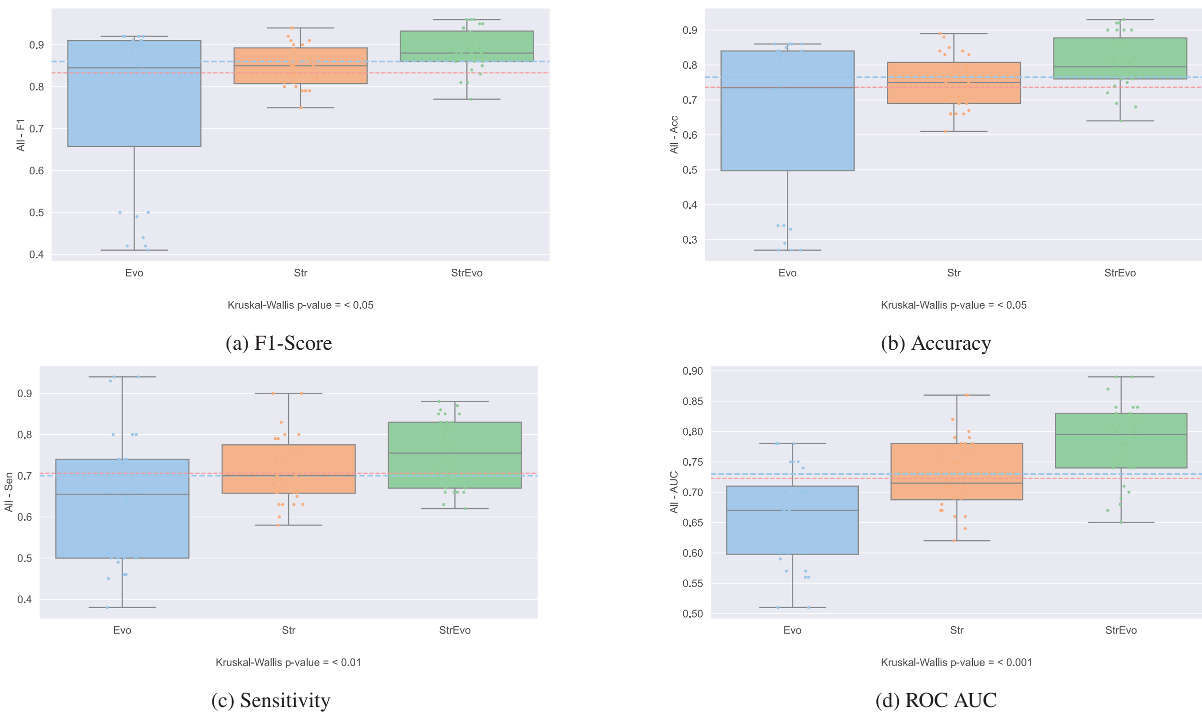
Table 4.4: Number of times each feature set reaches the best value for each indicator considering all systems

Indicator	Str	StrEvo	Evo
F1-Score	1	5	2
Accuracy	1	6	1
Sensitivity	1	3	3
ROC AUC	0	7	0

Response to RQ2: Models using both kinds of features, structural and evolution-based, are the ones with the best general performance.

4.3 RQ3 - HOW DO DIFFERENT WINDOW SIZES IN THE SLIDING WINDOW APPROACH AFFECT THE PREDICTION PERFORMANCE?

As mentioned in the last chapter, we investigated three values for S (2, 3, and 4). We empirically tested initial window sizes, starting with 2, the smallest possible window, as in the work of



(a) F1-Score

(b) Accuracy

(c) Sensitivity

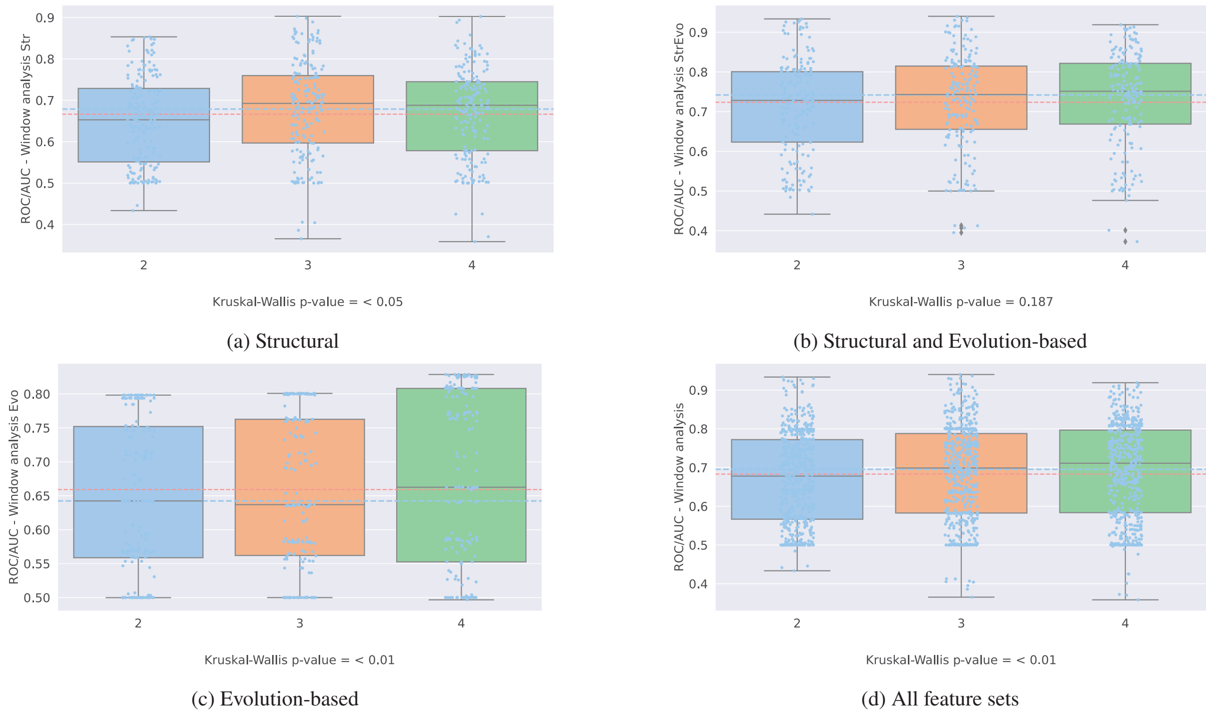
(d) ROC AUC

Figure 4.4: Statistical analysis regarding the performance of the feature sets

Tsoukalas et al. [136]. Larger window values did not imply an improvement in the results because the larger the window size, the greater the necessary history of the method, thus excluding methods added in more recent versions; this consequently reduces the training data and can generate bias in relation to older methods. By testing three feature sets, three window sizes, four algorithms, and seven resample techniques, we generated a total of 252 models for each application.

To answer this RQ we first performed two statistical analyses, using the Kruskal-Wallis test [77] and ROC AUC score. The first analysis considers the results of each feature set individually, as shown in Figures 4.5(a-c). For the feature set Str, the obtained p-value is less than or equal to 0.05, for the set StrEvo is 0.187, for Evo is less than or equal to 0.01. These results show that there are differences considering feature sets Str and Evo individually, but there is no statistical difference between window sizes for the feature set StrEvo. The second analysis joins all results from the three feature sets together and is shown in Figure 4.5(d). The results show a p-value less than or equal to 0.01, which means that there is a difference between the window sizes with statistical significance in this case.

Table 4.5 shows the Post hoc analysis of the ROC AUC score statistical analysis for the three feature sets for the seven target systems. The highest ROC AUC score mean results of each feature set as well as the maximum value of all are highlighted in bold. The lowest means of each feature set and the minimum result are marked in italic. Window size 3 got the highest mean value for the feature set Str, with 0.6792. The effect size analysis indicates that this feature set has a difference with a small magnitude compared with window $S = 2$ and negligible compared with $S = 4$. Window size 4 got the highest mean value (0.7316), marked by a star, in the feature set StrEvo, but the difference is insignificant compared with the other two sizes. To Evo feature set, we can observe that window size 4 has the higher ROC AUC score, on average, with statistical significance compared with the other window sizes in the analysis. The effect size shows a small difference compared with size 2 and is also negligible compared with size 3. The maximum value was reached in the StrEvo feature set and window size 3, with a value of



(a) Structural

(b) Structural and Evolution-based

(c) Evolution-based

(d) All feature sets

Figure 4.5: Statistical analysis regarding the performance of the feature set

0.9402. The lowest value, on average, was from the Evo feature set with window size 2. As a result, we observe that window size 4 appeared with the highest and with the maximum average score, followed by $S = 3$. We analyzed one more set called "All joined", composed of the joining of all results from the three feature sets, which is shown in Figure 6.5(d). The results show a p-value less than or equal to 0.01, which means that there is a difference between the window sizes with statistical significance in this case. As $S = 3$ obtained the best result in the Str feature set, the sizes do not present significant differences in the StrEvo feature set and $S = 4$ is the best result in the Evo feature set, the results indicate a tie between window sizes 3 and 4. To break the tie, we considered window size 4 because this window size, although there was no statistically significant difference, obtained the highest mean value of ROC AUC score from the StrEvo feature set and also obtained the best result with a statistical difference in the All joined feature set.

Response to RQ3: The models with $S = 4$ presented the highest mean ROC AUC score in the feature sets StrEvo, Evo and All joined. Models with $S = 3$ obtained the best results only in the feature set Str on which presented the maximum value. We considered window size 4 as the best result.

4.4 RQ4 - HOW DOES THE EFFECTIVENESS OF MODELS OBTAINED WITH THE SLIDING WINDOW APPROACH COMPARE TO THE MODELS OBTAINED WITH A TRADITIONAL REPRESENTATION?

To answer this RQ, we refer to Table 4.6, which presents the results of the best models for all three feature sets with $S = 4$, according to RQ3. This table presents, for each system and algorithm, the values of F1-Score, Accuracy, Sensitivity, and ROC AUC of the models obtained using the SW approach and the traditional one. The highest values for each indicator are highlighted

Table 4.5: Window statistical Post hoc analysis for ROC AUC indicator of the feature sets presented in Figure 4.5

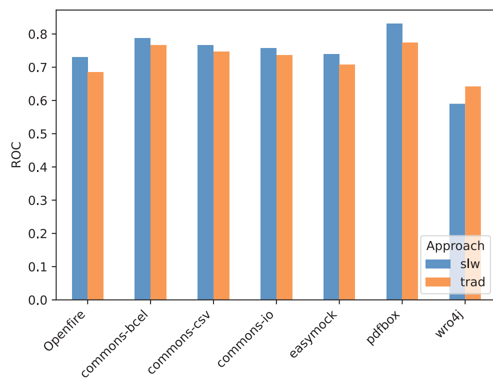
Feature set	Window	Mean	Std	Max	Min	Effect size
Str	2	0.6509	0.106	0.8533	0.4334	small
	3	**0.6792**	0.116	0.9029	0.3654	best result
	4	0.6691	0.109	0.9024	*0.3584*	neglibible
StrEvo	2	0.7130	0.120	0.9336	0.4417	insignificant
	3	0.7263	0.122	**0.9402**	0.3952	insignificant
	4	★**0.7316**	0.120	0.9189	0.3725	best result
Evo	2	*0.6477*	0.111	0.7983	0.4999	small
	3	0.6567	0.111	0.8008	0.5000	negligible
	4	**0.6732**	0.126	0.8287	0.4967	best result
All joined	2	0.6704	0.116	0.9336	0.4334	negligible
	3	0.6874	0.120	0.9402	0.3654	negligible
	4	**0.6913**	0.122	0.9189	0.3584	best result

in bold. The columns RS correspond to the resampling technique used to get the model that reaches the best performance. Considering ROC AUC, we can observe that the SW approach presented a higher value in almost all models, with only 7 exceptions from 84 cases. Thus, the SW approach improved the results in 91.7% of the best results. We calculated the mean values of the indicators, considering all algorithms and all feature sets from Table 4.6 (best results), to compare the models with the traditional dataset format with the models that employed the SW approach. The comparison is presented in Figure 4.6 for the best results and in Figure 4.7 for the StrEvo feature set only, which presented the best performance according to RQ2. If we consider ROC AUC, we can see that the SW approach got the best results in 6 out of 7 systems in both analyses. Regarding F1-Score and Accuracy, models with the SW dataset got higher values than the traditional on 4 out of 7 systems considering all best results and 5 of 7 in the StrEvo feature set. They also tied on one case. For the sensitivity indicator, SW approach had 5 outstanding results compared to the traditional approach for the analysis with all the best models and 6 of 7 for the StrEvo feature set. But we observe a better performance of the SW approach in most cases and great differences for systems `pdfbox`, and `Openfire`. This means that, for these systems, the SW approach tends to classify correctly the change-prone methods which are true positives. This is an important characteristic because the approach does not lead to spending time with false negatives. Moreover, except for the application `wro4j`, the best values of ROC AUC were obtained by models with the SW method, independently of the feature set and algorithms used. For the feature set StrEvo, the traditional approach got better results only in one of the systems for all four indicators.

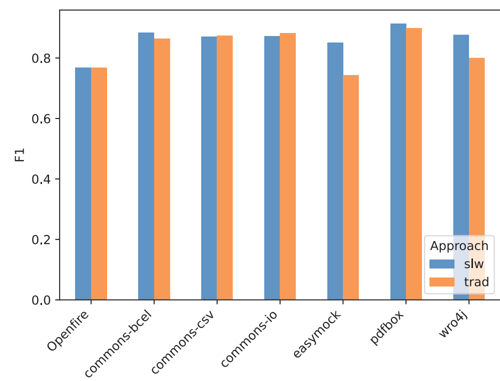
Table 4.7 corroborates this analysis. It presents, for each indicator, the number of times each approach was better or obtained an equivalent result in each feature set. The results confirm that the SW approach obtained the best values for the indicators compared to the traditional approach. We observe that for 84 cases analyzed in Table 4.7 (4 algorithms, 7 systems, and 3 sets), the models generated with the SW approach obtained the best values of ROC AUC in 75 (89.2%), and equivalent values in 2 models. The models of the traditional approach obtained the best ROC AUC values only in 7 (0.8%) cases. Considering the other indicators, the SW approach also overcomes the traditional. For F1-Score and Accuracy, the models with the SW approach obtained 61 (72.6%) best values, against only 14 cases where the traditional models were the best, with 9 ties. Regarding sensitivity, the results with the SW approach overcome

Table 4.6: Results from the best obtained models for each algorithm and feature set

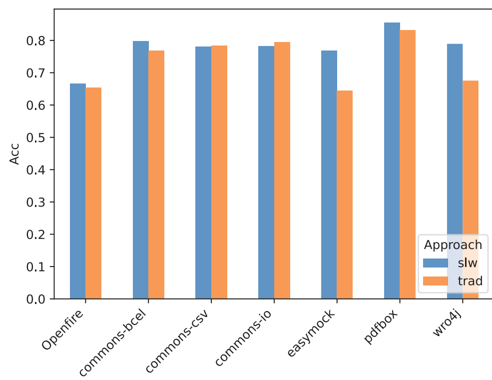
		Decision Tree					**Logistic Regression**					**MLP**					**Random Forest**				
Features	Method	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS
commons-bcel																					
Str	trad	0.79	0.66	0.75	0.71	RUS	0.81	0.69	0.67	0.68	ADA	0.79	0.66	0.79	0.72	RUS	0.81	0.69	0.73	0.71	RUS
Str	sw	0.89	0.80	0.60	0.70	ENN	0.81	0.69	0.69	0.69	ADA	0.86	0.76	0.72	0.74	ROS	0.83	0.71	0.77	0.74	RUS
StrEvo	trad	0.88	0.80	0.83	0.81	RUS	0.88	0.79	0.82	0.81	ADA	0.90	0.82	0.85	0.83	RUS	0.90	0.82	0.86	0.84	RUS
StrEvo	sw	0.92	0.86	0.86	0.86	RUS	0.91	0.84	0.86	0.85	SMOTE	0.92	0.86	0.90	0.88	RUS	0.91	0.84	0.89	0.86	RUS
Evo	trad	0.90	0.81	0.74	0.78	ROS	0.92	0.85	0.65	0.75	NONE	0.90	0.82	0.73	0.78	ROS	0.90	0.81	0.74	0.78	ROS
Evo	sw	0.93	0.86	0.79	0.83	NONE	0.93	0.86	0.79	0.83	ROS	0.93	0.86	0.79	0.83	SMOTE	0.93	0.86	0.79	0.83	ENN
commons-csv																					
Str	trad	0.85	0.75	0.80	0.77	RUS	0.89	0.80	0.69	0.75	ADA	0.91	0.84	0.75	0.79	ROS	0.86	0.76	0.83	0.80	RUS
Str	sw	0.88	0.79	0.80	0.79	RUS	0.89	0.80	0.76	0.78	ROS	0.93	0.87	0.75	0.81	ROS	0.88	0.79	0.84	0.82	RUS
StrEvo	trad	0.88	0.78	0.82	0.80	RUS	0.89	0.81	0.67	0.74	ROS	0.96	0.92	0.74	0.83	ROS	0.88	0.79	0.85	0.82	RUS
StrEvo	sw	0.88	0.79	0.89	0.84	RUS	0.89	0.80	0.77	0.79	ROS	0.98	**0.97**	0.73	0.85	ROS	0.90	0.82	0.89	0.86	RUS
Evo	trad	0.84	0.73	0.67	0.70	RUS	0.88	0.79	0.38	0.59	RUS	0.83	0.72	0.62	0.67	ROS	0.84	0.73	0.67	0.70	RUS
Evo	sw	0.87	0.78	0.85	0.81	RUS	0.73	0.59	0.64	0.61	RUS	0.88	0.79	0.84	0.82	ADA	0.90	0.82	0.79	0.81	NONE
commons-io																					
Str	trad	0.81	0.69	0.63	0.66	RUS	0.83	0.71	0.63	0.67	ROS	0.85	0.74	0.66	0.70	ROS	0.79	0.66	0.75	0.71	RUS
Str	sw	0.83	0.71	0.68	0.69	RUS	0.76	0.62	0.83	0.72	SMOTE	0.86	0.75	0.76	0.76	ROS	0.81	0.68	0.79	0.73	RUS
StrEvo	trad	0.96	0.93	0.66	0.80	ENN	0.87	0.78	0.70	0.74	SMOTE	0.95	0.90	0.77	0.84	ROS	0.88	0.79	0.83	0.81	RUS
StrEvo	sw	0.89	0.80	0.82	0.81	RUS	0.85	0.74	0.78	0.76	ROS	0.97	0.94	0.78	0.86	ROS	0.89	0.81	0.88	0.84	RUS
Evo	trad	0.91	0.84	0.66	0.75	ROS	0.90	0.83	0.51	0.67	RUS	0.91	0.84	0.64	0.74	ADA	0.91	0.84	0.66	0.75	ROS
Evo	sw	0.92	0.85	0.69	0.77	NONE	0.89	0.81	0.61	0.71	RUS	0.89	0.80	0.73	0.77	ROS	0.92	0.85	0.69	0.77	TL
easymock																					
Str	trad	0.94	0.89	0.66	0.78	ROS	0.83	0.71	0.66	0.69	ADA	0.92	0.85	0.79	0.82	ROS	0.94	0.88	0.70	0.79	ROS
Str	sw	0.96	0.92	0.66	0.79	NONE	0.86	0.76	0.68	0.72	ADA	0.94	0.88	0.79	0.84	ROS	0.86	0.75	0.85	0.80	RUS
StrEvo	trad	0.85	0.75	0.78	0.76	RUS	0.83	0.72	0.66	0.69	SMOTE	0.95	0.90	0.83	0.87	ROS	0.96	0.92	0.63	0.78	ROS
StrEvo	sw	0.98	0.96	0.71	0.84	ENN	0.89	0.80	0.71	0.75	ROS	0.98	0.95	0.78	0.87	ROS	0.87	0.78	0.90	0.84	RUS
Evo	trad	0.42	0.27	0.94	0.60	ENN	0.44	0.29	0.74	0.51	ENN	0.41	0.27	0.93	0.60	SMOTE	0.42	0.27	0.94	0.60	ENN
Evo	sw	0.93	0.86	0.51	0.69	RUS	0.91	0.84	0.44	0.64	ROS	0.93	0.87	0.51	0.69	RUS	0.53	0.36	**0.97**	0.66	SMOTE
openfire																					
Str	trad	0.86	0.76	0.74	0.75	RUS	0.86	0.76	0.65	0.71	ENN	0.91	0.84	0.68	0.76	ROS	0.86	0.76	0.80	0.78	RUS
Str	sw	0.91	0.84	0.85	0.84	RUS	0.87	0.78	0.68	0.73	ADA	0.97	0.95	0.86	0.90	ROS	0.91	0.83	0.89	0.86	RUS
StrEvo	trad	0.86	0.76	0.74	0.75	RUS	0.86	0.76	0.66	0.71	ADA	0.93	0.87	0.68	0.78	ROS	0.87	0.77	0.81	0.79	RUS
StrEvo	sw	0.91	0.84	0.85	0.85	RUS	0.88	0.79	0.69	0.74	SMOTE	0.98	0.95	0.85	0.90	ROS	0.91	0.83	0.90	0.86	RUS
Evo	trad	0.50	0.34	0.80	0.57	ADA	0.71	0.55	0.46	0.51	ROS	0.49	0.33	0.80	0.56	ROS	0.50	0.34	0.80	0.57	NONE
Evo	sw	0.54	0.38	0.80	0.59	ADA	0.36	0.23	0.89	0.55	ROS	0.53	0.37	0.80	0.58	SMOTE	0.54	0.38	0.80	0.59	ADA
pdfbox																					
Str	trad	0.90	0.83	0.90	0.86	ROS	0.80	0.69	0.58	0.64	ENN	0.87	0.79	0.77	0.78	ROS	0.90	0.83	0.90	0.86	ROS
Str	sw	0.95	0.91	0.82	0.87	TL	0.81	0.70	0.64	0.67	SMOTE	0.91	0.85	0.85	0.85	ROS	0.91	0.86	0.89	0.87	RUS
StrEvo	trad	0.94	0.90	0.88	0.89	ROS	0.86	0.77	0.71	0.74	ROS	0.89	0.82	0.85	0.84	ROS	0.94	0.90	0.87	0.89	ROS
StrEvo	sw	0.96	0.93	0.94	0.93	ENN	0.89	0.82	0.83	0.82	RUS	0.96	0.93	0.87	0.91	ROS	**0.99**	**0.97**	0.90	**0.94**	ADA
Evo	trad	0.92	0.86	0.50	0.70	RUS	0.92	0.86	0.50	0.70	RUS	0.92	0.86	0.50	0.70	RUS	0.92	0.86	0.50	0.70	RUS
Evo	sw	0.92	0.86	0.75	0.81	RUS	0.92	0.86	0.75	0.81	RUS	0.92	0.86	0.75	0.81	RUS	0.92	0.86	0.75	0.81	RUS
wro4j																					
Str	trad	0.75	0.61	0.63	0.62	RUS	0.85	0.75	0.60	0.68	SMOTE	0.80	0.67	0.70	0.69	RUS	0.79	0.66	0.66	0.66	RUS
Str	sw	0.92	0.86	0.25	0.56	SMOTE	0.83	0.72	0.67	0.70	NONE	0.88	0.78	0.47	0.63	ROS	0.97	0.93	0.16	0.56	SMOTE
StrEvo	trad	0.77	0.64	0.66	0.65	RUS	0.84	0.74	0.62	0.68	SMOTE	0.81	0.69	0.71	0.70	RUS	0.81	0.68	0.67	0.67	RUS
StrEvo	sw	0.94	0.88	0.25	0.57	ADA	0.83	0.71	0.67	0.69	ROS	0.89	0.81	0.42	0.62	ROS	0.97	0.94	0.15	0.56	ADA
Evo	trad	0.84	0.73	0.46	0.60	ENN	0.77	0.63	0.49	0.56	ADA	0.71	0.56	0.64	0.60	SMOTE	0.85	0.74	0.45	0.60	ENN
Evo	sw	0.83	0.71	0.50	0.61	ROS	0.77	0.64	0.52	0.58	SMOTE	0.73	0.58	0.61	0.60	SMOTE	0.83	0.71	0.50	0.61	ROS



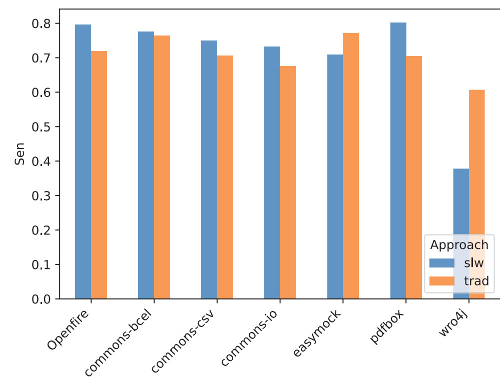
(a) ROC AUC



(b) F1-Score

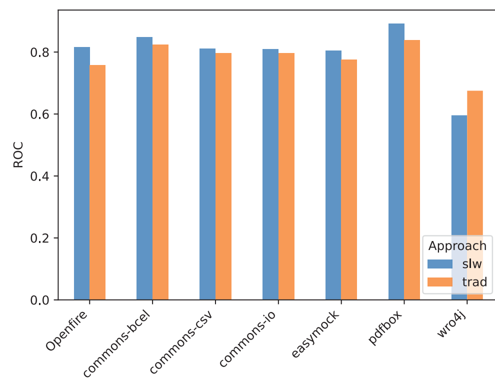


(c) Accuracy

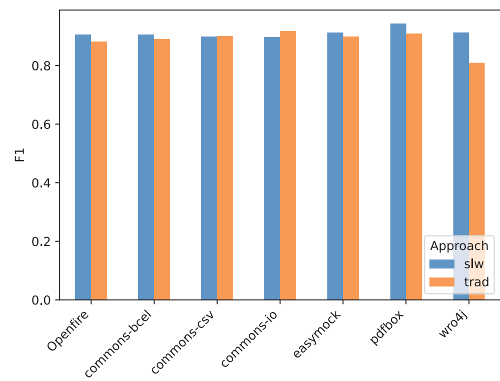


(d) Sensitivity

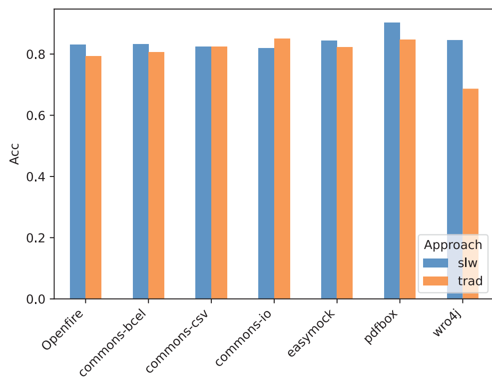
Figure 4.6: Comparing both approaches per indicator from the results of the best models



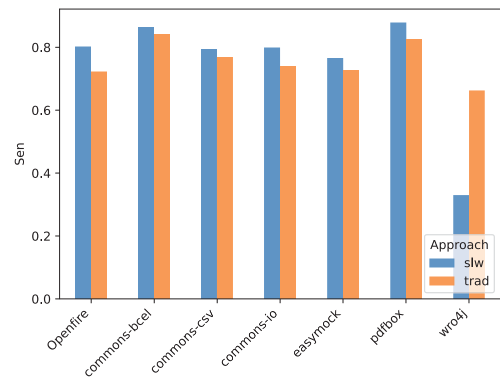
(a) ROC AUC



(b) F1-Score



(c) Accuracy

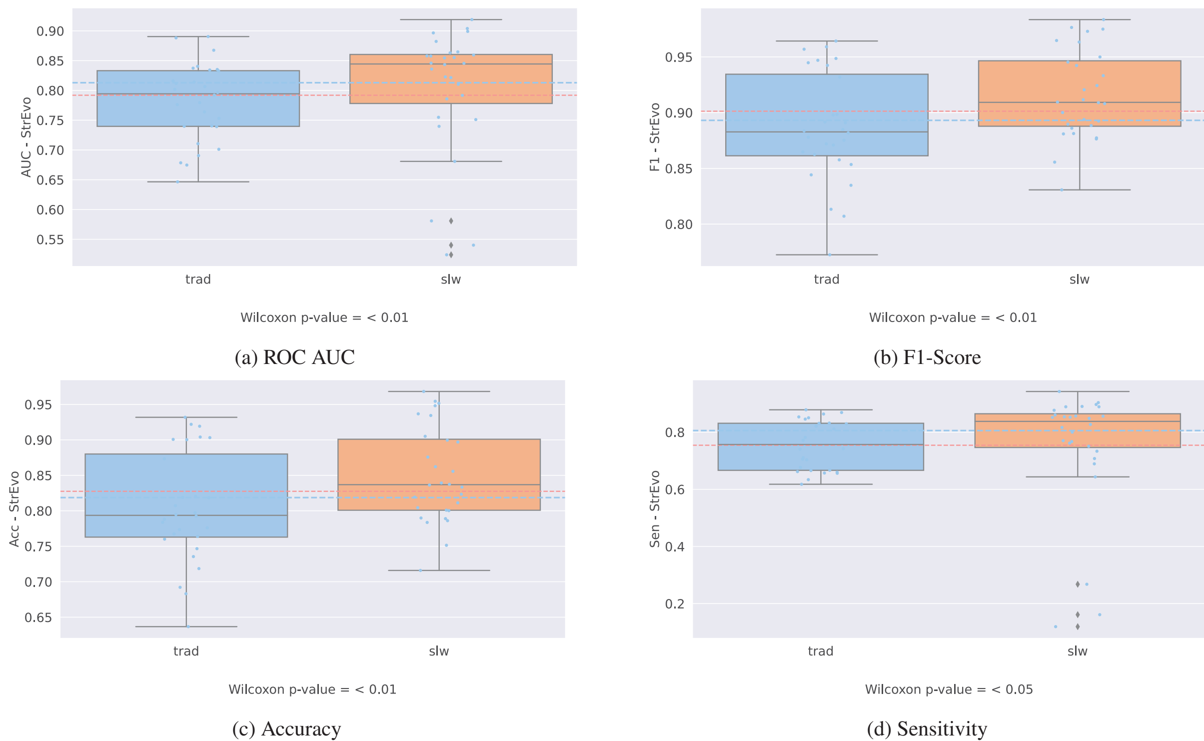


(d) Sensitivity

Figure 4.7: Comparing both approaches per indicator with StrEvo feature set

the traditional, reaching 60 (71.4%) of the best values, against 17 obtained by the traditional approach, and 7 ties.

If we consider all the indicators, algorithms, and all sets (336 cases), the SW approach produces the best results or equivalent ones in 260 cases (77.3%). To corroborate the findings we applied the Wilcoxon test [126] and calculated the p-value for all results of ROC AUC, F1-Score, Accuracy, and sensitivity performance indicators. For all indicators, there is a statistical difference in the distribution of SW approach and traditional approaches, with >95% of confidence (p-value=0.001), as Figure 4.8 shows. Conducting the same analysis for each kind of model obtained by adopting the three feature sets, we obtained similar results. The models with the SW approach perform better for all sets. These statistically significant differences highlight the advantages of the sliding window approach in accurately identifying code methods prone to changes, emphasizing its potential as a valuable approach in software maintenance and optimization efforts. The red dashed line indicates the mean and the blue the median. Regarding computational cost, the processing time of both approaches is similar. The biggest bottleneck is found in the data collection and pre-processing phase, but this is a problem that affects both approaches.



(a) ROC AUC

(b) F1-Score

(c) Accuracy

(d) Sensitivity

Figure 4.8: Statistical analysis comparing both approaches considering results of StrEvo set

Response to RQ4: For all the indicators analyzed, the models obtained by employing the SW approach overcome the models obtained with a traditional approach, in the great majority of the cases (83.13%). Moreover, the performance of the SW approach is better, independent of the feature set used. Then we can conclude that the adoption of the SW approach contributes to improving the performance of the models obtained by using all three feature sets evaluated.

Table 4.7: Comparing both approaches

	Str				StrEvo				Evo			
App	F1	Acc	Sen	AUC	F1	Acc	Sen	AUC	F1	Acc	Sen	AUC
Trad	3	3	7	4	5	5	6	3	6	6	4	0
SlW	23	23	17	24	21	23	22	24	17	18	21	27
Eq	2	2	4	0	2	0	0	1	5	4	3	1

4.5 RQ5 - WHICH FEATURES ARE THE MOST IMPORTANT FOR CHANGE-PRONE METHODS PREDICTION?

As mentioned in Chapter 3 (Section 3.5.4), to answer this RQ we employed MDI and Information Gain considering all the features. The values obtained with both techniques are in our repository [42]. For each system and technique, 3 ranks of features according to the best values were generated. They contain, respectively, the top-10 best features, top-5 and top-1. Next, we counted the number of times each metric appeared in these ranks, considering all systems, to create three general ranks. As a result, we generated the ranks of Table 4.8.

The first row of this table shows that FRCH (Frequency of Change), CountLine, CountInput and ATAF (Aggregated Change Normalized by Frequency Change) appear in the top-10 rank of all systems (7), according to MDI. According to the results of the Information Gain, no features appear in all the 7 subject systems in the top-10 rank. In the second row, the number of metrics that appear in 6 systems according to the Information Gain technique is greater than the number that appears according to MDI. This happens in almost all rows of the table.

Considering the three ranks, Information Gain pointed out as relevant 82 metrics, MDI 22. According to Information Gain, the metrics that stand out appearing in more than one rank are related to the count of declared, private, and public methods, as well as some metrics related to complexity.

Even though this greater number of metrics pointed out by Information Gain, there are some features that are important only for MDI, such as: line, CountLine, CountInput, CountOutput, CountLineComment, CountLineCodeExe, uniqueWordsQty and SumCyclomatic. On the other hand, the features pointed out by both methods are: i) structural: CoutLineCode, CountLineCodDec, AvgCyclomatic, AvgCyclomaticModified, rfc, modifiers, methodsInvokedQty, fanout, cbo, CountStmtDecl and anonymousClassesQty; and ii) evolution-based: BOM (BirthOfMethod), ATAF and FRCH. But it is interesting to observe that these last two, FRCH, and ATAF, stand out in the first positions in all ranks considering both techniques.

Response to RQ5: The most important features to predict change-prone methods are evolution-based, FRCH (Frequency of Change), and ATAF (Aggregated Change Normalized by Frequency Change), which appear in the first positions of the ranks generated by both techniques.

4.6 DISCUSSION

In this section, we discuss the implications of the findings obtained in the analysis of our RQs that may help developers in practice and also point out some research opportunities. These implications are presented as follows.

Table 4.8: Ranks with the most important features

Top10 features		
#Syst	Information Gain	MDI
7		FRCH, CountLine, CountInput, ATAF
6	innerClassesQty, TACH, PercentLackOfCohesion MaxInheritanceTree, MaxEssential, MaxCyclomaticStrict, MaxCyclomaticModified, MaxCyclomatic, CountDeclMethodPublic, CountDeclMethodProtected, CountDeclMethodPrivate, CountDeclMethodDefault, CountDeclMethodAll, CountDeclMethod, CountDeclInstanceVariable, CountDeclInstanceMethod, CountDeclFunction, CountDeclFile, CountDeclClassVariable, CountDeclClassMethod, CountDeclClass, CountClassDerived, CountClassCoupled, CountClassBase, CHO, CHD, AvgLineComment, AvgLineCode, AvgLineBlank, AvgLine, AvgEssential, AvgCyclomaticStrict,	line, CountOutput, CountLineCode AvgCyclomaticModified, AvgCyclomatic
5	maxNestedBlocksQty, mathOperationsQty, FRCH, ATAF, ACDF	uniqueWordsQty, CountLineCodeExe, BOM
4	parenthesizedExpsQty, parametersQty, logStatementsQty, hasJavaDoc, anonymousClassesQty, MaxNesting, CountLineBlank	
3	returnsQty, modifiers,loopQty, lambdasQty, WCH, WCD, SumCyclomaticStrict	SumCyclomatic
2	variablesQty, stringLiteralsQty, rfc, methodsInvokedQty, methodsInvokedLocalQty, fanout, cbo, WFR, SumCyclomaticModified, LCH, LCD, LCA, FCH, CSBS, CSB, BOM	CountLineCodeDecl
1	tryCatchQty, numbersQty, methodsInvokedIndirectLocalQty, loc, fanin, comparisonsQty, assignmentsQty, SumEssential CyclomaticModified, Cyclomatic, CountStmtDecl, CountPath, CountLineCodeDecl	rfc, modifiers, methodsInvokedQty fanout, cbo, CountStmtDecl, CountLineComment
Top5 features		
#Syst	Information Gain	MDI
6		FRCH, CountInput, ATAF
4	parenthesizedExpsQty, parametersQty,	CountOutput, anonymousClassesQty
3	lambdasQty, FRCH, ATAF	line, CountLine, BOM
2	maxNestedBlocksQty, mathOperationsQty, loopQty, innerClassesQty, hasJavaDoc, TACH, PercentLackOfCohesion, MaxNesting, MaxInheritanceTree, MaxEssential, MaxCyclomaticStrict, MaxCyclomaticModified, MaxCyclomatic CountLineBlank, CountDeclMethodPublic, CountDeclMethodProtected, CountDeclMethod-Private, CountDeclMethodDefault, CountDeclMethodAll, CountDeclMethod, CountDeclInstanceVariable, CountDeclInstanceMethod, CountDeclFunction, CountDeclFile, CountDeclClassVariable,CountDeclClassMethod, CountDeclClass, CountClassDerived, CountClassCoupled, CountClassBase, CHO, CHD, AvgLineComment, AvgLineCode, AvgLineBlank, AvgLine, AvgEssential, AvgCyclomaticStrict, AvgCyclomaticModified, AvgCyclomatic, ACDF	uniqueWordsQty
1	variablesQty, stringLiteralsQty, returnsQty, modifiers, methodsInvokedLocalQty, fanin, CountLineCode comparisonsQty, cbo,assignmentsQty, SumCyclomaticStrict, CyclomaticModified,CountStmtDecl, CountLineCodeDecl, BOM	CountLineCodeDecl,CountLineCode
Top1 features		
#Sys	Information Gain	MDI
3		FRCH
2	ATAF	ATAF
1	anonymousClassesQty, parenthesizedExpsQty, CountLineBlank, FRCH, maxNestedBlocksQty, parametersQty	BOM, CountInput

4.6.1 Prediction of change-prone methods

An initial concern of the work was whether the reduced amount of code of the methods in relation to the classes would have enough characteristics to predict the methods prone to change. The previous sections show that our methodology obtained results capable of predicting change-prone methods. Considering our best prediction results using the Random Forest algorithm, with a feature set composed of structural and evolution-based metrics reshaped with the sliding window approach we obtained a ROC AUC score average value of 0.82 for all systems evaluated, with similar or superior results for the other indicators. Six of the seven systems evaluated obtained a ROC AUC score greater than 0.84. This result represents an improvement over the state of the art [73] for the same problem.

A model that predicts 82% of the recommended methods on average may improve the development process and help a developer who needs to choose the next method to receive code maintenance. Anyway, the developer can still choose to reason on a list of methods selected with the recommendations made by the model. Considering the size of the systems, with classes containing an extensive number of methods, the purpose of this research is to help increase the code quality and the productivity of developers. It can be applied in a recommendation system for developers, for example.

There is a gap in developing new tools to the method level. Although some research indicates a preference for working at a more granular level such as the [84, 53] method, most tools work at the file level. Most tools work with code elements at the file level, which end up being easier to manipulate because they can work directly with the file system. Thus, we had to create and adapt tools for this research.

4.6.2 Choice of the algorithm

Random Forest presents the best performance for the change-prone methods prediction problem. The results obtained are similar to the results reported in the literature for software change prediction at the class level [105]. Thus, smarter algorithms lead our approach to producing better results. We also observe that the performance of resample techniques can vary according to the algorithms. Random Forest performed better with RUS and MLP obtained better results with ROS. This and other pre-processing techniques should be further investigated. Considering the number of feature sets (Table 4.2) chosen in the best results of all selected models from Table 4.1, MLP presented a balanced set of results. We can observe that MLP counts 11 cases for each of the three feature sets. The other algorithms got a majority of cases for the feature set Evo, Decision Tree with 16 cases out of 27, Logistic Regression with 12 out of 18, and Random Forest with 18 out of a total of 42. However, if we consider only the result with the maximum value of each system, shown in Table 4.4, the StrEvo feature set proves to be predominant. A research opportunity is to investigate other algorithms, for instance, Deep Learning neural networks, as they naturally have the power to analyze the state of variables in different versions, e.g. Long short-term memory (LSTM) networks can be used for speech recognition, and time series prediction, among others. For this, it would also be necessary to expand the datasets, since these networks demand a large amount of data.

4.6.3 Choice of sliding window size

The SW approach is applicable in the early stages of development to prevent future problems. According to the findings of RQ3, our approach does not require a long history to reach good performance; a time period of 4 releases seems to be enough, independently of the feature set and algorithms used. A more in-depth study needs to be carried out to verify if larger window values imply an improvement in the results, because the larger the window size, the greater the necessary history of the class, thus excluding classes added in more recent versions; this consequently reduces the training data and can generate bias in relation to older methods. However further studies should be conducted to better characterize the life of a method along the project and its relation with changes. It is expected that newborn methods are more change-prone, and otherwise for more mature methods. It would be interesting to test other window thresholds, to better explore the boundaries between smaller and larger windows.

4.6.4 The reshaping of data to a temporal dependent format

The use of the SW approach to reshape the datasets to a temporal dependent format seems to be the best option and can help developers, testers, and managers to concentrate effort on the correct methods that are more prone to change in the future. Our approach outperforms the traditional approach with a statistical difference in 89.3% of the cases, considering the ROC AUC score and 77.4% for all prediction performance indicators for the four ML algorithms, seven systems, and three feature sets. Another possible research opportunity is to investigate the limit of the ML models with the SW approach to generate models to predict the change-prone methods considering a number x of releases ahead.

4.6.5 Kinds of models adopting different feature sets

In the literature, there are many kinds of models which are classified according to the independent variables (features) adopted for change-prone class prediction. We derived our features from classes to apply to methods and they proved to be adequate for the task. We evaluated our

approach using different combinations of features. In terms of the highest prediction performance indicators, the combination of structural and evolution-based achieved the highest values with statistical significance for all indicators. Our findings complement the ones reported in the literature studies in showing how these metrics perform at the method level. Although studies do not show a consensus on the use of structural metrics [90, 144, 135], these metrics played an important role in improving the results of our models.

4.6.6 Features importance

In regard to the importance of the features to the models, we observe that evolutionary metrics play an important role, more than structural ones. We must highlight that features FRCH and ATAF have more power prediction than the others to identify change-prone methods. These results can help developers and researchers to create more robust and objective tools to predict change-prone methods. Additionally, there is a list of metrics that are also important for the prediction that we list in this work. In future work, the occurrences of these metrics can be analyzed in the different systems to correlate with the prediction performance.

4.7 THREATS TO VALIDITY

In this section, we present some threats to the validity of our results, according to the taxonomy of Wohlin et al. [140].

4.7.1 Construct validity

The main threat regarding the construct validity is related to the choice of features used. Other metrics could lead to other results. To mitigate this fact, we selected the features based on the most commonly found in the change prediction literature. Another threat is the fact that we have not excluded outliers. With the use of outliers, the models may present a bias that impairs their effectiveness. Additionally, we did not search for hyperparameters to tune the ML models due to the computational cost. The results are likely to improve significantly with this fine-tuning of the models. This search will be carried out in future work.

4.7.2 Internal validity

The threats in this category are related to the feature collection that involves a massive quantity of code elements of different systems. As the number of code elements is large, we cannot guarantee that we rule out unknown issues that may be happened during the automated process of data extraction. To mitigate this situation we debugged part of the execution of all applications to check a sample of the results obtained. The tools used may impact the results. To mitigate this we employed the tools CK, Understand, and ChangeDistiller also used in other works in literature [105, 24, 8]. Our dataset is very unbalanced and dealing with unbalanced datasets is a difficult task. We applied resampling techniques to deal with this unbalance and obtained significant improvement in the results. However, other techniques, e.g. to define weights to the less represented class of dependent variable, may be more appropriate and improve even more the results. The test data were not normalized to the same min-max values as the training and validation normalization. Therefore, the difference in scale between the trained model and the test data can lead to inaccuracy in the results.

4.7.3 Conclusion Validity

Our findings depend on the used indicators and the way we organize the results for analysis. We adopted common indicators to evaluate the algorithms and models and conducted statistical tests to mitigate this threat. We investigated feature importance by using two common techniques: Information Gain and MDI, but other techniques could be investigated, such as SHAP [91] that helps to explain the obtained models. The target data is being labeled using the ChangeDistiller and PerfoRT tools. Thus, as all conclusions are being drawn from the data collected by these tools, they depend on the correct functioning of these tools. ChangeDistiller is a more mature tool and is often used in the literature. PerfoRT is a new tool developed for this thesis and presents a major threat to the conclusion validity.

4.7.4 External validity

We analyze 7 systems with different application domains, sizes, and distinct developers. However, all systems are developed in Java. Therefore, it is not possible to generalize our results to contexts different from this. But the material in our repository can allow replicability and/or new evaluations. We created models by project, a single model that works in many projects is a good point to investigate in future work, as well as the use of the generated models for other contexts or languages.

4.8 CONCLUDING REMARKS

This chapter presented the experiments conducted to evaluate the prediction of change-prone methods with machine learning models and discusses the implications of the results obtained. The five RQs were formulated to evaluate if machine learning works on this problem and how it performs, to explore different feature sets and methods with the goal of improving its performance, and to identify the most important features. The experiments encompassed 7 systems containing 364 releases and more than 1 million instances of methods. We used four machine learning algorithms together with three feature sets, the original unbalanced dataset plus 6 resample techniques for balancing data. Moreover, we evaluate the use of the SW approach that considers the existence of a temporal dependency between training instances. The obtained models were compared by using four indicators.

The main results indicated that models obtained with the Random Forest algorithm, by using the combination of structural and evolution-based feature sets, random resample (RUS and ROS) techniques and the adoption of the sliding window approach led to the best general results, reaching mean ROC AUC score of 0.82. Evolution-based features demonstrated more prediction power, especially the metrics FRCH and ATAF identified as the most important features. A possible drawback to using methods scope could be the lack of sufficient characteristics to predict methods prone to change. However, the results indicated the opposite. In this sense, this work contributes to defining a methodology that addresses the challenges of creating models for predicting change-prone methods effectively in real-world scenarios. Change-prone methods prediction allows focusing on a low level of granularity and on more specific regions of the code to help the developers find parts to change with more accuracy, mainly in classes with a large number of methods. Also, classes contain many different characteristics of software attributes and some software behaviors are more granular and better captured at the method level. We highlight the practical applications of these prediction models in software maintenance, optimization, and resource allocation.

5 PERFORMANCE IMPACTFUL CHANGES PREDICTION

Building upon the methodologies and techniques discussed in previous chapters, this chapter focuses on analyzing the effectiveness and reliability of prediction models in identifying performance impactful method changes. Similar to the last chapter, the next five sections report the obtained results of the experiments designed to answer a specific research question. Then, in Section 5.6 we analyze the results and discuss the findings. Section 5.7 presents the threats to the validity of this study. Finally, Section 5.8 contains the concluding remarks of the chapter.

5.1 RQ1 - WHAT IS THE EFFECTIVENESS OF THE MACHINE LEARNING TECHNIQUES TO PREDICT PERFORMANCE IMPACTFUL METHOD CHANGES?

We evaluated the performance of models to predict performance impactful method changes for all algorithms, feature sets, and resample techniques for each target system. Table 5.1 presents the results of the indicators F1-Score, Accuracy, Sensitivity, and ROC AUC score of the models with the highest ROC AUC scores, i.e., the best models.

We observe that the F1-Score values range from 0.37 to 0.98, accuracy from 0.31 to 0.97, sensitivity from 0.35 to 0.99, and ROC AUC from 0.55 to 0.98. These values are highlighted in bold in the table. We also evaluated the prediction performance of the algorithms, feature sets, and systems for this research problem. Observing the model with the highest ROC AUC, with a value of 0.98, it uses the Random Forest algorithm, feature set StrEvo, and occurred in `commons-bcel` system. Considering all five systems, this model with Random Forest and StrEvo obtained a ROC AUC mean score of 0.77, an F1-Score mean value of 0.86, a mean accuracy of 0.79, and a Sensitivity value of 0.74, on average.

Table 5.1: Performance impactful method changes selected models according to AUC ROC score

	Decision Tree					**Logistic Regression**					**MLP**					**Random Forest**				
Features	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS
commons-bcel																				
Str	0.84	0.80	0.87	0.82	ENN	0.68	0.64	0.81	0.69	ADA	0.82	0.77	0.84	0.79	RUS	0.82	0.78	0.91	0.82	ENN
StrEvo	0.92	0.89	0.85	0.88	NONE	0.74	0.68	0.72	0.69	SMOTE	0.95	0.93	0.97	0.94	ADA	**0.98**	**0.97**	**0.99**	**0.98**	ROS
Evo	0.73	0.66	0.57	0.63	NONE	0.74	0.66	0.48	0.61	ENN	0.73	0.66	0.57	0.63	RUS	0.73	0.66	0.57	0.63	NONE
commons-csv																				
Str	0.76	0.66	0.64	0.65	SMOTE	0.64	0.54	0.77	0.63	ADA	0.78	0.68	0.68	0.68	ROS	0.83	0.75	0.66	0.72	RUS
StrEvo	0.89	0.83	0.78	0.81	TL	0.73	0.63	0.71	0.66	RUS	0.87	0.80	0.79	0.80	RUS	0.92	0.87	0.83	0.85	RUS
Evo	0.92	0.88	0.70	0.81	ROS	0.75	0.65	0.72	0.68	SMOTE	0.87	0.81	0.81	0.81	ADA	0.95	0.92	0.80	0.88	ADA
easymock																				
Str	0.84	0.75	0.85	0.79	NONE	0.80	0.69	0.52	0.62	SMOTE	0.84	0.75	0.58	0.68	ROS	0.84	0.75	0.80	0.77	ROS
StrEvo	0.83	0.73	0.67	0.70	ENN	0.79	0.67	0.53	0.61	RUS	0.91	0.85	0.38	0.64	ENN	0.85	0.76	0.71	0.73	ENN
Evo	0.84	0.75	0.85	0.79	NONE	0.80	0.69	0.53	0.62	NONE	0.83	0.73	0.61	0.68	ROS	0.84	0.75	0.80	0.77	ROS
jgit																				
Str	0.73	0.63	0.71	0.66	TL	0.67	0.56	0.63	0.59	ADA	0.80	0.70	0.52	0.63	ENN	0.80	0.70	0.58	0.65	ROS
StrEvo	0.78	0.69	0.63	0.66	ROS	0.70	0.60	0.69	0.63	SMOTE	0.79	0.69	0.60	0.66	ENN	0.73	0.63	0.70	0.66	RUS
Evo	0.67	0.58	0.76	0.65	ADA	0.61	0.52	0.81	0.63	NONE	0.59	0.51	0.81	0.62	ROS	0.65	0.56	0.79	0.65	RUS
openfire																				
Str	0.71	0.59	0.70	0.64	ENN	0.77	0.65	0.61	0.63	ADA	0.90	0.82	**0.35**	0.62	ENN	0.77	0.66	0.62	0.64	ADA
StrEvo	0.80	0.69	0.50	0.61	ROS	0.72	0.60	0.68	0.63	ADA	0.88	0.80	0.40	0.63	ENN	0.85	0.75	0.50	0.65	NONE
Evo	**0.37**	**0.31**	0.88	**0.55**	ENN	**0.37**	**0.31**	0.88	**0.55**	ENN	**0.37**	**0.31**	0.88	**0.55**	ENN	**0.37**	**0.31**	0.88	**0.55**	ENN

To compare the performance of the algorithms, we derived Table 5.6 that summarizes the number of times each algorithm reaches the best result (or equivalent to the best one) considering all systems. We can see in the columns of totals (T) of each algorithm that Random Forest shows the greatest number of models. For F1-Score there are 8 models on which Random Forest surpassed others with the same feature set. Similarly, there are 9 best models considering Accuracy, 7 for Sensitivity, and 11 for ROC AUC score, the values are in bold in the table. In Sensitivity Decision Tree tied with Random Forest. Considering the indicator ROC AUC score,

Decision Tree obtained 4 occurrences with feature set Str, while Random Forest 3. In the feature set StrEvo, Random Forest counted 4 and Decision Tree 1. In the Evo set they tied with 4 times each one. Logistic Regression and MLP had worse results, with the exception of Sensitivity for MLP in the Evo set, the case where it was the best in 4 cases, and presents the best general performance for this indicator. We can observe that when the feature set Evo is used, all the algorithms performed better.

Table 5.2: Number of times each algorithm reaches the best value for each indicator considering all systems

	Decision Tree				**Logistic Regression**				**MLP**				**Random Forest**			
Indicator	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T
F1-Score	2	1	4	7	0	0	2	2	3	2	1	6	3	2	3	**8**
Accuracy	2	1	4	7	0	0	2	2	3	2	2	7	3	2	4	**9**
Sensitivity	3	1	3	**7**	1	0	2	3	0	0	4	4	1	4	2	**7**
ROC AUC	4	1	4	9	0	0	1	1	0	0	2	2	3	4	4	**11**
Total	11	4	**15**	30	1	0	7	8	6	4	9	19	10	12	13	**35**

Legend: structural model (Str); structural and evolution model (StrEvo); evolution-based model (Evo); Total (T).

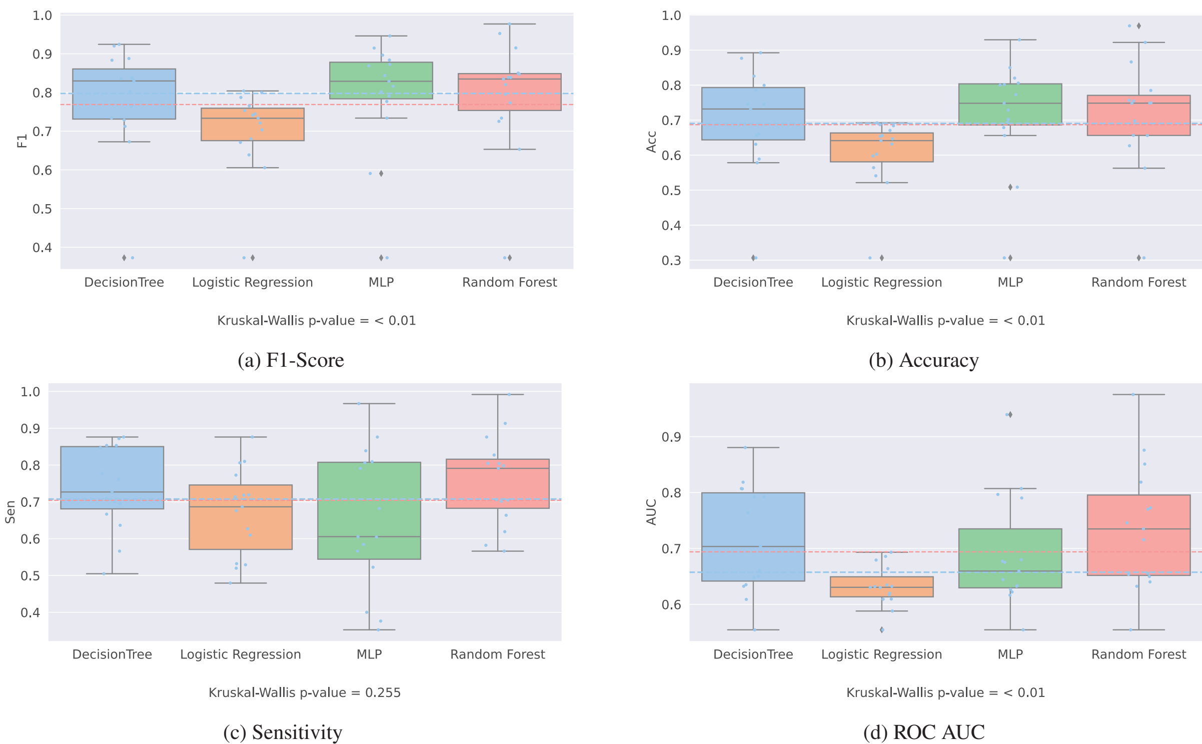
We also used the Kruskal-Wallis test and calculated the p-value considering models of Table 5.1. Figure 5.1 shows the obtained results per indicator. Random Forest has the best result in F1-Score, sensitivity, and ROC AUC score and MLP has the best accuracy. For F1-Score, Random Forest presents a difference from Logistic Regression with statistical significance, as we can observe p-value $<= 0.01$. The effect size between Random Forest, Decision Tree, and MLP has an insignificant magnitude. Regarding Accuracy, MLP has the highest value on average but Decision Tree and Random Forest have very close values, while Logistic Regression is visibly lower. For Sensitivity and ROC AUC, Random Forest has the highest values. Considering ROC AUC, Random Forest presents a difference from Logistic Regression with statistical significance, while p-value $<= 0.01$. In relation to the other algorithms, except Logistic Regression, which did not show statistical differences. The red dashed line indicates the mean and the blue the median.

The use of resampling techniques also improved the results in the scope of performance impactful method changes. Table 5.3 contains the number of times each resample technique is used in the best models according to ROC AUC derived from Table 5.1. We can observe that only seven results did not use any resample technique. The results are evenly distributed, except for TL which appears only 2. We can highlight ENN that appears 14 times and is the best resample technique from the results.

Table 5.3: Resample techniques with the best performance for PIC

Alg	**ADA**	**ENN**	**NONE**	**ROS**	**RUS**	**SMOTE**	**TL**
DT	1	4	4	3		1	2
LR	5	2	1	0	2	4	0
MLP	2	5	0	4	2	0	0
RF	2	3	2	4	4	0	0
Total	10	**14**	7	11	8	5	2

Figure 5.2 shows boxplots of each system for the four indicators with the p-value of the Kruskal-Wallis test. The best results in F1-Score, accuracy, and ROC AUC score were obtained for `commons-csv`, on average. In sensitivity, `commons-bcel` got the best result. The worst results are obtained for `jgit` in F1-Score and accuracy and for `openfire` in sensitivity and



(a) F1-Score

(b) Accuracy

(c) Sensitivity

(d) ROC AUC

Figure 5.1: Statistical analysis of the performance of the algorithms

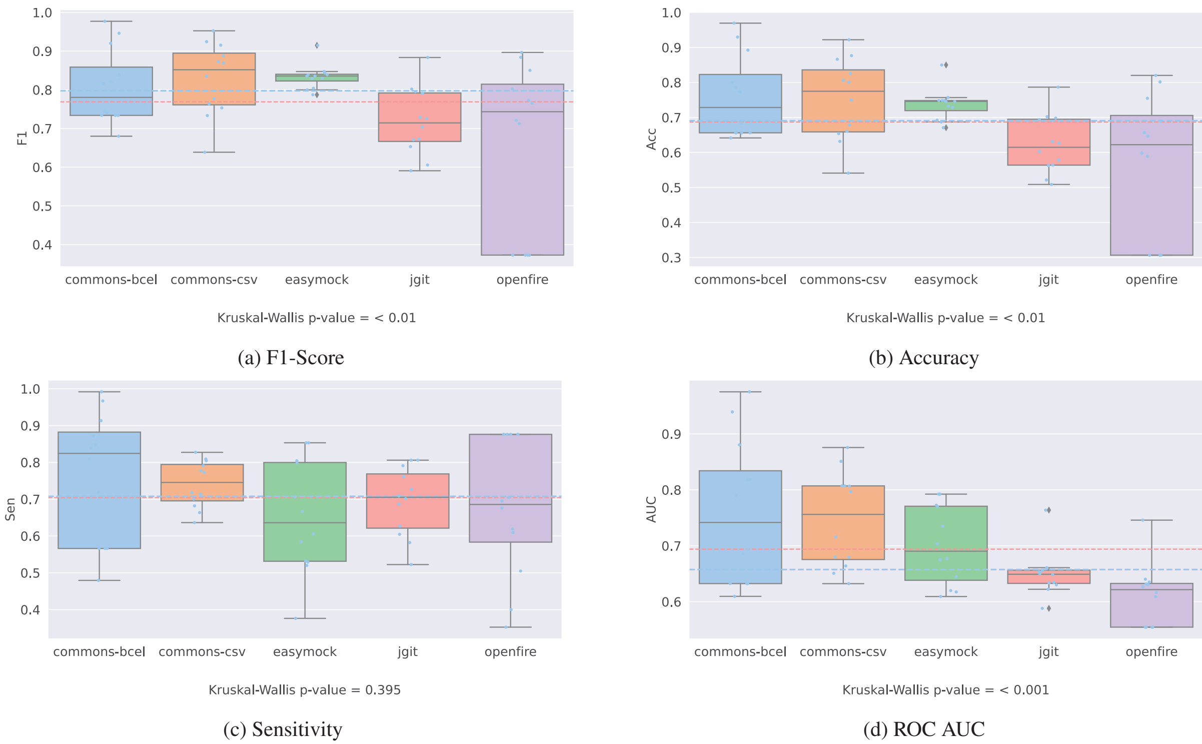
ROC AUC score. Except for sensitivity, there is a statistical difference for all the other indicators, with p-value <= 0.01. We can observe that the results variate between the analyzed systems.

Response to RQ1: We observed that the model obtained by Random Forest with StrEvo feature set achieved a mean ROC AUC score of 0.77. The analysis indicates that Random Forest presents the best general performance, followed by Decision Tree. Both algorithms present high ROC AUC and sensitivity values. On the other hand, Logistic Regression presents the worst general performance. In general, the ML techniques presented a lower ROC AUC rate than change-prone methods. The ENN undersample technique achieved the best performance and TL the worst.

5.2 RQ2 - WHICH FEATURE SETS YIELD THE MOST ACCURATE PREDICTION OF PERFORMANCE IMPACTFUL METHOD CHANGES?

Comparing the feature sets, we derived Table 5.4. It shows the number of times a model is considered the best for each indicator, using the respective feature set, and considering all systems.

We can observe that set StrEvo, which combines both kinds of features presents better results than the other feature sets, except for the sensitivity indicator. Figure 5.3 shows boxplot charts of the feature sets. We can observe that the feature set StrEvo achieved the best results for F1-Score, accuracy, and ROC AUC score on average, while Evo set obtained the highest sensitivity result. Considering the maximum value, StrEvo set wins in all indicators. The feature set Evo had the worst results in all indicators, except sensitivity. The results present statistical differences in F1-Score and accuracy.



(a) F1-Score

(b) Accuracy

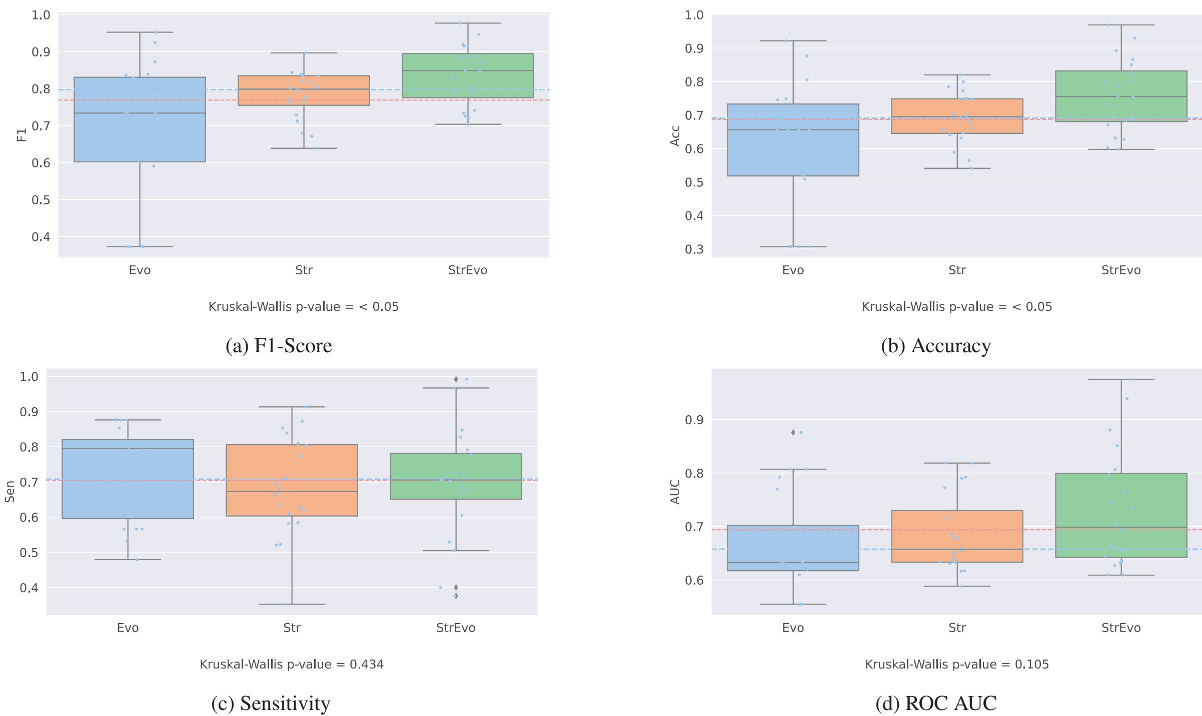
(c) Sensitivity

(d) ROC AUC

Figure 5.2: Statistical analysis of the algorithms' performance in each system

Table 5.4: Number of times each set of features reaches the best value for each indicator considering all systems

Indicator	Str	StrEvo	Evo
F1-Score	1	3	1
Accuracy	1	3	1
Sensitivity	1	2	3
AUC	1	3	1
Total	4	**11**	6



(a) F1-Score

(b) Accuracy

(c) Sensitivity

(d) ROC AUC

Figure 5.3: Statistical analysis regarding the performance of the feature set

Response to RQ2: The combination of both feature sets, structural and evolution-based (StrEvo), shows the best performance.

5.3 RQ3 - HOW DO DIFFERENT WINDOW SIZES IN THE SLIDING WINDOW APPROACH AFFECT THE PREDICTION PERFORMANCE?

Three window sizes for S (2, 3, and 4) were investigated with two statistical analyses, using the Kruskal-Wallis statistical test [77] and the ROC AUC score. The first analysis considers the three feature sets and can be seen in Figure 5.4. The boxplot chart shows that window size 3 obtained the highest ROC AUC on average, while size 4 obtained the lowest value between the three sizes evaluated. The statistical analysis shows a p-value of 0.523, which means that there is no statistical difference between the size of the windows. The second analysis considers each kind of model individually and is presented in Figure 5.5. For the set Str, the p-value is 0.469, for the feature set StrEvo is 0.813, and for Evo is 0.591. These results show that even considering each model individually, there is no statistical difference between window sizes for the prediction of performance impactful method changes. Although there was no statistical difference between the feature sets, we can observe in the charts that window 3 presents better results than the other sizes in the analysis with all feature sets, with structural (Str) and with structural and evolution-based combined (StrEvo), losing to window size 2 in the analysis with structural feature set only.

Response to RQ3: In a graphical analysis we observed that a window size equal to 3 presents the ROC AUC score with the highest values in the majority of the feature sets, although the sizes do not have statistically significant differences.

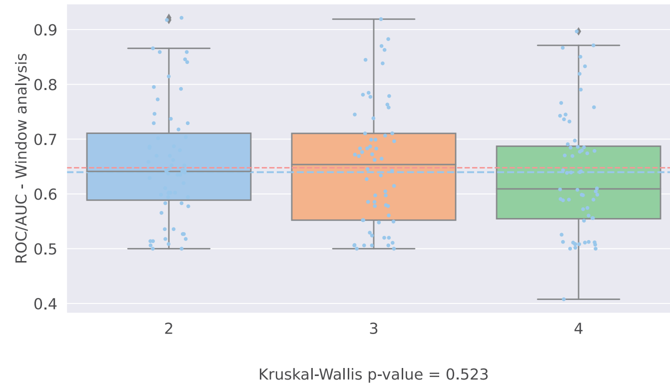
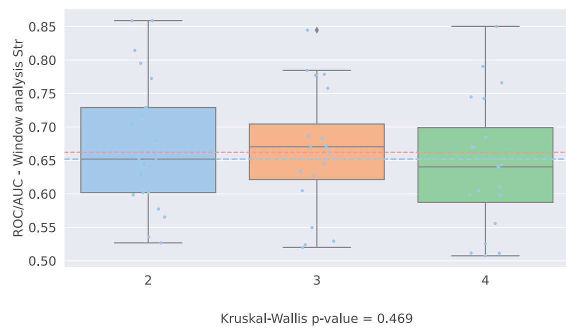
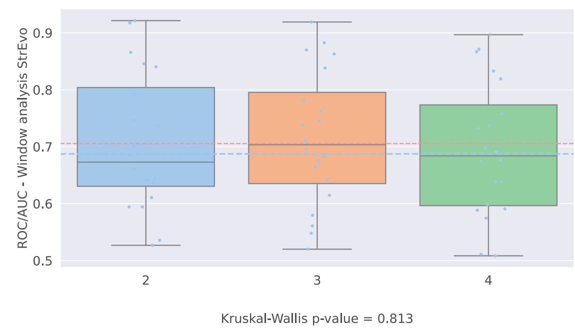


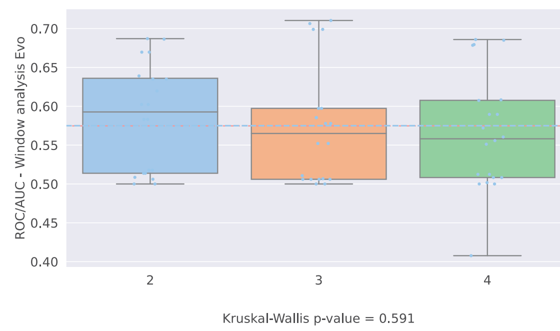
Figure 5.4: Window size statistical analysis for ROC AUC score on all feature sets



(a) Str



(b) StrEvo



(c) Evo

Figure 5.5: Window size statistical analysis of ROC AUC score on each feature set

5.4 RQ4 - HOW DOES THE PREDICTIVE EFFECTIVENESS OF MODELS OBTAINED WITH THE SLIDING WINDOW APPROACH COMPARE TO THE MODELS OBTAINED WITH A TRADITIONAL REPRESENTATION?

To answer this RQ, we use Table 5.5, which presents the results of the best models using $S = 3$, according to RQ3, for all three feature sets. This table presents, for each system and algorithm, the values of F1-Score, Accuracy, Sensitivity, and ROC AUC of the models obtained using the sliding window approach and the traditional one. The best result was obtained by the Random Forest algorithm, StrEvo feature set, and traditional approach with the values of 0.98 for F1-Score, 0.97 for Accuracy, 0.99 for Sensitivity, and 0.98 for ROC AUC. These results are highlighted in bold. The columns RS correspond to the resample technique used to get the model that reaches the highest value of the ROC AUC score.

Table 5.5: Results from the best obtained models for each algorithm and set of features

		Decision Tree					**Logistic Regression**					**MLP**					**Random Forest**				
Fs	Met	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS
												commons-bcel									
Str	trad	0.84	0.80	0.87	0.82	ENN	0.68	0.64	0.81	0.69	ADA	0.82	0.77	0.84	0.79	RUS	0.82	0.78	0.91	0.82	ENN
Str	slw	0.82	0.77	0.85	0.79	ENN	0.67	0.61	0.84	0.68	ADA	0.80	0.75	0.89	0.80	RUS	0.89	0.85	0.88	0.86	ENN
StrEvo	trad	0.92	0.89	0.85	0.88	NONE	0.74	0.68	0.72	0.69	SMOTE	0.95	0.93	0.97	0.94	ADA	**0.98**	**0.97**	**0.99**	**0.98**	ROS
StrEvo	slw	0.95	0.92	0.92	0.92	NONE	0.79	0.71	0.71	0.71	SMOTE	0.88	0.84	0.86	0.85	ADA	0.96	0.94	0.88	0.92	ROS
Evo	trad	0.73	0.66	0.57	0.63	NONE	0.74	0.66	0.48	0.61	ENN	0.73	0.66	0.57	0.63	RUS	0.73	0.66	0.57	0.63	NONE
Evo	slw	0.80	0.71	0.37	0.61	NONE	0.03	0.27	1.00	0.51	ENN	0.04	0.27	1.00	0.51	RUS	0.81	0.71	0.37	0.61	NONE
												commons-csv									
Str	trad	0.76	0.66	0.64	0.65	SMOTE	0.64	0.54	0.77	0.63	ADA	0.78	0.68	0.68	0.68	ROS	0.83	0.75	0.66	0.72	RUS
Str	slw	0.86	0.78	0.43	0.63	ENN	0.74	0.64	0.63	0.64	ADA	0.80	0.70	0.60	0.66	ROS	0.80	0.71	0.87	0.78	RUS
StrEvo	trad	0.89	0.83	0.78	0.81	TL	0.73	0.63	0.71	0.66	RUS	0.87	0.80	0.79	0.80	RUS	0.92	0.87	0.83	0.85	RUS
StrEvo	slw	0.87	0.80	0.97	0.87	NONE	0.82	0.73	0.75	0.74	RUS	0.85	0.77	0.76	0.76	RUS	0.90	0.84	0.95	0.88	RUS
Evo	trad	0.92	0.88	0.70	0.81	ROS	0.75	0.65	0.72	0.68	SMOTE	0.87	0.81	0.81	0.81	ADA	0.95	0.92	0.80	0.88	ADA
Evo	slw	0.76	0.65	0.76	0.70	NONE	0.76	0.66	0.79	0.71	SMOTE	0.75	0.65	0.79	0.71	ADA	0.76	0.65	0.76	0.70	ADA
												easymock									
Str	trad	0.84	0.75	0.85	0.79	NONE	0.80	0.69	0.52	0.62	SMOTE	0.84	0.75	0.58	0.68	ROS	0.84	0.75	0.80	0.77	ROS
Str	slw	0.92	0.86	0.49	0.70	ENN	0.86	0.77	0.51	0.66	SMOTE	0.89	0.82	0.58	0.72	ROS	0.90	0.84	0.79	0.81	ROS
StrEvo	trad	0.83	0.73	0.67	0.70	ENN	0.79	0.67	0.53	0.61	RUS	0.91	0.85	0.38	0.64	ENN	0.85	0.76	0.71	0.73	ENN
StrEvo	slw	0.88	0.81	0.67	0.75	NONE	0.87	0.78	0.54	0.68	RUS	0.91	0.84	0.43	0.66	ENN	0.90	0.83	0.74	0.79	ENN
Evo	trad	0.84	0.75	0.85	0.79	NONE	0.80	0.69	0.53	0.62	NONE	0.83	0.73	0.61	0.68	ROS	0.84	0.75	0.80	0.77	ROS
Evo	slw	0.12	0.18	0.96	0.51	NONE	0.17	0.21	0.94	0.52	NONE	0.17	0.21	0.94	0.52	ROS	0.12	0.18	0.96	0.51	ROS
												jgit									
Str	trad	0.73	0.63	0.71	0.66	TL	0.67	0.56	0.63	0.59	ADA	0.80	0.70	0.52	0.63	ENN	0.80	0.70	0.58	0.65	ROS
Str	slw	0.88	0.79	0.37	0.60	ENN	0.69	0.58	0.58	0.58	ADA	0.77	0.67	0.49	0.60	ENN	0.88	0.80	0.51	0.67	ROS
StrEvo	trad	0.78	0.69	0.63	0.66	ROS	0.70	0.60	0.69	0.63	SMOTE	0.79	0.69	0.60	0.66	ENN	0.73	0.63	0.70	0.66	RUS
StrEvo	slw	0.72	0.62	0.79	0.69	NONE	0.72	0.62	0.67	0.64	SMOTE	0.86	0.77	0.35	0.61	ENN	0.75	0.63	0.66	0.64	RUS
Evo	trad	0.67	0.58	0.76	0.65	ADA	0.61	0.52	0.81	0.63	NONE	0.59	0.51	0.81	0.62	ROS	0.65	0.56	0.79	0.65	RUS
Evo	slw	0.69	0.59	0.72	0.64	NONE	0.61	0.52	0.82	0.63	NONE	0.61	0.52	0.82	0.63	ROS	0.66	0.56	0.72	0.62	RUS
												openfire									
Str	trad	0.71	0.59	0.70	0.64	ENN	0.77	0.65	0.61	0.63	ADA	0.90	0.82	0.35	0.62	ENN	0.77	0.66	0.62	0.64	ADA
Str	slw	0.82	0.72	0.63	0.68	NONE	0.94	0.89	0.06	0.53	NONE	0.93	0.87	0.10	0.55	NONE	0.82	0.73	0.63	0.68	NONE
StrEvo	trad	0.80	0.69	0.50	0.61	ROS	0.72	0.60	0.68	0.63	ADA	0.88	0.80	0.40	0.63	ENN	0.85	0.75	0.50	0.65	NONE
StrEvo	slw	0.76	0.65	0.76	0.70	NONE	0.94	0.89	0.06	0.53	NONE	0.93	0.87	0.10	0.55	NONE	0.82	0.73	0.63	0.68	NONE
Evo	trad	0.37	0.31	0.88	0.55	ENN	0.37	0.31	0.88	0.55	ENN	0.37	0.31	0.88	0.55	ENN	0.37	0.31	0.88	0.55	ENN
Evo	slw	0.45	0.38	0.88	0.59	NONE	0.94	0.89	0.00	0.50	NONE	0.94	0.89	0.00	0.50	NONE	0.45	0.38	0.88	0.59	NONE

To a better visualization, we calculated, for both approaches, the mean values for all indicators, considering all algorithms and all sets from Table 5.5. The comparison is presented in Figure 5.6. Evaluating the ROC AUC score, the traditional approach surpassed the sliding window approach on all five systems. These results indicate that the prediction of performance impactful method changes is not improved with the sliding window approach. Considering F1-Score and Accuracy, the traditional approach was better in three systems (`commons-bcel`, `commons-csv`, and `easymock`), and the SW approach was better in two (`Openfire` and `jgit`). For the sensitivity indicator, the sliding window approach presented slightly better results on `commons-bcel`, `commons-csv`, and `easymock`.

To confirm the findings, we applied the Wilcoxon statistical test [126] and calculated the p-value for all results of ROC AUC, F1-Score, Accuracy, and Sensitivity. Figure 5.7 shows the statistical analysis comparing the approaches. Our analysis reveals in Figure 5.7(a) a notable difference with statistical significance between the traditional and sliding window approaches to ROC AUC score. The traditional approach outperformed the sliding window approach in

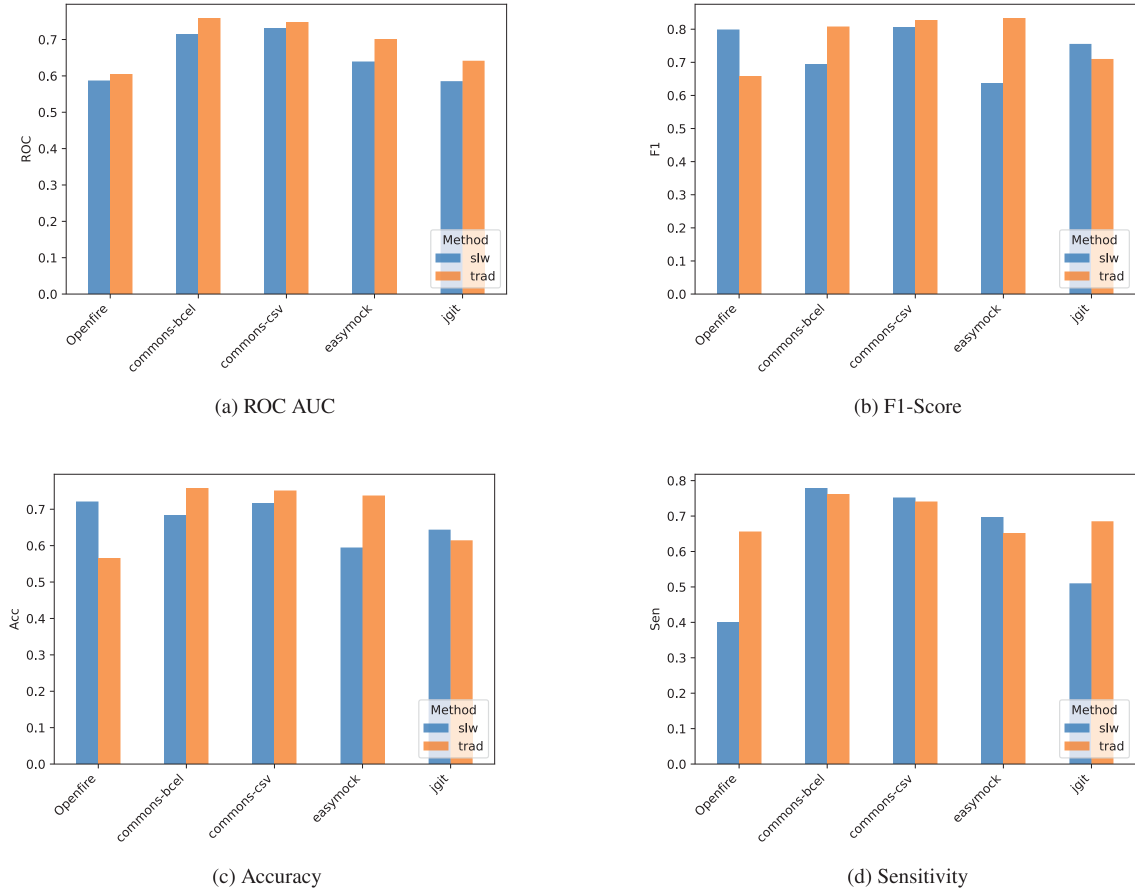


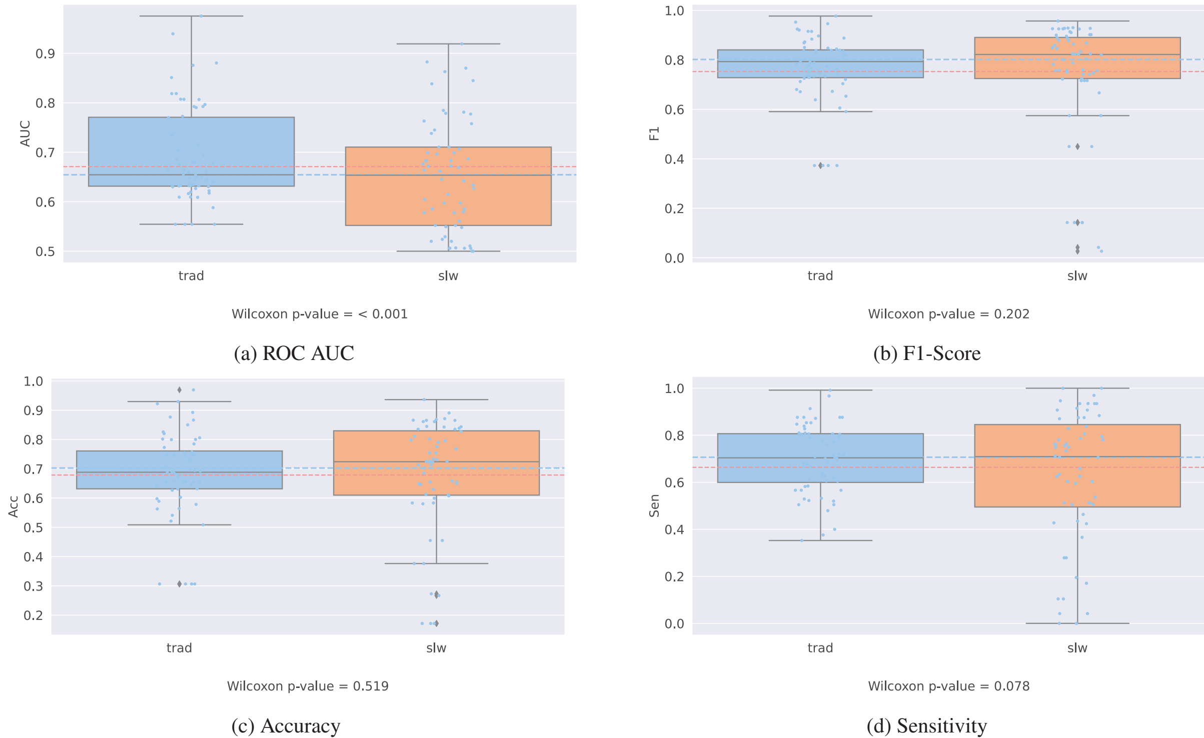
Figure 5.6: Comparing both approaches per indicator

Table 5.6: Number of times each algorithm reaches the best value for each indicator considering all systems

	Decision Tree				**Logistic Regression**				**MLP**				**Random Forest**			
Indicator	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T	Str	StrEvo	Evo	T
F1-Score	2	1	4	7	0	0	2	2	3	2	1	6	3	2	3	**8**
Accuracy	2	1	4	7	0	0	2	2	3	2	2	7	3	2	4	**9**
Sensitivity	3	1	3	**7**	1	0	2	3	0	0	4	4	1	4	2	**7**
ROC AUC	4	1	4	9	0	0	1	1	0	0	2	2	3	4	4	**11**
Total	11	4	**15**	30	1	0	7	8	6	4	9	19	10	12	13	**35**

Legend: structural model (Str); structural and evolution model (StrEvo); evolution-based model (Evo); Total (T).

terms of the ROC AUC score and sensitivity. On the other hand, considering F1-Score and Accuracy, the SW approach obtained better results, however, does not present a difference with statistical significance. Taking into account the results of ROC AUC, our main indicator, on which the traditional approach surpassed the SW approach for all systems and there is a significant difference between them, we conclude that the temporal patterns and changes over time do not impact prediction accuracy of performance impactful method changes.



(a) ROC AUC

(b) F1-Score

(c) Accuracy

(d) Sensitivity

Figure 5.7: Statistical analysis comparing both approaches considering results of all feature sets

> **Response to RQ4**: The use of the sliding window approach did not improve the performance of the prediction models.

5.5 RQ5 - WHICH FEATURES ARE THE MOST IMPORTANT FOR PERFORMANCE IMPACTFUL METHOD CHANGES PREDICTION?

We present the results of feature importance analysis using both Information Gain and MDI techniques to answer the RQ5. By comparing the results obtained from these two techniques, we aim to gain a comprehensive understanding of the importance of features in the data used for the performance impactful method changes prediction. For each system, 3 ranks of features according to the best values were generated by using Information Gain and MDI. These ranks contain, respectively, the top-10 best features, top-5, and top-1. Next, we counted the number of times each metric appeared in the ranks of each system, to create three general ranks. As a result, we generated the ranks of Table 5.7.

The first row of this table shows that innerClassesQty and MaxNesting appear in the top-10 ranking of all five systems, considering Information Gain. Additionally in this ranking, CountInput, CountLine, line, and uniqueWordsQty appear, according to MDI. According to

Information Gain, no features appear in the top-10 rank of all systems. In the second row, the number of metrics that appear in 4 systems according to the Information Gain technique is greater than the number that appears according to MDI. This happens in all the next rows of the table.

Considering the three ranks, Information Gain pointed out as relevant 99 metrics, MDI 19. According to Information Gain, the metrics that stand out appearing in more than one rank are related to the classes of the methods (anonymousClassQty, and innerClassQty), inheritance (MaxNesting), and internal structures (loopQty). Even though this greater number of metrics is pointed out by Information Gain, there are some features that are important only for MDI, such as line, CountLine, CountInput, and uniqueWordsQty.

The two feature importance techniques diverge in the results reporting different metrics as the most important. This reinforces the difficulty of predicting changes that impact performance. An important aspect that we observe is that structural metrics appeared as more important features than evolution-based metrics in more systems evaluated.

Table 5.7: Ranks with the most important features

Top10 features

#Syst	Information Gain	MDI
5	innerClassesQty, MaxNesting	CountInput, CountLine, line, uniqueWordsQty
4	ACDF, anonymousClassesQty, AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, CountClassBase, CountClassCoupled, CountClassDerived, CountDeclClass, CountDeclClassMethod, CountDeclClassVariable, CountDeclFile, CountDeclFunction, CountDeclInstanceMethod, CountDeclInstanceVariable, CountDeclMethod, CountDeclMethodAll, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethodProtected, CountDeclMethodPublic, CountLineBlank, CSB, CSBS, FCH, hasJavaDoc, LCA, LCD, LCH, loopQty, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxEssential, MaxInheritanceTree, maxNestedBlocksQty, PercentLackOfCohesion, RatioCommentToCode, WCD, WCH, WFR	cbo, cboModified
3	ATAF, CHD, CHO, comparisonsQty, CountLineCodeDecl, Essential, fanout, FRCH, logStatementsQty, methodsInvokedLocalQty, parametersQty, parenthesizedExpsQty, TACH, tryCatchQty	fanin
2	BOM, CountStmtDecl, CyclomaticModified, lambdasQty, mathOperationsQty, methodsInvokedIndirectLocalQty, modifiers, numbersQty, returnsQty, stringLiteralsQty, SumCyclomaticStrict, SumEssential, variablesQty, wmc	ATAF, BOM, CountOutput, FRCH, modifiers, stringLiteralsQty, SumCyclomatic
1	assignmentsQty, cbo, cboModified, CountInput, CountLine, CountLineCode, CountLineCodeExe, CountLine-Comment, CountOutput, CountPath, CountSemicolon, CountStmt, CountStmtExe, Cyclomatic, CyclomaticStrict, methodsInvokedQty, rfc, SumCyclomatic, SumCyclomaticModified, uniqueWordsQty	CountLineComment, hasJavaDoc, numbersQty, returnsQty, rfc

Top5 features

#Syst	Information Gain	MDI
5		line, uniqueWordsQty
4	loopQty, MaxNesting	
3	anonymousClassesQty, innerClassesQty, logStatementsQty, parenthesizedExpsQty	CountInput
2	ACDF, AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, BOM,comparisonsQty, CountClassBase, CountClassCoupled, CountClassDerived, CountDeclClass, Count-DeclClassMethod, CountDeclClassVariable, CountDeclFile, CountDeclFunction, CountDeclInstanceMethod, CountDeclInstance-Variable, CountDeclMethod, CountDeclMethodAll, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethod-Protected, CountDeclMethodPublic, CountStmtDecl, CSB, CSBS, CyclomaticModified,fanout, FCH, hasJavaDoc, lambdasQty, LCA, LCD, LCH, mathOperationsQty, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxEssential, MaxInheritanceTree, maxNestedBlocksQty, PercentLackOfCohesion, RatioCommentToCode, SumCyclomaticStrict, WCD, WCH, WFR	BOM, cbo, fanin
1	variablesQty, stringLiteralsQty, returnsQty, modifiers, methodsInvokedLocalQty, fanin, CountLineCode cbo, CHD, CHO, CountInput, CountLine, CountLineCode, CountLineCodeExe, CountLineComment, CountStmt, CountStmtExe, Cyclomatic, Essential, loc, modifiers, parametersQty, returnsQty, stringLiteralsQty, SumCyclomatic, SumCyclomaticModified, TACH, uniqueWordsQty, variablesQty	ATAF, cboModified, FRCH, hasJavaDoc, numbersQty, SumCyclomatic

Top1 features

#Sys	Information Gain	MDI
4		line
1	anonymousClassesQty, CountLineCodeExe, CountStmt, CountStmtExe, comparisonsQty, lambdasQty, loopQty, variablesQty, WCD, WCH	BOM

Response to RQ5: The most important features to predict change-prone methods that affect performance are structural, especially related to method classes and internal details of the methods like innerClassesQty, MaxNesting, loopQty, CountInput, and uniqueWordsQty.

5.6 DISCUSSION

This section discusses the key points of the results obtained in response to our RQs about performance impactful method changes prediction and also points out some possible future research. The points are presented as follows.

5.6.1 Predicting performance impactful method changes

Considering one of the best models with the Random Forest algorithm, structural and evolution-based (StrEvo) feature set, and the traditional dataset format, we obtained an average greater or equal to 0.75 in all prediction performance indicators. The ROC AUC score presents the mean value of 0.77 for the Model obtained by the Random Forest algorithm, StrEvo feature set, and traditional dataset format for the five target systems. The performance of the best predictors is comparable with the results obtained by change-prone class prediction results from the literature [105]. Our results enable developers to identify candidate performance impactful method changes in their systems that may significantly affect their execution time when changed. Developers can be notified of these impacts and proactively investigate them to search for bugs before being committed to production. Also, they can develop performance regression tests focusing on these methods to check that they will not present problems during the execution of the system. Software performance is an important quality attribute that is difficult to deal with. It can be time-consuming and resource-intensive and generates a huge quantity of data to be analyzed. Performance bugs are difficult to localize in code, often detected only during runtime, reported manually, hard to reproduce, and developers have to spend more time discussing to find solutions [119, 143]. Using predictive models for this goal, based on static metrics, is a good alternative because they do not need to spend time and resources generating load, measuring, and analyzing metrics, and can be integrated into linters, IDEs, or continuous integration tools.

5.6.2 Results of the sliding window approach

Unlike the study of change-prone methods prediction, the sliding window approach does not improve the results significantly. Starting with the window size analysis, the results of sizes 2, 3, and 4 obtained results without statistically significant differences. As any size stood out, we chose the one with the highest value in the boxplot chart. The results with the ROC AUC score show a lower performance of the sliding window approach in all systems with statistical significance. This evidence indicates that this kind of data does not have a historical dependence. One example to explain this behavior better is the case in which the developer adds a loop inside another loop, changing the time complexity from linear to quadratic. In the next change of this method, there is no tendency to insert a new loop or remove the existing loop and make a new change that affects the performance of the software. More investigations with larger window sizes and more data need to be carried out to confirm this phenomenon.

5.6.3 Features importance

Differently from the study for change-prone methods prediction, the prediction of performance impactful method changes has the most discriminant features in the structural set. Inner classes are classes declared entirely within another class or an interface and have strong prediction power. We can observe that loopQty is considered one of the most important features since actually loops have a direct impact on performance. The evolution-based considered most important metrics for predicting change-prone methods ATAF, and FRCH also appear in the list of performance impactful method changes prediction, but in a smaller number of systems. Complexity metrics also are important features. We can observe the metrics avgCyclomatic, avgCyclomaticModified, avgCyclomaticStrict, and avgEssential appearing in the Top 10 and Top 5 lists.

5.6.4 Comparison Between Change-Prone Method Prediction and Performance Impactful Method Changes

We compare the results of change-prone method changes and performance impactful method changes in Table 5.8. We can observe that there are some differences and similarities between the two experiments. First, the change-prone methods prediction presented a higher ROC AUC rate than performance impactful methods change. The former obtained 0.81 for the traditional approach and 0.82 for the SW approach, while the latter obtained 0.77 and 0.78 for the both approaches, respectively. The maximum ROC AUC rate was obtained with MLP in the change-prone method prediction, while for performance impactful methods change reached the maximum value with Random Forest and Decision Tree algorithms from traditional and SW approaches, respectively. On average, the best ROC AUC rate was obtained with the Random Forest algorithm in all results. The best resample technique for change-prone method prediction is RUS and for performance-impactful methods change is ENN. The feature set with the best results for both experiments is StrEvo. The window size with the highest results for the first experiment is 4 and for the second experiment is 3. Lastly, the most important features of the first experiment are evolution-based, and of the second are structural.

Table 5.8: Comparison Between Change-Prone Method Prediction and Performance Impactful Method Changes

Approach	Metric	CPMP	PIMC
Traditional	Highest AUC	0.81	0.77
	Best Algorithm (Max)	MLP	Random Forest
	Best Algorithm (Avg)	Random Forest	Random Forest
	Best Resample Technique	RUS	ENN
	Feature Set	StrEvo	StrEvo
Sliding Window	Window Size	4	3
	Highest AUC	0.82	0.78
	Best Algorithm (Max)	MLP	Decision Tree
	Best Algorithm (Avg)	Random Forest	Random Forest
	Important Features	Evolution-based (ATAF, FRCH)	Structural (LoopQty, Complexity)

Legend: Change-Prone Method Prediction (CPMP); PerformanceImpactful Method Changes (PIMC).

5.7 THREATS TO VALIDITY

In this section, we present some threats to the validity of the results related to the investigation of the use of machine learning models to predict performance impactful method changes.

5.7.1 Construct validity

The main threat regarding the construct validity is also related to the choice of features used. Other metrics could lead to other results, for example, CPU and memory usage instead of execution time. To mitigate this fact, we selected the features based on the most commonly found in the literature. Another threat is the fact that we have not excluded outliers. With the use of outliers, the models may present a bias that impairs their effectiveness.

5.7.2 Internal validity

The threats in this category are also related to the feature collection that involves a massive quantity of code elements of different systems. As the number of code elements is large, we cannot guarantee that we rule out unknown issues that may happen during the automated process of data extraction. To mitigate this situation we debugged part of the execution of all applications to check a sample of the results obtained. We debugged the data extraction of change-prone and performance data. The threats involving the used tools also may impact these results. As already explained, we employed the tools CK, Understand, and ChangeDistiller also used in other works in literature [105, 24, 8] to mitigate this threat. When evaluating software performance, it is essential to consider various factors such as system configurations, workload variations, and environmental conditions. Conducting a limited number of experiments may not adequately capture the variability and complexity of real-world scenarios. We used PerfoRT to collect the execution time of the methods. The tool executed all test cases available on the subject systems and the PerfoRT-Tracer collected the performance metrics on the executed methods. We executed the measurements of each test case 5 times to deal with the high variability of performance metrics, following the work of Alcocer et al. [4]. This number of executions may not be sufficient to deal with the variability of the times measured and lead to some inaccurate performance data. Our dataset is very unbalanced and dealing with unbalanced datasets is a difficult task. We applied resampling techniques to deal with this unbalance and obtained significant improvement in the results. However, other techniques, e.g. to define weights to the less represented class of dependent variable, may be more appropriate and improve even more the results. The test data were not normalized to the same min-max values as the training and validation normalization. Therefore, the difference in scale between the trained model and the test data can lead to inaccuracy in the results.

5.7.3 Conclusion Validity

Our findings depend on the used indicators and the way we organize the results for analysis. We adopted common indicators to evaluate the algorithms and models and conducted statistical tests to mitigate this threat. The execution time of the methods depends on the test coverage of the systems. We did not evaluate the test coverage of the methods used in this research. Thus, if the test coverage is narrow, it may represent a threat to the conclusion of the experiments of the systems. The number of evaluated systems may be small to generalize the conclusions. We tried to measure the performance of more target systems, but we could not run them without errors. That is the cause of the limited number of systems. We investigated feature importance by using two common techniques: Information Gain and MDI, but other techniques could be investigated, such as SHAP [91] that helps to explain the obtained models.

5.7.4 External validity

We analyze 5 systems with different application domains, sizes, and distinct developers. However, all systems are developed in Java. Therefore, it is not possible to generalize our results to contexts different from this. But the material in our repository [42] can allow replicability and/or new evaluations. We created models by project, a single model that works in many projects is a good point to investigate in future work, as well as the use of the generated models for other contexts or languages.

5.8 CONCLUDING REMARKS

In this study, we presented the results of the experiments performed to investigate the effectiveness of machine learning models with feature sets composed of structural and evolution-based metrics to predict performance impactful method changes. We conclude that machine learning algorithms are effective and might indeed support developers in making faster and more educated decisions on identifying change-prone methods that may impact software performance in Java projects. Random Forest models outperform other ML models and the combination of structural and evolution-based metrics had better results instead of the individual sets.

The investigation into the effects of the temporal representation of the datasets concluded that the SW approach does not improve the prediction performance significantly. In terms of the ROC AUC score, the traditional approach presented better results. No window size showed better results than the others, the results indicate that they have insignificant differences.

Furthermore, analyzed the importance of the features used by the models. In this analysis, we list 99 relevant features evaluated as important by the Information Gain technique and 19 by MDI. Structural metrics related to the method classes and internal details of the methods like innerClassesQty, MaxNesting, loopQty, CountInput, and uniqueWordsQty are identified for a greater number of systems with the Information Gain technique. We also observed that the evolution-based metrics BOM, ATAF, and FRCH also appeared but on a smaller number of systems.

6 FINAL REMARKS AND FUTURE WORK

In this thesis, we have explored the use of machine learning models to predict change-prone methods and to predict performance impactful method changes. In addition, we evaluated the effects of using the Sliding Window approach [36] to represent temporal dependence between instances of the methods in the prediction models. We also evaluated the importance of the used features from structural and evolution-based metrics, with the aim of finding the subset of predictors most relevant. By conducting a comprehensive study encompassing the related work review, methodology description, results, and discussion, we have made significant contributions to the field of software change prediction. This chapter serves as a culmination of our research, revisiting the contributions, presenting published papers, and insights for future work.

6.1 REVISITING OUR CONTRIBUTIONS

The main contributions of this study are summarized in this section. We present new research on software change prediction. To accomplish this, we developed a new performance regression tool, models that reduced the scope of change-proneness often applied from classes to methods, predicted performance impactful method changes, evaluated the use of the sliding window approach to reshape the data in a temporal dependent format, and analyzed the importance of features for the predictions.

From existing research on software change prediction, we developed the idea that the class scope, commonly used in existing research, is too broad. Considering that there are classes with hundreds of methods and thousands of lines, method-level prediction can improve the accuracy of locating the code parts to be modified. Thus, we implemented and evaluated the performance of ML models to predict change-prone methods in seven systems (see Chapter 4). The performance of machine learning models in predicting change-prone methods has demonstrated good results and effectively can be used in the prediction of change-prone methods. The experimental results indicate achieved a 0.82 ROC AUC, surpassing state-of-the-art results for the same problem [73]. Through rigorous evaluation and analysis, it has been observed that supervised machine learning models can effectively capture patterns and dependencies within software systems, enabling accurate predictions of code methods prone to changes.

We compare the performance of the traditional approach with the sliding window approach in predicting change-prone methods. Through rigorous evaluation measures and statistical analyses, we demonstrate a statistically significant difference between the two approaches. The superiority of the sliding window approach, with its ability to capture temporal patterns and changes over time, highlights its potential for accurately identifying change-prone methods.

We provide a comprehensive analysis of different feature sets for predicting change-prone methods. By evaluating a wide range of features derived from structural and evolution-based metrics we observed that evolution-based metrics have more predictive capacity than structural metrics. The ATAF and FRCH metrics stood out from the others. However, there is an extensive list of metrics that are also important. The good results of the temporal dependency representation corroborate with the identification of evolution-based metrics as relevant features.

Performance is a very important software quality attribute. We introduced the concept of performance impactful method changes. In simple words, the idea is to predict change-prone methods that, after being changed, can impact significant variations in system performance. This idea contributes to creating proactive systems that warn developers that a suggested method for

change may introduce performance variation. Thus, preventive actions can be taken to alleviate the performance degradation of the developed systems. We implemented and evaluated the performance of ML models to predict performance impactful method changes in five systems (see Chapter 5). The performance of machine learning models has demonstrated promising results and holds great potential to be used in the prediction of performance impactful method changes and be evaluated for other purposes of changes like security, or robustness. Through rigorous evaluation and analysis, it has been observed that supervised machine learning models can effectively capture patterns and dependencies within software systems, enabling accurate predictions of code methods prone to changes.

The prediction of performance impactful method changes does not seem to be influenced by the temporal dependence between the instances of methods. We can observe it in the results that indicate a worse performance with the sliding window approach. Also, the analysis of the importance of features also reinforces this perception because structural metrics seem to have more discriminant power than evolution-based metrics.

Overall, this study's contributions provide practical guidance for software engineering practitioners in building robust prediction models for software change. By identifying the most informative feature sets and showcasing the advantages of the sliding window approach, and identifying the purpose of the recommended methods to receive future changes this research offers actionable insights for improving software maintenance, performance optimization, and resource allocation in software development projects.

6.2 PUBLICATIONS

The research presented in this thesis has resulted in significant contributions to the fields of software quality and maintenance, predicting change-prone methods, and their impact on software performance. These contributions have been recognized and validated through the publication of several papers in prestigious conferences and collaborations with other research groups. The publications are listed as follows.

- Colanzi, T.E. and Assunção, W.K.G. and Vergilio, S.R. and Farah, P.R. and Guizzo, G. A Review of Ten Years of the Symposium on Search-Based Software Engineering. Proceedings of the 11th International Symposium on Search-Based Software Engineering (SSBSE). Springer, 2019 [29].

- Silva, H.N. and Farah, P.R. and Mendonça, W.D.F. and Vergilio, S.R. Assessing Android Test Data Generation Tools via Mutation Testing. Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing (SAST). ACM, 2019 [33].

- Colanzi, T.E. and Assunção, W.K.G. and Vergilio, S.R. and Farah, P.R. and Guizzo, G. The Symposium on Search-Based Software Engineering: Past, Present, and Future. Information and Software Technology (IST). Elsevier, 2020 [30].

- Soares, V. and Oliveira, A. and Pereira, J. A. and Bibiano, A.C. and Garcia, A.F. and Farah, P.R. and Vergilio, S.R. and Schots, M. and Silva, C. and Coutinho, D. and Oliveira, D. and Uchôa, A.G. On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns. Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES). ACM, 2020 [131].

- Farah, P.R. and Mariani, T. and Roza, E. A. and Silva, R.C. and Vergilio, S.R. Unsupervised Learning for Refactoring Pattern Detection. Proceedings of the IEEE Congress on Evolutionary Computation (CEC). IEEE, 2021 [43].

- Silva, R. C. and Farah, P.R. and Vergilio, S. R. Machine Learning for Change-Prone Class Prediction: A History-Based Approach. Proceedings of the XXXVI Brazilian Symposium on Software Engineering (SBES). ACM, 2022 [130].
- Farah, P.R. and Vergilio, S. R.PerfoRT: A Tool for Software Performance Regression. In Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion) [45].
- Farah, P.R. and Silva, R. C. and Vergilio, S. R. An Assessment of Machine Learning Algorithms and Models for Prediction of Change-Prone Java Methods. Proceedings of the 37th Brazilian Symposium on Software Engineering (SBES 2023) [44].

6.3 FUTURE WORK

In future work, we intend to study other aspects of change-prone method prediction as other conventional machine learning and deep learning algorithms, different types of features, more preprocessing techniques, and use other performance metrics as targets. We also want to analyze the importance of the features of each system separately and use SHAP technique to help explain the obtained models. As we created the models by each system, we intend to create only one model for all systems and investigate if it generalizes more and obtains better results.

We want to conduct more studies to better characterize the life cycle of the methods along the project and its relation with changes and extend the size of the sliding window approach to forecast a longer horizon on change-proneness types to explore the boundaries between smaller and larger window sizes.

We want to investigate other non-functional attributes like, for example, robustness and security and create models to predict and classify these types of change-prone methods. Also, other aspects may be studied, like developer or company preferences about changes or change behavior as input features, and prioritization of change-prone methods, among others.

REFERENCES

[1] R. Abbas, F. A. Albalooshi, and M. Hammad. Software change proneness prediction using machine learning. In *2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT)*, pages 1–7. IEEE, 2020.

[2] A. Agrawal and R. K. Singh. Empirical validation of OO metrics and machine learning algorithms for software change proneness prediction. In *Towards Extensible and Adaptable Methods in Computing*, pages 69–84. Springer, 2018.

[3] M. Al-Khiaty, R. Abdel-Aal, and M. Elish. Abductive network ensembles for improved prediction of future change-prone classes in object-oriented software. *International Arab Journal of Information Technology (IAJIT)*, 14(6), 2017.

[4] J. P. S. Alcocer and A. Bergel. Tracking down performance variation against source code evolution. *SIGPLAN Not.*, 51(2):129–139, oct 2015.

[5] C. Alhawi and A. Abdilrahim. *Studying the Relation Between Change- and Fault-proneness - Are Change-prone Classes More Fault-prone, and Vice-versa?* Bachelor's thesis, Linnaeus University, 2020.

[6] D. Alshoaibi, I. Chaabane, K. Hannigan, A. Ouni, and M. W. Mkaouer. On the detection of performance regression introducing code changes: Experience from the git project. In *2022 IEEE 29th Annual Software Technology Conference (STC)*, pages 206–217, 2022.

[7] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara. Empirical analysis of change-proneness in methods having local variables with long names and comments. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–4, 2015.

[8] M. Aniche, E. Maziero, R. Durelli, and V. H. S. Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE TSE*, 48(4):1432–1450, 2022.

[9] E. Arisholm, L. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[10] D. Azar. A genetic algorithm for improving accuracy of software quality predictive models: a search-based software engineering approach. *International Journal of Computational Intelligence and Applications*, 9(02):125–136, 2010.

[11] A. Bansal. Empirical analysis of search based algorithms to identify change prone classes of open source software. *Computer Languages, Systems & Structures*, 47:211–231, 2017.

[12] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[13] F. Bantelay, M. B. Zanjani, and H. Kagdi. Comparing and combining evolutionary couplings from interactions and commits. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 311–320, 2013.

[14] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, 6(1):20–29, jun 2004.

[15] R. Bell, T. Ostrand, and E. Weyuker. The limited impact of individual developer data on software defect prediction. *Empirical Software Engineer*, 18:478–505, 2013.

[16] J. Bieman, G. Straw, H. Wang, P. Munger, and R. Alexander. Design patterns and change proneness: an examination of five evolving systems. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*, pages 40–49, 2003.

[17] W. Brown, R. Malveau, W. Brown, H. I. McCormick, and T. Mowbray. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[18] J. Brownlee. *Data Preparation for Machine Learning: Data Cleaning, Feature Selection, and Data Transforms in Python*. Machine Learning Mastery, 2020.

[19] A. E. Burhandenny, H. Aman, and M. Kawahara. Change-prone java method prediction by focusing on individual differences in comment density. *IEICE Trans. on Information and Systems*, 100(5):1128–1131, 2017.

[20] G. Catolino and F. Ferrucci. Ensemble techniques for software change prediction: A preliminary investigation. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 25–30. IEEE, 2018.

[21] G. Catolino and F. Ferrucci. An extensive evaluation of ensemble techniques for software change prediction. *Journal of Software: Evolution and Process*, 31(9):e2156, 2019.

[22] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman. Developer-related factors in change prediction: An empirical assessment. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 186–195, 2017.

[23] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman. Enhancing change prediction models using developer-related factors. *Journal of Systems and Software*, 143:14–28, 2018.

[24] G. Catolino, F. Palomba, F. A. Fontana, A. De Lucia, Z. Andy, and F. Ferrucci. Improving change prediction models with code smell-related information. *Empirical Software Engineer*, 25:49–95, 2020.

[25] K. Chaturvedi, P. Kapur, S. Anand, and V. B. Singh. Predicting the complexity of code changes using entropy based measures. *Int J Syst Assur Eng Manag*, (5):155—164, 2014.

[26] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, jun 2002.

[27] J. Chen and W. Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352, 2017.

[28] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[29] T. E. Colanzi, W. K. G. Assunção, P. R. Farah, S. R. Vergilio, and G. Guizzo. A review of ten years of the symposium on search-based software engineering. In S. Nejati and G. Gay, editors, *Proceedings of the 11th International Symposium on Search-Based Software Engineering, SSBSE 2019, Tallinn, Estonia, August 31 - September 1, 2019*, volume 11664 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2019.

[30] T. E. Colanzi, W. K. G. Assunção, S. R. Vergilio, P. R. Farah, and G. Guizzo. The symposium on search-based software engineering: Past, present and future. *Information and Software Technology*, 127:106372, 2020.

[31] U. B. Corrêa, L. Lamb, L. Carro, L. Brisolara, and J. Mattos. Towards estimating physical properties of embedded systems using software quality metrics. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 2381–2386, 2010.

[32] B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering*, SE-5(2):96–104, 1979.

[33] H. N. da Silva, P. R. Farah, W. D. F. Mendonça, and S. R. Vergilio. Assessing android test data generation tools via mutation testing. In I. Machado, R. Souza, R. S. P. Maciel, and C. Sant'Anna, editors, *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing, SAST 2019, Salvador, Brazil, September 23-27, 2019*, pages 32–41. ACM, 2019.

[34] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2018.

[35] M. Di Penta, L. Cerulo, Y.-G. Gueheneuc, and G. Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *2008 IEEE International Conference on Software Maintenance*, pages 217–226, 2008.

[36] T. G. Dietterich. Machine learning for sequential data: A review. In T. Caelli, A. Amin, R. P. W. Duin, D. de Ridder, and M. Kamel, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[37] T. G. Dietterich. Machine learning for sequential data: A review. In T. Caelli, A. Amin, R. P. W. Duin, D. de Ridder, and M. Kamel, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[38] G. Dotzler, M. Kamp, P. Kreutzer, and M. Philippsen. More accurate recommendations for method-level changes. In *Proceedings of the 11th FSE*, ESEC/FSE 2017, page 798–808. ACM, 2017.

[39] M. O. Elish and M. Al-Rahman Al-Khiaty. A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. *Journal of Software: Evolution and Process*, 25(5):407–437, 2013.

[40] M. O. Elish, H. Aljamaan, and I. Ahmad. Three empirical studies on predicting software maintainability using ensemble methods. *Soft Computing*, 19(9):2511–2524, 2015.

[41] S. Eski and F. Buzluca. An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 566–571, 2011.

[42] P. Farah. Thesis supplementary material. `https://github.com/paulorfarah/thesis_supplementary_material`, 2023.

[43] P. R. Farah, T. Mariani, E. A. Roza, R. C. Silva, and S. R. Vergilio. Unsupervised learning for refactoring pattern detection. In *IEEE Congress on Evolutionary Computation, CEC 2021, Kraków, Poland, July 28 - Aug 1, 2021*, pages 1–8. IEEE, 2021.

[44] P. R. Farah, R. D. C. Silva, and S. R. Vergilio. Machine learning prediction of change-prone methods and performance impactful changes. In *SBES '23: 37th Brazilian Symposium on Software Engineering, Campo Grande, Brazil, September 25-29, 2023*. ACM, 2023.

[45] P. R. Farah and S. R. Vergilio. Perfort: A tool for software performance regression. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion, page 119–120, New York, NY, USA, 2023. Association for Computing Machinery.

[46] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change Distilling:Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[47] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[48] H. C. Gall, B. Fluri, and M. Pinzger. Change analysis with Evolizer and ChangeDistiller. *IEEE software*, 26(1):26–33, 2009.

[49] Y. Ge, M. Chen, C. Liu, F. Chen, S. Huang, and H. Wang. Deep metric learning for software change-proneness prediction. In *International Conference on Intelligent Science and Big Data Engineering*, pages 287–300. Springer, 2018.

[50] E. Giger, M. Pinzger, and H. C. Gall. Can we predict types of code changes? an empirical analysis. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 217–226. IEEE, 2012.

[51] D. Godara, A. Choudhary, , and R. K. Singh. Predicting change prone classes in open source software. *International Journal of Information Retrieval Research (IJIRR)*, 8(4):1–23, Oct. 2018.

[52] D. Godara and R. Singh. A review of studies on change proneness prediction in object oriented software. *International Journal of Computer Applications*, 105(3):0975–8887, 2014.

[53] F. Grund, S. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes. Codeshovel: A reusable and available tool for extracting source code histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 221–222, 2021.

[54] A.-R. Han, S.-U. Jeon, D.-H. Bae, and J.-E. Hong. Measuring behavioral dependency for improving change-proneness prediction in uml-based design models. *Journal of Systems and Software*, 83(2):222–234, 2010.

[55] J. Han, J. Pei, and H. Tong. *Data mining: concepts and techniques*. Morgan kaufmann, 2022.

[56] X. Han and T. Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, New York, NY, USA, 2016. Association for Computing Machinery.

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

[58] H. He, Y. Bai, E. A. Garcia, and S. Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 1322–1328, 2008.

[59] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, pages 60–71, 2014.

[60] IBM. Identifying and resolving performance issues. `https://www.ibm.com/docs/en/txseries/8.2?topic=troubleshooting-identifying-resolving-performance-issues`, 2019.

[61] I. Ilyas and X. Chu. *Data Cleaning*. ACM Collection II Series. Association for Computing Machinery, 2019.

[62] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[63] A. Kaur, K. Kaur, and S. Jain. Predicting software change-proneness with code smells and class imbalance learning. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 746–754, 2016.

[64] K. Kaur and S. Jain. Evaluation of machine learning approaches for change-proneness prediction using code smells. *Advances in Intelligent Systems and Computing*, 515, 2017.

[65] L. Kaur. *Selection of Change-prone Software Components using the Expertise of Semantic Web and Intelligent Computing Methods*. PhD thesis, Thapar Institute of Engineering & Technology, 2021.

[66] L. Kaur and A. Mishra. A comparative analysis of evolutionary algorithms for the prediction of software change. In *2018 International Conference on Innovations in Information Technology (IIT)*, pages 187–192, 2018.

[67] Y. A. Khan, M. O. Elish, and M. El-Attar. A systematic review on the impact of CK metrics on the functional correctness of object-oriented classes. In *International Conference on Computational Science and Its Applications*, pages 258–273. Springer, 2012.

[68] M. Khanna, S. Priya, and D. Mehra. Software change prediction with homogeneous ensemble learners on large scale open-source systems. In *17th IFIP International Conference on Open Source Systems (OSS)*, pages 68–86. Springer International Publishing, 2021.

[69] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84, 2009.

[70] F. Khomh, Y.-G. Di Penta, Massimiliano an Guéhéneuc, and G. Antonio. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empir Software Eng*, 17:243–275, 2011.

[71] T. Khoshgoftaar, N. Seliya, and Y. Liu. Genetic programming-based decision trees for software quality classification. In *Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 374–383, 2003.

[72] D. K. Kim. Finding bad code smells with neural network models. *International Journal of Electrical and Computer Engineering*, 7(6):3613, 2017.

[73] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, 2009.

[74] E. Kitsu, T. Omori, and K. Maruyama. Detecting program changes from edit history of source code. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 299–306, 2013.

[75] A. Koru and J. Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transactions on Software Engineering*, 31(8):625–642, 2005.

[76] A. G. Koru and H. Liu. Identifying and characterizing change-prone classes in two large-scale open-source products. *Journal of Systems and Software*, 80(1):63–73, 2007.

[77] W. H. Kruskal and W. A. Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[78] L. Kumar, R. K. Behera, S. Rath, and A. Sureka. Transfer learning for cross-project change-proneness prediction in object-oriented software systems: A feasibility analysis. *ACM SIGSOFT Software Engineering Notes*, 42(3):1–11, 2017.

[79] L. Kumar, S. Lal, A. Goyal, and N. L. B. Murthy. Change-proneness of object-oriented software using combination of feature selection techniques and ensemble learning techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC'19, New York, NY, USA, 2019. Association for Computing Machinery.

[80] L. Kumar, S. K. Rath, and A. Sureka. Empirical analysis on effectiveness of source code metrics for predicting change-proneness. In *Proceedings of the 10th Innovations in Software Engineering Conference*, pages 4–14, 2017.

[81] L. Kumar, S. K. Rath, and A. Sureka. Using source code metrics to predict change-prone web services: A case-study on eBay services. In *2017 IEEE workshop on machine learning techniques for software quality evaluation (MaLTeSQuE)*, pages 1–7. IEEE, 2017.

[82] C. Laaber and P. Leitner. (h|g)opper: Performance history mining and analysis. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 167–168, New York, NY, USA, 2017. Association for Computing Machinery.

[83] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2010.

[84] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, New York, NY, USA, 2010. Association for Computing Machinery.

[85] M. Lindvall. Are large C++ classes change-prone? An empirical investigation. *Journal of Software: Practice and Experience*, 28(15):1551–1558, 1998.

[86] C. Liu, D. Yang, X. Xia, M. Yan, and X. Zhang. Cross-project change-proneness prediction. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 64–73. IEEE, 2018.

[87] Y. Liu and T. Khoshgoftaar. Genetic programming model for software quality classification. In *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*, pages 127–136, 2001.

[88] W.-Y. Loh. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(1):14–23, 2011.

[89] M. Lorenz and J. Kidd. *Object-oriented software metrics - a practical guide*. Prentice-Hall, Inc., 1994.

[90] H. Lu, Y. Zhou, B. X, H. Leung, and L. Chen. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Software Engineer*, 17:200–242, 2012.

[91] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[92] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518, 2015.

[93] R. Malhotra and B. A. Predicting change using software metrics: A review. In *IEEE International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 1–6, 2015.

[94] R. Malhotra and A. Bansal. Investigation of various data analysis techniques to identify change prone parts of an open source software. *International Journal of System Assurance Engineering and Management*, 9(2):401–426, 2018.

[95] R. Malhotra and A. J. Bansal. Cross project change prediction using open source projects. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 201–207. IEEE, 2014.

[96] R. Malhotra and R. Jangra. Prediction & assessment of change prone classes using statistical & machine learning techniques. *Journal of Information Processing Systems*, 13(4):778–804, 2017.

[97] R. Malhotra, R. Kapoor, D. Aggarwal, and P. Garg. Comparative study of feature reduction techniques in software change prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 18–28, 2021.

[98] R. Malhotra and M. Khanna. Investigation of relationship between object-oriented metrics and change proneness. *International Journal of Machine Learning and Cybernetics*, 4:273–286, 2013.

[99] R. Malhotra and M. Khanna. A new metric for predicting software change using gene expression programming. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, pages 8–14, 2014.

[100] R. Malhotra and M. Khanna. Mining the impact of object oriented metrics for change prediction using machine learning and search-based techniques. In *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 228–234, 2015.

[101] R. Malhotra and M. Khanna. An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering*, 22:2806–2851, 2017.

[102] R. Malhotra and M. Khanna. An exploratory study for software change prediction in object-oriented systems using hybridized techniques. *Automated Software Engineering*, 24(3):673–717, 2017.

[103] R. Malhotra and M. Khanna. Particle swarm optimization-based ensemble learning for software change prediction. *Information and Software Technology*, 102:65–84, 2018.

[104] R. Malhotra and M. Khanna. Prediction of change prone classes using evolution-based and object-oriented metrics. *Journal of Intelligent & Fuzzy Systems*, 34:1755–1766, 2018.

[105] R. Malhotra and M. Khanna. Software change prediction: A systematic review and future guidelines. *e-Informatica Software Engineering Journal*, 13(1):227–259, 2019.

[106] R. Malhotra and M. Khanna. On the applicability of search-based algorithms for software change prediction. *International Journal of Systems Assurance Engineering and Management*, April 2021.

[107] R. Malhotra and K. Lata. An empirical study on predictability of software maintainability using imbalanced data. *Software Quality Journal*, 28, 2020.

[108] C. Marinescu. How good is genetic programming at predicting changes and defects? In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 544–548. IEEE, 2014.

[109] A. D. F. Martins, C. S. Melo, J. M. Monteiro, and J. de Castro Machado. Empirical study about class change proneness prediction using software metrics and code smells. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 140–147, 2020.

[110] M. Massoudi, N. K. Jain, and P. Bansal. Software defect prediction using dimensionality reduction and deep learning. In *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, pages 884–893, 2021.

[111] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[112] C. S. Melo. Supporting change-prone class prediction. Master's thesis, Post-Graduation Program in Computer Science - Federal University of Ceará, Fortaleza - SC, 2020.

[113] C. S. Melo, M. M. L. da Cruz, A. D. F. Martins, J. M. da Silva Monteiro Filho, and J. de Castro Machado. Time-series approaches to change-prone class prediction problem. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 122–132, 2020.

[114] C. S. Melo, M. M. L. da Cruz, A. D. F. Martins, T. Matos, J. M. da Silva Monteiro Filho, and J. de Castro Machado. A practical guide to support change-proneness prediction. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 269–276, 2019.

[115] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511, 2013.

[116] C. E. Metz. Basic principles of roc analysis. *Seminars in Nuclear Medicine*, 8(4):283–298, 1978.

[117] T. Mitchell. *Machine Learning*. McGraw Hill, New York, 1997.

[118] A. Nielsen. *Practical Time Series Analysis: Prediction with Statistics and Machine Learning*. O'Reilly Media, 1 edition, 2019.

[119] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*, pages 237–246. IEEE, 2013.

[120] E. E. Ogheneovo et al. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 2(14):1, 2014.

[121] Oracle. Troubleshoot performance issues using flight recorder. `https://docs.oracle.com/en/java/javase/17/troubleshoot/troubleshoot-performance-issues-using-jfr.html`, 2022. September 6, 2022.

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

[123] N. Pritam, M. Khari, L. Hoang Son, R. Kumar, S. Jha, I. Priyadarshini, M. Abdel-Basset, and H. Viet Long. Assessment of code smell for predicting class change proneness using machine learning. *IEEE Access*, 7:37414–37425, 2019.

[124] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 354–363, 2007.

[125] D. G. Reichelt and S. Kühne. How to detect performance changes in software history: Performance analysis of software system versions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 183–188, New York, NY, USA, 2018. Association for Computing Machinery.

[126] D. Rey and M. Neuhäuser. *Wilcoxon-Signed-Rank Test*, pages 1658–1659. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[127] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *39th ICSE*, pages 404–415, 2017.

[128] D. Romano and M. Pinzger. Using source code metrics to predict change-prone Java interfaces. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 303–312, 2011.

[129] A. R. Sharafat and L. Tahvildari. Change prediction in object-oriented software systems: A probabilistic approach. *J. Softw.*, 3(5):26–39, 2008.

[130] R. D. C. Silva, P. R. Farah, and S. R. Vergilio. Machine learning for change-prone class prediction: A history-based approach. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, SBES '22, page 289–298, New York, NY, USA, 2022. Association for Computing Machinery.

[131] V. Soares, A. Oliveira, J. A. Pereira, A. C. Bibiano, A. F. Garcia, P. R. Farah, S. R. Vergilio, M. Schots, C. Silva, D. Coutinho, D. Oliveira, and A. G. Uchôa. On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns. In E. Cavalcante, F. Dantas, and T. Batista, editors, *SBES '20: 34th Brazilian Symposium on Software Engineering, Natal, Brazil, October 19-23, 2020*, pages 788–797. ACM, 2020.

[132] D. Spadini, M. Aniche, and A. Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[133] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[134] K. Z. Sultana, V. Anu, and T.-Y. Chong. Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach. *Journal of Software: Evolution and Process*, 33(3):e2303, 2021.

[135] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, 31(7):601–614, 2005.

[136] D. Tsoukalas, D. Kehagias, M. Siavvas, and A. Chatzigeorgiou. Technical debt forecasting: An empirical study on open-source repositories. *Journal of Systems and Software*, 170:110777, 2020.

[137] A. Vieira, P. Faustini, L. Carro, and E. Cota. Nfrs early estimation through software metrics. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 329–332, 2015.

[138] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering*, 26(12):1147–1156, 2000.

[139] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, and M. Data. *Practical machine learning tools and techniques*, volume 2. 2005.

[140] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[141] M. Yan, X. Zhang, C. Liu, L. Xu, M. Yang, and D. Yang. Automated change-prone class prediction on unlabeled dataset using unsupervised method. *Information and Software Technology*, 92:1–16, 2017.

[142] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[143] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 199–208, 2012.

[144] Y. Zhou, H. Leung, and B. Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. *IEEE Transactions on Software Engineering*, 35(5):607–623, 2009.

[145] X. Zhu, Y. He, L. Cheng, X. Jia, and L. Zhu. Software change-proneness prediction through combination of bagging and resampling methods. *Journal of Software: Evolution and Process*, 30(12):e2111, 2018.

[146] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

APPENDIX A – RELATED WORK

Table A.1: Previous work that employed supervised learning methods to predict code change-proneness.

Source	Year	Element	Metrics	Prediction Methods	Datasets
Al-Khiaty et al. [3]	2017		Product, Process	Group Method of Data Handling (GMDH)	VSSPLUGIN
Catolino et al. [22]	2017	Class	Product: (Cohesion) LCOM1, LCOM2, LCOM3, LCOM4, Co, Co', LCOM5, Coh, TCC, LCC, ICH, OCC, PCC, DCd, DC1, CAMC, NHD, SNHD. Product:(Coupling) CBO, RFC, MPC, DAC, ICP, IH-ICP, NIH-ICP, ACAIC, ACMIC, DCAEC, DCMEC, OCAIC, OCAEC, OCMIC, OCMEC, AMMIC, DMMEC, OMMIC, OMMEC, CBI. (Inheritance) DIT, AID, CLD, NOC, NOP, NOD, NOA, NMO, NMI, NMA, SIX, SPA, SPD, SP, DPA, DPD, DP. Developer: entropy of changes, structural and distance between every pair of classes modified, degree of textual similarity between every pair of classes modified, number of developers	ADTree, Decision Table Majority , Logistic Regression, Multilayer Perceptron, Support Vector Machine, and Naive Bayes	ArgoUML, Apache Ant, Apache Cassandra, Apache Xerces, aTunes, FreeMind, JEdit, JFreeChart, JHotDraw, JVLT.

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Kumar et al. [80]	2017		product	Logistic Regression (LOGR), Naive Bayes Classifier (NBC), Extreme Learning Machine (ELM) with linear (LIN), polynomial (PLY) and Radial Basis Function (RBF) kernels, Support Vector Machine (SVM) with linear (LIN), RBF and Sigmoid kernel (SIG) and two ensemble techniques such as Best-in-Training (BTE) and Majority Voting (MV)	Eclipse
Kumar et al. [81]	2017		Product	Least Squares Support Vector Machines (LS-SVM) (linear, polynomial and RBF)	eBay web services
Malhotra & Jangra [96]	2017		Product	Correlation-based feature selection, Random Forest (RF), AdaBoost, Bagging, Multilayer perceptron, Bayes Net, Naïve Bayes, J48, LogitBoost, NNge	AOI, SweetHom e 3D

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Malhotra & Khanna [102]	2017			genetic fuzzy system Logitboost (GFS_Logit) and hierarchical decision rules (HIDER), [decision tress with genetic algorithms (DT_GA),neural net evolutionary programming (NNEP), particle swarm optimization (PSO) with a statistical technique linear discriminant analysis (LDA), genetic algorithm based classifier algorithm with adaptive discretization intervals (GA_ADI) and memetic Pittsburgh learning classifier system (MPLCS),X-classifier system (XCS) and SUpervised classifier system (SUCS), constricted particle swarm optimization (CPSO), decision tree (C4.5), support vector machine (SVM), classification and regression trees (CART) and multilayer perceptron (multilayer perceptron with conjugate learning (MLP_CG), linear discriminant analysis (LDA)	Android packages

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Kumar et al. [78]	2017		Product	LOGR (Logistic Regression), ANN (Artificial Neural Networks), RBFN (Radial Basis Function Network), DT (Decision Trees), RF (Random Forests), BTE (Best Training Ensemble), MVE (Majority Voting Ensemble) and NDTF (Non Linear Decision Tree Forest)	Eclipse plug-ins: compare, webdav, debug, update, core, swt, team, pde, ui, jdt
Kaur & Jain [64]	2017	Class	Smells: feature envy (F_E), long method (L_M), God class (GOD), empty catch block (ECB), unprotected main program (UMP), dummy handler (DH), nested try statement (NTS), careless cleanup (CC), exceptions thrown from finally block (ETFFB) and overlogging (OL)	probabilistic and general regression neural network (P\G), group method of data handling polynomial network (GM), cascade correlation network (CCR), treeboost, gene expression programming (GEP), and discriminant analysis (DA)	Mobac, Jajuk, Gogui, Openrocket
Malhotra & Khanna [101]	2017	Class	Product	Random Forest, AdaBoost, LogitBoost, Bagging	Android calendar, Android bluetooth, Android MMS, Apache IO, Apache net, Apache Log4j
Yan et al. [141]	2017		Product	CLAMI, CLAMI+, LogitBoost, MLP, Radial basis function network (RBF), SVM, k-means	ant, antlr, argouml, azureus, freecol, freemind, hibernate, jgraph, jmeter, jstock, jung, junit, lucene, weka
Agrawal & Singh [2]	2018		Product	MLP, Logistic Regression, Random Forest, KStar, PARTial decision lists, Bagging, Baÿes Net	GATE, Tuxguitar, FreeCol, KolMafia, Legatus

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Catolino & Ferrucci [20]	2018		Product, Process, Developer-related	AdaBoost, Random Forest, Bagging	
Catolino et al. [20]	2018				
Ge et al. [49]	2018		Product	Logistic Regression, Naïve Bayes, Decision Tree, SVM, Decision Table, Deep Metric Learning	ArgoUML, FreeCol, JMeter, Jung, Weka
Liu et at. [86]	2018		Product	Baïes Net, CLAMI+	Ant, Antlr, ArgoUML, Azureus, FreeCol, FreeMind, Hibernate, JGraph, JMeter, JStock, Jung, JUnit, Lucene, Weka
Malhotra & Bansal [94]	2018			Bagging, Random Forest, Logit Boost, AdaBoost	
Malhotra & Khanna [104]	2018		Product, Process	Logistic Regression, MLP, Naïve Bayes, Random Forest, ADaBoost, Bagging, Logit Boost	Android Contacts, Android Gallery 2
Malhotra & Khanna [103]	2018		Product	Random Forest, Bagging, ADaBoost, Logit Boost, Constricted for Particle Swarm Optimization	Android Calendar, Android Contacts, Android Gallery, Android Bluetooth, Android MMS and Android Telephony, Apache Commons IO, Apache Commons Math, Apache Log4j and Apache Net
Zhu et al. [145]	2018		Complexity metrics, word metrics, file network metrics	C4.5, Naïve Bayes, SVM, Bagging	Ant, Eclipse, JEdit, Itextpdf, Liferay, Lucene, Struts, Tomcat
Catolino and Ferrucci [21]	2019				

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Pritam et al. [123]	2019	Class	Smells	Naive Bayes Classifier, Multi-layer Perceptron, Logit Boost, Bagging, Random Forest, and Decision Tree	
Kumar et al. [79]	2019	Class	DIT, NOA, CBO, RFC, LCOM, LCOM3, CAM, CAMC, ICH, MPC, DAC, MFA, NPM, IC, CBM, CLOC-L, LOC, AND SLOC-P	Logistic Regression, Polynomial Regression, Decision Tree, SVM (Linear, Polynomial and RBF), Extreme ML and Least-Square SVM, Simple Logistic, ANN, Best in Training, Majority Voting, Non-Linear Decision Tree Forest	Eclipse
Melo et al [114]	2019	Class	LOC, CBO, DIT, LCOM, NOC, RFC, WMC, CC	Logistic Regression, LightGBM, XGBoost, Decision Tree, Random Forest, KNN, Adaboost, Gradient Boost, SVM with Linear Kernel and SVM with RBF kernel	Proprietary
Abbas & Al-balooshi [1]	2020	Class	LOC, CBO, DIT, LCOM, NOC, RFC, WMC, CC"	"Instance-Bases Learning with parameter K (IBK), Decision Tree J48, Random Forest (RF), Bootstrap aggregating (Bagging), Adaptive boosting (AdaBoost), LogitBoost, Multilayer Perceptron (MLP), Naïve Bayes, Bayesian Networks (BN), and One-R. Combining methods: Voting, Select-Best, and staking scheme"	"Proprietary, Apache Ant, Apache Beam, Apache Cassandra

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Abdilrahim & Alhawi [5]	2020	Class	LOC, Number of bugs, Defect density	Boxplot, outlier analysis	Beam, Camel, Ignite, Jenkins, and JMeter

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Malhotra & Lata [107]	2020	Class	CK: WMC: weighted methods per class, DIT: depth of inheritance tree, NOC: number of children of a class, CBO: coupling between the objects, LOCM: lack of cohesion among methods, and RFC: response for class. QMOOD: MOA: measure of aggregation, DAM: data access metric, MFA: measure of functional abstraction, NPM: number of public methods, and CAM: cohesion among methods of a class. Martin: Ce: efferent coupling and Ca: afferent coupling. Other: IC: inheritance coupling, CBM: coupling between the class methods and AMC: average method complexity, LOC: lines of source code, and LCOM3. LCOM3 is the variation of LCOM given by Henderson-Sellers. The metrics WMC, NPM, LOC, DAM, and AMC are indicators of the size of a class. The metrics CBO, RFC, Ca, Ce, IC, and CBM measure the coupling. The inheritance property is measured with the help of NOC, DIT, and MFC. The metrics LCOM, CAM, and LCOM3 are indicators ofclass cohesion, whereas MOA measures composition.	C4.5, Multilayer perceptron with conjugate learning (MLP-CG), Radial basis function neural network (RBFNN), Increment radial basis function neural network (IRBFNN), BG, AdaBoost, KNN	Apache Bcel, Betwixt, Io, Ivy, Jcs, Lang, Log4j, Ode

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Martins et al. [109]	2020	Class	Class Between Object (CBO), Cyclomatic Complexity (CC), Depth of Inheritance Tree (DIT), Lines Of Code (LOC), Number Of Children (NOC) andWeighted Methods per Class (WMC)	Logistic Regression (LR), Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF), K-Nearest Neighbours (KNN), Light Gradi- ent Boost Machine (LGBM), and eXtreme Gradient Boost Machine (XGB)	Proprietary web application
Melo et al. [113]	2020	Class	Class Between Object (CBO), Cyclomatic Complexity (CC), Depth of Inheritance Tree (DIT), Lack of Cohesion in Methods (LCOM), Line of Code (LOC), Number of Child (NOC), Response for a Class (RFC), Weighted Methods per Class (WMC)	Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), MLP, Gated Recurrent Unit (GRU)	Apache Ant, Apache Beam, and Apache Cassandra

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Malhotra et al. [97]	2021	Class	"OO: Average Cyclomatic Complexity, Average Modified Cyclomatic Complexity, Average Strict Cyclomatic Complexity, Average Essential Cyclomatic Complexity, Average Number of Lines, Average Number of Blank Lines, Average Number of Lines of Code, Average Number of Lines with Comments, A Base Classes, Coupling Between Objects, Count of Modified Coupling Between Classes, Number of Children, Classes, Class Methods, Class Variables, Executable Unit, Number of Files, Function, Instance Methods, Instance Variables, Local Methods, CountDeclMethodAll, Local Default Visibility Methods, Private Methods, Protected Methods, Public Methods, Inputs, Physical Lines, Blank Lines of Code, Lines of Code, Declarative Lines of Code, Executable Lines of Code, Lines with Comments, Outputs, Paths, Paths Log(x), Semicolons, Statements, Declarative Statements, Executable Statements, Cyclomatic Complexity, Modified Cyclomatic Complexity, Strict Cyclomatic Complexity, Essential Complexity, Knots, Max Cyclomatic Complexity, Max Modified Cyclomatic Complexity, Max Strict Cyclomatic Complexity	"Naïve Bayes, SVM, Decision Tree"	"Ant, Tomcat, Wro4j, Hibernate, Javaclient, Neuroph"

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Melo et al. [112]	2020	Class	LOC, CBO, DIT, LCOM, NOC, RFC, WMC, CC	Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), and MLP, GRU	Proprietary, Apache Ant, Apache Beam, Apache Cassandra
Kaur [65]	2021	Component	SLOC, Total Cyclomatic Complexity (CycloC), Cumulative Halstead Length (CHL), Cumulative Halstead Volume (CHV), Cumulative Halstead Effort (CHE), Cumulative Halstead Bugs(CHB), Maintainability Index (MI), Afferent Coupling(AC), Efferent Coupling(EC)	Bayesian Classification: Bayesian Network, Naïve Bayes Functions: Fisher's Linear Discriminant (FLDA), Logistic Regression (LR), MLP, Radial Basis Function Neural Network (RBFN), Sequential Minimal Optimization (SMO), SPegasos (SPEG)	

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Kaur [65] (continuation)	2021	Component	Instability, Weighted Methods per Class(WMC), Depth of inheritance (DIT), Number of children (NOC), Coupling between objects (CBO), Response for a class (RFC), Lack of cohesion (LCOM), Cognitive Complexity (CogC)	Meta-classification: ADaBoost (ADB), Bagging (BAG), Dagging (DAG), Filtered Classifier (FC), LogitBoost (LB), Random Sub Space (RSS) Miscellaneous: Composite Hypercubes on Iterated Random Projections (CHIRP), Fuzzy Latice Reasoning (FLR), Voting Feature Intervals (VFI), Ant Miner (AM) Decision Rules: Decision Table/Naïve Bayes hybrid classifier (DTNB), Fuzzy Unordered Rule Induction Algorithm (FURIA), Modlem (MOD), Non-Nested Generalised Exemplars (NNGE), PARTial decision lists (PART) Decision Trees: Hoeffding Trees (HFT), J48, Random Forest, Classification and Regression Trees, SysFor, Evolutionary Approach: Genetic Algorithm with Neural Network (GANN), Neural Network Evolutionary (NNEP), Multi Objective Evolutionary Fuzzy Classifier (MOEFC)	JFreeChart, Heritrix

Continued on next page

Table A.1 – continued from previous page

Source	Year	Element	Metrics	Prediction Methods	Datasets
Khanna et al. [68]	2021	Class	Product	Homogeneous Ensemble Learners: AdaBoost (AB), Bagging (BG), Dagging (DG), Decorate (DC), MultiBoostAB (MB), Random Forest (RF), Random SubSpace (RSS) and Rotation Forest (ROF), Classification and Regression Trees (CART), Instance-based learner (IB), J48, JRip, Logistic Regression (LR), Multilayer Perceptron (MLP), Naive Bayes (NB), OneR and Sequential Minimal Optimization (SMO))	
Malhotra & Khanna [106]	2021	Class	CK: Coupling Between Objects, Depth of Inheritance Tree, Lack of Cohesion amongst Methods, Number of Children, Response for a Class and Weighted Methods per Class metrics Source Lines of Code (SLOC)	Statistical: LDA. ML: SVM, Naïve Bayes, adaBoost, Classification and Regression Tree, MLP SB: Constricted Particle Swarm Optimization, Genetic Algorithm Based Classifier System with Adaptive Discretization Intervals and with Intervalar Rules, Gene Experssion Programming, Hierarchical Decision Rules, Memetic Pittsburgh Learning Classifier System, Supervised Classifier System, X-Classifier System	AOI, Apollo, AviSync, Celestia, DrJava, Dspace, Eclipse, Frinika, Glest, Jmeter, PMD, Robocode, Simutrans, Subsonic

APPENDIX B – SOFTWARE METRICS

Table B.1: Structural metrics.

Metric	Definition	Source
anonymousClassesQty	Quantity of Anonymous classes	CK
assignmentsQty	The number of assignments	CK
AvgCyclomatic	Average Cyclomatic Complexity	Understand
AvgCyclomatic-Modified	Average cyclomatic complexity for all nested functions or methods	Understand
AvgCyclomaticStrict	Average strict cyclomatic complexity for all nested functions or methods	Understand
AvgEssential	Average Essential complexity. [aka Ev(G)]	Understand
AvgLine	The average number of physical lines between methods of a class	Understand
AvgLineBlank	The average number of blanks for all nested functions or methods	Understand
AvgLineCode	The average number of lines containing source code for all nested functions or methods	Understand
AvgLineComment	The average number of lines containing comments between methods of a class	Understand
CBO	Coupling Between Objects	CK
CBOModified	Coupling Between Objects in both directions	CK
comparisonsQty	The number of comparisons (i.e., == and !=)	CK
CountClassBase	IFANIN	Understand
CountClassCoupled	Chidamber & Kemerer - Coupling Between Objects (CBO)	Understand
CountClassDerived	Chidamber & Kemerer - Number of Children (NOC)	Understand
CountDeclClass	Number of declared classes	Understand
CountDeclClass-Method	Number of Class Methods	Understand
CountDeclClass-Variable	Lorenz & Kidd - Number of Variables (NV)	Understand
CountDeclFile	Number of Files	Understand
CountDeclFunction	The number of declared functions	Understand
CountDeclInstance-Method	Number of Instance Methods (NIM)	Understand
CountDeclInstance-Variable	Number of Instance Variables (NIV)	Understand
CountInput	FANIN (Infomational fan-in)	Understand
CountLine	Number of Lines (NL)	Understand
CountLineBlank	Blank Lines of Code (BLOC)	Understand
CountLineCode	Lines of Code (LOC), Source Lines of Code (SLOC)	Understand
CountLineCodeDecl	The number of Declarative Code Lines	Understand
CountLineCodeExe	The number of Executable Code Lines	Understand
CountLineComment	Comment Lines of Code (CLOC)	Understand
CountOutput	FANOUT (Infomational fan-out)	Understand
CountPath	The number of NPATH	Understand
CountSemicolon	The number of Semicolons	Understand
CountStmt	The number of Statements	Understand

Continued on next page

Table B.1 – continued from previous page

Metric	Definition	Source
CountStmtDecl	The number of Declarative Statements	Understand
CountStmtExe	The number of Executable Statements	Understand
Cyclomatic	McCabe - McCabe Cyclomatic Complexity, CC	Understand
CyclomaticModified	McCabe - McCabe Modified Cyclomatic Complexity, CC3	Understand
CyclomaticStrict	McCabe - McCabe Strict Cyclomatic Complexity, CC2	Understand
Essential	Essential Complexity	Understand
FANIN	number of input dependencies	CK
FANOUT	number of output dependencies	CK
hasJavaDoc	Boolean indicating whether a method has javadoc	CK
innerClassesQty	Quantity of inner classes	CK
lambdasQty	Quantity of lambda classes	CK
LINE	Method start line	CK
LOC	Lines of code	CK
logStatementsQty	Number of log statements compatible with SLF4J and Log4J	CK
loopQty	The number of loops (i.e., for, while, do while, enhanced for)	CK
mathOperationsQty	The number of math operations	CK
MaxCyclomatic	Max Cyclomatic Complexity	Understand
MaxCyclomatic-Modified	Max Modified Cyclomatic Compexity	Understand
MaxCyclomaticStrict	Max Strict Cyclomatic Complexity	Understand
MaxEssential	Max Essential Complexity	Understand
MaxInheritanceTree	Chidamber & Kemerer - Depth of Inheritance Tree (DIT)	Understand
maxNestedBlocksQty	The highest number of blocks nested together	CK
MaxNesting	Max nesting	Understand
methodsInvoked-IndirectLocalQty	Quantity of methods invoked indirectally	CK
methodsInvoked-LocalQty	Quantity of methods invoked locally	CK
methodsInvokedQty	Quantity of invoked methods	CK
modifiers	Native modifiers of methods	CK
numbersQty	The number of numbers (i.e., int, long, double, float) literals	CK
parametersQty	The number of parameters	CK
parenthesizedExpsQty	The number of expressions inside parenthesis	CK
PercentLackOf-Cohesion	Chidamber & Kemerer - Lack of Cohesion in Methods (LCOM/LOCM), LCOM2	Understand
RatioComment-ToCode	Comment to Code Ratio	Understand
returnsQty	Quantity of returns	CK
RFC	number of unique method invocations	CK
stringLiteralsQty	The number of string literals (e.g., "John Doe").	CK
SumCyclomatic	Chidamber & Kemerer - Weighted Methods per Class (WMC)	Understand
SumCyclomatic-Modified	Sum Modified Cyclomatic Complexity	Understand
SumCyclomaticStrict	Sum Strict Cyclomatic Complexity	Understand
SumEssential	Sum Essential Complexity	Understand
tryCatchQty	The number of try/catches	CK
uniqueWordsQty	Number of unique words in the source code	CK
variablesQty	Quantity of declared variables	CK

Continued on next page

Table B.1 – continued from previous page

Metric	Definition	Source
WMC	Weight Method Class (McCabe's complexity)	CK

Table B.2: Software evolution-based metrics (Source: Elish and Al-Khiaty, 2013 [39]).

Metric	Definition
BOM	**Birth of a Method**. The first time the method appears.
TACH	**Total Amount of Changes**. It is the sum of added lines, deleted lines, and twice changed lines between release $n - 1$ and release n.
FCH	**First Change**. The first time the method has been exposed to changes.
LCH	**Last Change**. The last time the method has been exposed to changes.
CHO	**Change Occurred**. It is a binary metric that indicates whether or not the class has been exposed to changes from release $n - 1$ to n.
FRCH	**Frequency of Changes**. The number of times (in terms of releases) the method has been changed.
CHD	**Change Density**. The change density of a method C is its change size ($TACH(C)$) normalized by the size of the method (its total lines of code (LOC)).
WCD	**Weighted Change Density**. It is a cumulative frequency of change density (CHD) that favor the latest occurrence of changes over the old ones.
WFR	**Weighted Frequency of Changes**. Is a cumulative frequency of changes that favor the latest occurrence of changes over the old ones.
ATAF	**Aggregated Change Size Normalized by Frequency of Change**. This is obtained from accumulating the size of changes of the method in the past and normalizing by frequency of changes.
LCA	**Last Change Amount**. It is defined as the last change size of the method when moving from release $i - 1$ to release i.
LCD	**Last Change Density**. This metric is defined as its last change size (LCA) normalized by the size of the method.
CSB	**Changes since the Birth**. It is computed by comparing the size of the first version of a method with its current version.
CSBS	**Changes since the Birth Normalized by Size**. It is the CSB normalized by the size of the first version of the method
ACDF	**Aggregated Change Density Frequency**. It is obtained from the cumulating density of changes introduced to the method in the past, and then this accumulated amount is normalized by the frequency of changes.

APPENDIX C – PERFORT: A TOOL FOR SOFTWARE PERFORMANCE REGRESSION

This appendix presents PerfoRT, a tool that automates software performance measurement for Java applications. The overall idea is that readily available tests are used to provide feedback on how the performance of the application varies according to recent changes in development. The same workload is usually applied across regression tests so that the test results of prior commits are used as an informal baseline and compared against the current commit. Thus, it is an effective way to reveal if recent versions of the software are behaving correctly or if performance variation has been introduced. The main motivation for proposing PerfoRT is the automation of data collection used for the performance impactful method changes prediction. As stated previously, the existing tools are focused on selecting test cases to measure software performance. However, in our work, we also need the cases when there are no variations in performance to compose the training dataset.

The appendix is structured as follows. Section C.1 presents the architecture of PerfoRT. Section C.2 depicts the workflow of how PerfoRT works. Section C.3 describes the metrics set measured. Section C.4 illustrates a usage example. Finally, Section 6 concludes this appendix.

C.1 ARCHITECTURE

PerfoRT profiles and traces different project versions using existing tests in the system repository. It encompasses three main modules depicted in Figure C.1: PerfoRT-Core, PerfoRT-Tracer, and PerfoRT-Analizer[1]. PerfoRT requires Ubuntu operational system and Maven to run. The Java projects needs to be compiled using JDK 8 or superior and installed by Maven without errors.

PerfoRT-Core is responsible for cloning and extracting properties of the target system such as authors, committers, files, and changes. After this, it identifies which building tool is being used in the project, compiling, installing, and collecting existing regression tests using Apache Maven. It instruments classes using JaCoCo Java code coverage library[2] to collect coverage metrics, and executes tests using JUnit[3]. When a test case is executed, the agent JFR and the module PerfoRT-Tracer run in parallel. JFR[4] is a profiling and event collection framework built into the JDK. The module PerfoRT-Tracer uses Byte Buddy[5] to modify and record the time of executed methods during runtime. Methods are filtered from the monitored package specified in the settings. Then, PerfoRT-Core retrieves information on test running processes and system utilization using gopsutil[6] library. The raw data is recorded in a MySQL database. Lastly, PerfoRT-Analyzer processes raw data and allows visualization of the results. It outputs performance regression measured metrics to CSV files and charts. In this initial version of PerfoRT this module is capable of: generating a statistical descriptive analysis of versions, a violin density chart by versions, including all methods of all collected classes and by an specific method; generating a multiple area chart to evaluate performance of all methods of a class; and

[1]The source code of these modules is available at https://github.com/paulorfarah/perfort

[2]https://www.eclemma.org/jacoco/

[3]https://junit.org/junit5/

[4]https://docs.oracle.com/en/java/java-components/jdk-mission-control/8/user-guide/using-jdk-flight-recorder.html

[5]https://bytebuddy.net/

[6]https://github.com/shirou/gopsutil

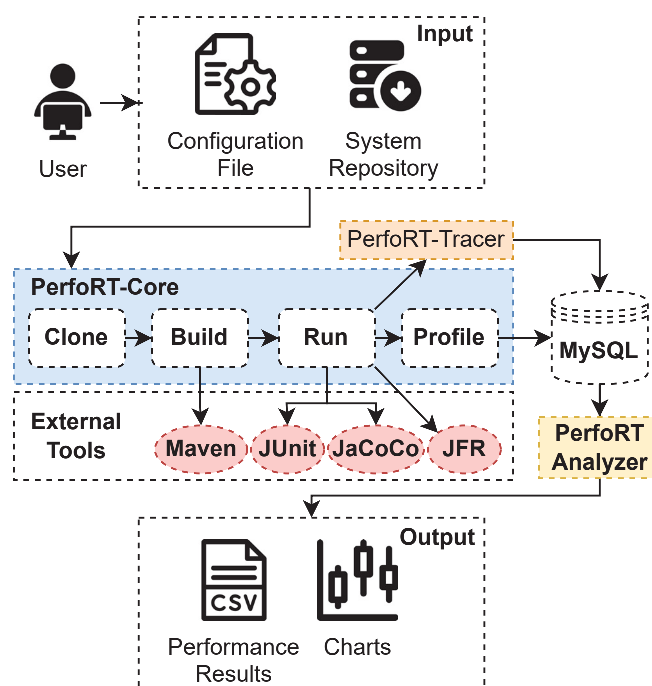


Figure C.1: PERFORT architecture.

generating a waterfall chart of execution time of called methods of the test cases. As this module uses Python, other analysis can be done directly by the users to better attend their needs. These main functionalities are illustrated in Section C.4.

C.2 WORKFLOW

Our tool automates the steps shown in Figure C.2 in order to measure performance regression testing. A configuration file must be provided by the user containing the paths for the system repository database, the package to profile, the number of executions and threads, the minimum execution time of the test cases, the monitoring interval time, as well as, the hashes list of commits to be measured. Next, the target system is cloned, the versions are read, and the application is prepared. For each version, PERFORT will checkout the hash of the corresponding commit, detect the building tool used, and the Java version, compile, install, list modules of the application, and read a parameter to define the minimum time that a test should have to be executed. This parameter can let the tool ignore tests that have a very small time.

For each system module of the target system, the tool will read its test list, inject the code to calculate coverage, and execute each test case. When the test is run, three events occur: a new lightweight thread (goroutine[7]) is started to monitor the process and system resources usage, a Flight Recorder agent for collecting diagnostic and profiling data about a running Java application is started and the methods are instrumented with the tracer agent to collect timestamps recorded at the entry and exit of the executed methods. Finally, performance raw data is recorded.

C.3 METRICS

PERFORT collects three groups of metrics: 1) method execution, 2) process and system, and 3) JVM metrics. Method execution metrics include execution time (own_duration and

[7]https://go.dev/tour/concurrency/1

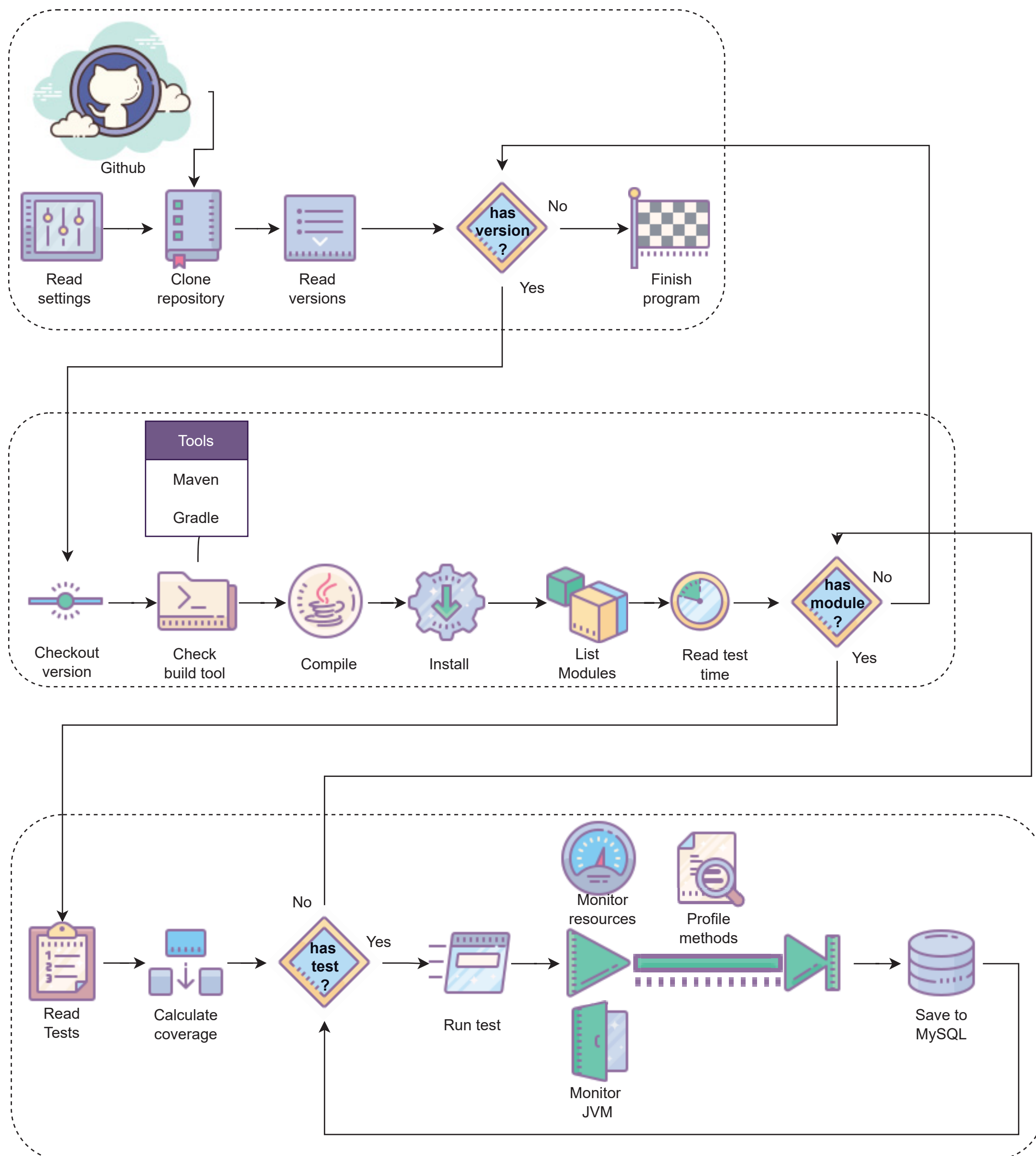


Figure C.2: PERFORT automated workflow.

cumulative_duration), and call frequency of methods and are collected by PERFORT-TRACER. Process and the system metrics are described in Table C.1.

The metrics monitored in JVM are grouped into five sets used to identify performance issues: find bottlenecks, garbage collection, synchronization, I/O, code execution, and memory usage [121]. The performance metrics collected from the JVM are presented in Table C.2. These metrics show different aspects of the system execution including CPU, memory systems, disk, network, faults, thread stalling, small and big objects allocated in memory, waiting time of the processor. Causes of contention points can be more accurately identified because of the greater variety of metrics monitored.

Table C.1: Process performance metrics collected by PerfoRT using gopsutil.

Name	Description
CPUPercent	Returns how many percent of the CPU time the process uses.
MemPercent	Returns how many percent of the total RAM the process uses.
RSS	Resident Set Size is used to show how much memory is allocated to the process and is in RAM.
VMS	Virtual Memory Size which is the virtual memory the process is using.
HWM	High Water Mark, when memory usage exceeds this level, system performance might be affected and you might look for ways to reduce the memory usage.
DRS	Data Resident Set is the amount of physical memory devoted to other than executable code.
Locked	Prevents the operating system from paging out heap or off-heap memory.
Swap	Extends the memory available to processes, storing infrequently used pages there.
ReadCount	number of disk read operations.
WriteCount	number of disk write operations.
ReadBytes	number of bytes read from disk.
WriteBytes	number of bytes written to disk.
MinorFaults	Occurs when a process needs data that is in memory assigned to another process.
MajorFaults	Occurs when a process attempts to access memory that exceeds its permissions.
ChildMinor Faults	number of minor faults with child's.
ChildMajor Faults	number of major faults with child's.

C.4 USAGE EXAMPLE

This section illustrates the inputs and outputs produced by PerfoRT through a usage example. To this end, we use the Apache Commons BCEL[8] byte code engineering library. BCEL has been used successfully in several projects such as compilers, optimizers, obfuscators, code generators, and analysis tools. Listing C.1 shows the configuration file `.env`, input of our tool, which contains: database settings (db_user, db_pass, db_name, db_host, db_port), repository URL of the target system (repository), root package of the classes to be monitored (package), number of runs (runs), number of threads to execute tests in parallel (threads), minimum test execution time to execute (min_test_time), time interval to monitor process and system (monitoring_time), and test case timeout to ignore possible stuck test cases (testcase_timeout).

Listing C.1: Input settings file .env.

```
db_user="root"
db_pass="password"
db_name="perfort"
db_host="localhost"
db_port="3306"
repository="https://github.com/apache/commons-bcel"
```

[8]https://commons.apache.org/proper/commons-bcel/

Table C.2: Process performance metrics collected by PerfoRT using JFR.

Issue	Name	Description
Bottleneck	ThreadStart	Notification of a new running thread in the VM.
	ThreadEnd	Notification the end of a thread in the VM.
	ThreadSleep	Causes the current thread to suspend execution for a specified period.
	ThreadPark	Shows when a thread is parked.
	JavaErrorThrow	An object derived from java.lang.Error has been created. Out of memory errors are ignored.
	JavaExceptionThrow	An object derived from java.lang.Exception has been created.
	JavaMonitorEnter	Java statistics monitor blocked.
Bottleneck/ Synchronization	JavaMonitorWait	Shows how much time a thread spends waiting for a monitor.
Bottleneck/IO	FileRead	Reading data from a file.
	FileWrite	Writing data to a file.
	SocketRead	Reading data from a socket.
	SocketWrite	Writing data to a socket.
Code Execution	CPULoad	OS CPU Load.
	ThreadCPULoad	Identifies the threads that use the most CPU time.
Memory Leak	OldObjectSample	Contains information about potential memory leaks.
	ClassLoadingStatistics	Contains statistical information about class loading in VM.
	ClassLoader Statistics	Contains statistical information about classes that load classes in VM.
	ObjectAllocationInNewTLAB	Shows new small objects allocated in TLAB area.
	ObjectAllocationOutsideTLAB	Shows new large objects allocated outside a TLAB area.
	Heap Statistics	Contains statistics about the VM heap.
Garbage Collection (GC)	AvgSumOfPauses	Average amount of time that the application was paused during a GC.
	MaxSumOfPauses	Maximum time that the application was paused during a GC.
	TotalSumOfPauses	Total amount of time that the application was paused during a GC.

```
package="org.apache.bcel."
runs=1
threads=1
min_test_time=0.0
monitoring_time=0.01
testcase_timeout=1800
```

This example contains four releases of Apache Commons BCEL. Figure C.3 presents a resources measurement sample output extracted from the raw collected data. We calculated the average of all resource usage metrics for the executed methods in each release. For example, the average own duration of method $isStatic()$ is $316ms$ and the average CPU is 33.94%.

1	class_name	method_name	own_dur	cum_dur	cpu_per	mem_
2	src/main/java/org/apache/bcel/classfile/AccessFlags.java	public final boolean org.apache.bcel.classfile.AccessFlags.isAbstract()	3	3	55.29	2.4
3	src/main/java/org/apache/bcel/classfile/AccessFlags.java	public final boolean org.apache.bcel.classfile.AccessFlags.isAbstract()	5	5	199.76	2.1
4	src/main/java/org/apache/bcel/classfile/AccessFlags.java	public final boolean org.apache.bcel.classfile.AccessFlags.isAbstract()	195	195	34.80	2.4
5	src/main/java/org/apache/bcel/classfile/AccessFlags.java	public final boolean org.apache.bcel.classfile.AccessFlags.isNative()	248	248	34.42	2.4
6	src/main/java/org/apache/bcel/classfile/AccessFlags.java	public final boolean org.apache.bcel.classfile.AccessFlags.isStatic()	316	316	33.94	2.4
7	src/main/java/org/apache/bcel/classfile/AccessFlags.java	public final int org.apache.bcel.classfile.AccessFlags.getAccessFlags()	4	4	148.41	2.2

Figure C.3: Sample of methods resource usage from Apache Commons BCEL.

To better illustrate the main functionalities of the module PerfoRT-Analyzer we show its output in a top-down way. We start analyzing performance changes at the version level, next we consider the class and method levels. To analyze the version scope, we exhibit a violin chart, shown in Figure C.4 for the CPU average usage as example. First, we calculated the average of each metric, grouped by version and used the mean value of CPU. This chart shows the average, quartiles, and density of the values. We can observe that the second version, which occurred in 2019-09-20, used lower CPU.

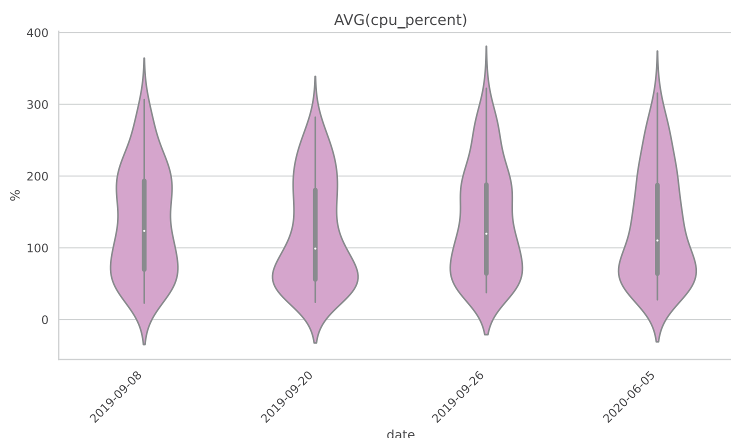


Figure C.4: CPU usage for Apache Commons BCEL system.

In a second moment, we analyzed each release. To do this, we generated Table C.3 that includes, for each metric (own duration, cumulative duration, CPU and RAM memory percentage usage), the mean, standard deviation, median, minimum and maximum values. We can observe that the standard deviation of cumulative duration is very high. For example, the value for release fa271c is 1799.58 and may indicate that the results have outliers.

The goal of the analysis at the version level is show a performance's overview of the including all measured methods of all classes. This analysis can be done in the scope of versions, packages, classes and methods to compare the performance of code elements. Figure C.5 shows and example of the class scope. It contains the performance analysis of the test cases of the test file PLSETestCase.java of four releases identified by the hashes fa271c5, bbaf623, bebe70d,

Table C.3: Statistics by commit.

		a9c13e	bbaf62	bebe70	fa271c
Own duration	Mean	35.39	27.85	30.01	32.64
	Std Dev.	198.62	115.74	164.54	187.71
	Median	8.00	9.00	8.00	9.00
	Min	1.0	2.00	2.00	2.00
	Max	9877.00	3453.00	6714.00	11366.00
Cumulative duration	Mean	157.56	143.60	141.53	173.75
	Std Dev.	1604.22	1429.44	1568.06	1799.58
	Median	14.00	15.00	14.00	15.00
	Min	1.0	2.00	2.00	2.00
	Max	73841.00	98469.00	102621.00	58820.00
CPUPercent	Mean	98.66	108.13	98.56	86.08
	Std Dev.	57.55	54.46	48.34	57.08
	Median	93.90	96.78	91.34	67.64
	Min	1.0	2.00	2.00	2.00
	Max	282.10	270.81	289.60	287.84
MemPercent	Mean	1.67	1.70	1.64	1.69
	Std Dev.	0.30	0.29	0.27	0.30
	Median	1.71	1.77	1.69	1.84
	Min	0.74	0.80	0.77	0.74
	Max	2.05	2.07	1.93	2.01

and A9C13ED. The file contains four methods: testB208(), testB262() testB295() and testB79(). Three metrics are presented: cumulative duration of the execution time, and CPU and RAM memory percent usage. This output provides the developer with information about the set of test cases between evaluated versions. For example, we can observe that testB79() presented a significant increase in the cumulative duration for the release BBAF623. On the other hand, it shows the smallest CPU usage of the evaluated releases.

At the method level we can observe in Figure C.6 an example of a waterfall chart of the cumulative duration of all methods called by test case testB79(), from test file PLSETestCase.java of release BBAF623. The chart shows the time in seconds and the methods are in order of the bottom to the top. The testcase testB79() at the base of the figure calls AbstractTestCase.getTestClass(java.lang.String) and so on. As the chart is cumulative, bars at the bottom sum the time of all bars above them. Each color of method call represents a different class. Thus, the developers can analyze and look for possible causes of a degradation in performance.

C.5 CONCLUDING REMARKS

This Appendix introduces PERFORT, a tool to measure software performance using existing regression tests. The tool measures miscellaneous metrics of Java programs including method calls, execution time, testing code coverage, process and system utilization, and JVM events. In this way, PERFORT allows developers, testers, and researchers to mine a robust set of software performance aspects of real-world projects in an automated fashion. It automates performance regression testing tasks since clone the project from GitHub to measure performance metrics and trace the target system and save results in a database. The results can help measure and analyze performance of systems and create software performance detailed datasets for varied purposes.

This tool was designed to measure the performance metrics to be used in the prediction of performance impactful method changes. As depicted in the chapter 3, we used PERFORT to measure the five subject systems used in this research. Although we measured all the described

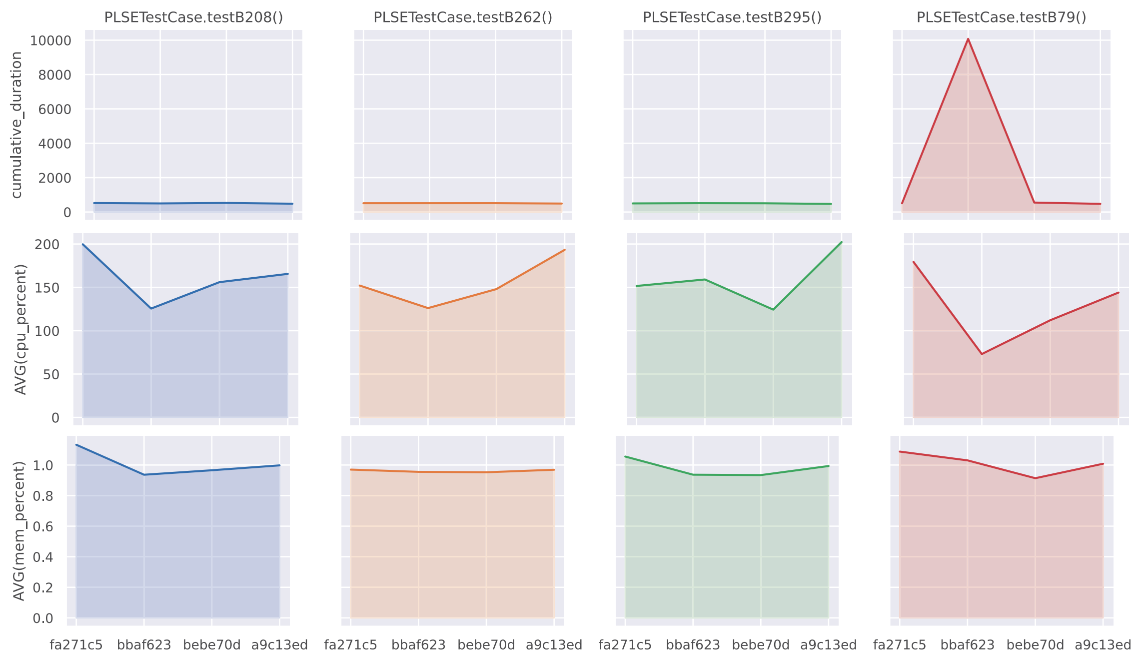


Figure C.5: Test cases monitored from PLSETestCase.java.

metrics in Section C.3, in this thesis we used only the own duration of the methods that represent their execution time.

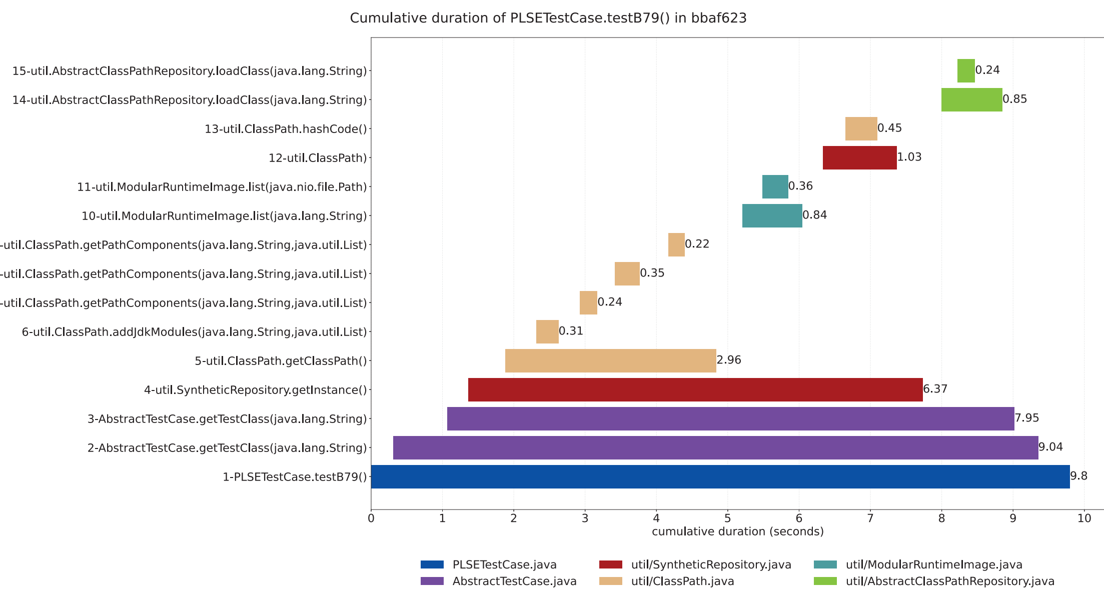


Figure C.6: Cumulative execution time (in seconds) of methods called by the test case testRemoveLocalVariables() from generic.MethodGenTestCase.