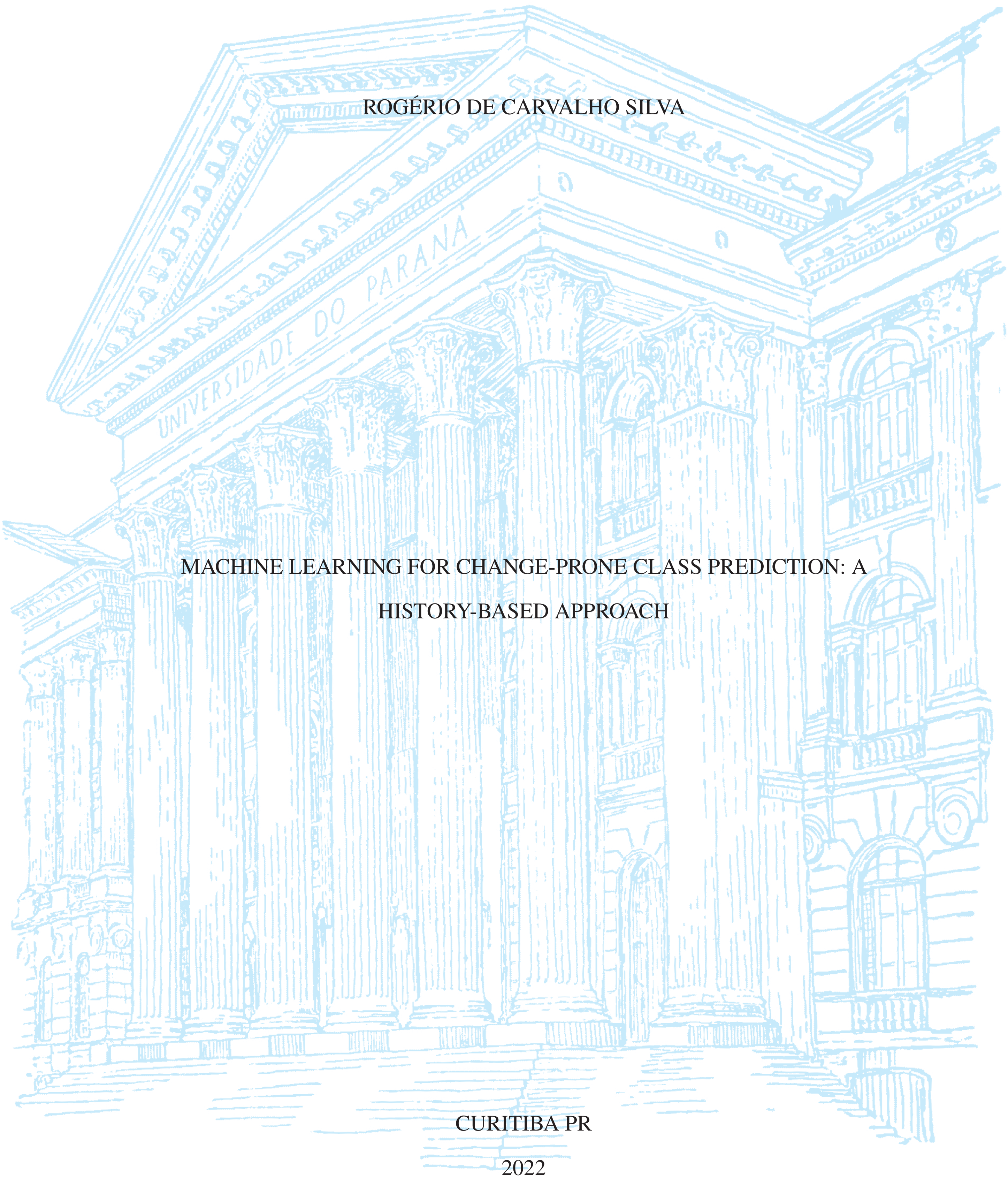
UNIVERSIDADE FEDERAL DO PARANÁ

ROGÉRIO DE CARVALHO SILVA

MACHINE LEARNING FOR CHANGE-PRONE CLASS PREDICTION: A HISTORY-BASED APPROACH

CURITIBA PR

2022



ROGÉRIO DE CARVALHO SILVA

MACHINE LEARNING FOR CHANGE-PRONE CLASS PREDICTION: A HISTORY-BASED APPROACH

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

Área de concentração: *Ciência da Computação*.

Orientador: Silvia Regina Vergilio.

CURITIBA PR

2022

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

Silva, Rogério de Carvalho
Machine learning for change-prone class prediction: a history-based approach / Rogério de Carvalho Silva. – Curitiba, 2022.
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.

Orientadora: Silvia Regina Vergilio.

1. Software - Desenvolvimento. 2. Software - Manutenção. I. Vergilio, Silvia Regina. II. Universidade Federal do Paraná. Programa de pós-Graduação em Informática. IV. Título.

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

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 Dissertação de Mestrado de **ROGÉRIO DE CARVALHO SILVA** intitulada: **Machine Learning for Change-Prone Class Prediciton: A History-Based Approach**, 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 mestre 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, 24 de Agosto de 2022.

Assinatura Eletrônica
25/08/2022 10:58:14.0
SILVIA REGINA VERGILIO
Presidente da Banca Examinadora

Assinatura Eletrônica
25/08/2022 14:24:39.0
EDUARDO JAQUES SPINOSA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
27/08/2022 12:42:11.0
JULIANA ALVES PEREIRA
Avaliador Externo (PONTIFíCIA UNIVERSIDADE CATóLICA PUC-RJ)

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: 217668
Para autenticar este documento/assinatura, acesse https://www.prppg.ufpr.br/siga/visitante/autenticacaoassinaturas.jsp e insira o codigo 217668

Dedico este trabalho de pesquisa a toda minha família, meus pais, irmãos, esposa, amigos e todos que sempre me deram suporte nos meus estudos, trabalhos, projetos e pesquisas.

ACKNOWLEDGEMENTS

Agradeço a conclusão deste trabalho a minha família, esposa e amigos que sempre me apoiaram e estiveram ao meu lado em todos os momentos.

Agradeço a minha orientadora Silvia Vergilio por toda paciência e suporte ao longo dessa jornada.

Agradeço também aos meus colegas de laboratório, Paulo Farah, Enrique Rosa, Jackson Prado, Bruno Meyer e Cristina Morimoto pela ajuda e orientações ao longo do projeto.

Agradeço ao meu amigo de longa data Leuson Mário por todo auxílio prestado quando precisei.

Agradeço a Universidade Federal do Paraná pelo suporte e infra-estrutura, e professores e funcionários envolvidos nos trabalhos de pesquisa e ensino da Universidade.

RESUMO

À medida que os projetos de software evoluem, novos artefatos são criados, modificados ou removidos. Um dos principais artefatos gerados no desenvolvimento de software orientado a objetos é a classe. A classe tem um ciclo de vida muito dinâmico em projetos de software orientados a objetos. Classes são criadas, modificadas ou removidas devido a diferentes motivos, como por exemplo a necessidade de refatoração, resultando em custos adicionais ao projeto. Uma maneira de mitigar isso é detectar, logo nos estágios iniciais do projeto, classes que estão propensas a mudanças. Isso pode impactar positivamente na produtividade da equipe, na alocação de recursos e na qualidade do software desenvolvido. Vários trabalhos na literatura tentam prever a propensão a mudanças de uma classe. Alguns deles adotam abordagens baseadas em estatística, outros usam aprendizado de máquina (AM), muitas vezes coletando métricas de classe ou informações relacionadas a *code smells* como base para previsões. O problema com essas abordagens tradicionais é que elas não consideram a dependência temporal entre diferentes versões da classe ao longo do projeto, isto é, elas consideram que as instâncias de treinamento são independentes. Para preencher essa lacuna, este trabalho apresenta uma abordagem para predizer classes propensas a mudanças baseada no histórico de mudanças da classe. A abordagem utiliza o método de janela deslizante e adota como preditores, métricas estruturais, evolutivas, bem como frequência e diversidade de *code smells*. O trabalho também explora algumas técnicas de *resample* que geram dados sintéticos para aumentar o conjunto de dados e melhorar a capacidade preditiva dos modelos. Cinco projetos e quatro algoritmos de AM são usados na avaliação. Na grande maioria dos casos, a abordagem proposta supera a abordagem tradicional considerando a maioria dos indicadores. Além disso, observa-se que as medidas evolutivas desempenham um papel importante na predição de mudança em classes. Entre os quatro algoritmos de AM testados, *Random Forest* apresenta o melhor desempenho, e o uso de informações relacionadas aos *code smells* não impacta os resultados.

Palavras-chave: Classes propensas a mudança, aprendizado de máquina, dependência temporal, evolução de software, manutenção de software

ABSTRACT

As software projects evolve, new artifacts are created, modified or removed. One of the main artifacts generated in the development of object-oriented software is the class. Classes have a very dynamic life cycle in object-oriented software projects. They can be created, modified or removed due to different reasons, e.g. need of refactoring, which results in additional costs to the project. One way to mitigate this is to detect, in the early stages of the project, classes that are prone to change. This can positively impact the team's productivity, the allocation of resources and the quality of the software developed. To predict the change proneness of a class, most works in the literature adopt approaches based on statistics, others use machine learning (ML), often collecting class metrics or smells information as the basis for predictions. The problem with these traditional approaches is that they do not consider the temporal dependency between different versions of the class throughout the project, that is, they consider the training instances are independent. To fill this gap, this work presents an approach for predicting class change proneness, based on the history of the class changes. The approach uses the sliding window method and adopts structural and evolutionary metrics as predictors, as well as the frequency and diversity of code smells. The work also explores some resample techniques to generate synthetic data to increase the dataset and improve the predictive ability of the models. Five projects and four ML algorithms are used in the evaluation. In the great majority of the cases, our approach overcomes the traditional approach considering most of the indicators. We also highlight that evolutionary measures play an important role in class change-prone prediction. Among the four ML algorithms tested, Random Forest has the best performance, and the use of smell-based information does not impact the results.

Keywords: class change proneness, machine learning, temporal dependency, software evolution, software maintenance

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADA	Adaptive Synthetic
ANN	Artificial Neural Network
CPCP	Change-Prone Class Prediction
DL	Deep Learning
DT	Decision Tree
GRU	Gated Recurrent Unit
LR	Logistic Regression
LSTM	Long Short-Term Memory
MDI	Mean Decrease in impurity
ML	Machine Learning
MLP	Multilayer Perceptron
RF	Random Forest
ROC	Receiver Operating Characteristic Curve
ROS	Random Over-Sampler
RUS	Random Under-Sampler
SLOC	Source Lines of Code
SMOTE	Synthetic Minority Oversampling Technique
SVM	Support Vector Machine
SW	Sliding Window

CONTENTS

1 INTRODUCTION

Software projects are constantly evolving. As a consequence, artifacts are created, edited, or removed over time. One of the most important artifact produced in the object-oriented software development is the class. Through classes, software components are structured, and the business logic is implemented (Lindvall, 1998).

Classes may change throughout the project due to many reasons (Malhotra and Khanna, 2019): bugs resolution, implementation of new features, code refactoring or even technology update. As a result of this continuous evolution, maintaining project quality is expensive, demanding time from the development team and increasing project costs (Benestad et al., 2010).

Change-prone classes are the ones that are likely to change across different versions of a software product. Along the software evolution, a class is subject to different kinds of changes (Malhotra and Khanna, 2019). One kind of change regards the addition of new functionalities. Classes related to the implementation of essential features of the systems are more likely to change in the future versions. With the addition of new functionalities, a class becomes more complex and fault-proneness, increasing the probability of changes to fix bugs. This change process possibly erodes the original design with a subsequent reduction of the quality class structure, by introducing smells (Catolino et al., 2020), what leads to refactoring necessity and future changes to improve the overall quality of the class structure.

Thus, managing and controlling class changes is fundamental; the early detection of classes that may change in future versions can support technical leaders and managers to take decisions in an objective and anticipated way, mitigating, for example, that poorly designed classes remain implemented in the project. This problem is known in the literature as the *Change-Prone Class Prediction (CPCP)* problem (Malhotra and Khanna, 2019).

Previous work addresses the problem using different approaches, Malhotra and Bansal (2016) performed a systematic review of the literature on predicting changes in software. Among the results, the authors report available studies in the area of change proneness in software projects. The systematic review shows that more than 50% of the works use class metrics as a basis for the analysis and that about 25% use mixed approaches, that is, class metrics combined with other approaches, such as code smells. The authors highlight that machine learning is a promising approach and is a tendency, adopted in more recent works.

Godara et al. (2018) and Kumar et al. (2019) investigate the use of metrics to predict change-prone classes. While the work of Godara et al. (2018) relied on the bee colony algorithm (Karaboga, 2005) to perform its predictions, Kumar et al. (2019) addresses the use of machine learning. The work of Kumar et al. (2019) used 18 machine learning algorithms, including Logistic Regression (LR), Decision Tree (DT), Support Vector Machine (SVM), among others. The results show that code metrics are useful in predicting changes in software modules and that the results may vary according to the choice of metrics.

Pritam et al. (2019b) and Khomh et al. (2011) investigate the relationship of code smells with the class change proneness through machine learning and statistical analysis. As a result, the authors point out that classes with more code smells are more likely to change. These works most often use metric values associated with the class in a software version to predict its change proneness, not taking into account its history, that is, how these values have changed over the course of the development of different past versions of the software.

The works of Melo (2020) and Melo et al. (2020a) consider the temporal aspect between the instances, and time windows, proposing two representations based on time series, called:

Concatenated and Recurrent. The first allows the use of traditional machine learning algorithms such as Logistic Regression, Decision Tree, Random Forest (RF), and Multilayer Perceptron (MLP) and the second allows the use of *Deep Learning* algorithms (DL). For example, the Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) algorithms. The work, however, is limited to 3 repositories. Moreover, the work considers only structural metrics and a simple approach based on lines of code to define the change prone classes.

1.1 MOTIVATION

From the context described above, we can then observe that the CPCP problem encompasses different situations and is a consequence of many causes during the software evolution process. These causes are inter-related. The addition of a functionality makes the class more complex and this can lead to bug introduction; the bug fixing can lead to an erosion of the class structure by introducing a smell; smells should be removed through refactoring. To capture all these changes that a class may suffer different metrics need to be used. Moreover, we observe that these changes occur over the time as a sequence, that is, they are time-dependent.

All these characteristics make CPCP a problem largely studied in the literature (Godara and Singh, 2014; Malhotra and A., 2015; Malhotra and Khanna, 2019). The great majority of approaches uses Machine Learning (ML) to derive the prediction model. ML techniques overcome statistical methods (Malhotra and Khanna, 2013). A great number of algorithms have been used, but supervised ones are the preferred, and Random Forest has presented the best results in most studies (Malhotra and Khanna, 2019).

Existing techniques employ features related to different aspects and dimensions of the software development, each one of them is captured by a particular set of software class metrics. As a consequence, different models exist, taking into account distinct kinds of metrics to be used as independent variables (predictors). Structural models use metrics such as size, complexity, and coupling (Lu et al., 2012; Malhotra and Khanna, 2021). Evolutionary models uses process metrics that capture some evolution aspects of the class (Al-Khiaty et al., 2017; Catolino and Ferrucci, 2018, 2019; Catolino et al., 2018; Elish and Al-Rahman Al-Khiaty, 2013; Malhotra and Khanna, 2018a,b). Smell-based models uses metrics based on the design and structure quality, such as intensity of smells or anti-patterns (Catolino et al., 2020; Kaur and Jain, 2017; Pritam et al., 2019a).

The works mentioned above have shown that the use of different kinds of metrics contributes to improve the performance of the ML models in different degrees. But existing approaches present some limitations, as stated by Malhotra and Khanna (2019). Their review discusses trends and challenges to be investigated regarding CPCP. One challenge mentioned is the necessity of assessing the capability of all these metrics as different predictors, used (or not) in a combined way, and taking into account the temporal dependency between the classes changes.

1.2 GOALS

Given the context and motivation presented above, this work introduces a ML approach for the CPCP problem that considers the class change history. The introduced approach uses the *Sliding Window (SW)* method (Dietterich, 2002). This method allows evaluating the dependency that exists between different instances of a class in a time window. In this way, the approach better captures the class change history and avoids that the prediction is impacted only by the current metrics values. In addition to this, the approach is elaborated using different kind of metrics as predictors: structural, evolutionary and smell-based. The approach is evaluated with five applications extracted from the GitHub repositories and compared with a traditional approach.

The results show that the history-based approach using the Sliding Window method improves significantly the prediction performance. We also observed that the use of sliding window size of 3 obtained a better result than windows with sizes 2 and 4. Additionally, smell-based metrics did not prove to be an impactful prediction feature, and Random Forest achieved the highest performance of the evaluated algorithms.

In short, this work has as main contribution to present and evaluate a history-based approach for the CPCP problem. This approach allows considering the structure of the problem and the dependency between the instances obtained from different releases. This leads to a better performance. Our approach overcomes the traditional approaches in the great majority of the cases and for all indicators. The correct detection of change-prone classes can allow developers better focus on these change-prone parts to minimize the number of future changes, to guide the maintenance team, and resource distributions, in order to reduce future efforts and costs. Moreover we evaluate the impact of using smell-related information in our history-based approach and the performance of four algorithms, commonly adopted in the literature.

1.3 WORK ORGANIZATION

The remainder of this document is divided into chapters, organized as follows. Chapter 2 addresses the theoretical foundation related to our work, bringing important concepts to the reader. Chapter 3 presents related work. In Chapter 4 our approach is described. Chapter 5 describes the experiment that was conducted, the methodology, research questions and results. Chapter 6 presents concluding remarks and future work.

2 THEORETICAL BASIS

This chapter presents concepts about important themes of this dissertation. Section 2.1 presents the overall concepts of machine learning. Section 2.2 presents evaluation measures adopted when evaluating machine learning models. Section 2.3 presents the concepts for resampling techniques. Section 2.4 presents the factors related to class change.

2.1 MACHINE LEARNING

Machine learning (ML) is a sub-area of artificial intelligence, which seeks to learn and make predictions about a set of data (Bishop, 2006). Machine learning methods can be categorized into two main areas, supervised learning and unsupervised learning.

Unsupervised learning has no prior knowledge of the data. In turn, it tries to find and group data patterns through statistical analysis. For example, it is possible to identify the profile of customers in online stores. Unsupervised learning can also be used to reduce the dimensionality of data, seeking to maintain as much integrity as possible. For example, it is complicated to use a five-dimensional dataset. To solve this problem, it is possible to reduce the 5-dimensional dataset to a 3-dimensional dataset, where the data can be visualized and interpreted in the form of a three-dimensional graph.

Unlike unsupervised learning, supervised learning uses a pre-labeled dataset to train models and make predictions. Within supervised learning there are two types of problems: Classification and Regression (Kubat, 2015). In the classification task, the classifier aims to predict pre-established categories. For example, true or false, dog or cat. On the other hand, regression problems aim to predict real values. For example, the price of a property or the rate of population growth. Examples of supervised learning algorithms are *Decision Tree* (DT) and *Artificial Neural Network* (ANN) (Kubat, 2015). It is also possible to use multiple algorithms on supervised learning problems. Techniques that use this strategy are called *ensemble methods* (Bengio et al., 2017). In summary, the models are trained using different algorithms and the models vote on the output for test examples. Examples of ensemble methods are bagging, boosting and Random Forest algorithms (Livingston, 2005).

This work addresses a classification problem, since one of the objectives is to predict whether a class will changes in future versions. To this work, we are interested in four supervised learning techniques: Random Forest, Multi Layer Perceptron, Linear Regression and Decision Tree. These techniques were also adopted in related work (Malhotra and Khanna, 2013), (Malhotra and Khanna, 2019), (Catolino et al., 2020).

2.1.1 Multi Layer Perceptron

The Multi Layer Perceptron (MLP) is a type of Artificial Neural Network (ANN) (Riedmiller, 1994). ANNs are computational structures based on the human central nervous system, specifically the brain. This type of model is composed of nodes, called artificial neurons, also known as Perceptron (Rosenblatt, 1958), which are connected with each other creating a network where they transmit signals between them, simulating synapses of the brain.

The MLP structure is typically composed by three layers of nodes: an input layer, a hidden layer and an output layer, as shown in Figure 2.1. Information flows from the input layer through the intermediate layers to the output layer. For information to flow, neurons are interconnected and can perform mathematical operations based on pre-established conditions, passing information to other neurons in the network.

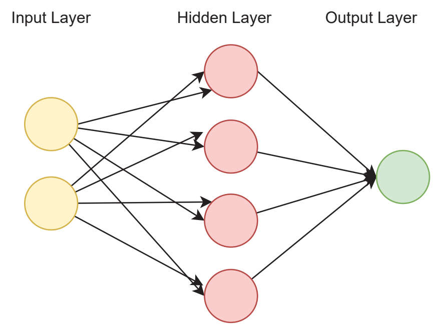


Figure 2.1: MLP Example

2.1.2 Logistic Regression

The Logistic Regression (LR) algorithm is based on statistical models and is often used for classification (Kubat, 2015). It estimates the probability of an event to occur based on a given data set of independent variables. The target has a binary nature and therefore the result will be, for example, true or false.

The LR calculates the probability of an event, so its value varies between 0 and 1, and cannot exceed this limit, thus generating a sigmoid function S also called logistic function. To classify the late, the concept of limit value is used. Values above the threshold value tend to true and a value below the threshold values tend to false, as illustrated on Figure 2.2.

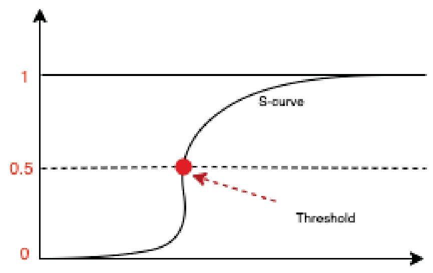


Figure 2.2: Logistic Regression Example

2.1.3 Decision Tree

Decision trees (DTs) are widely used algorithms in ML. The tree is composed of nodes and leaves, always starting from a root node, as shown on Figure 2.3. Each node represents a decision point that can lead to one path or another. These paths are known as branches. At the end of each branch there is a leaf, which indicates the result of the target variable. In a tree, there may be several options for different paths to be followed, each path leading to a result.

2.1.4 Random Forest

Random Forests (RF) is an ensemble learning method that can be applied for classification or regression (Livingston, 2005). The technique combines multiple decision trees which perform individual predictions. The main difference between the Random Forest and the Decision Trees classifier is that the Random Forest, instead of creating a single complex tree, creates several smaller trees and combines their result. The results of each prediction are compared against each other and a vote is taken, choosing the majority or average value (Livingston, 2005).

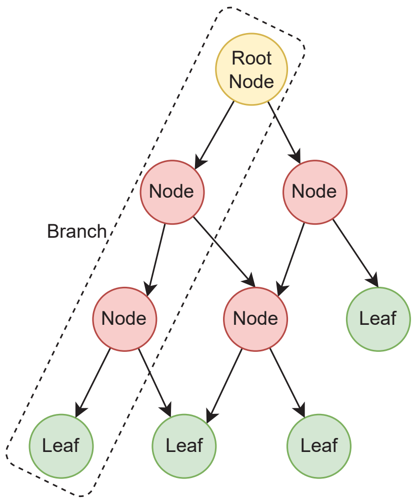


Figure 2.3: Decision Tree Example

Each tree generated internally by the RF algorithm may contain different paths to find the solution to the problem. Figure 2.4 abstractly illustrates how the algorithm works. From a dataset, three decision trees are generated, each tree is executed independently, and its result is computed. Finally, the algorithm considers the average or majority of the results to decide the output.

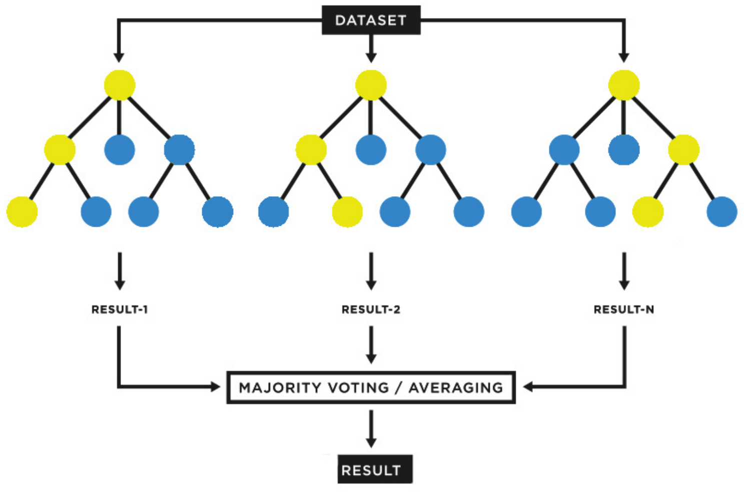
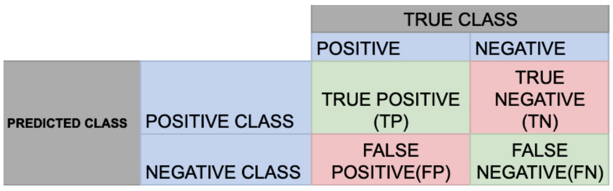


Figure 2.4: Random Forest Example

2.2 EVALUATION MEASURES

To evaluate the performance of the models, some metrics can be applied. The main ones are precision, sensitivity, f1-score, accuracy and Receiver Operating Characteristic Curve - ROC.

To understand these measures, it is first necessary to understand the concept of confusion matrix, which is a way of visualizing the errors and successes generated by the classifiers (Stehman, 1997). The confusion matrix summarizes true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). As shown in Figure 2.5, the TP indicates when the model correctly predicts the positive class. The TN indicates when the model correctly predicts the negative class. The FP indicates that the model has incorrectly predicted the positive class. The FN indicates that the model has incorrectly predicted the negative class.



		TRUE CLASS	
		POSITIVE	NEGATIVE
PREDICTED CLASS	POSITIVE CLASS	TRUE POSITIVE (TP)	TRUE NEGATIVE (TN)
	NEGATIVE CLASS	FALSE POSITIVE(FP)	FALSE NEGATIVE(FN)

Figure 2.5: Confusion matrix

Given this the confusion matrix, the metrics can be defined as follows:

- *Accuracy*:
$$\frac{TP+TN}{TP+TN+FP+FN} \tag{2.1}$$
True Positive (TP), True Negative (TN), False Positive (FP), False Negative (FN).

- *Sensitivity*:
$$\frac{TP}{TP+FN} \tag{2.2}$$
Proportion of actual positives that were identified correctly.

- *Precision*:
$$\frac{TP}{TP+FP} \tag{2.3}$$
Proportion of positive identifications that were identified correctly.

- *F1-Score*:
$$2 * \frac{precision * sensitivity}{precision + sensitivity} \tag{2.4}$$
Harmonic mean between precision (proportion of positive identifications) and sensitivity; it is a way of evaluating precision and sensitivity in a single metric;

- ROC: Graph that shows the representation of the rate of true positives by the rate of false positives. The ROC curve is usually analyzed using the Area Under the Curve (AUC), which is a measure of performance for classification problems at various threshold settings. This metric is more suitable for unbalanced data (He and Ma, 2013).

2.3 RESAMPLING TECHNIQUES

In real world problems (non-trivial problems) there may be an imbalance between the variables to be predicted. This imbalance can make the models to bias around the class that has a higher frequency, generating erroneous results. To mitigate this, there are two state-of-the-art techniques: Undersampling and Oversampling.

Undersampling consists on reduce the number of samples in the majority class in order to balance with the minority class. We can highlight three state-of-art techniques:

- Condensed Nearest Neighbors (CNN) (Hart, 1968), looks for a consistent minimum set, that is, a subset of a collection of samples that does not lead to a loss of model performance;

- Tomek Links method (Tomek, 1976), is a modification of the CNN method, which consists of finding a pair of near neighbors, each pair contains a majority class and a minority class. The examples in the majority class that are closest to the minority class, can be removed;
- Random Under-Sampler (RUS)(He and Ma, 2013), randomly picks instances of the majority class and removes them.

Oversampling consists of generating synthetic data in order to balance the proportion of data. We can highlight three commonly applied techniques:

- Synthetic Minority Oversampling Technique (SMOTE)(Chawla et al., 2002), generates new synthetic instances obtained by combining the features of an instance and its k-nearest neighbors;
- Random Over-Sampler (ROS)(Batista et al., 2004), randomly chooses an instance from the minority class and repeats it;
- Adaptive Synthetic (ADA)(He et al., 2008), is similar to SMOTE, but generates different number of samples depending on an estimate of the local distribution of the class to be oversampled.

2.4 FACTORS RELATED TO CLASS CHANGE

Iterative and incremental software projects have a very dynamic lifecycle, artifacts are created, changed or deleted as new versions of the project are created. This occurs naturally throughout the life of the project (Benestad et al., 2010). Class changes occur constantly due to the evolving nature of software projects. Classes may change due to malfunction, addition/removal of features or the need to refactor them.

Software refactoring aims to improve the maintainability and performance of software by changing the internal structure of the code, without modifying its external behavior. Fowler and Beck (1999) identified, in the object-oriented context, a set of simple code restructurings that aim to improve code quality. For example, renaming methods or separating classes. Other refactorings seek to remove *Code Smells*. *Code Smells* are code patterns that indicate there are problems with the code, they are symptoms of poor design and poor implementation choices. *Smells* may not affect the behavior of the software, but bring quality and maintainability risks as the system evolves. The presence of *smells* in the project indicates that there may be a better way to rewrite the code for quality and maintainability. The presence of *Code Smells* in classes may indicate that they are candidates for refactoring in future versions.

2.5 CONCLUDING REMARKS

In this chapter, the essential topics were presented to contextualize the work developed in this dissertation. Machine learning concepts and algorithms were presented as well the main evaluation measures. This chapter also introduced the concepts of resampling and factors related to class change.

In the next chapter, research in the area of change-prone classes is presented, describing existing work related to the theme of this dissertation.

3 RELATED WORK

This chapter presents works that address the main themes related to this dissertation. Section 3.1 describes the CPCP problem and Section 3.2 presents some approaches adopted in the literature to deal with this problem.

3.1 THE CPCP PROBLEM

The Change-Prone Class Prediction (CPCP) problem concerns the identification of classes that are most prone to change in order to reduce the cost and resource consumption in activities such as refactoring, peer-code reviews and software maintenance (Elish and Al-Rahman Al-Khiaty, 2013; Catolino et al., 2017). To define if a class changed, there are many methods. 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 (Malhotra and Khanna, 2019; Arisholm et al., 2004; Kaur and Jain, 2017; Khanna et al., 2021; Koru and Tian, 2005; Lu et al., 2012; Malhotra and Khanna, 2021; Martins et al., 2020; Zhou et al., 2009).

Other studies (Bieman et al., 2003; Eski and Buzluca, 2011; Massoudi et al., 2021) consider structural changes in the code elements, in which they extract changes comparing the abstract syntax tree between two versions. This approach usually uses the *ChangeDistiller* tool. A description of the tool and the complete list of operations are presented in (Fluri et al., 2007; Gall et al., 2009). This method is known as *Fine-Grained Changes* (FGC) in the literature. Some works (Catolino et al., 2020; Romano and Pinzger, 2011) 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.

The SLOC-based method identifies that a class has changed even if only a few SLOCs have been changed. This ensures that all changes will be detected. On the other hand, this method has as disadvantages to have a huge quantity of varied types of changes, including even trivial changes and do not capture the details about the semantics of changes (Giger et al., 2012). The FGC method identifies which kind of the code structure has changed (loop, control structure, statement) and can provide a more detailed information about the changes on the level of statement and declaration changes. The drawback of the FGC method is that it is more complex to implement and can let some small changes not be considered. Taking into account the robustness of method, in this work we adopt the FGC method. The method is also adopted in other works (Catolino et al., 2020; Romano and Pinzger, 2011), its adoption allows comparison with state-of-the-art approaches.

Many works addressing the Change-Prone Class Prediction (CPCP) problem are found in the literature. Due to its importance, the problem has been subject of surveys and systematic reviews (Godara and Singh, 2014; Malhotra and A., 2015; Malhotra and Khanna, 2019). These studies show that the great majority of approaches use *Machine Learning (ML)* to derive the prediction models. ML overcome statistical methods (Malhotra and Khanna, 2013). A great number of algorithms have been used, but supervised ones are the preferred. Logistic regression, MLP, Decision Trees, and Random Forest are the most employed and evaluated algorithms. Random Forest has presented the best results in most studies (Malhotra and Khanna, 2019).

3.2 APPROACHES TO THE CPCP PROBLEM

As mentioned before, the CPCP problem encompasses different aspects and dimensions of the software development. Existing ML approaches adopt as predictors different kind of metrics (Malhotra and Khanna, 2019). Table 3.1 contains a brief description of the approaches adopted by previous works. The categories of static metrics are: structural, evolutionary, network, developer-related, word vector, code smells and bugs.

Table 3.1: Software metrics used for the CPCP problem.

Metric Category	**Description**	**Studies**
Static metrics		
Structural	These metrics are software internal quality metrics, which depict structural attributes of Object-Oriented elements as size, coupling, complexity, cohesion, etc. These structural metrics include suites such as CK (Chidamber and Kemerer, 1994), QMOOD (Bansiya and Davis, 2002), among others.	All
Evolutionary	These metrics quantify the change history of a software evolution, from one release to another, e.g., Birth of a class, amount of changes, frequency of changes, change density, and others.	(Al-Khiaty et al., 2017; Catolino and Ferrucci, 2018, 2019; Catolino et al., 2018; Elish and Al-Rahman Al-Khiaty, 2013; Malhotra and Khanna, 2018a,b)
Developer-related	Entropy of changes introduced by a developer in a time of period, number of developers employed on an specific segment in a specific time, structural and semantic scattering of developers in a specific time period.	(Catolino and Ferrucci, 2018, 2019; Catolino et al., 2017, 2018)
Code Smells	These metrics are related with poor Object-Oriented design structures, which can cause source code degradation as code smells (Fowler, 1999), antipatterns (Brown et al., 1999) and identity disharmonies (Lanza and Marinescu, 2010).	(Kaur and Jain, 2017; Pritam et al., 2019a)
Others	Less used metrics based in word vector method, the dependency graph of the software and bugs collected in issue tracking repositories.	(Alhawi and Abdilrahim, 2020; Giger et al., 2012; Zhu et al., 2018)

The structural approaches use product-based metrics that capture structural aspects of the classes, such as size, complexity, coupling, inheritance, and cohesion (Lu et al., 2012;

Malhotra and Khanna, 2021; Catolino et al., 2020; Godara and Singh, 2014; Khomh et al., 2011; Lu et al., 2012; Malhotra and A., 2015; Malhotra and Khanna, 2019; Tsantalis et al., 2005). They consider that more complex classes are more change-prone. The evolutionary approaches use process-based metrics that capture some evolution aspects of the class, such as birth date, change frequency and density (Al-Khiaty et al., 2017; Catolino and Ferrucci, 2018, 2019; Catolino et al., 2018; Elish and Al-Rahman Al-Khiaty, 2013; Malhotra and Khanna, 2018a,b). The smell-based approach uses metrics based on the design and structure quality, such as intensity of smells or anti-patterns (Catolino et al., 2020; Kaur and Jain, 2017; Pritam et al., 2019a; Khomh et al., 2009). There are approaches based on information extracted from the class text vocabulary or the class dependency graph, and other approaches are developer-based and use metrics related to the number of developers working in a time period, or related to social and other aspects of their communication (Zhu et al., 2018; Giger et al., 2012; Zhu et al., 2018; Catolino and Ferrucci, 2018, 2019; Catolino et al., 2017, 2018).

Although there are a large number of approaches, few studies compare their performance or use them in a combined way (Malhotra and Khanna, 2019). There is also a lack of benchmarks, because the metric values are collected in different contexts and ways. Studies do not show a consensus on the use of structural metrics, despite all CPCP researches use them. Tsantalis et al. (2005) concluded that structural metrics are insignificant features for CPCP, except class size. Lu et al. (2012) analyzed the highest number of systems in literature and confirmed that size structural metrics have moderate prediction capacity. On the other hand, Zhou et al. (2009) indicate that size metrics present a strong confound effect and are not good predictors.

On the other hand studies show that classes with code smells are more prone to change than classes without smells (Kaur and Jain, 2017; Pritam et al., 2019a). The work of Catolino et al. (2020) shows that the adoption of code smell-related information improves the performance of different categories of prediction models performance: structural, evolutionary and developer-based. The results show that the smell intensity is a better predictor than anti-pattern metrics and that a combined change prediction model including product, process, developer-based, and smell-related features presents notably higher performance than other models.

A limitation of existing approaches is that they do not take into account the CPCP problem structure, which resembles a temporal sequence. Some works consider class evolution aspects and history (Al-Khiaty et al., 2017; Catolino and Ferrucci, 2018, 2019; Catolino et al., 2018; Elish and Al-Rahman Al-Khiaty, 2013; Malhotra and Khanna, 2018a,b) to derive the predictors, but the problem of these and all the works mentioned above is that they use as training inputs each instance individually and do not take temporal dependencies between them. Some initiatives exist to model the problem by using time series (Caprio et al., 2001; Melo et al., 2020b). The work of Caprio et al. (2001) employs ARIMA to model the problem. The work of Melo et al. (2020b) introduces two representations to capture the temporal aspect, called by the authors concatenated and recurrent. They evaluated the first one with conventional ML algorithms and the second representation is used with *Recurrent Neural Network (RNN)* based on the *Gated Recurrent Units - GRU* architecture. The problem with these two works is that they use a limited number of independent variables. They do not combine and diversify metrics, using only a few structural metrics. Another problem is that to define the change prone class as simple SLOC approach is used. This approach works with structural metrics, but would create a bias when working with evolutionary metrics, as evolutionary metrics already capture these SLOC-related changes in their nature. The mentioned limitations make not feasible a direct comparison between our work and the work of Caprio et al. (2001) or Melo et al. (2020b).

3.3 CONCLUDING REMARKS

In this chapter we discuss the concepts involved in the CPCP problem (Section 3.1), we also discuss how related works approach the problem, Section 3.2. In this dissertation, we apply an approach that considers the temporal factor in CPCP problems, but differently from related work, our approach uses different kinds of features in a combined way, and adopts the FGC method to define a class-change, similarly to the work of Catolino et al. (2020). This allows comparison with a traditional state-of-the-art approach. The approach we propose is described in the next chapter.

4 PROPOSED APPROACH

Based on the content and works described in the previous chapter, this chapter introduces a history based approach to Change-Prone Class Prediction (CPCP). The chapter is organized as follows: Section 4.1 presents an overview of our approach. Section 4.2 describes the independent variables, Section 4.3 describes how the dependent variable is calculated and extracted. Section 4.4 describes how the data is represented.

4.1 OVERVIEW

Figure 4.1 presents a summarized overview of the approach. We divided the approach in five main steps. Initially, we studied the CPCP literature and identified relevant metrics to act as predictors. To choose the dependent and independent variables, we followed the work of Catolino et al. (2020) and we used the metrics in a combined way. We selected the following kind of metrics: structural, evolutionary and smell-related. On the second step, we created the dataset by mining public repositories of the GitHub and collected the metrics using static analysis tools. In the third step, we conducted a data preprocessing. We first built an input representation for the ML algorithms, by adopting the Sliding Window (SW) method (Dietterich, 2002), and we applied some methods for data normalization and balancing.

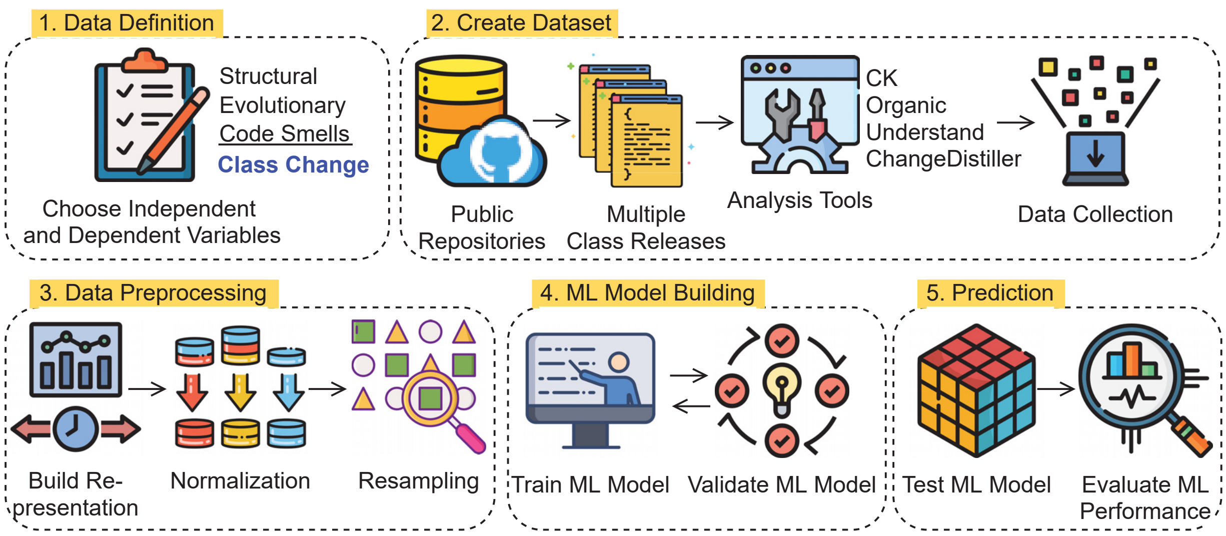


Figure 4.1: Overview of our approach

On the fourth step, after the preprocessing, we used the resulting dataset to train and test the ML models. On the last step, we evaluated the performance of the ML algorithms. These steps were adopted in all experiments of each project individually. More details about the independent and dependent variables used, as well as the representation adopted as input for the ML algorithms are provided next.

4.2 INDEPENDENT VARIABLES

The independent variables of our approach comprises metrics from three categories (Malhotra and Khanna, 2019), also used by the traditional approaches (Catolino et al., 2020). They are: i) structural: these metrics are related to software internal quality metrics, which depict structural attributes of Object-Oriented elements. These structural metrics include suites such as CK (Chidamber and Kemerer, 1994), QMOOD (Bansiya and Davis, 2002), among others; ii) evolutionary: these metrics are related to the software evolution from one release to another, e.g., birth of a class, amount of changes, frequency of changes, change density, and so on. It is worth mentioning that, unlike our approach, the changes captured by the evolutionary metrics are related to a simple SLOC change between two versions, whereas our approach is able to test longer horizons and with a more complex change definition; and iii) smell-based: these metrics capture poor Object-Oriented design structures, which can cause source code degradation as code smells (Fowler, 1999) and anti-patterns (Brown et al., 1999; Lanza and Marinescu, 2010), (Kaur and Jain, 2017; Pritam et al., 2019a). We employed two smell-based metrics which are density and diversity of smells. Density calculates the number of smells found in the class, diversity calculates how many types of smells the class has.

Table 4.1 presents the structural metrics covered in this work, presenting the metric category, name, brief description and source. To extract the structural metrics, we use two extraction tools Understand[1] and CK (Aniche, 2015). Understand is a static code analyzer which collects product metrics of code elements. CK is a tool that calculates class-level and method-level code metrics in Java projects.

Table 4.2 presents the name and description of the code smells covered in this work to calculate density and diversity. To extract these smells, we use the tool Organic[2]. Organic can identify all code smells considered in this research. To extract density and diversity, from the data collected, we built our own tool, available on our replication package (Silva, 2022).

Table 4.3 presents the software evolution-based metrics proposed by Elish and Al-Khiaty (Elish and Al-Rahman Al-Khiaty, 2013). We could not find an updated tool to extract these metrics, thus, we implemented our own tool, available on our replication package (Silva, 2022).

Table 4.1: Structural metrics.

Metric	Definition	Source
Cohesion metrics		
LCOM2	**Lack of cohesion in methods**. The number of pairs of methods in the class using no common attributes minus the number of pairs of methods that do. If this difference is negative, then, LCOM2 is set to zero.	(Chidamber and Kemerer, 1994)
LCOM3	**Lack of cohesion in methods**. Treats each method pair as an individual entity, and determines the difference between the amount of similar and different pairs.	(Li and Henry, 1993)
TCC	**Tight class cohesion**. The percentage of pairs of public methods of the class which are directly connected methods. Two methods are called connected, if they use common attributes, directly or indirectly.	(Bieman and Kang, 1995)
LCC	**Loose class cohesion**. The percentage of pairs of public methods of the class which are directly or indirectly connected. If there are methods $m_1, \ldots, m_n$, such that m_i and m_{i+1} are connected for $i = 1, \ldots, n-1$, then m_1 and m_n are indirectly connected.	(Bieman and Kang, 1995)

Continued on next page

[1]https://www.scitools.com/

[2]https://github.com/opus-research/organic

Table 4.1 – continued from previous page

Metric	Definition	Source
Coupling metrics		
CBO	**Coupling Between Object Classes**. A count of the number of other classes to which it is coupled.	(Chidamber and Kemerer, 1994)
RFC	**Response For a Class**. A set of methods that can potentially be executed in response to a message received by an object of that class.	(Chidamber and Kemerer, 1994)
FANIN	**Fan-in**. The number of external classes that invoke methods from the analyzed class.	(Henry and Kafura, 1981)
FANOUT	**Fan-out**. The number of external method invocations made by the analyzed class	(Henry and Kafura, 1981)
Complexity metrics		
WMC	**Weighted Methods Per Class**. The sum of the cyclomatic complexity of the methods of a class.	(Chidamber and Kemerer, 1994)
AvgCyclomatic	**Average Cyclomatic Complexity**. Average cyclomatic complexity for all nested functions or methods.	(McCabe, 1976)
SumCyclomatic	**Sum Cyclomatic Complexity**. Sum of cyclomatic complexity of all nested functions or methods.	(McCabe, 1976)
MaxCyclomatic	**Max. Cyclomatic Complexity**. Maximum cyclomatic complexity of all nested functions or methods.	(McCabe, 1976)
Inheritance metrics		
IFANIN	**Base Classes**. The number of immediate base classes and interfaces.	(Destefanis et al., 2014)
DIT	**Depth of Inheritance Tree**. The number of nodes between the root of the inheritance tree and the analyzed class.	(Chidamber and Kemerer, 1994)
NOC	**Number of Children**. Number of immediate sub classes subordinated to a class in the class hierarchy.	(Chidamber and Kemerer, 1994)
OR	**Override Ratio**. Ratio of methods in a class that are overrides from a superclass.	(Lanza and Marinescu, 2010).
Size metrics		
NIM	**Instance Methods**. The number of instance methods in a class.	(Lorenz and Kidd, 1994)

Continued on next page

Table 4.1 – continued from previous page

Metric	Definition	Source
NIV	**Instance Variables**. The number of instance variables in a class.	(Lorenz and Kidd, 1994)
LOC	**Total lines of code**. Count all lines of executable code within the system, class, or method.	(Lorenz and Kidd, 1994)
CLOC	**Lines with Comments**. The number of lines containing comments in a class.	(Lorenz and Kidd, 1994)
NOPA	**Public Fields**. Number of Public Fields.	(Lanza and Marinescu, 2010)
STMTC	**Statements**. The number of statements in a class.	(Lorenz and Kidd, 1994)
WOC	**Weight of a Class**. The number of methods in the interface of the class, divided by the total number of interface members.	(Lanza and Marinescu, 2010)
CountSemicolon	**Semicolons**. The number of semicolons in a class.	(Scitools, 2021)
CountStmtDecl	**Declarative Statements**. The number of declarative statements in a class.	(Scitools, 2021)
RatioCommentToCode	**Comment to Code Ratio**. The ratio of comment lines to code lines in a class.	(Scitools, 2021)
CountDeclMethodDeflt	**Local Default Visibility Methods**. The number of local methods with default visibility in a class.	(Scitools, 2021)
AvgLine	**Average Number of Lines**. The average number of physical lines between methods of a class.	(Scitools, 2021)
AvgLineComment	**Average Number of Lines with Comments**. The average number of lines containing comments between methods of a class.	(Scitools, 2021)
CountDeclClassMethod	**Class Methods**. The number of class methods in a class.	(Scitools, 2021)
CountDeclClassVariable	**Class Variables**. The number of class variables in a class.	(Scitools, 2021)
NPM	**Private Methods**. The number of local private methods in a class.	(Scitools, 2021)
CountDeclMeProtected	**Protected Methods**. The number of local protected methods in a class.	(Scitools, 2021)
NPRM	**Public Methods**. The number of local public methods in a class.	(Scitools, 2021)

Table 4.2: Types of code smells (Source: Rêgo (2018)).

Metric	Definition
Brain Class	Complex classes that centralize functionalities
Brain Method	Methods that centralize the intelligence of a class
Complex Class	Classes presenting a overly high cyclomatic complexity
Data Class	A class exposes its attributes, thus violating the information hiding principle
Dispersed Coupling	Occurs when a method calls methods from a large number of provider classes
Feature Envy	A method accesses the data of another object more than its own data
God Class	One class monopolizes the processing, and other classes primarily encapsulate data
Intensive Coupling	When a method calls many other methods from a few classes
Lazy Class	Understanding and maintaining classes always costs time and money. So if a class doesn't do enough to earn your attention, it should be deleted.
Long Method	A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions
Long Parameter List	More than three or four parameters for a method
Message Chains	Message chains occur when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.
Refused Bequest	Subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions.
Shotgun Surgery	Making any modifications requires that you make many small changes to many different classes.
Spaghetti Code	Declare a number of long methods without parameters
Speculative Generality	There's an unused class, method, field or parameter.

4.3 DEPENDENT VARIABLE

The dependent variable represents if a class changed or not in a period of time, that is in our approach the time between two subsequent releases. In our work, releases are official versions of the project that have been approved. Each release is identified by the project developers through tags. To decide whether a class has changed in a given context we considered the *Fine-Grained Changes* (FGC) method, mentioned in Section 3.1. To compute the dependent variable we use the same definition of other works (Catolino et al., 2020; Romano and Pinzger, 2011). A class is change-prone if, in a period of time t, the number of structural changes of a class is higher than the median of the distribution of the number of changes experienced by all the classes of the system. For each pair of commits (c_i, c_{i+1}) of t we run the *ChangeDistiller* tool (Fluri et al., 2007). This tool implements a differencing algorithm that generates and compares the syntax tree between

Table 4.3: Software evolution-based metrics (Source: Elish and Al-Rahman Al-Khiaty (2013)).

Metric	Definition
BOC	**Birth of a Class**. The first time the class 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 class has been exposed to changes.
LCH	**Last Change**. The last time the class 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 term of releases) the class has been changed.
CHD	**Change Density**. Change density of a class C is its change size ($TACH(C)$) normalized by the size of the class (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 size of changes of the class in the past and normalizing by frequency of changes.
LCA	**Last Change Amount**. It is defined as the last change size of the class 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 class.
CSB	**Changes since the Birth**. It is computed by comparing the size of the first version of a class 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 class
ACDF	**Aggregated Change Density Frequency**. It is obtained from cumulating density of changes introduced to the class in the past, and then this accumulated amount is normalized by the frequency of changes.

two versions of a project. A description of the tool and the complete list of changes it identifies are presented in (Fluri et al., 2007). Change examples are: attribute renaming change, parameter ordering change, and other methods and classes declaration changes. It is worth mentioning that the tool ignores white space-related differences and documentation-related updates.

Note that, we compute the independent variables considering the release before the one where the dependent variable is computed, i.e., we compute 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. An example of the data collected following the chronological order is presented in Table 4.4. This representation is used by the traditional approaches found in the literature. The model would predict true in case the class should undergo through a change, or false otherwise. In this structure, each instance represents a version of the class in a given release of the project, each instance contains the metrics M as the independent variables.

4.4 DATA REPRESENTATION

In the build representation step of our approach, the data structure is changed to a representation that captures the temporal dependency between the data instances. To this end, we employ the Sliding Window (SW) method (Dietterich, 2002). 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 class changes. The method works as follows. First, we select metrics from the versions of a class C for all releases, if the number of versions (releases) of the class is less than S, the class 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 end of the releases, always following a chronological order.

In summary, the main difference between the traditional and Sliding Window methods 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 metrics history over this period. Table 4.5 shows the dataset from Table 4.4 reshaped with sliding window method with $S = 2$. Thus, the instance 0 contains metrics M_1, M_2, and M_3 from releases 0 and 1 and will predict the dependent variable $changed$ from release R, so 1 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.

Table 4.4: Traditional representation

Release	M_1	M_2	M_3	Changed
0	1	10	100	-
1	2	20	200	true
2	3	30	300	true
3	4	40	400	false
4	5	50	500	true
5	2	20	400	false

Table 4.5: Representation of Table 4.4 reshaped with sliding window method

	Release i			Release i+1			Release i+2
Instance	M_1	M_2	M_3	M_1	M_2	M_3	Changed
0	1	10	100	2	20	200	true
1	2	20	200	3	30	300	false
2	3	30	300	4	40	400	true
3	4	40	400	5	50	500	false

4.5 CONCLUDING REMARKS

In this chapter, the detailed steps of the approach were presented. Section 4.1 presented an overview of the approach. The independent and dependent variables were described in Section 4.2 and Section 4.3. Section 4.4 presented the traditional and temporal structures adopted in this work. The proposed approach has well-defined steps and can be implemented relatively easily. The biggest difficulty lies in the pre-processing of the metrics, since data from several tools are aggregated into a single dataset. In summary, the main difference between the traditional and sliding window methods is that the first 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 metrics history over this period.The next chapter presents the experiments conducted for the evaluation of our history-based approach.

5 EVALUATION DESCRIPTION

The current chapter presents the experiments conducted for the evaluation of our history-based approach compared with a traditional approach. The use of different metrics in a combined way is also evaluated, as well as the impact of the smell-based features in the performance of temporal models obtained by our approach. This chapter is organized as follows: Section 5.1 describes the Research Questions (RQs). Section 5.2 describes the repositories adopted by our study. Section 5.3 presents the generated datasets. Section 5.4 describes how the models were build. Section 5.5 presents and analyzes the results of each RQ.

5.1 RESEARCH QUESTIONS

- RQ1. **How different window sizes affect the prediction performance of the models?** In this RQ, we intend to evaluate the performance of the models produced by the most used algorithms according to different window sizes. To this end we use sizes of 2, 3 and 4 for each application. These values were adopted based on literature (Tsoukalas et al., 2020).

- RQ2. **What is the performance of our approach with respect to the performance of a traditional approach?** In this RQ, we intend to compare the performance of the algorithms using our representation based on a temporal dependency with respect to the algorithms' performance using a traditional approach. The work of Catolino et al. (2020) shows that when traditional and non-temporal approaches are adopted, the best models are obtained using a set that combines all the kinds of features. Then to answer this RQ, our approach uses the set of all features. The best value for window size pointed in RQ1 is used in comparison with a traditional approach using the same features.

- RQ3. **What is the impact of using smell-related information to predict change-proneness using our approach?** As mentioned in Chapter 3 the use of smell-based features in traditional approaches shows a performance improvement in different kinds of models (Catolino et al., 2020), but existing approaches that consider the temporal dependency do not include smell-based metrics (Caprio et al., 2001; Melo et al., 2020b). Then this RQ evaluates if the use of such metrics contributes to improve the performance of the models obtained by our temporal approach.

- RQ4. **What is the algorithm that leads our approach to produce the best results when answering all RQs?** In the literature, the algorithm Random Forest presented the best performance in most of the reports (Malhotra and Khanna, 2019). But it is relevant to investigate that this also holds for our temporal approach.

- RQ5. **Which metrics are the most important for change-proneness prediction?** We analyzed different sets of features, algorithms and resample techniques, we also compared the effects of considering smell-related information. However, it is important to analyze the individual importance of each metric. We apply two techniques to measure the importance of features in predicting change-proneness: Mean Decrease in Impurity (MDI) (Loh, 2011) and Information Gain.

5.2 TARGET SYSTEMS

We extracted features from five open-source Java applications from the GitHub [1] online repository. The choice of the datasets were based on their popularity and number of releases (at least greater than 20). We choose projects with a diverse number of classes and releases, and already used by previous change-proneness works (Malhotra and Lata, 2020; Malhotra et al., 2021). Although the number of projects is small we have a large amount of classes (111.917). For each application, we collected a subsequent number of releases. This approach led to five datasets containing from 23 up to 61 releases of each application. The applications are described in Table 5.1.

Table 5.1: Used Java Applications

Application	Releases	Samples	Avg Classes	Period
Commos-bcel	23	5383	234	Jul/2011 to Sep/2019
Commons-io	61	34556	566	Mar/2012 to Sep/2020
Junit4	26	7987	307	Nov/2012 to Feb/2021
PdfBox	57	47079	826	Jun/2018 to Mar/2021
Wro4j	50	16912	338	Mar/2010 to Nov/2021

5.3 DATASET CREATION AND PREPROCESSING

To answer RQ3 that evaluates the impact of using smell-based information in our history-based approach, we created 6 sets, as illustrated in Figure 5.1. Set 1 contains all features combined, Set 2 combines structural metrics with evolutionary metrics. Set 3 combines smell-based metrics with evolutionary metrics. Set 4 combines smell-based metrics with structural metrics. Sets 5 and 6 contain structural metrics and evolutionary metrics alone, respectively.

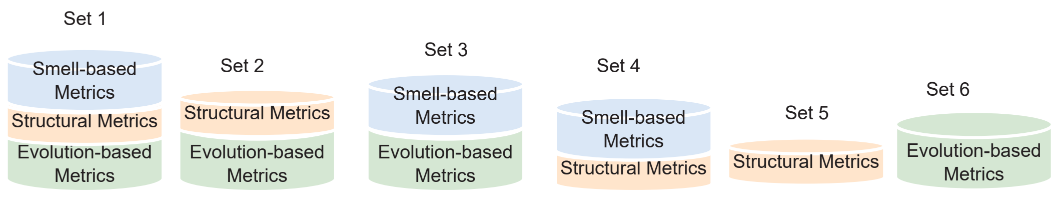


Figure 5.1: All Sets

In order to answer RQ3, in addition to the datasets using our representation (Table 4.5), we also build datasets using the representation of the traditional approaches (Table 4.4). To ensure that the data are on the same scale, they were normalized using the *min-max* technique, where the data are scaled with values ranging from 0 to 1.

After 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 chose to test three state-of-the-art oversampling techniques: Synthetic Minority Oversampling Technique (SMOTE), (Chawla et al., 2002), Random Over-Sampler (ROS) (Batista et al., 2004) and Adaptive Synthetic (ADA) (He et al., 2008). Additionally, we also tried to use undersampling techniques,

[1]https://github.com

but they further reduced our dataset until we did not have enough data left for the split step used in validation.

5.4 ML MODEL BUILDING AND PREDICTION

In the training step, the sets were divided into training (70% of the instances) and testing (30%) sets. 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. Test data are used for a final evaluation of the trained models. To increase the reliability and accuracy of the results we performed a stratified 10-fold cross-validation (Stone, 1974), which consists of dividing the training data into distinct subsets and using part of this subset for training and the rest for testing (i.e., 9 folds for training and 1 fold for testing). To implement the cross-validation we used the scikit-learn tool (Pedregosa et al., 2011).

To evaluate the performance of the algorithms, we used: Accuracy (Acc), Sensitivity (Sen), F1-Score, and Receiver Operating Characteristic Curve - ROC. For our study and answer RQ4, we employed the algorithms widely used in the literature (Malhotra and Khanna, 2019): Decision Tree (DT); Logistic Regression (LR); Multi Layer Perceptron (MLP), and Random Forest (RF). The normalization and resample methods, as well as to implement the algorithms, we used the scikit-learn tool (Pedregosa et al., 2011). All the raw data and scripts used are available in our repository (Silva, 2022).

At the end, our approach generated a total of 1440 models. Each algorithm was executed for each dataset (Set 1 up to Set 6) using three window sizes, and three or none resample techniques (4 algorithms x 6 sets x 3 windows sizes x 4 techniques x 5 applications). Default parameters of the scikit-learn tool were used. For each application 96 models were generated. Similarly, the traditional approach generated 480 models; 96 models generated for each application. All these models were used to evaluate the impact of the sliding window size (RQ1). The values of the indicators described above obtained for these models are presented in our supplementary material (Silva, 2022).

But, as mentioned before, to answer the other RQs we consider only the models adopting the best value for the window size pointed out by RQ1, as well as the best resample technique for each algorithm and application.

5.5 ANALYSIS OF THE RESULTS

5.5.1 RQ1: How different window sizes affect the prediction performance of the models?

As mentioned in the last section, we investigated three values for the window size (2, 3 and 4). There is no standard rule for optimal window size, so we empirically tested initial window sizes, starting with the smallest possible window two, as in the work of Tsoukalas et al. (2020) and Melo (2020). Larger window values did not 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 classes. By testing three values of windows, we generated a total of 288 models for each application.

To answer this RQ we first perform a statistical analysis, using Kruskal Wallis test (Kruskal and Wallis, 1952), using ROC/AUC score. Figure 5.2 presents the result and shows that the p-value was 0.927 considering all results for each size. As the results do not have difference with statistical significance between evaluated values, we decided to create a raking of the top 10 best models, in order to choose the most outstanding window size. We selected the top 10 best models from each of the 5 application, according to the ROC/AUC score. We aggregated the top 10 results from each application, thus creating a list of 50 models.

With this list, we counted the number of times each size appears. The result is presented in Figure 5.3. We can see that 20 (out of 50) of the best models uses 3 for the window size, followed by size 2 that is used in 17 models, and size 4, used in 13. Models using windows with size of 4 have almost the worst distribution in the ranking, which could be an indication that this size value is too large for our problem, when considering the parameters tested. On the other hand, models with window size 3 stand out and appear more often among the best.

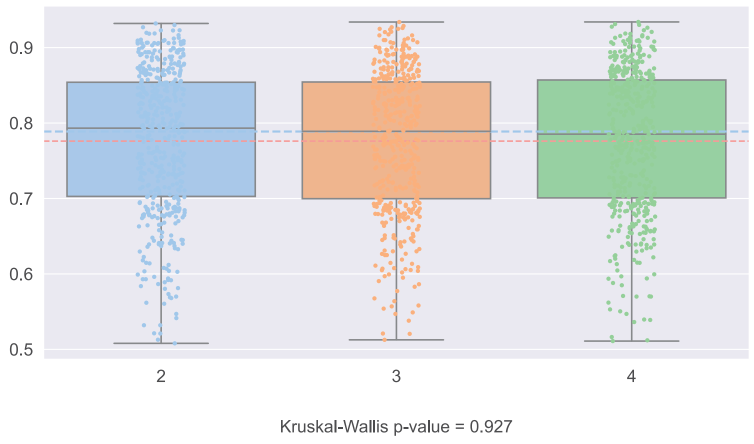


Figure 5.2: Window size statistical analysis - ROC/AUC

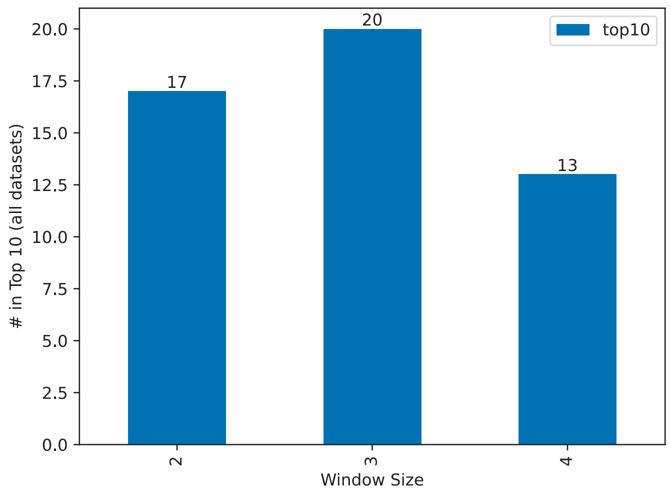


Figure 5.3: Window size ranking

Answer to RQ1: Although there was no statistical difference between the models using the different window sizes evaluated, the models with an intermediate size, equals to 3, appear most frequently among the best ones.

Implications: based on the ranking created, the results can be view as indicative of how much the change-proneness of a class depends on its history. It seems that it is not necessary a long history to reach good performance; a time period of 3 releases seems enough. But further studies should be conducted to better characterize the life of a class along the project and its relation with changes. It is expected that newborn classes are more change-prone, and otherwise for more mature classes. This finding points out new research directions.

5.5.2 RQ2: What is the performance of our approach with respect to the performance of a traditional approach?

To answer this and the next RQs, we refer to Table 5.2, which presents the results of the best models using window size equals to 3, according to RQ1. This table presents for each application and algorithm the indicator values of F1-Score, Accuracy, Sensitivity, ROC/AUC of the models obtained using our approach and the traditional one, for all sets. The columns RS correspond to the resample technique used in the model, which reaches the best performance. We can observe that for the great majority of the cases, high values of AUC imply high values of the others indicators. Moreover, except for the application `commons-io` the best values of AUC were obtained by our approach, independently of the set and algorithm used.

To a better visualization, we calculated, for all approaches, the mean values for all indicators, considering all algorithms and all sets from Table 5.2. A comparison is presented in Figure 5.4. If we consider AUC, we can see that our approach got the best results in 3 out of 5 projects. For the commons-io and commons-bcel projects, both approaches have similar results. Regarding F1-Score and Accuracy, our approach is better than the traditional approach on 4 out five systems, almost tied on commons-io dataset. The traditional approach has no outstanding results compared to our approach, having slightly better results on `commons-io`. For sensitivity indicator we observe the greatest differences for systems `Junit4`, `PdfBox`, and `Wro4j`, what represents a better performance of our approach in identifying correctly the prone-change classes.

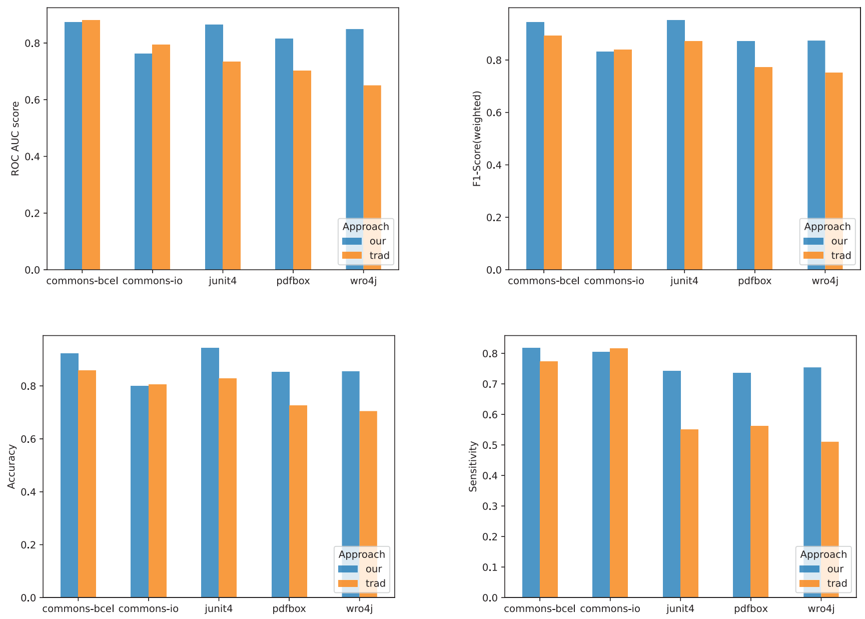


Figure 5.4: Comparing both approaches

Table 5.3 counts how many times each approach was better or obtained the same result in each set, for all evaluation measures. The results confirm that our approach obtained the best values for the indicators compared to the traditional approach. We observed that for 120 cases analyzed in Table 5.3 (4 algorithms and 5 systems and 6 sets), the models generated by our approach obtained the best values of AUC in 96, and equivalent values in 8 models. The models

Table 5.2: Best results

	DT					LR					MLP					RF					
										Commos-bcel											
	App	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**
Set 1	Our	**0,98**	**0,98**	**0,73**	**0,86**	A	**0,93**	**0,91**	**0,95**	**0,92**	S	**0,98**	**0,98**	**0,78**	**0,88**	R	**0,99**	**0,99**	**0,78**	**0,89**	A
Set 1	Trad	0,95	0,94	0,68	0,82	A	0,92	0,90	0,82	0,86	S	0,96	0,96	0,63	0,80	S	0,97	0,97	0,78	0,88	A
Set 2	Our	**0,98**	**0,98**	0,78	**0,89**	R	**0,93**	**0,90**	**0,95**	**0,92**	R	**0,98**	**0,98**	**0,76**	**0,87**	S	**0,99**	**0,99**	0,78	**0,89**	A
Set 2	Trad	0,94	0,93	**0,79**	0,86	A	0,92	**0,90**	0,83	0,87	S	0,96	0,96	0,68	0,83	S	0,97	0,97	**0,80**	**0,89**	A
Set 3	Our	**0,98**	**0,98**	0,73	0,86	A	**0,88**	**0,83**	**0,95**	**0,89**	A	**0,95**	**0,94**	0,78	**0,87**	A	**0,99**	**0,99**	**0,78**	**0,89**	R
Set 3	Trad	0,95	0,94	**0,80**	**0,87**	S	0,83	0,77	0,88	0,82	R	0,92	0,91	**0,79**	0,85	A	0,95	0,95	0,80	0,88	S
Set 4	Our	**0,96**	**0,96**	0,54	**0,76**	R	0,84	0,76	**0,78**	**0,77**	A	0,90	0,85	**0,78**	**0,82**	A	**0,97**	**0,97**	0,38	0,68	S
Set 4	Trad	0,90	0,88	**0,56**	0,73	A	**0,94**	**0,94**	0,51	0,74	S	**0,93**	**0,94**	0,22	0,60	R	0,88	0,85	**0,76**	**0,81**	S
Set 5	Our	**0,96**	**0,95**	**0,54**	**0,75**	N	0,81	0,72	**0,81**	**0,77**	A	0,88	0,82	**0,81**	**0,82**	R	**0,97**	**0,97**	0,38	0,68	S
Set 5	Trad	0,89	0,87	0,39	0,65	R	**0,93**	**0,92**	0,48	0,71	S	**0,91**	**0,91**	0,24	0,60	A	0,89	0,86	**0,73**	**0,80**	A
Set 6	Our	**0,98**	**0,98**	0,73	0,86	R	**0,87**	**0,81**	**1,00**	**0,90**	S	**0,94**	**0,92**	0,81	**0,87**	R	**0,98**	**0,98**	0,73	0,86	R
Set 6	Trad	0,94	0,93	**0,93**	**0,93**	A	0,81	0,75	0,89	0,82	R	0,88	0,84	**0,89**	0,86	S	0,94	0,93	**0,90**	**0,92**	A
										Commons-io											
	App	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**
Set 1	Our	0,94	0,93	0,83	0,89	S	**0,81**	**0,77**	**0,73**	**0,75**	R	**0,94**	**0,94**	**0,80**	**0,88**	A	**0,97**	**0,97**	0,83	**0,91**	R
Set 1	Trad	**0,96**	**0,96**	**0,86**	**0,91**	R	0,78	0,73	0,72	0,73	S	0,92	0,92	0,83	0,88	R	0,96	0,96	**0,88**	0,93	R
Set 2	Our	0,94	0,94	0,80	0,87	S	**0,81**	**0,77**	**0,73**	**0,75**	R	**0,94**	**0,94**	**0,86**	**0,90**	R	**0,97**	**0,97**	0,84	0,91	R
Set 2	Trad	**0,95**	**0,95**	**0,88**	**0,92**	S	0,78	0,74	**0,73**	0,73	S	0,92	0,92	0,78	0,86	A	0,96	0,96	**0,88**	**0,93**	R
Set 3	Our	0,89	0,88	0,64	0,78	A	**0,73**	**0,67**	**0,71**	**0,69**	S	**0,84**	**0,82**	**0,77**	**0,80**	R	0,95	0,95	0,79	0,88	A
Set 3	Trad	**0,94**	**0,93**	**0,86**	**0,90**	A	0,70	0,62	0,67	0,64	R	0,82	0,78	0,65	0,72	S	**0,96**	**0,96**	**0,90**	**0,94**	A
Set 4	Our	**0,93**	**0,93**	**0,86**	**0,90**	A	**0,76**	**0,71**	0,68	**0,70**	S	**0,90**	**0,89**	0,83	**0,86**	S	**0,95**	**0,95**	0,85	**0,91**	A
Set 4	Trad	**0,93**	**0,93**	0,83	0,89	N	0,73	0,67	**0,71**	0,68	S	0,86	0,84	**0,88**	0,85	S	**0,95**	**0,95**	**0,86**	**0,91**	S
Set 5	Our	**0,93**	**0,92**	**0,85**	**0,89**	A	**0,76**	**0,71**	0,68	**0,70**	S	**0,89**	**0,87**	0,84	**0,86**	A	**0,95**	**0,95**	0,85	**0,91**	R
Set 5	Trad	**0,93**	**0,92**	**0,85**	**0,89**	S	0,73	0,67	**0,70**	0,68	R	0,84	0,81	**0,89**	0,85	R	**0,95**	0,94	**0,86**	**0,91**	A
Set 6	Our	0,88	0,86	0,66	0,77	S	**0,69**	**0,62**	**0,70**	**0,65**	S	**0,81**	**0,78**	**0,72**	**0,75**	R	0,95	0,95	0,78	0,88	S
Set 6	Trad	**0,94**	**0,93**	**0,86**	**0,90**	A	0,65	0,57	0,69	0,63	S	0,80	0,76	0,64	0,71	S	**0,96**	**0,96**	**0,91**	**0,94**	A
										Junit4											
	App	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**
Set 1	Our	**0,97**	**0,97**	**0,82**	**0,90**	S	**0,90**	**0,87**	**0,78**	**0,83**	R	**0,96**	**0,96**	0,56	**0,77**	A	**0,98**	**0,98**	**0,80**	**0,90**	S
Set 1	Trad	0,92	0,90	0,53	0,72	A	0,83	0,76	0,71	0,74	N	0,91	0,88	**0,64**	**0,77**	R	0,94	0,95	0,40	0,69	S
Set 2	Our	**0,97**	**0,97**	**0,79**	**0,88**	S	**0,90**	**0,87**	**0,79**	**0,83**	A	**0,96**	**0,96**	**0,62**	**0,80**	S	**0,98**	**0,98**	**0,79**	**0,89**	S
Set 2	Trad	0,92	0,91	0,51	0,72	S	0,83	0,76	0,71	0,74	N	0,91	0,88	0,60	0,75	R	0,95	0,95	0,38	0,68	A
Set 3	Our	**0,97**	**0,97**	**0,75**	**0,86**	A	**0,91**	**0,88**	**0,78**	**0,83**	S	**0,96**	**0,96**	0,70	**0,84**	S	**0,98**	**0,98**	**0,88**	**0,93**	S
Set 3	Trad	0,92	0,90	0,59	0,75	S	0,80	0,72	0,74	0,73	N	0,84	0,78	**0,78**	0,78	R	0,95	0,94	0,59	0,78	S
Set 4	Our	**0,89**	**0,86**	**0,41**	**0,65**	S	**0,82**	**0,75**	0,60	**0,68**	N	**0,90**	**0,87**	**0,43**	0,66	R	**0,93**	**0,94**	0,22	**0,60**	A
Set 4	Trad	0,88	0,84	0,40	0,63	A	0,80	0,71	**0,65**	**0,68**	N	0,85	0,80	0,57	**0,69**	A	0,92	0,92	**0,23**	0,59	A
Set 5	Our	**0,93**	**0,93**	0,24	0,60	N	**0,83**	**0,77**	0,57	**0,68**	R	**0,90**	**0,88**	0,38	0,64	S	**0,93**	**0,94**	0,21	**0,60**	A
Set 5	Trad	0,88	0,84	**0,46**	**0,66**	A	0,80	0,71	**0,64**	**0,68**	N	0,81	0,74	**0,63**	**0,69**	R	0,92	0,92	**0,23**	0,59	A
Set 6	Our	**0,97**	**0,97**	**0,69**	**0,84**	S	**0,90**	**0,86**	**0,77**	**0,82**	A	**0,97**	**0,97**	**0,81**	**0,89**	R	**0,97**	**0,97**	**0,84**	**0,91**	S
Set 6	Trad	0,90	0,87	0,52	0,70	A	0,78	0,70	0,71	0,70	R	0,86	0,81	0,71	0,77	R	0,94	0,94	0,56	0,76	S
										PdfBox											
	App	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**
Set 1	Our	**0,92**	**0,91**	**0,75**	**0,84**	A	**0,81**	**0,77**	**0,71**	**0,74**	R	**0,90**	**0,89**	**0,78**	**0,84**	R	**0,94**	**0,94**	**0,73**	**0,85**	A
Set 1	Trad	0,81	0,79	0,58	0,70	S	0,75	0,69	0,63	0,67	R	0,81	0,78	0,60	0,70	R	0,86	0,86	0,42	0,67	S
Set 2	Our	**0,92**	**0,92**	**0,73**	**0,84**	A	**0,80**	**0,77**	**0,71**	**0,74**	S	**0,91**	**0,91**	**0,77**	**0,85**	R	**0,94**	**0,94**	**0,73**	**0,85**	S
Set 2	Trad	0,83	0,81	0,52	0,69	S	0,75	0,69	0,64	0,67	S	0,80	0,76	0,67	0,72	R	0,86	0,86	0,43	0,67	A
Set 3	Our	**0,91**	**0,90**	**0,71**	**0,82**	A	**0,79**	**0,74**	**0,69**	**0,72**	A	**0,90**	**0,89**	**0,81**	**0,85**	A	**0,93**	**0,92**	**0,79**	**0,86**	A
Set 3	Trad	0,82	0,79	0,56	0,69	A	0,74	0,68	0,59	0,64	A	0,78	0,74	0,70	0,72	S	0,87	0,86	0,61	0,75	A
Set 4	Our	**0,87**	**0,87**	0,46	**0,69**	R	**0,76**	**0,71**	**0,65**	**0,69**	R	**0,82**	**0,78**	**0,80**	**0,79**	R	**0,88**	**0,88**	**0,53**	**0,73**	A
Set 4	Trad	0,76	0,71	**0,49**	0,61	R	0,75	0,70	0,61	**0,66**	R	0,75	0,69	0,71	0,70	R	0,76	0,71	0,51	0,62	R
Set 5	Our	**0,84**	**0,82**	**0,54**	**0,69**	S	**0,76**	**0,71**	**0,65**	**0,69**	S	**0,83**	**0,80**	**0,78**	**0,79**	R	**0,88**	**0,88**	**0,54**	**0,73**	A
Set 5	Trad	0,76	0,71	0,50	0,62	R	0,73	0,67	**0,65**	0,66	A	0,75	0,70	0,71	0,70	R	0,76	0,71	0,51	0,62	R
Set 6	Our	**0,88**	**0,87**	**0,74**	**0,81**	A	**0,83**	**0,80**	**0,61**	**0,72**	A	**0,88**	**0,86**	**0,85**	**0,86**	A	**0,89**	**0,88**	**0,85**	**0,87**	A
Set 6	Trad	0,79	0,75	0,66	0,71	A	0,72	0,66	0,59	0,63	N	0,75	0,69	0,74	0,72	S	0,83	0,80	0,70	0,76	A
										Wro4j											
	App	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**	**F1**	**Acc**	**Sen**	**AUC**	**RS**
Set 1	Our	**0,93**	**0,92**	**0,76**	**0,85**	S	**0,77**	**0,73**	**0,80**	**0,76**	R	**0,89**	**0,88**	**0,71**	**0,81**	A	**0,93**	**0,93**	**0,78**	**0,87**	A
Set 1	Trad	0,80	**0,77**	0,44	0,63	A	0,75	0,69	0,67	0,68	R	0,81	0,79	0,51	0,67	R	0,86	0,86	0,36	0,64	A
Set 2	Our	**0,93**	**0,92**	**0,78**	**0,86**	A	**0,77**	**0,73**	**0,79**	**0,75**	R	**0,89**	**0,89**	**0,64**	**0,78**	A	**0,94**	**0,93**	**0,80**	**0,88**	A
Set 2	Trad	0,79	**0,76**	0,45	0,62	A	0,75	0,69	0,65	0,68	N	0,78	0,74	0,59	0,67	R	0,86	0,86	0,36	0,64	S
Set 3	Our	**0,92**	**0,92**	**0,76**	**0,85**	A	**0,72**	**0,66**	**0,81**	**0,73**	S	**0,90**	**0,89**	**0,86**	**0,88**	R	**0,94**	**0,94**	**0,91**	**0,92**	A
Set 3	Trad	0,77	**0,73**	0,51	0,64	S	0,65	0,58	0,75	0,65	N	0,76	0,71	0,65	0,68	S	0,84	0,83	0,49	0,68	S
Set 4	Our	0,80	0,77	**0,47**	**0,64**	S	0,72	0,66	**0,71**	**0,69**	A	**0,83**	**0,81**	0,59	**0,72**	R	**0,86**	**0,86**	**0,37**	**0,65**	A
Set 4	Trad	**0,83**	**0,82**	0,32	0,60	R	**0,74**	**0,69**	0,63	0,66	R	0,76	0,71	**0,64**	0,68	A	0,83	0,83	0,35	0,62	A
Set 5	Our	**0,79**	0,76	**0,46**	**0,63**	A	0,73	0,68	**0,69**	**0,68**	A	**0,82**	**0,79**	**0,59**	**0,70**	R	**0,86**	**0,86**	**0,37**	**0,65**	A
Set 5	Trad	0,83	**0,82**	0,31	0,60	R	**0,75**	**0,70**	0,63	0,67	S	0,78	0,74	**0,59**	0,67	R	0,84	0,83	0,36	0,63	S
Set 6	Our	**0,92**	**0,92**	**0,75**	**0,85**	A	**0,72**	**0,66**	**0,80**	**0,72**	R	**0,91**	**0,90**	**0,89**	**0,90**	R	**0,94**	**0,93**	**0,92**	**0,93**	A
Set 6	Trad	0,76	0,72	0,52	0,63	S	0,68	0,61	0,60	0,61	A	0,72	0,66	0,68	0,67	S	0,84	0,83	0,51	0,69	S

Legend: Approach (App); Traditional (Trad); Decision Tree (DT); Logistic Regression (LR); Multi Layer Perceptron (MLP); Random Forest (RF); F1-Score (F1); Accuracy (Acc); Sensitivity (Sen); ROC AUC Score (AUC), Resample (RS), ADA(A); SMOTE (S); ROS (R); NONE (N) of the traditional approach obtained the best AUC values only in 16 cases. Considering the other indicators, our approach also overcomes the traditional. For F1-score the models of our approach obtained 102 best values, against only 14 of the traditional and 4 ties. Related to accuracy, both approaches tied 4 times and the models of our approach obtained 102 best values against 14 of the traditional. Regarding sensitivity, our approach overcomes the traditional reaching 78 of the best values, against 37 of the traditional and 5 ties.

If we consider all the indicators, algorithms and all sets (480 cases). Our approach produces the best results or equivalent ones in 399 cases (83.13%). To corroborate the findings we applied the Wilcoxon test (Rey and Neuhäuser, 2011) and calculated the p-value for all results of ROC/AUC, F1-score, accuracy and sensitivity performance metrics. For all metrics, there's a statistical difference in the distribution of history-based and traditional approaches with >95% of confidence (p-value=0.001), as Figure 5.5 shows. To confirm that our approach overcomes the traditional, we analyzed individually each set. We decided to evaluate the ROC/AUC, since it is a robust evaluation measure. We can see in Figure 5.6 that our approach perfom better in all sets.

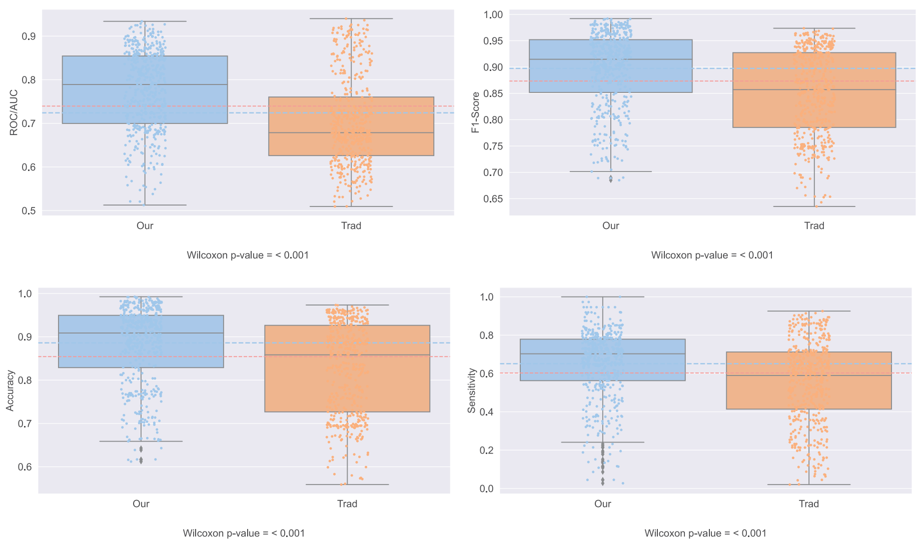


Figure 5.5: Statistical analysis of both approaches in all sets

Answer to RQ2: For all the indicators analyzed, our approach overcomes the traditional approach in the great majority of the cases (83.13%). Then we can conclude that our approach contributes to improve the performance of the models.

Implications: These results show that our history-based approach, considering the temporal dependency between the instances of the datasets, contributes to improve performance in the great majority of the cases. This happens independently of the algorithm and set of features used. Only for one indicator and one system the traditional approach presented a better mean value. The use of our approach seems to be the best option and can help developers, testers and managers to concentrate effort in the correct classes, that ones that really will change in the future, what helps to improve project quality, ease maintenance and evolution, and decrease costs.

Table 5.3: Comparing both approaches

Set 1					Set 2				
App	**F1**	**Acc**	**Sen**	**AUC**	**App**	**F1**	**Acc**	**Sen**	**AUC**
Our	19	19	15	16	Our	19	18	15	17
Trad	1	1	4	2	Trad	1	1	4	2
Eq	0	0	1	2	Eq	0	1	1	1
Set 3					Set 4				
App	**F1**	**Acc**	**Sen**	**AUC**	**App**	**F1**	**Acc**	**Sen**	**AUC**
Our	18	18	14	17	Our	14	14	10	16
Trad	2	2	6	3	Trad	4	4	10	2
Eq	0	0	0	0	Eq	2	2	0	2
Set 5					Set 6				
App	**F1**	**Acc**	**Sen**	**AUC**	**App**	**F1**	**Acc**	**Sen**	**AUC**
Our	14	15	9	14	Our	18	18	15	16
Trad	4	4	8	3	Trad	2	2	5	4
Eq	2	1	3	3	Eq	0	0	0	0

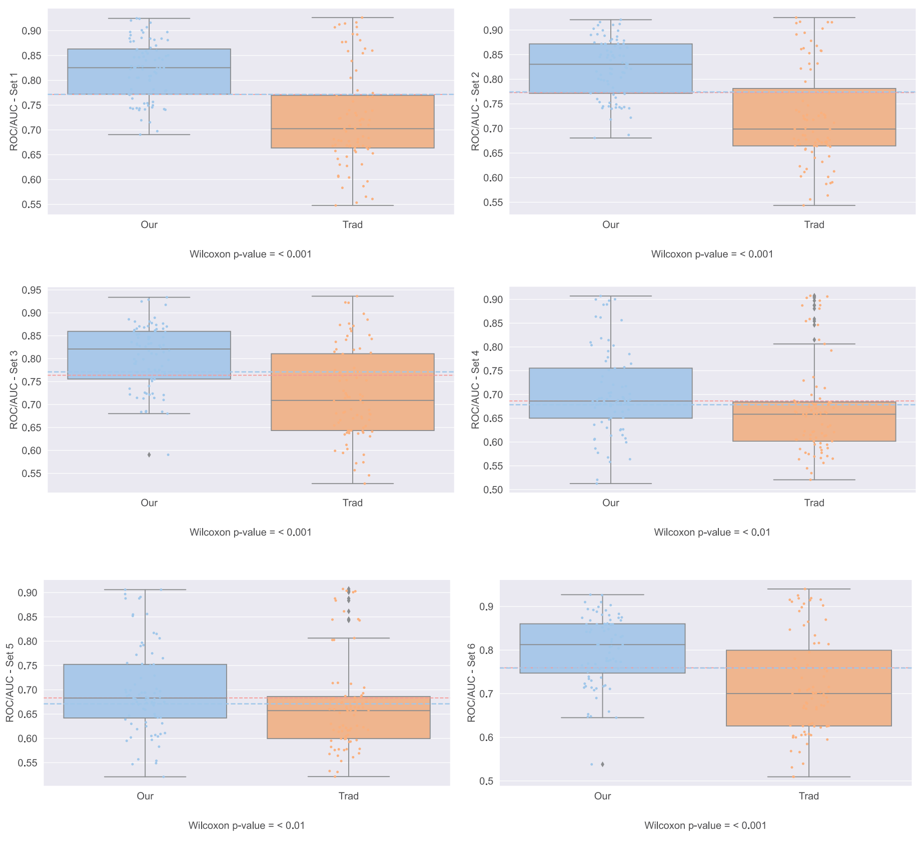


Figure 5.6: Statistical analysis of ROC/AUC of each set

5.5.3 RQ3: What is the impact of using smells related information to predict change-proneness using our approach?

To analyze the impact of using or not smells in the performance of our approach, we refer to Figure 5.7. This figure presents a comparison regarding the average values of AUC for all sets considering all algorithms. We can compare models with and without the smell-based metrics, Set 1 (combination of smell-based metrics, structural metrics and evolution-based metrics) against Set 2 (combination of structural metrics and evolution-based metrics); Set 3 (combination of smell-based metrics and evolution-based metrics) against Set 6 (only evolution-based metrics); and Set 4 (combination of smell-based metrics and structural metrics) against Set 5 (only structural metrics). We can see little difference when comparing the sets across all projects. This means that using smell-based metrics did not impact the prediction.

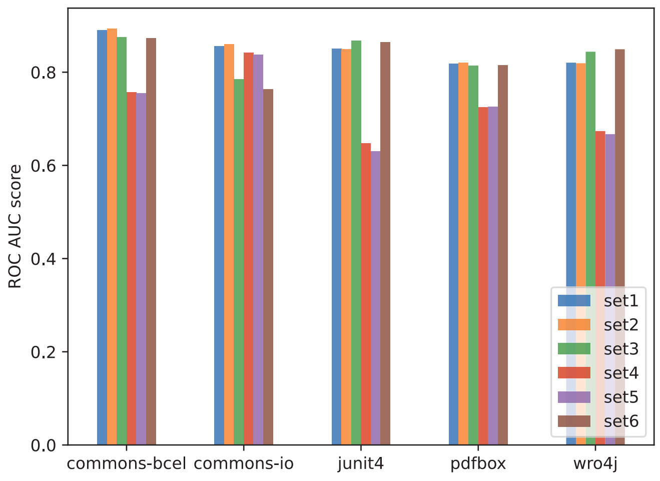


Figure 5.7: Impact of smells related information

To corroborate this hypothesis, we performed Wilcoxon test (Rey and Neuhäuser, 2011) and calculated the p-value for all results of ROC/AUC , between each comparison. Figure 5.8 shows the obtained results of the Wilcoxon test. For Set 1 against Set 2 the p-value was 0.871, which means no statistical difference between the sets. When comparing Set 3 against Set 4 and Set 4 against Set 5, we have p-value less than 0.05 and 0.001 respectively, which means a statistical difference on the results.

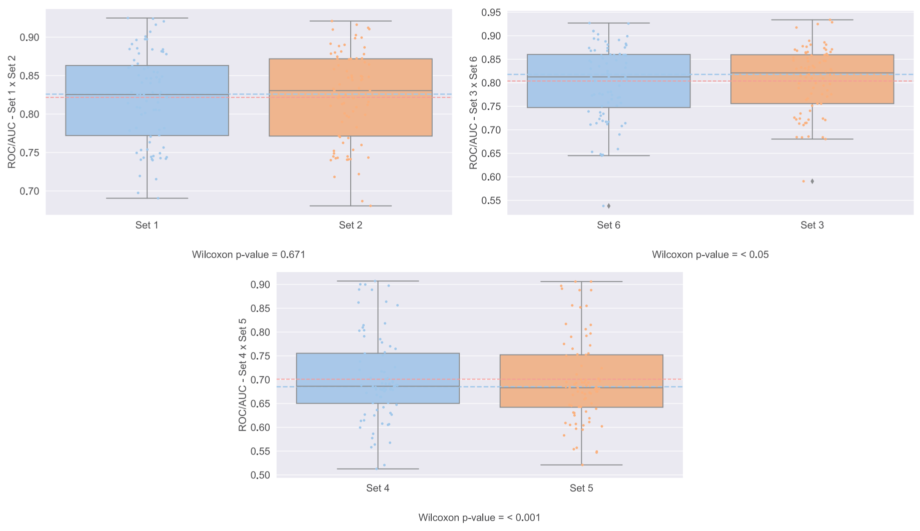


Figure 5.8: Statistical analysis - smell related information

To explore how different the results are between each set, we calculated the effect size. For effect size analyses, we used Vargha and Delaney's metric $\hat{D}$ (Vargha and Delaney, 2000). This measure ranges from 0 to 1 and defines the probability of a value, taken randomly from the first sample, is higher than a value taken randomly from the second sample. A $Negligible$ magnitude ($(\hat{D} < 0,56)$) represents a very small difference among the values and usually does not yield statistical difference. The $Small$ $(0,56 \leq \hat{D} < 0,64)$ and $medium$ $(0,64 \leq \hat{D} < 0,71)$ magnitudes represent small and medium differences among the values, and may or not yield statistical differences. Finally, a $Large$ magnitude $(0,71 \leq \hat{D})$ represents a significantly large difference that usually can be seen in the numbers without much effort.

We can observe in Tables 5.4 and 5.5 that sets 3 and 4 has the best ROC/AUC when compared to sets without smell-based metrics. However, when analyzing closely, we can see that the effect size was negligible, that is, it had little significance.

Table 5.4: Effect size - Set 3 x Set 6

Set	**ROC**	**Effect Size**
`Set 3`	0.8060 ± 0.0690	best result
`Set 6`	0.8015 ± 0.0800	negligible magnitude

Table 5.5: Effect size - Set 4 x Set 5

Set	**ROC**	**Effect Size**
`Set 4`	0.7034 ± 0.0920	best result
`Set 5`	0.6985 ± 0.0910	negligible magnitude

Answer to RQ3: The results show that the use of smell-based metrics does not have a significant impact on the performance of the models generated by our approach. Similar results were obtained for all sets of features, for all algorithms and projects.

Implications: our findings are different from the ones reported in the literature. For instance the work of Catolino et al. (2020) shows that the adoption of code smell-related information improves the performance of different categories of models obtained with the traditional approaches. Moreover, studies do not show a consensus on the use of structural metrics (Tsantalis et al., 2005; Lu et al., 2012; Zhou et al., 2009). Our results shows that the addition of density and diversity of smells does not contribute to improve performance of our approach. The use of a lower number of features can be advantageous because makes the models simpler. The combination of different kind of features and their influence in our approach should be better evaluated in further studies.

5.5.4 RQ4: What is the algorithm that leads our approach to produce the best results when answering all RQs?

Answering previous RQs, we did not find great differences in the indicators values; great values of AUC imply great values of other indicators. Similarly, we did not observe performance improvements using the smell-based features. Then to answer RQ4 we consider the AUC indicator and all sets.

Figure 5.9 presents a summary of the results. We can see that RF stands out in all projects, being the best in 3, out of 5 projects, and only surpassed by LR and MLP respectively, in `commons-bcel` and MLP in `Pdfbox`. It is worth to observe that `commons-bcel` has the smallest average number of classes and samples in comparison with the others. We can also highlight the DT algorithm, which reaches ROC/AUC values greater than 0.8 for all projects. LR and MLP also reach values greater than 0.7, but they do not reach values greater than 0.8 in all projects.

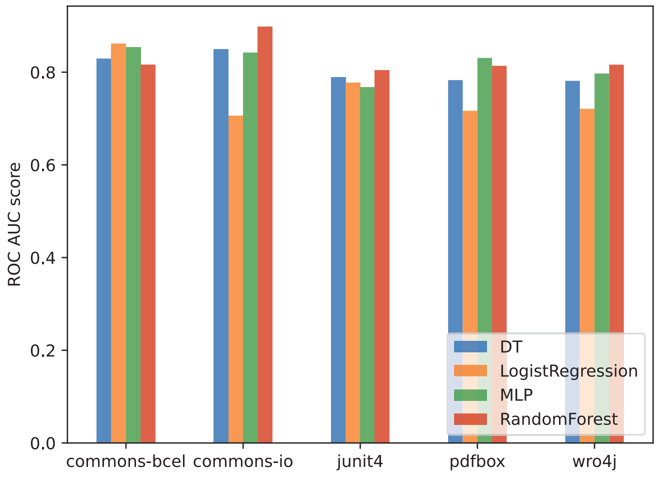


Figure 5.9: Algorithms performance

We also applied Kruskal Wallis test (Kruskal and Wallis, 1952) and calculated the p-value for all results of ROC/AUC . The statistical difference for the algorithms has 95% of confidence, as shown in Figure 5.10. The effect size presented small and negligible magnitudes and can be seen in Table 5.6. Random forest obtained the best result among the algorithms

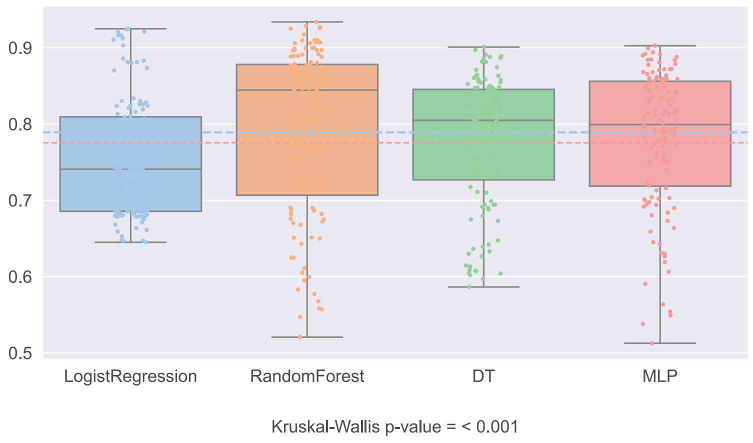


Figure 5.10: Statistical analysis algorithms performance - ROC/AUC

To complement our discussion, by using information from Table 5.2, we generated Table 5.7. This table shows for each algorithm how many times it uses each one of the resample techniques in all approaches. We observe that for our approach, in all 30 cases (5 projects and 6 sets of metrics), RF obtained the best results using ADA (17 times), and the same occurred for the traditional approach, using ADA 15 times. The DT algorithm for our approach performed

Table 5.6: Effect Size - ROC/AUC

Algorithm	**ROC**	**Effect Size**
`DT`	0.7780 ± 0.0850	small magnitude
`LogistRegression`	0.7506 ± 0.0760	negligible magnitude
`MLP`	0.7812 ± 0.0910	negligible magnitude
`RandomForest`	0.7919 ± 0.1110	best result

better with the ADA technique (14 times), on both approaches. For LR, SMOTE stands out for both approaches, appearing 10 times for our approach and 10 for the traditional one. The MLP algorithm performed better in both approach using ROS, in our approach 16 times, while for the traditional 15 times.

Table 5.7: Comparing resample techniques

	Our				Trad			
	ADA	NONE	ROS	SMOTE	ADA	NONE	ROS	SMOTE
DT	14	2	4	10	14	1	6	9
LR	9	1	10	10	3	8	9	10
MLP	9	0	16	5	5	0	15	10
RF	17	0	5	8	15	0	4	11

Answer to RQ4: The analysis indicates that our approach produced the best results with Random Forest. This happens for 3 out of 5 projects. MLP and LR leads our approach to reach the second best performance, being best in 1 project each. We can highlight that ADA is the resample technique that stood out the most when applying Random Forest algorithm, in both approaches

Implications: the results obtained with our approach are similar to the ones reported in the literature when the traditional approaches are applied, which reports that RF algorithm presents the best performance for the CPCP problem (Malhotra and Khanna, 2019). Thus smarter algorithms (e.g. Random Forest and Decision Trees) leads our approach to produce better results, but it is interesting to observe that our approach works well with simple/less sophisticated learners (e.g. Logistic Regression), which reach AUC values greater than 0.7. We also observe that the use of resample techniques is important and their performance may vary according to the algorithms. This and other pre-processing techniques should be further investigated.

5.5.5 RQ5: Which metrics are the most important for change-proneness prediction?

To calculate the most important features in the prediction of change-proneness we applied two methods: Mean Decrease in Impurity (MDI) (Loh, 2011) and Information gain. To calculate MDI we ran ExtraTreesClassifier, of scikit-learn, for each dataset. In both methods we evaluate the set containing all features (Set 1).

In the first method we computed the MDI for each dataset individually and ranked the 10 most important features in each dataset, Figures 5.11. We then created a unique set of 50 features, the top 10 of each dataset. In this set of 50 features, we intersected the 10 most repeated features. Figure 5.12 presents the result of this ranking. The LCH, WFR and FRCH appeared

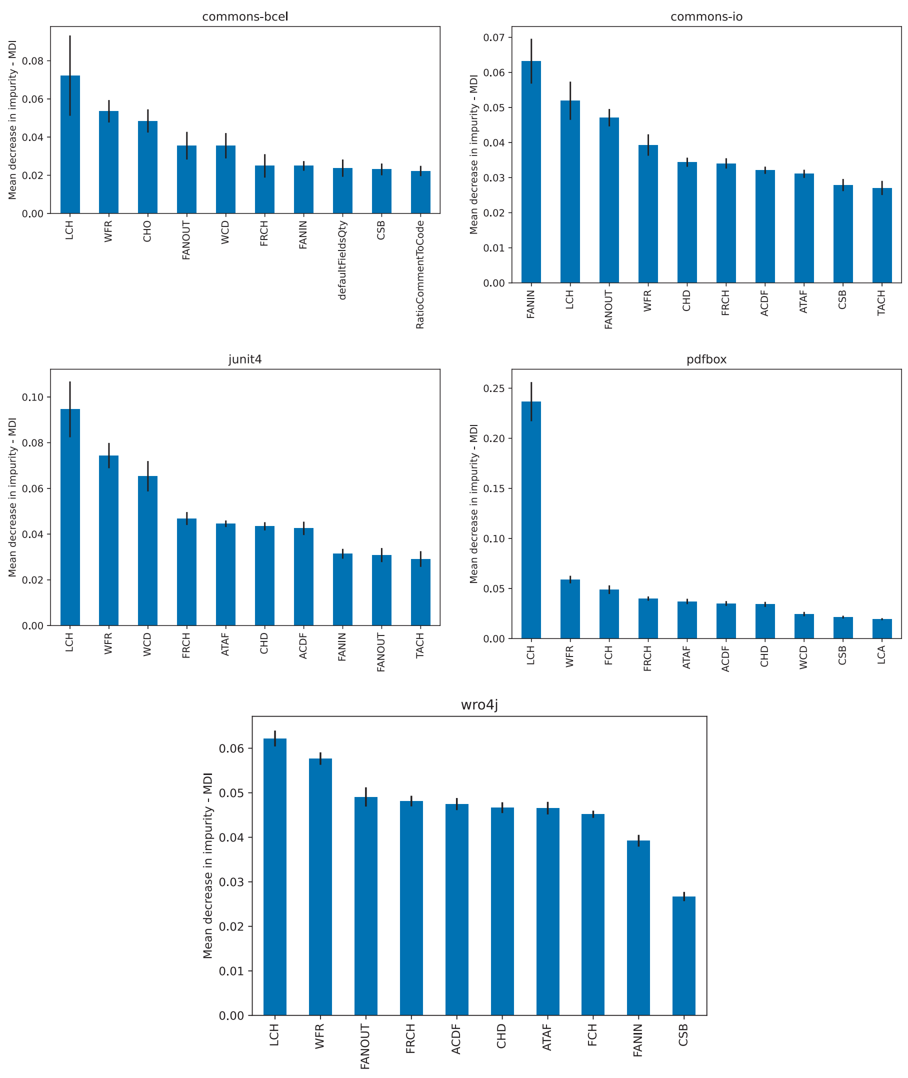


Figure 5.11: Feature importance

in the top 10 across all datasets. ACD, ATAF, CHD, CSB, FANIN and FANOUT features are among the top 10 in four out of five datasets, followed by WCD, with three datasets. We can notice that only two of the top 10 are structural metrics (FANIN, FANOUT), which are associated with class coupling.

To evaluate the information gain of features, we use the method `mutual_info_classif` of scikit-learn with default settings, which is equivalent to the entropy calculation of Weka (Witten et al., 1999). The information gain (as known as entropy) is calculated for each output variable. This value ranges from 0 (no gain) to 1 (maximum of information gain).

In the information gain method, we computed the top 10 features for each dataset individually, Table 5.8 presents the results. We then created a unique set of 50 features, the top 10 of each dataset. In this set of 50 features, we intersected the 10 most repeated features, Figure 5.13. The features CHD, FRCH, sumCyclomaticModified, and TACH performed best across all datasets, followed by BOC, FCH, finalFieldsQty, LCH, and WFR with four out of five datasets. The CHO feature is the last one in the ranking, ranking among the top 10 in just two datasets. As in the MDI method, in the information gain method we can observe that the evolutionary metrics performed better.

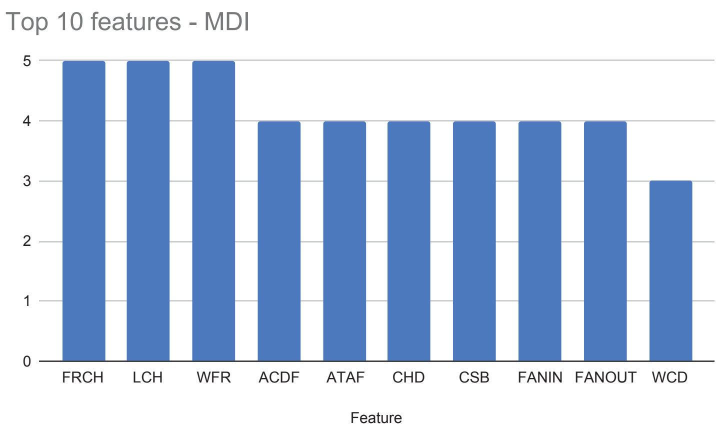


Figure 5.12: Feature importance using MDI

Table 5.8: Top 10 information gain of each dataset

Commons-bcel		Common-io		Junit4		pdfbox		wro4j	
Metric	**Info gain**	**Metric**	**Info gain**	**Metric**	**Info gain**	**Metric**	**Info gain**	**Metric**	**Info gain**
TACH	0,154	TACH	0,301	TACH	0,178	TACH	0,199	TACH	0,294
FRCH	0,096	FRCH	0,152	WFR	0,112	WFR	0,094	WFR	0,147
CHD	0,083	WFR	0,149	FRCH	0,109	BOC	0,073	FRCH	0,122
innerClassesQty	0,081	BOC	0,116	BOC	0,101	FRCH	0,071	BOC	0,108
finalFieldsQty	0,079	CHD	0,113	CHD	0,092	SumCyclomaticModified	0,063	LCH	0,079
MaxCyclomaticModified	0,065	LCH	0,105	LCH	0,091	FCH	0,049	CHD	0,071
CHO	0,055	FCH	0,075	FCH	0,065	LCH	0,047	WCD	0,067
stringLiteralsQty	0,055	finalFieldsQty	0,069	SumCyclomaticModified	0,064	CHD	0,046	FCH	0,061
wmc	0,054	ATAF	0,066	CHO	0,051	finalFieldsQty	0,042	finalFieldsQty	0,045
SumCyclomaticModified	0,053	SumCyclomaticModified	0,065	SumEssential	0,040	LCA	0,039	SumCyclomaticModified	0,037

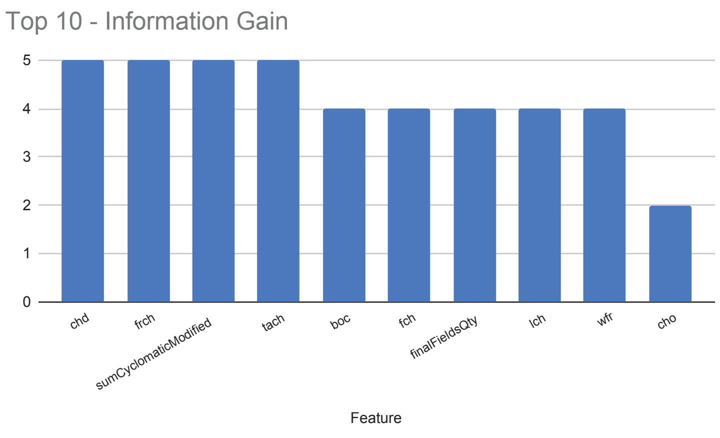


Figure 5.13: Information Gain

Answer to RQ5: The MDI and information gain analyzes show that evolutionary metrics were present in 80% of the top 10 features in both methods. Structural metrics appeared in only 20% of the top 10 features. The smell-based metrics did not appear among the top 10, which reinforces the RQ3 findings, which states that these metrics have no impact on the change-proneness prediction.

Implications: The result can be seen as an indication that the class history of changes plays a predominant role in the chance of the class being changed in future versions. We also know that this history does not necessarily have to be long, according to our RQ1, indicating that a period of three releases would be enough. These results can help developers to create more robust and objective tools to predict change-prone classes.

5.6 THREATS TO VALIDITY

Although we attempted to mitigate them to the best of our effort, there are some threats to the validity of our results. Below we present them, according to the taxonomy of Wohlin et al. (2000).

Internal Validity: We could make mistakes in the extraction, selection, and preparation of the data. A possible threat is related to the data treatment and preprocessing, as well as the techniques used. To minimize them, we followed a methodology including a set of steps commonly reported in the literature, and make the data available for a possible replication study. The algorithms were configured with standard parameters, then a study testing and optimizing other parameters can lead to better results.

External Validity: We analyze 5 systems with different sizes, diverse application domains and from distinct developers. However, all systems were developed in Java. Therefore, the limited number of projects and the predominance of only one language is a threat, and make it not possible to generalize the obtained results to a different context.

Conclusion Validity: The conclusions may be dependent on the indicators used in our analysis. To minimize such a threat, we employed different indicators and statistical tests.

5.7 CONCLUDING REMARKS

This chapter presented the experiments conducted for the evaluation of our history-based approach in relation to the traditional one. We introduced and evaluated the results of 5 research questions, as well their implications. The results show that our approach surpasses the traditional approach. We also showed that the smell-based metrics had no impact on the prediction of change-prone classes, and that the Random Forest algorithm obtained better results in most tested cases. We also highlight, through feature importance techniques, that evolutionary metrics stood out in the prediction of change-prone classes. In RQ5 we carried out a study to detect the most important features, but due to time, we did not do a feature selection to create specialized models, this would be interesting in future work. The next chapter discusses the final conclusions and research directions.

6 CONCLUSIONS

This work introduced a history-based approach for the CPCP problem. This approach adopts the sliding window method that allows considering the structure of the problem and the temporal dependency between the instances obtained from different releases. In this way, each learning instance is not limited to a fixed release, but contains the total or partial history of the class changes. Our approach combines structural, evolutionary and smell-based metrics to predict change-prone classes. We compare our work with the traditional approach, that associates metric values with the class in a single version to predict its change-proneness, without taking into account the history of the class throughout the project.

We analyzed 3 window sizes (2, 3, 4), the results point that to predict change-proneness of a class is not necessary a long history; in the tested applications, 3 releases seems enough. When comparing our approach with the traditional, results show that our approach, independently of the algorithm and set of features, contributes to improve performance in the great majority of the cases. Regarding performance, the execution 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.

Other aspect of the work was to test if smell-based features have influence on the performance of the models, for that we have tested models with and without density and diversity of smells. The results show that the smell-based metrics does not contribute to improve performance of our approach. When analyzing the individual performance of the algorithms, Random Forest algorithm was the best. We can also observe that our approach works well with other tested algorithms, e.g. Decision Tree, with ROC/AUC above 0.7. Allied to the algorithms, the use of resample proved to be effective in improving performance of the models. We can highlight that the best results were obtained when some resample technique was applied, e.g. ADA. We also evaluated the feature importance in the prediction of change-proneness. The results show that evolutionary metrics are predominant on the change-proneness. Structural metrics stood out discreetly, 2 out of 10 most important. Smell-based metrics did not stand out among the most important.

In this way, our work can lead researchers to understand the aspects of history-based representation and the role of evolutionary, structural and smell-related metrics in predicting change-prone classes. Researches can develop precise tools to recommend change-prone classes. In the practical field, our approach helps developers to better focus on change-prone classes to minimize the number of future changes, guide maintenance staff and resource distributions in order to reduce future efforts and costs.

6.1 ACADEMIC CONTRIBUTION

Throughout the work, some aspects of the research were submitted to the academic community, generating the following publications:

- R. C. Silva, P. R. Farah, S. R. Vergilio. - "Machine Learning for Change-Prone Class Prediction: A History-Based Approach", Brazilian Symposium on Software Engineering (SBES), 2022.

- E. A. da Roza, J. A. P. Lima, R. C. Silva and S. R. Vergilio, "Machine Learning Regression Techniques for Test Case Prioritization in Continuous Integration Environment", IEEE

International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 196-206, doi: 10.1109/SANER53432.2022.00034.

- P. R. Farah, T. Mariani, E. A. da Roza, R. C. Silva, S. R. Vergilio, "Unsupervised Learning For Refactoring Pattern Detection", IEEE Congress on Evolutionary Computation (CEC), 2021, pp. 2377-2384, doi: 10.1109/CEC45853.2021.9504804.

In order that the study can be replicated, and new academic discussions can be fostered, we make all source codes and results available in our repository (Silva, 2022). It contains the set of scripts used to extract all the metrics used in the system, as well as scripts for executing the approach. Scripts for data analysis are also available.

6.2 LIMITATIONS AND FUTURE WORK

Due to the difficulty of extracting and processing the metrics, the approach studied a limited set of repositories. It would be interesting to expand the dataset. We only used three window sizes, it would be interesting to test other window thresholds, to better explore the boundaries between small windows and large windows. Another limitation is that the study is limited to the Java programming language. An interesting extension of the work would be to cover other languages, so that other types of projects can be analyzed. We also intend to investigate sets of applications from the same domain, to create more specific models for these problems. We did an analysis on the features, but we did not generate a model with the best features, it would be interesting in future work to make this comparison.

As future work, we intend to expand the approach by applying Deep Learning neural networks, since 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, time series prediction, among others. For this, it would also be necessary to expand the dataset, since these networks demand a large amount of data.

We also plan further analysis on how other types of metrics sets can impact the change-prone class prediction. We also plan on investigating other history-based representations along with others ML algorithms.

REFERENCES

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

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

Aniche, M. (2015). *Java code metrics calculator (CK).* Available in https://github.com/mauricioaniche/ck/.

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

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

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

Benestad, H. C., Anda, B., and Arisholm, E. (2010). Understanding cost drivers of software evolution: A quantitative and qualitative investigation of change effort in two evolving software systems. *Empirical Software Engineering*, 15(2):166–203.

Bengio, Y., Goodfellow, I., and Courville, A. (2017). *Deep learning*, volume 1. MIT press Massachusetts, USA:.

Bieman, J., Straw, G., Wang, H., Munger, P., and Alexander, R. (2003). 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.

Bieman, J. M. and Kang, B.-K. (1995). Cohesion and reuse in an object-oriented system. *ACM SIGSOFT Software Engineering Notes*, 20(SI):259–262.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Information science and statistics. Springer, 1st ed. 2006. corr. 2nd printing edition.

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

Caprio, F., Casazza, G., Penta, M., and Villano, U. (2001). Measuring and predicting the linux kernel evolution. In *Proceedings of the International Workshop of Empirical Studies on Software Maintenance*.

Catolino, G. and Ferrucci, F. (2018). 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.

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

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

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

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

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

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

Destefanis, G., Counsell, S., Concas, G., and Tonelli, R. (2014). Software metrics in agile software: An empirical study. In *International Conference on Agile Software Development*, pages 157–170. Springer.

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

Elish, M. O. and Al-Rahman Al-Khiaty, M. (2013). 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.

Eski, S. and Buzluca, F. (2011). 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.

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

Fowler, M. (1999). *Refactoring – Improving the Design of Existing Code*. Addison-Wesley.

Fowler, M. and Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

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

Giger, E., Pinzger, M., and Gall, H. C. (2012). 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.

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

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

Hart, P. (1968). The condensed nearest neighbor rule (corresp.). *IEEE Transactions on Information Theory*, 14(3):515–516.

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

He, H. and Ma, Y. (2013). *Imbalanced Learning: Foundations, Algorithms, and Applications*. Wiley-IEEE Press, 1st edition.

Henry, S. and Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 1(5):510–518.

Karaboga, D. (2005). An idea based on honey bee swarm for numerical optimization, technical report - tr06. *Technical Report, Erciyes University*.

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

Khanna, M., Priya, S., and Mehra, D. (2021). 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.

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

Khomh, F., Di Penta, Massimiliano an Guéhéneuc, Y.-G., and Antonio, G. (2011). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17:243–275.

Koru, A. and Tian, J. (2005). 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.

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

Kubat, M. (2015). *An Introduction to Machine Learning*. Springer Publishing Company, Incorporated, 1st edition.

Kumar, L., Lal, S., Goyal, A., and Murthy, N. L. B. (2019). 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. Association for Computing Machinery.

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

Li, W. and Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122. Object-Oriented Software.

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

Livingston, F. (2005). Implementation of breiman's random forest machine learning algorithm. *Machine Learning Journal Paper*.

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

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

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

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

Malhotra, R. and Bansal, A. J. (2016). Software change prediction: a literature review. *International Journal of Computer Applications in Technology*, 54(4):240–256.

Malhotra, R., Kapoor, R., Aggarwal, D., and Garg, P. (2021). 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.

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

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

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

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

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

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

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

Massoudi, M., Jain, N. K., and Bansal, P. (2021). 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.

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

Melo, C., Lima da Cruz, M., Martins, A., Filho, J., and Machado, J. (2020a). Time-series approaches to change-prone class prediction problem. In *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS*, pages 122–132. INSTICC, SciTePress.

Melo, C. S. (2020). Supporting change-prone class prediction. Master's thesis, Universidade Federal do Ceará.

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

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

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

Pritam, N., Khari, M., Kumar, R., Jha, S., Priyadarshini, I., Abdel-Basset, M., Long, H. V., et al. (2019b). Assessment of code smell for predicting class change proneness using machine learning. *IEEE Access*, 7:37414–37425.

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

Riedmiller, M. (1994). Advanced supervised learning in multi-layer perceptrons — from backpropagation to adaptive learning algorithms. *Computer Standards & Interfaces*, 16(3):265–278.

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

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rêgo, D. C. G. (2018). *Understanding and Improving Batch Refactoring in Software Systems*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

Scitools (2021). What metrics does undertand have? `https://support.scitools.com/t/what-metrics-does-undertand-have/66`.

Silva, R. d. C. (2022). Supplementary Material - Machine Learning for Change-Prone Class Prediction: A History-Based Approach. URL https://github.com/carvalho7976/ChangeProneTools.

Stehman, S. V. (1997). Selecting and interpreting measures of thematic classification accuracy. *Remote sensing of Environment*, 62(1):77–89.

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

Tomek, I. (1976). Two modifications of cnn. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):769–772.

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

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

Vargha, A. and Delaney, H. (2000). A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics - J EDUC BEHAV STAT*, 25.

Witten, I. H., Frank, E., Trigg, L., Hall, M., Holmes, G., and Cunningham, S. J. (1999). Weka: Practical machine learning tools and techniques with java implementations.

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

Zhou, Y., Leung, H., and Xu, B. (2009). 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.

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