



# On the use of Change History Data to Enhance Class Change-Proneness Prediction Models

Rogério de Carvalho Silva  [ Federal University of Paraná | [rcsilva@inf.ufpr.br](mailto:rcsilva@inf.ufpr.br) ]

Paulo Roberto Farah   [ Santa Catarina State University | [paulo.farah@udesc.br](mailto:paulo.farah@udesc.br) ]

Silvia Regina Vergilio  [ Federal University of Paraná | [silvia@inf.ufpr.br](mailto:silvia@inf.ufpr.br) ]

 Software Engineering Department, UDESC, Rua Dr. Getúlio Vargas, 2822, Bela Vista, Ibirama, SC, 89.140-000, Brazil.

**Received:** 23 October 2023 • **Accepted:** 06 August 2024 • **Published:** 05 October 2024

**Abstract** As software evolves, new artifacts are created, modified, or removed. One of these main artifacts generated in the development of object-oriented software is the class. Classes have a very dynamic life cycle that can result in additional costs to the project. One way to mitigate this is to detect, in the early stages of the development, classes that are prone to change. Some approaches in the literature adopt *Machine Learning (ML)* algorithms to predict the change-proneness of a class. However, most of these approaches do not consider the temporal dependency between training instances, i.e., they consider that the instances are independent. To overcome such a limitation, this study presents an approach for predicting change-proneness based on the class change history. The approach adopts the sliding window method and is evaluated to obtain six kinds of models, which are derived by using, as predictors, different sets of metrics: structural, evolutionary, and smell-based. The evaluation uses five systems, four ML algorithms, and also explores some resample techniques to deal with imbalanced data. Regardless of the kind of model analyzed and the algorithm used, our approach overcomes the traditional one in 378 (~80%) cases, out of 420, considering all systems, kinds of models, indicators, and algorithms. Moreover, the results show that our approach presents the best performance when the set of evolutionary metrics is used as predictors. There is no improvement when smell-based metrics are added. The Random Forest algorithm with the resampling technique ADA reaches the best performance among the ML algorithms evaluated.

**Keywords:** class change-proneness, software maintenance, software evolution

## 1 Introduction

The adoption of iterative and incremental software development by the software industry encompasses the generation of newer software versions and releases over time. As the systems evolve, they become larger and more complex. This makes software maintenance expensive and arduous. Changes in the software can occur due to many reasons, for instance, accommodation of customers' demands, quality improvement, fault corrections, changes in technology, etc. Thus, managing and controlling changes is critical [Malhotra and Khanna, 2019]. However, monitoring equally all the parts of the code is expensive and resource consuming [El-ish and Al-Khiaty, 2013]. Hence, the industry has adopted change-proneness prediction models to identify parts of the code more prone to change in subsequent versions of the software [Malhotra *et al.*, 2021a].

Some models focus on different code segments, such as packages, files, classes, and methods, according to some importance criterion. However, models for predicting change-prone code at class-level granularity are the most used and popular. This is because the class is considered the foundation of an *Object-Oriented (OO)* software. It contains everything required to execute only one aspect of a desired functionality and is a relatively independent and interchangeable module [Lindvall, 1998]. In this sense, change-prone classes are those that are likely to change across different

versions of a software product. Predicting these classes in the early stages of the development is known in the literature as *Change-Prone Class Prediction (CPCP)* [Malhotra and Khanna, 2019].

A class is subject to different kinds of changes along the software evolution. One of these changes regards the addition of new functionalities. Classes implementing essential system features are more likely to change in future versions. A class becomes more complex and fault-proneness by adding new functionalities, increasing the probability of changes to fix bugs. This change process possibly erodes the original design and reduces the quality of class structure by introducing smells [Catolino *et al.*, 2020] that are sub-optimal design and/or implementation choices adopted by practitioners. Smells also imply the need for refactoring and future changes to improve the overall quality of the class structure.

Therefore, efficient solutions to the CPCP problem are essential. They help to plan preventive maintenance operations to ensure product quality, keep the software operational, allocate resources more efficiently, assure customer satisfaction, and, reduce technical debt. This avoids the propagation of problems to later stages with higher costs. Due to its importance, the CPCP problem has been a theme of surveys and systematic reviews [Malhotra and Khanna, 2019; Godara and Singh, 2014; Malhotra and Bansal, 2015]. These studies show that the most approaches use *Machine Learning*

(ML) techniques to derive the prediction models, overcoming statistical methods [Malhotra and Khanna, 2013]. Current models use different sets of metrics (dependent variables) to capture the different CPCP dimensions and characteristics. Most of them are structural-based models [Malhotra and Khanna, 2019]. However, these metrics are often combined with other metrics, such as the intensity of smells or anti-patterns [Catolino *et al.*, 2020; Kaur and Jain, 2017; Pritam *et al.*, 2019], and process metrics [Elish and Al-Khiaty, 2013; Al-Khiaty *et al.*, 2017; Catolino and Ferrucci, 2018, 2019; Catolino *et al.*, 2018; Malhotra and Khanna, 2018a,b]. These studies showed that the use of different kinds of metrics improves the performance of the ML models in different degrees. However, studies on the performance of different kinds of models are required [Malhotra and Khanna, 2019].

The current approaches present some limitations. They do not consider the temporal dependency between the class changes, i.e., they consider that all instances used in the training phase are independent. In our previous study [Silva *et al.*, 2022], we introduced an ML approach for the CPCP problem using the *Sliding Window (SW)* method [Dietterich, 2002] to overcome this limitation. We proposed an approach considering that any changes to a class may affect their subsequent modifications. We also considered that the changes often depend on previous changes. Then, the class change history is considered. The approach was evaluated with four ML algorithms by using structural, evolutionary, and smell-based metrics. The use of the *SW* method significantly improved the prediction performance compared to the traditional approach, which consider independent learning instances. However, we did not evaluate the performance of the approach using different sets of metrics in our previous study. Such an evaluation is important to better guide developers using our approach and selecting the best set of metrics (predictors).

Considering this fact, in the present study, we evaluate our *SW* approach to obtain six types of models, which are usually found in the CPCP literature. The models are derived by using different sets of metrics (features). We also assess the importance of these features for the prediction models. Moreover, we conduct a broader analysis with new indicators and statistical tests. In this way, the main contributions of this study are as follows:

- We propose a history-based approach for the CPCP problem: this approach better captures the problem structure. It allows a better representation, considering the dependency between the instances obtained from different releases. This leads to a better performance than the traditional approaches proposed in the literature, which consider the instances are independent;
- We evaluate the performance of the approach by adopting six kinds of models found in the literature. These models use different sets of features as predictors. In this way, we can identify the best model for our approach, and derive some insights on the use of the approach;
- We provide a replication package containing the implemented code, datasets, and results, available online in our repository [Silva *et al.*, 2024]. It contains the set of

scripts used to extract all the metrics as well as scripts for executing the approach. This allows replication and future research.

We also present some implications of our results for guiding the research community and practitioners. In short, evolutionary models are the best when the *SW* method is applied. The Random Forest algorithm presents the best performance. Correct detection of change-prone classes can allow developers to better focus on change-prone classes. This minimizes the number of future changes, guides the maintenance team, and efficiently distributes resources and effort. The results of this study can be used for the early detection of change-prone classes that can be monitored during software development, reducing technical debt and decreasing maintenance costs.

The remainder of this paper is organized as follows. Section 2 reviews the CPCP problem and existing approaches. In Section 3, we present our history-based approach. Section 4 describes how our evaluation was conducted, the adopted methodology, and research questions. Section 5 presents and analyzes the results. Section 6 discusses the main implications of our findings. Section 7 describes the main threats to the validity of our results. Section 8 concludes the paper and proposes future studies.

## 2 Related Work

The main source of information includes surveys and systematic reviews on CPCP [Malhotra and Khanna, 2019; Godara and Singh, 2014; Malhotra and Bansal, 2015]. The systematic review of Malhotra and Khanna [2019] shows that more than 50% of the studies use structural metrics, and around 25% combine two or more types of metrics. The authors highlight that ML is a trend adopted in more recent studies. The studies that are most related to ours are the focus of this section, which is organized as follows. First, we present the main ways adopted to decide whether a class changes. Then, we describe the main sets of features adopted in the literature, the models generated, as well as some pieces of work with a goal similar to ours that compare these models. After, we discuss the main ML algorithms and indicators used in the existing studies. Lastly, we present some works that are based on time series that also consider temporal dependency.

### 2.1 Methods for Computing Class Changes

There are different approaches in the literature to determine changes in classes. One is based on the number of *Source Lines of Code (SLOC)*. If the SLOC values of the current version are different from the values of the previous release, a change occurred in the class [Malhotra and Khanna, 2019; Kaur and Jain, 2017; Arisholm *et al.*, 2004; 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 approaches consider structural changes in the code elements [Bieman *et al.*, 2003; Eski and Buzluca, 2011; Mas-soudi *et al.*, 2021]. They extract changes comparing the abstract syntax tree between two different versions. This approach, known in the literature as *Fine-Grained Changes*

(FGC), usually uses the *ChangeDistiller* tool [Fluri et al., 2007; Gall et al., 2009]. Some studies consider a change occurs if the number of structural changes is higher than the median of the distribution of the number of changes experienced by all the classes of the system [Catolino et al., 2020; Romano and Pinzger, 2011].

The SLOC method can detect all small changes, such as a refactoring that renames a variable. However, this method includes several types of changes, even trivial ones. Thus, it may not capture the details about the semantics of changes [Giger et al., 2012]. The FGC method identifies which type of code structure has changed (loop, control structure, statement) and can provide detailed information about the changes. As a drawback, it is more complex to implement and may ignore small changes. In our study, we adopt the FGC method, based on other works with goals similar to ours. These works also evaluate different kinds of models [Catolino et al., 2020; Romano and Pinzger, 2011]. Thus, we can compare our results with these existing studies in the literature.

## 2.2 Predictors adopted in the studies

Different kinds of models have been investigated in the literature. They use different types of metrics as predictors. Structural models use class-based measure metrics such as size, complexity, coupling, inheritance, and cohesion [Malhotra and Khanna, 2019; Catolino et al., 2020; Godara and Singh, 2014; Malhotra and Bansal, 2015; Lu et al., 2012; Malhotra and Khanna, 2021; Khomh et al., 2011; Tsantalis et al., 2005]. They consider that more complex classes are more change-prone. Evolutionary models use process metrics that capture some evolution aspects of the class, such as birth date, change frequency, and density [Elish and Al-Khiaty, 2013; Al-Khiaty et al., 2017; Catolino and Ferrucci, 2018, 2019; Catolino et al., 2018; Malhotra and Khanna, 2018a,b]. Smell-based models use metrics based on the design and structure quality, such as the intensity of smells or anti-patterns [Catolino et al., 2020; Kaur and Jain, 2017; Pritam et al., 2019; Khomh et al., 2009].

Other models are based on information extracted from the class text vocabulary or the class dependency graph. Some 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 [Catolino and Ferrucci, 2018, 2019; Catolino et al., 2018, 2017; Giger et al., 2012; Zhu et al., 2018].

Although several metrics are used and different kinds of models can be derived, few studies compare their performance by combining different metrics [Malhotra and Khanna, 2019]. There is also a lack of benchmarks since the metric values are collected in different contexts and ways. There is no consensus on the use of structural metrics. Some conflicting results are reported in the literature [Lu et al., 2012; Zhou et al., 2009; Tsantalis et al., 2005]. Tsantalis et al. [2005] concluded that structural metrics, except class size, are insignificant features for CPCP. Lu et al. [2012] analyzed the highest number of systems in the literature and confirmed that size structural metrics have moderate prediction capacity. In contrast, Zhou et al. [2009] indicated that

size structural metrics present a strong confounding effect and are not good predictors.

Some studies show that classes with code smells are more change-prone than classes without smells [Kaur and Jain, 2017; Pritam et al., 2019]. Catolino et al. [2020] report that code smell-related information improves the performance of three categories of prediction models: structural, evolutionary, and developer-based. The results show that smell intensity is a better predictor than anti-pattern metrics. A combined change prediction model including product, process, developer-based, and smell-related features presents notably higher performance than other models. Then, we can observe conflicting results in studies evaluating the models using different sets of metrics. The present study contributes to overcoming this gap by evaluating our approach considering models based on six kinds of metrics.

## 2.3 Adopted ML algorithms and indicators

Malhotra and Khanna [2019] provided an overview of CPCP and showed that several different classification and prediction techniques have been adopted. Logistic regression, Naïve Bayes, Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), Decision Trees, and Random Forest are the main algorithms employed and evaluated. The most used indicators for evaluating the performance of the models are: accuracy, *AUC (Area Under Receiver Operating Characteristic Curve)*, recall, precision, and F1-score. Ensemble algorithms, such as Random Forest, have presented the best results in most studies [Malhotra and Khanna, 2019]. However, we observed that these approaches consider that all the training instances are independent, i.e., no temporal dependency is considered.

## 2.4 Use of dependent instances

Temporal dependency involves extracting meaningful summary and statistical information from data points arranged in chronological order. It diagnoses past behavior and predicts future behavior [Nielsen, 2019]. In general, temporal dependency between instances is not directly supported by ML models. Approaches based on time series proposed in the literature can capture this aspect [Caprio et al., 2001; Melo et al., 2020]. Caprio et al. [2001] employed ARIMA to model the problem. Melo et al. [2020] introduced concatenated and recurrent representations to capture the temporal aspect. They evaluated the concatenated representation using conventional ML algorithms and recurrent representation with a *Recurrent Neural Network (RNN)* based on the *Gated Recurrent Units (GRU)* architecture.

These two studies used a limited number of independent variables, with a few structural metrics. They also used the simple SLOC approach to identify class changes. In a previous study [Silva et al., 2022], we proposed an approach considering the dependency between the repository instances to address this limitation. The difference between related work and the present study is that we combine distinct kinds of features. Moreover, we adopt the FGC method to identify changes in classes. This approach is described in the next section.

### 3 Proposed Approach

The proposed approach adopts the *Sliding Window (SW)* method [Dietterich, 2002] to capture the temporal dependency between the instances. First, we identified relevant metrics to act as predictors reviewing the CPCP literature. We selected the dependent and independent variables based on the work of Catolino *et al.* [2020]. The chosen metrics are structural, evolutionary, and smell-based. Figure 1 presents an overview of the proposed approach. After selecting the metrics in the Data Definition step, we created the dataset by mining public GitHub repositories. We collected the metrics with static analysis tools in the DataSet Creation step. In the Data Preprocessing step, we built an input representation for the ML algorithms and applied some methods for data normalization and balancing. We used the resulting dataset to train and test the ML models bluein the ML Model Building step. In the last step, Prediction, we evaluated the performance of the ML algorithms by using data science indicators.

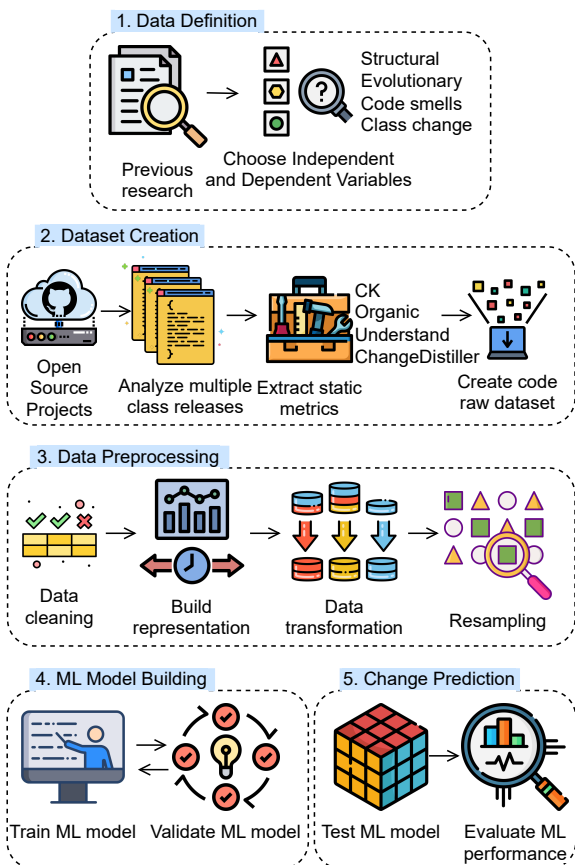


Figure 1. Overview of the proposed approach. Adapted from [Silva *et al.*, 2022]

#### 3.1 Data Definition

The independent and dependent variables were selected based on the CPCP literature review. The independent variables comprise metrics from three categories [Malhotra and Khanna, 2019], which were also used by the traditional approaches [Catolino *et al.*, 2020]. They are: i) structural: these metrics are related to internal software quality met-

rics, 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], and others; ii) evolutionary: these metrics are related to the software evolution from one release to another, e.g., the birth of a class, amount of changes, frequency of changes, change density, and so on. It is worth mentioning that 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 OO design structures, which can cause source code degradation as code smells [Fowler, 1999] and anti-patterns [Brown *et al.*, 1999; Lanza *et al.*, 2010]. We employed two smell-based metrics: density and diversity of smells. Density calculates the number of smells found in the class. Diversity calculates the number of types of smells in the class. Table 1 presents the list of metrics used to create the set of independent variables for the models.

The dependent variable represents whether a class changed or not in a period of time that corresponds, in our approach, to the time between two subsequent releases. In the present study, releases are official versions of the project that have been approved. Each release is identified by the project developers using tags. We considered the FGC method to decide whether a class has changed in a given context. We used the same definition as [Catolino *et al.*, 2020; Romano and Pinzger, 2011] to compute the dependent variable. A class changes if, in a time period  $t$ , the number of structural changes of this class is higher than the median of the distribution of the number of changes experienced by all classes of the system. To determine if a class changes, we calculate its number of changes for each pair of commits ( $c_i, c_{i+1}$ ) belonging to the interval between the releases.

#### 3.2 DataSet Creation

In this step, we built the dataset with the independent and dependent variables of the classes in each release. We used the Understand<sup>1</sup> and CK [Aniche, 2015] tools to extract the structural metrics. The Understand tool is a static code analyzer that collects product metrics of code elements. The CK tool calculates class-level and method-level code metrics in Java projects. We implemented our own tool to calculate the evolutionary metrics based on Elish and Al-Khiaty [2013]. We used the Organic<sup>2</sup> tool to extract the smells reported in Table 1. The implementations and the description of each metric are available in our repository [Silva *et al.*, 2024]. To compute the dependent variable, we use the *ChangeDistiller* tool. This tool implements a difference algorithm that generates and compares the abstract syntax tree between the 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 include attribute renaming change, parameter ordering change, and other methods and class declaration changes. The tool ignores white space-related differences and documentation-related updates.

<sup>1</sup><https://www.scitools.com/>

<sup>2</sup><https://github.com/opus-research/organic>

**Table 1.** Metrics for change-prone classes prediction

Structural Metrics
LackOfCohesioninMethods (LCOM), TighClassCohesion (TCC), LooseClassCohesion (LCC), CouplingBetweenObject (CBO), ResponseForaClass (RFC), FANIN, FANOUT, WeightedMethodsperClass (WMC), totalMethodsQty, staticMethodsQty, publicMethodsQty, privateMethodsQty, protectedMethodsQty, defaultMethodsQty, abstractMethodsQty, finalMethodsQty, synchronizedMethodsQty, totalFieldsQty, staticFieldsQty, publicFieldsQty, privateFieldsQty, protectedFieldsQty, defaultFieldsQty, visibleFieldsQty, finalFieldsQty, NOSI, LOC, returnQty, loopQty, comparisonsQty, tryCatchQty, parenthesizedExpsQty, stringLiteralsQty, numbersQty, assignmentsQty, mathOperationsQty, variablesQty, maxNestedBlocksQty, anonymousClassesQty, innerClassesQty, lambdasQty, uniqueWordsQty, modifiers, logStatementsQty, AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxEssential, MaxInheritanceTree, MaxNesting, PercentLackOfCohesion, RatioCommentToCode, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential.
Evolutionary Metrics
BirthOfClass (BOC), TotalAmountOfChanges (TACH), FirstChange (FCH), LastChange (LCH), ChangeOccurred (CHO), FrequencyOfChanges (FRCH), ChangeDensity (CHD), WeightedChangeDensity (WCD), WeightedFrequencyOfChanges(WFR), AggregatedChange-SizeNormalizedbyFrequencyOfChanges (ATAF), LastChangeAmount (LCA), LastChangeDensity (LCD), ChangesSinceBirth (CSB), ChangesSinceTheBirthNormalizedBySize (CSBS), AggregatedChange-DensityFrequency (ACDF).
Smells
LazyClass, DataClass, ComplexClass, SpaghettiCode, SpeculativeGenerality, GodClass, BrainClass, ShotgunSurgery, LongParameterList, LongMethod, FeatureEnvy, DispersedCoupling, MessageChain, IntensiveCoupling, BrainMethod, RefusedBequest, ClassDataShouldBePrivate.

Our model is designed to be independent of the type, motivation, or intent of changes. Thus, we used structural, evolutionary, and smell-based metrics. Our target is classified only as changed or not changed. When a class is created, the possible added code is not considered until it suffers a modification, as defined by our approach. When this happens, the dependent variable corresponding to the new class is altered to capture that change. But, if the creation of a new class is associated with changes in other classes, for instance, those resulting from some refactoring, such as Extract Class, these changes will affect the value of the dependent variable of the existing classes that were modified.

The values of the independent variables were computed considering the release before the one in which the dependent variable is calculated, i.e., we computed the independent variables in the release  $R_i$  and the dependent variable between releases  $R_i$  and  $R_{i+1}$ . Thus, we avoid biases due to the computation of the change-proneness in the same periods as the independent ones. Table 2 illustrates how the data is collected. We supposed the system has only one class, and only three metrics ( $M_1$ ,  $M_2$  and  $M_3$ ) are extracted. In Release 0, the metrics values are 1, 10, and 100. But, at this time, we do not have information about changes. In Release 1, the metric values for the same class are calculated again as 2, 20, and 200. Now, we can set a Boolean value for the dependent variable Changed, for which *true* represents the class changed using the FGC approach, and *false* otherwise.

**Table 2.** Example of how metrics are collected by release

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

### 3.3 Data Preprocessing

In the Data Preprocessing step, we first cleaned the raw dataset, then we built the data representation that captured the temporal dependency between the data instances. The dataset is transformed and balanced before being used by the ML algorithms. After collecting variables, we performed a Data cleaning. Cleaning implies identifying and discarding instances without values for any independent variables. This may occur due to some errors in the collection tools. Next, empty values were filled with zero and duplicate rows were removed [Ilyas and Chu, 2019]. Duplicate data rows could be useless to the modeling process, if not dangerously misleading during model evaluation. For instance, a duplicate row or rows can appear in both train and test datasets, and any evaluation of the model on these rows will be correct and result in an optimistically biased estimate of performance on unseen data [Brownlee, 2020]. To finish the data preparation, we unified the dataset by merging all the metrics collected by the different tools. The features obtained with each tool were merged based on the key fields: project name, release hash, and class name. The scripts to be adopted in this step are available in our repository. In the end, the values of the features were normalized because the metrics were collected in different scales.

### 3.4 Build Representation

Then, a dataset is built from the dataset containing the values of the metrics represented chronologically by employing the SW method. This 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 SW method works as follows. First, we select metric values 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 last release, always following the order of the chronological dataset.

## 4 Evaluation Description

To better illustrate the adopted representation, we use Tables 3 and 4, both of which are derived from Table 2. Table 3 shows the representation adopted by the traditional approach. In this structure, each instance represents a version of the



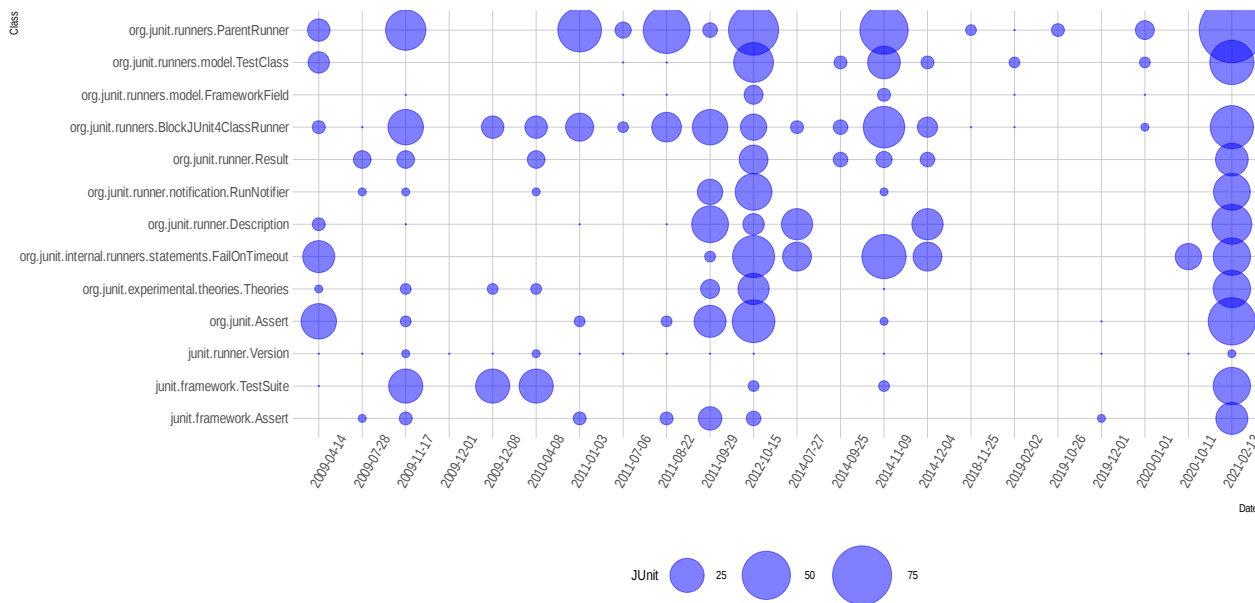


Figure 2. Classes of JUnit system that changed in more than 6 releases

class in a given project release, and each instance contains the value of the metrics  $M$  as the independent variables. The values collected in Release  $i$  predict the dependent variable *Changed* in Release  $i + 1$ . Table 4 shows the new dataset built by reshaping the original one using the SW method and  $S = 2$ . Instance 0 contains metrics values  $M_1, M_2,$  and  $M_3$  from Releases  $i$  and  $i + 1$  to predict the dependent variable *Changed* from Release  $i + 2$ , so one step ahead. The process is repeated by shifting the two boxes simultaneously over the samples, one step at a time, creating new rows until the window reaches the end of the table.

Table 3. Dataset used by traditional approaches

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

Table 4. Representation of Table 3 reshaped with  $S = 2$

Instance	Release $i$			Release $i+1$			Release $i+2$
	$M_1$	$M_2$	$M_3$	$M_1$	$M_2$	$M_3$	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

In summary, the main difference between the methods is that the traditional one considers the metric values of a single release, and, in the SW method, the metric values of the  $S$  previous releases are used in a single structure. The role of

the window is to say how many releases will be aggregated at a time, i.e., the size of the metrics history over this period. In this sense, the change history of the class is considered.

### 4.1 Motivation for a Change-History Approach

Figure 2 presents an example of a change history approach for the CPCP problem. It contains a list of classes of the JUnit system used in our evaluation, which changed in, at least, seven different releases. The x-axis contains the dates of the releases, and the y-axis contains the names of the classes. Each bubble represents the number of changes of a class in one release. At the top, we can observe the org.junit.runners.ParentRunner class changed in the release of 2009-04-14, did not change in the next release, and then changed in the subsequent one. After three releases without any changes, it changed again in a sequence of five releases. This is a clear case of a change-prone class. But, if only the information of release 2010-04-08 were used to predict the change-proneness of release 2011-01-03, this would result in an incorrect prediction. Thus, our approach allows the creation of instances containing the change history of this class, leading to better prediction performance.

In our previous study [Silva et al., 2022], we evaluated the proposed approach by combining structural, evolutionary, and smell-based metrics. We compared models based on these predictors using our approach and the traditional one which does not adopt the SW method. We also analyzed the impact of using smell-related features. Our approach overcame the traditional one, and we did not observe any impact on the use of smell-related information. By adopting the same methodology and target systems, in the evaluation described in this section, we use our approach and the traditional one to obtain a broader set of models to the CPCP problem, as well as to investigate the best kind of features

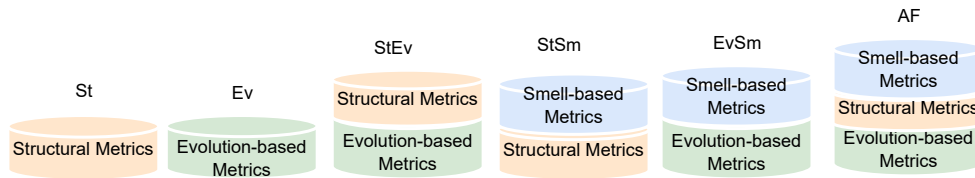


Figure 3. Kinds of models evaluated

to be used with our approach. Following the most common models found in the literature, we evaluate six kinds of models, as illustrated in Figure 3. They were obtained by varying the categories of features: 1) Structure-Based Model (St); 2) Evolution-Based Model (Ev); 3) Structure and Evolution-Based Model (StEv); 4) Structure and Smell-Based Model (StSm); 5) Evolution and Smell-Based Model (EvSm); and 6) All Features-Based Model (AF). Note that smell-based models were not considered since this model is not common. As mentioned earlier, smell-based metrics are only combined with other types of features in the literature [Catolino *et al.*, 2020].

## 4.2 Research Questions

We stated the following *Research Questions (RQs)* based on our goals:

**RQ1 - How do different window sizes affect the prediction performance of the models?** In this RQ, we intend to evaluate the performance of the six kinds of models generated by the most used ML algorithms, according to different window sizes. To this end, we use sizes 2, 3, and 4 for each application. These sizes were adopted based on the work of Tsoukalas *et al.* [2020];

**RQ2 - What is the performance of our approach regarding the performance of a traditional approach?** In this RQ, we intend to compare the performance of the algorithms using our approach based on a temporal dependency with the performance using a traditional approach. To this end, we consider the six kinds of models to evaluate if the set of features affects the performance of both approaches;

**RQ3 - What is the performance of the different kinds of models generated by our approach?** In this RQ, we evaluate the performance of the six kinds of models that can be derived using our approach. We aim to identify the best set of features for our approach;

**RQ4 - Which algorithm leads our approach to produce the best results?** The Random Forest algorithm presents the best performance in most studies [Malhotra and Khanna, 2019]. However, it is relevant to investigate whether this also holds for our temporal approach, considering the six kinds of models based on different features;

**RQ5 - Which metrics are the most important for change-proneness class prediction?** In the previous RQs, we analyze different sets of features (models) and algorithms. However, it is important to analyze the individual prediction

power of each metric.

## 4.3 Target Systems

We extracted features from five open-source Java applications from the GitHub<sup>3</sup> online repository. The criteria for selecting systems used in our research are described as follows.

- **Java language:** We consider systems written in Java since it is supported by the tools used in this study, increasing the replicability;
- **Public access:** We selected open-source software systems to allow full replication of our study since access to private projects is usually difficult to obtain. Additionally, the systems use Git as the version control system to track the evolution of their source code. This allows access to the complete history of changes in the source code of a software system and multiple analyses to address our RQs;
- **Activeness and maturity:** To classify a software system as active and mature, we considered two criteria: (i) it has the status defined as active to ensure the systems have a development activity and are subject to the changes and (ii) its GitHub repository has more than 20 releases to ensure change-history.
- **Use in change-prone classes domain:** We selected projects already used by other studies on change-prone class prediction [Malhotra and Lata, 2020; Malhotra *et al.*, 2021b].

We performed a search in the GitHub search engine. We sorted the results by popularity, based on the highest number of stars. Table 5 presents the main characteristics of the applications, such as the number of releases and samples, the average number of classes (Avg Classes), as well as the average number of classes changed (or not), the period considered and the average number of days between releases (Days). The average days between each release vary from 30 in PdfBox to 218 in Junit4. The standard deviation (Std) of the days between the releases is high, with an average of 220.2.

## 4.4 Data Preprocessing

We consider six kinds of models to answer the RQs. Therefore, we derived six datasets, containing the corresponding set of features. To answer RQ2, in addition to the datasets

<sup>3</sup><https://github.com>

**Table 5.** Used Java systems

System	Releases	Samples	Avg Classes	Not Changed	Changed	Period	Days	Std
Commons-bcel	23	5383	234	5110	273	Jul/2011 to Sep/2019	114	212
Commons-io	61	34556	566	30844	3712	Mar/2012 to Sep/2020	78	225
Junit4	26	7987	307	7579	408	Nov/2012 to Feb/2021	218	397
PdfBox	57	47079	826	41584	5495	Jun/2018 to Mar/2021	30	45
Wro4j	50	16912	338	14922	1990	Mar/2010 to Nov/2021	90	222
Total	217	111917	2271	100039	11878	Mar/2010 to Nov/2021	106	220.2

using our representation (Table 4), we also build datasets using the representation of the traditional approaches (Table 3). To ensure 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, for each set of features and by analyzing the data distribution, we found a high imbalance in relation to the dependent variable. Training unbalanced models can make the models tend to predict the class that has the highest frequency, generating erroneous results. To mitigate this, we selected 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]. In addition, we tried to use undersampling techniques, but they further reduced our dataset until we did not have enough data left for the split step used in validation.

#### 4.5 ML Model Building

In the ML Model Building and Prediction steps, the datasets 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. Testing sets 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]. This consists of dividing the training set 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). We used the scikit-learn tool [Pedregosa *et al.*, 2011] to implement the cross-validation.

We employed the algorithms widely used in the literature [Malhotra and Khanna, 2019], such as Decision Tree (DT); Logistic Regression (LR); Multi-Layer Perceptron (MLP), and Random Forest (RF). We used the normalization and resampling methods and algorithms implemented by the scikit-learn tool [Pedregosa *et al.*, 2011]. All the raw data and scripts are available in our repository.

Our approach generated a total of 1440 models. Each algorithm was executed for each of the six datasets using or not one of the three resample techniques (4 algorithms x 6 datasets x 3 windows sizes x 4 resampling techniques x 5 applications). Default parameters of the scikit-learn were used. For each system, 288 models were generated. Similarly, the traditional approach generated 480 models, 96 models for each system. All these models were used to evaluate the impact of the sliding window size (RQ1). The values of these indicators for these models are presented in our repository.

However, to answer the other RQs, we consider only the models adopting the best value for the window size pointed out by RQ1 and the best resampling technique for each algorithm and system.

#### 4.6 Prediction Evaluation

We use common data science performance indicators and often used for the CPCP problem to evaluate the generated models (see Section 2). The indicators are defined as follows, considering that TP, TN, FP, and FN as the number of true positives, true negatives, false positives, and false negatives, respectively.

- **Accuracy:** it is the proportion of correct predictions, both, true positives and true negatives, among the total number of cases evaluated [Metz, 1978];

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

- **Sensitivity:** it is also known as Recall, is given by the fraction of the study population that is actually positive [Metz, 1978];

$$Sensitivity = \frac{TP}{TP + FN} \quad (2)$$

- **F1-score:** it is a weighted average of Precision and Sensitivity [Witten *et al.*, 2005]. Precision is the ability of the classifier to avoid labeling negative samples as positive. The Precision is the ratio  $TP/(TP + FP)$ .

$$F1-score = \frac{2 * Precision * Sensitivity}{Precision + Sensitivity} = \frac{2 * TP}{2 * TP + FP + FN} \quad (3)$$

- **ROC/AUC:** the *ROC (Receiver Operating Characteristic)* curve shows the trade-off between the TP and FP rates. An *Area Under the ROC Curve (AUC)* close to 1 indicates a high-performance model while an area about 0.5 indicates a low-performance model. An increase in Sensitivity (TP rate) also increases the FP rate. Therefore, the ROC measure helps track the optimum point where the metric performs well for the TP and FP rates [Sultana *et al.*, 2021].

ROC/AUC is the main performance indicator used in our work to compare all models and choose the models with the best results. It considers equally positive and negative classes. We selected this indicator because it is more suitable to compare classification models, especially with unbalanced data [Han *et al.*, 2022]. The other indicators are used to complement the analysis.



We used non-parametric statistical tests, specifically Kruskal-Wallis [Kruskal and Wallis, 1952] (multiple groups) and Wilcoxon rank-sum [Mann and Whitney, 1947] (comparison pair), with a confidence level of 95%. This selection was guided by the nature of our data [Arcuri and Briand, 2011] since it does not follow a normal distribution. This fact was corroborated by the Gaussian curve and the Kolmogorov–Smirnov test [Massey, 1951] in the datasets of all systems. These results are available in our repository.

We use Vargha and Delaney’s  $\hat{A}_{12}$  metric to calculate the effect size magnitude of the difference between two groups. This defines the probability that a value randomly taken from the first sample is higher than a value randomly taken from the second sample [Vargha and Delaney, 2000]. The magnitude can be: *negligible* ( $\hat{A}_{12} < 0.56$ ); *small* ( $0.56 \leq \hat{A}_{12} < 0.64$ ); *medium* ( $0.64 \leq \hat{A}_{12} < 0.71$ ); and *large* ( $0.71 \leq \hat{A}_{12}$ ). A negligible magnitude represents a tiny difference between the values and usually does not yield statistical difference. The small and medium magnitudes may yield statistical differences (or not). Finally, a large magnitude represents a statistically significant difference, generally evident in the numbers without much effort.

#### 4.7 Calculation of Feature Importance

To answer RQ5, we analyzed the individual importance of each metric by applying *Mean Decrease in Impurity (MDI)* and *Information Gain* methods. These methods work as follows.

The MDI, also known as Impurity Importance, is commonly calculated with decision tree algorithms [Breiman, 2001]. It measures how much each predictor variable (feature) contributes to reducing impurity at tree nodes during the splitting process. Thus, it measures how much the model’s accuracy decreases when a certain variable is excluded. The greater the decrease in impurity, the more important the variable. The importance of a variable can be calculated as the average of the change in impurity, considering all nodes at which the variable was selected.

According to Mitchell [1997], Information Gain measures how well a given attribute (feature) separates the training instances according to their target classification. It measures the effectiveness of an attribute in classifying the training data and ranges from 0 (no gain) to 1 (maximum of information gain). The measure is the expected reduction in entropy caused by partitioning the instances according to this attribute. More precisely, the Information Gain,  $Gain(S, A)$  of an attribute  $A$  relative to a collection of instances  $S$ , is defined as:

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where  $Values(A)$  is the set of all possible values for attribute  $A$  and  $S_v$  is the subset of  $S$  for which attribute  $A$  has value  $v$  (i.e.,  $S_v = \{s \in S | A(s) = v\}$ ).  $Entropy(S)$  characterizes the (im)purity of the collection  $S$  and is given by:

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

where  $p_{\oplus}$  and  $p_{\ominus}$  are the proportion of positive and negative instances in  $S$ , respectively. In all entropy calculations, to avoid undefined values  $\log_0$  is considered equals to 0. Entropy quantifies the similarity or the difference between the number of instances of the different classes. Its highest value is 1, when all classes have the same number of instances. It becomes lower when the difference in the number of instances of classes increases.

## 5 Analysis of the Results

This section presents the results, analysis, and answers for our RQs.

### 5.1 RQ1: How do different window sizes affect the prediction performance of the models?

As mentioned in Section 4.2, we investigated three values for  $S$  (2, 3, and 4). There is no standard rule for optimal window size, then, we empirically tested initial window sizes, starting with 2, since this is the smallest possible window size, similar to the work of Tsoukalas *et al.* [2020]. Larger window size values did not improve the results. The larger the window size, the greater the class history required, thus excluding classes added in more recent versions. This reduces the training data and can generate bias in relation to older classes. By testing six models, three window sizes, four algorithms, and four resampling techniques, we generated a total of 288 models for each system.

First, we applied the Kruskal-Wallis test considering all models to answer RQ1. We aim to evaluate if there is a statistical difference between the window sizes. The p-value obtained is 0.927, which means statistical equivalence. Figure 4 shows this result.

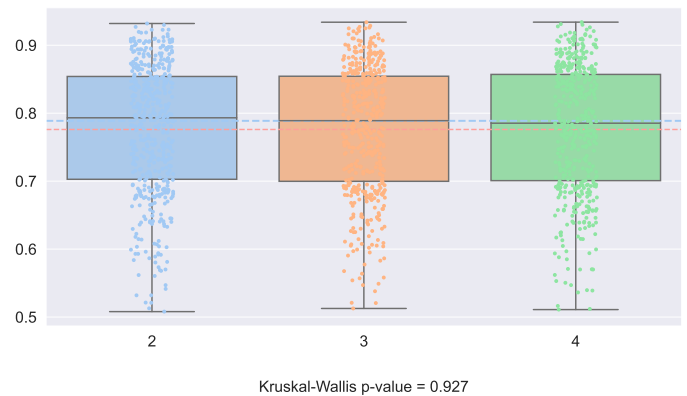


Figure 4. Boxplots regarding window size and AUC considering all models

After this, we performed a second analysis by applying the test considering each model individually, as shown in Figure 5. The p-values for AF, Ev, EvSm, St, StEv, and StSm were 0.857, 0.979, 0.882, 0.875, 0.807, and 0.812, respectively. These results show that even considering each model individually, there is no statistical difference between the models using the different window sizes.

As the results did not show statistical differences, we decided to create a ranking of the top 10 models to select the

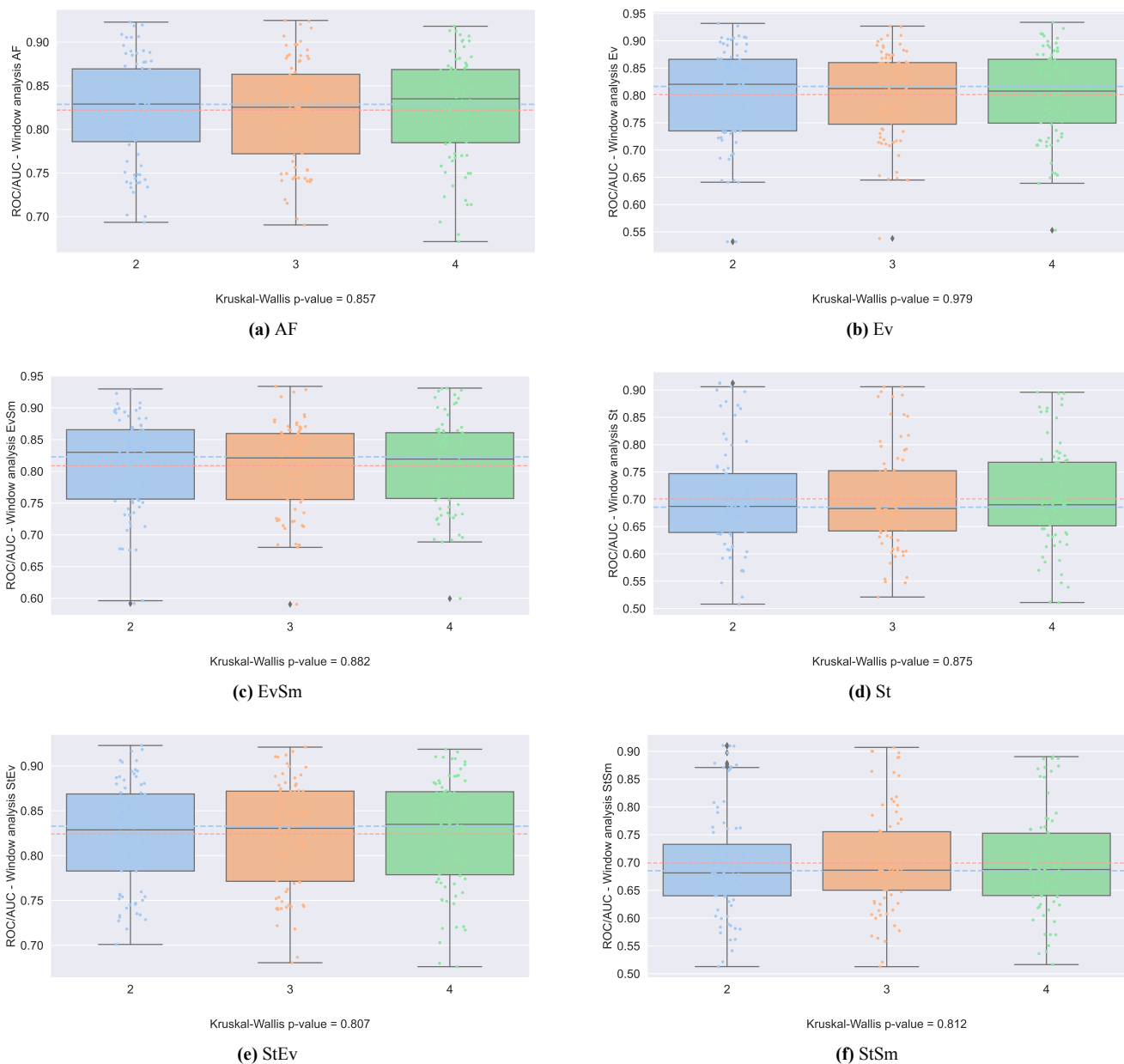


Figure 5. Boxplots regarding the window size and AUC for each kind of model

most outstanding window size. We selected the 10 best models for each of the 5 systems based on the AUC score. We aggregated the top 10 results from each application, creating a list of 50 models. With this list, we counted the number of times each size appeared. Figure 6 presents this result. We can observe that 20 (out of 50) or 40% of the best models use  $S = 3$ , followed by  $S = 2$  which is used in 17 (34%) models, and size  $S = 4$ , used in 13 (26%). Models using  $S = 4$  have almost the worst distribution in the ranking. This could indicate that increasing  $S$  would not contribute to a better performance. In contrast, models with  $S = 3$  stand out and appear more often among the best ones.

**Response to RQ1:** Although there was no statistical difference between the models using the evaluated window sizes, the models with an intermediate size, ( $S = 3$ ), ap-

peared most frequently among the best ones.

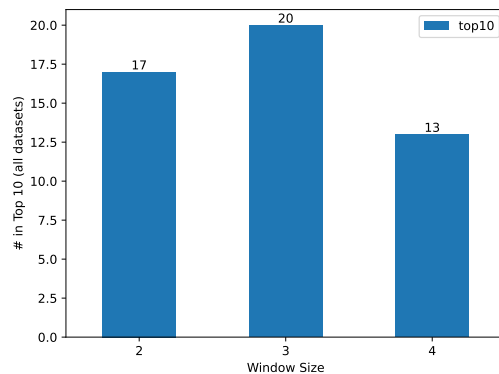


Figure 6. Window size ranking

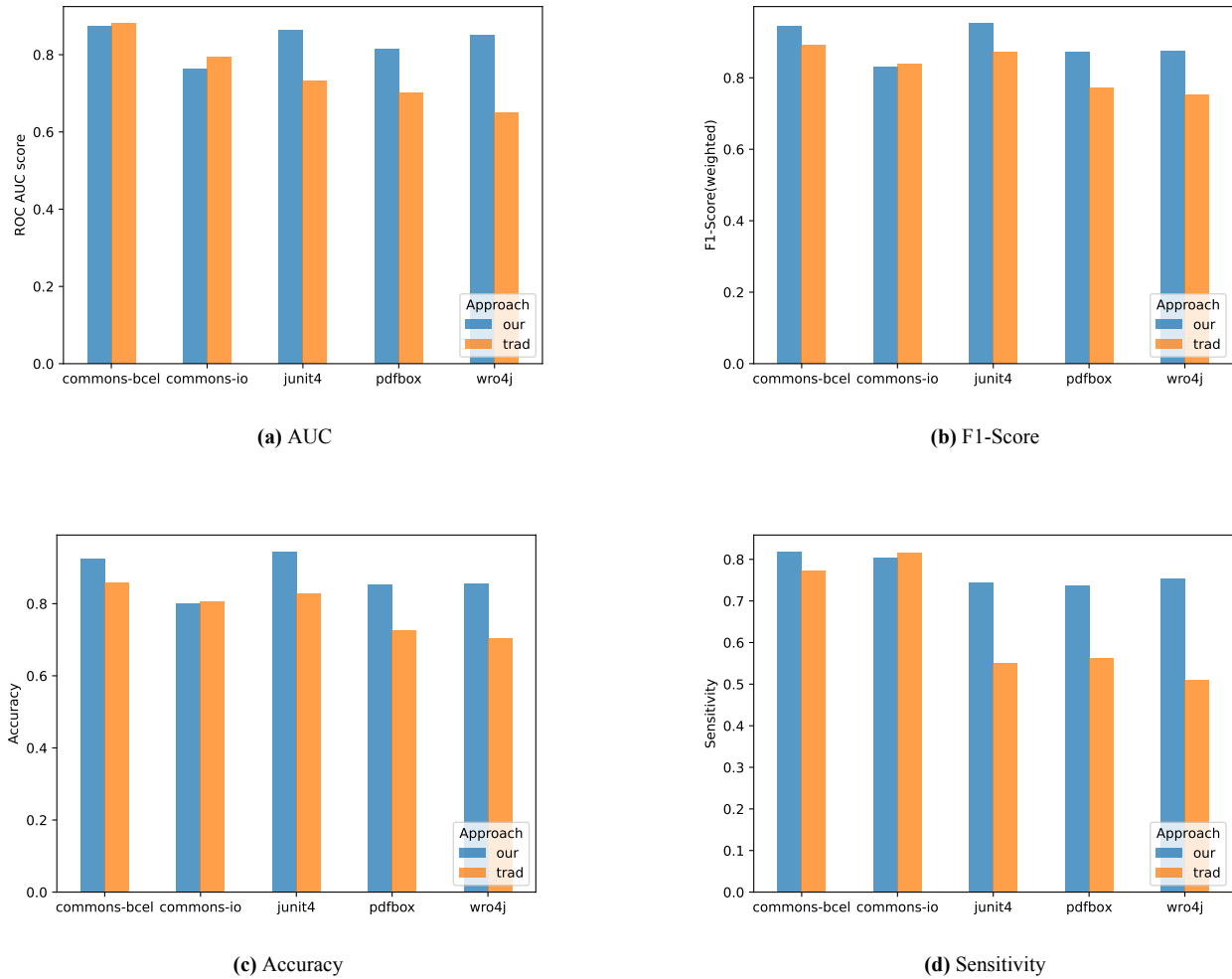


Figure 7. Comparing both approaches per indicator

## 5.2 RQ2: What is the performance of our approach regarding the performance of a traditional approach?

Table 6 presents the results of the best models using  $S = 3$ , according to RQ1, for the six sets of features. For each system and algorithm, it presents the values of the F1-score, Accuracy, Sensitivity, and AUC of the models obtained using our approach and the traditional one. The best values are highlighted in bold. The RS columns correspond to the resampling method used to obtain the model with the best performance.

In both approaches, we calculated the mean and median values for the indicators, considering all algorithms and sets of features from Table 6. Figure 7 compares the means for both approaches. For AUC, we can observe that our approach obtained the best results in 3 out of 5 systems with the mean values, and in 4 out of 5 with the median values. Both approaches had similar results for `commons-io` and `commons-bcel` systems. This may be due to both systems belonging to the same family and having a similar nature. Our approach presented better results for F1-score and Accuracy than the traditional one in 4 out of 5 systems, almost tied on `commons-io`. For Sensitivity, the traditional approach

had slightly better results on the `commons-io` system but did not show outstanding results compared to our approach. However, our approach presented better performance in most cases and great differences for `JUnit4`, `PDFBox`, and `Wro4j` systems. This means that, for these systems, our approach tends to correctly classify the change-prone classes as true positives. This is an important characteristic because the approach does not spend time with false negatives. In addition, except for `commons-bcel` and `commons-io` systems, our approach obtained the best values of AUC, regardless of the set of features and algorithms used. The results comparing the approaches with median values are similar and are available in our repository.

Table 7 presents the number of times each approach was better or obtained an equivalent result for each indicator and each set of features. These results confirm that our approach obtained the best values for the indicators compared to the traditional approach. We can observe that, for 120 models analyzed in Table 7 (4 algorithms, 5 systems, and 6 sets), our approach obtained the best results of AUC in 96 (80%) models and equivalents in 8 models. The traditional approach only reached the best values of AUC in 16 (~13%) models. Considering the other indicators, our approach also overcame the traditional approach. For F1-score and Accuracy, the mod-

**Table 6.** Results from the best obtained models for each algorithm and set of features

		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
AF	Our	<b>0.98</b>	<b>0.98</b>	<b>0.73</b>	<b>0.86</b>	A	<b>0.93</b>	<b>0.91</b>	<b>0.95</b>	<b>0.92</b>	S	<b>0.98</b>	<b>0.98</b>	<b>0.78</b>	<b>0.88</b>	R	<b>0.99</b>	<b>0.99</b>	<b>0.78</b>	<b>0.89</b>	A
	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
StEv	Our	<b>0.98</b>	<b>0.98</b>	0.78	<b>0.89</b>	R	<b>0.93</b>	<b>0.90</b>	<b>0.95</b>	<b>0.92</b>	R	<b>0.98</b>	<b>0.98</b>	<b>0.76</b>	<b>0.87</b>	S	<b>0.99</b>	<b>0.99</b>	0.78	<b>0.89</b>	A
	Trad	0.94	0.93	<b>0.79</b>	0.86	A	0.92	<b>0.90</b>	0.83	0.87	S	0.96	0.96	0.68	0.83	S	0.97	0.97	<b>0.80</b>	<b>0.89</b>	A
EvSm	Our	<b>0.98</b>	<b>0.98</b>	0.73	0.86	A	<b>0.88</b>	<b>0.83</b>	<b>0.95</b>	<b>0.89</b>	A	<b>0.95</b>	<b>0.94</b>	0.78	<b>0.87</b>	A	<b>0.99</b>	<b>0.99</b>	<b>0.78</b>	<b>0.89</b>	R
	Trad	0.95	0.94	<b>0.80</b>	<b>0.87</b>	S	0.83	0.77	0.88	0.82	R	0.92	0.91	<b>0.79</b>	0.85	A	0.95	0.95	0.80	0.88	S
StSm	Our	<b>0.96</b>	<b>0.96</b>	0.54	<b>0.76</b>	R	0.84	0.76	<b>0.78</b>	<b>0.77</b>	A	0.90	0.85	<b>0.78</b>	<b>0.82</b>	A	<b>0.97</b>	<b>0.97</b>	0.38	0.68	S
	Trad	0.90	0.88	<b>0.56</b>	0.73	A	<b>0.94</b>	<b>0.94</b>	0.51	0.74	S	<b>0.93</b>	<b>0.94</b>	0.22	0.60	R	0.88	0.85	<b>0.76</b>	<b>0.81</b>	S
St	Our	<b>0.96</b>	<b>0.95</b>	<b>0.54</b>	<b>0.75</b>	N	0.81	0.72	<b>0.81</b>	<b>0.77</b>	A	0.88	0.82	<b>0.81</b>	<b>0.82</b>	R	<b>0.97</b>	<b>0.97</b>	0.38	0.68	S
	Trad	0.89	0.87	0.39	0.65	R	<b>0.93</b>	<b>0.92</b>	0.48	0.71	S	<b>0.91</b>	<b>0.91</b>	0.24	0.60	A	0.89	0.86	<b>0.73</b>	<b>0.80</b>	A
Ev	Our	<b>0.98</b>	<b>0.98</b>	0.73	0.86	R	<b>0.87</b>	<b>0.81</b>	<b>1.00</b>	<b>0.90</b>	S	<b>0.94</b>	<b>0.92</b>	0.81	<b>0.87</b>	R	<b>0.98</b>	<b>0.98</b>	0.73	0.86	R
	Trad	0.94	0.93	<b>0.93</b>	<b>0.93</b>	A	0.81	0.75	0.89	0.82	R	0.88	0.84	<b>0.89</b>	0.86	S	0.94	0.93	<b>0.90</b>	<b>0.92</b>	A
		Commons-io																			
	App	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS
AF	Our	0.94	0.93	0.83	0.89	S	<b>0.81</b>	<b>0.77</b>	<b>0.73</b>	<b>0.75</b>	R	<b>0.94</b>	<b>0.94</b>	<b>0.80</b>	<b>0.88</b>	A	<b>0.97</b>	<b>0.97</b>	0.83	<b>0.91</b>	R
	Trad	<b>0.96</b>	<b>0.96</b>	<b>0.86</b>	<b>0.91</b>	R	0.78	0.73	0.72	0.73	S	0.92	0.92	0.83	0.88	R	0.96	0.96	<b>0.88</b>	0.93	R
StEv	Our	0.94	0.94	0.80	0.87	S	<b>0.81</b>	<b>0.77</b>	<b>0.73</b>	<b>0.75</b>	R	<b>0.94</b>	<b>0.94</b>	<b>0.86</b>	<b>0.90</b>	R	<b>0.97</b>	<b>0.97</b>	0.84	0.91	R
	Trad	<b>0.95</b>	<b>0.95</b>	<b>0.88</b>	<b>0.92</b>	S	0.78	0.74	<b>0.73</b>	0.73	S	0.92	0.92	0.78	0.86	A	0.96	0.96	<b>0.88</b>	<b>0.93</b>	R
EvSm	Our	0.89	0.88	0.64	0.78	A	<b>0.73</b>	<b>0.67</b>	<b>0.71</b>	<b>0.69</b>	S	<b>0.84</b>	<b>0.82</b>	<b>0.77</b>	<b>0.80</b>	R	0.95	0.95	0.79	0.88	A
	Trad	<b>0.94</b>	<b>0.93</b>	<b>0.86</b>	<b>0.90</b>	A	0.70	0.62	0.67	0.64	R	0.82	0.78	0.65	0.72	S	<b>0.96</b>	<b>0.96</b>	<b>0.90</b>	<b>0.94</b>	A
StSm	Our	<b>0.93</b>	<b>0.93</b>	<b>0.86</b>	<b>0.90</b>	A	<b>0.76</b>	<b>0.71</b>	0.68	<b>0.70</b>	S	<b>0.90</b>	<b>0.89</b>	0.83	<b>0.86</b>	S	<b>0.95</b>	<b>0.95</b>	0.85	<b>0.91</b>	A
	Trad	<b>0.93</b>	<b>0.93</b>	0.83	0.89	N	0.73	0.67	<b>0.71</b>	0.68	S	0.86	0.84	<b>0.88</b>	0.85	S	<b>0.95</b>	<b>0.95</b>	<b>0.86</b>	<b>0.91</b>	S
St	Our	<b>0.93</b>	<b>0.92</b>	<b>0.85</b>	<b>0.89</b>	A	<b>0.76</b>	<b>0.71</b>	0.68	<b>0.70</b>	S	<b>0.89</b>	<b>0.87</b>	0.84	<b>0.86</b>	A	<b>0.95</b>	<b>0.95</b>	0.85	<b>0.91</b>	R
	Trad	<b>0.93</b>	<b>0.92</b>	<b>0.85</b>	<b>0.89</b>	S	0.73	0.67	<b>0.70</b>	0.68	R	0.84	0.81	<b>0.89</b>	0.85	R	<b>0.95</b>	0.94	<b>0.86</b>	<b>0.91</b>	A
Ev	Our	0.88	0.86	0.66	0.77	S	<b>0.69</b>	<b>0.62</b>	<b>0.70</b>	<b>0.65</b>	S	<b>0.81</b>	<b>0.78</b>	<b>0.72</b>	<b>0.75</b>	R	0.95	0.95	0.78	0.88	S
	Trad	<b>0.94</b>	<b>0.93</b>	<b>0.86</b>	<b>0.90</b>	A	0.65	0.57	0.69	0.63	S	0.80	0.76	0.64	0.71	S	<b>0.96</b>	<b>0.96</b>	<b>0.91</b>	<b>0.94</b>	A
		Junit4																			
	App	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS	F1	Acc	Sen	AUC	RS
AF	Our	<b>0.97</b>	<b>0.97</b>	<b>0.82</b>	<b>0.90</b>	S	<b>0.90</b>	<b>0.87</b>	<b>0.78</b>	<b>0.83</b>	R	<b>0.96</b>	<b>0.96</b>	0.56	<b>0.77</b>	A	<b>0.98</b>	<b>0.98</b>	<b>0.80</b>	<b>0.90</b>	S
	Trad	0.92	0.90	0.53	0.72	A	0.83	0.76	0.71	0.74	N	0.91	0.88	<b>0.64</b>	<b>0.77</b>	R	0.94	0.95	0.40	0.69	S
StEv	Our	<b>0.97</b>	<b>0.97</b>	<b>0.79</b>	<b>0.88</b>	S	<b>0.90</b>	<b>0.87</b>	<b>0.79</b>	<b>0.83</b>	A	<b>0.96</b>	<b>0.96</b>	<b>0.62</b>	<b>0.80</b>	S	<b>0.98</b>	<b>0.98</b>	<b>0.79</b>	<b>0.89</b>	S
	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
EvSm	Our	<b>0.97</b>	<b>0.97</b>	<b>0.75</b>	<b>0.86</b>	A	<b>0.91</b>	<b>0.88</b>	<b>0.78</b>	<b>0.83</b>	S	<b>0.96</b>	<b>0.96</b>	0.70	<b>0.84</b>	S	<b>0.98</b>	<b>0.98</b>	<b>0.88</b>	<b>0.93</b>	S
	Trad	0.92	0.90	0.59	0.75	S	0.80	0.72	0.74	0.73	N	0.84	0.78	<b>0.78</b>	0.78	R	0.95	0.94	0.59	0.78	S
StSm	Our	<b>0.89</b>	<b>0.86</b>	<b>0.41</b>	<b>0.65</b>	S	<b>0.82</b>	<b>0.75</b>	0.60	<b>0.68</b>	N	<b>0.90</b>	<b>0.87</b>	<b>0.43</b>	0.66	R	<b>0.93</b>	<b>0.94</b>	0.22	<b>0.60</b>	A
	Trad	0.88	0.84	0.40	0.63	A	0.80	0.71	<b>0.65</b>	<b>0.68</b>	N	0.85	0.80	0.57	<b>0.69</b>	A	0.92	0.92	<b>0.23</b>	0.59	A
St	Our	<b>0.93</b>	<b>0.93</b>	0.24	0.60	N	<b>0.83</b>	<b>0.77</b>	0.57	<b>0.68</b>	R	<b>0.90</b>	<b>0.88</b>	0.38	0.64	S	<b>0.93</b>	<b>0.94</b>	0.21	<b>0.60</b>	A
	Trad	0.88	0.84	<b>0.46</b>	<b>0.66</b>	A	0.80	0.71	<b>0.64</b>	<b>0.68</b>	N	0.81	0.74	<b>0.63</b>	<b>0.69</b>	R	0.92	0.92	<b>0.23</b>	0.59	A
Ev	Our	<b>0.97</b>	<b>0.97</b>	<b>0.69</b>	<b>0.84</b>	S	<b>0.90</b>	<b>0.86</b>	<b>0.77</b>	<b>0.82</b>	A	<b>0.97</b>	<b>0.97</b>	<b>0.81</b>	<b>0.89</b>	R	<b>0.97</b>	<b>0.97</b>	<b>0.84</b>	<b>0.91</b>	S
	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
AF	Our	<b>0.92</b>	<b>0.91</b>	<b>0.75</b>	<b>0.84</b>	A	<b>0.81</b>	<b>0.77</b>	<b>0.71</b>	<b>0.74</b>	R	<b>0.90</b>	<b>0.89</b>	<b>0.78</b>	<b>0.84</b>	R	<b>0.94</b>	<b>0.94</b>	<b>0.73</b>	<b>0.85</b>	A
	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
StEv	Our	<b>0.92</b>	<b>0.92</b>	<b>0.73</b>	<b>0.84</b>	A	<b>0.80</b>	<b>0.77</b>	<b>0.71</b>	<b>0.74</b>	S	<b>0.91</b>	<b>0.91</b>	<b>0.77</b>	<b>0.85</b>	R	<b>0.94</b>	<b>0.94</b>	<b>0.73</b>	<b>0.85</b>	S
	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
EvSm	Our	<b>0.91</b>	<b>0.90</b>	<b>0.71</b>	<b>0.82</b>	A	<b>0.79</b>	<b>0.74</b>	<b>0.69</b>	<b>0.72</b>	A	<b>0.90</b>	<b>0.89</b>	<b>0.81</b>	<b>0.85</b>	A	<b>0.93</b>	<b>0.92</b>	<b>0.79</b>	<b>0.86</b>	A
	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
StSm	Our	<b>0.87</b>	<b>0.87</b>	0.46	<b>0.69</b>	R	<b>0.76</b>	<b>0.71</b>	<b>0.65</b>	<b>0.69</b>	R	<b>0.82</b>	<b>0.78</b>	<b>0.80</b>	<b>0.79</b>	R	<b>0.88</b>	<b>0.88</b>	<b>0.53</b>	<b>0.73</b>	A
	Trad	0.76	0.71	<b>0.49</b>	0.61	R	0.75	0.70	0.61	<b>0.66</b>	R	0.75	0.69	0.71	0.70	R	0.76	0.71	0.51	0.62	R
St	Our	<b>0.84</b>	<b>0.82</b>	<b>0.54</b>	<b>0.69</b>	S	<b>0.76</b>	<b>0.71</b>	<b>0.65</b>	<b>0.69</b>	S	<b>0.83</b>	<b>0.80</b>	<b>0.78</b>	<b>0.79</b>	R	<b>0.88</b>	<b>0.88</b>	<b>0.54</b>	<b>0.73</b>	A
	Trad	0.76	0.71	0.50	0.62	R	0.73	0.67	<b>0.65</b>	0.66	A	0.75	0.70	0.71	0.70	R	0.76	0.71	0.51	0.62	R
Ev	Our	<b>0.88</b>	<b>0.87</b>	<b>0.74</b>	<b>0.81</b>	A	<b>0.83</b>	<b>0.80</b>	<b>0.61</b>	<b>0.72</b>	A	<b>0.88</b>	<b>0.86</b>	<b>0.85</b>	<b>0.86</b>	A	<b>0.89</b>	<b>0.88</b>	<b>0.85</b>	<b>0.87</b>	A
	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
AF	Our	<b>0.93</b>	<b>0.92</b>	<b>0.76</b>	<b>0.85</b>	S	<b>0.77</b>	<b>0.73</b>	<b>0.80</b>	<b>0.76</b>	R	<b>0.89</b>	<b>0.88</b>	<b>0.71</b>	<b>0.81</b>	A	<b>0.93</b>	<b>0.93</b>	<b>0.78</b>	<b>0.87</b>	A
	Trad	0.80	<b>0.77</b>	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
StEv	Our	<b>0.93</b>	<b>0.92</b>	<b>0.78</b>	<b>0.86</b>	A	<b>0.77</b>	<b>0.73</b>	<b>0.79</b>	<b>0.75</b>	R	<b>0.89</b>	<b>0.89</b>	<b>0.64</b>	<b>0.78</b>	A	<b>0.94</b>	<b>0.93</b>	<b>0</b>		

**Table 7.** Number of times each approach reached the best value for each indicator and model considering all systems

AF					StEv				
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
EvSm					StSm				
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
St					Ev				
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

**Table 8.** Effect size results obtained by comparing both approaches

Indicator	Approach	Avg	Std	Magnitude
AUC	Trad	0.7036	± 0.104	medium
	Our	0.7755	± 0.093	best
Accuracy	Trad	0.8282	± 0.107	small
	Our	0.8803	± 0.092	best
F1-score	Trad	0.8517	± 0.082	small
	Our	0.8952	± 0.073	best
Sensitivity	Trad	0.5541	± 0.217	small
	Our	0.6510	± 0.182	best

els of our approach obtained the best values in 102 (85%) models, against only 14 in the traditional one and 4 equivalents. Regarding Sensitivity, our approach also overcame the traditional one reaching the best values in 78 (65%) models, against 37 in the traditional approach and 5 equivalents. If we consider all indicators, algorithms, and sets (480 cases), our approach achieved the best results in 378 cases (~79%), and equivalent ones in 21 (4.3%).

Figure 8 shows the boxplots comparing both approaches for each indicator using the Wilcoxon test. In all cases, there is a statistical difference in the distribution between our and traditional approaches with confidence level >95% ( $p\text{-value} \leq 0.001$ ). Table 8 shows that the magnitude was medium for AUC and small for the other indicators. We conducted the same analysis for each model individually and obtained similar results. Our approach performs better for all the models, obtained with the six sets of features. The statistical analysis is detailed in our repository.

The time to execute the algorithms is similar for both approaches. The biggest bottleneck is found in the data collection and preprocessing steps. However, this problem affects both approaches.

**Response to RQ2:** Our approach overcomes the traditional one in ~ 79% of the cases. The performance of our approach is better regardless of the indicators and the set of features. Then, we can conclude that our approach improves the performance of the six kinds of models analyzed.

### 5.3 RQ3: What is the performance of the different kinds of models generated by our approach?

Figure 9 compares the mean values of the indicators for each type of model considering all algorithms to the five systems. We derived this figure from Table 6 to analyze the effect of different sets of features on the performance of our approach. We can observe that the performance of the models varies according to the system. In general, when considering the results for the five systems, our approach presented a better performance for AF, StEv, EvSm, and Ev but showed poor results for St and StSm. The St and StSm models do not include evolutionary metrics. In our approach, structural metrics are not good predictors when used alone or combined with smell-based metrics. Some studies on the traditional approach reported similar results [Lu *et al.*, 2012; Zhou *et al.*, 2009; Tsantalis *et al.*, 2005]. In contrast, evolutionary metrics were good predictors in our approach, even when used alone (Ev). We also analyzed the median values of each set of features. We did not observe any impact on the results. Additional figures presenting median values are available in our repository.

Table 9 presents the number of times each kind of model achieved the best values for each indicators considering all the systems. We can observe that evolutionary models (Ev) achieve the best values of Sensitivity in 4 out of 5 systems. The StEv set achieves three best values for AUC, F1-score, and Accuracy. The AF set reaches two best values for AUC and one for F1-score and Accuracy. The mean values for each model for all the indicators are available in our repository.

**Table 9.** Number of times each type of model reached the best value for each indicator considering all systems

Indicator	AF	Ev	EvSm	St	StEv	StSm
AUC	2	0	0	0	3	0
F1-score	1	1	0	0	3	0
Sensitivity	0	4	0	0	0	1
Accuracy	1	0	1	0	3	0
Total	4	5	1	0	9	1

We also applied the Kruskal-Wallis test to compare the models. Figure 10 shows the results for each indicator. The p-values were less than 0.01, indicating a statistical difference between the models. The StSm and St sets have the lowest median values for all indicators. Table 10 presents the effect size values. The StEv obtained the best values of AUC, Accuracy, and F1-score with no statistical difference compared to AF, Ev, and EvSm. However, there was a statistically significant difference compared to St and StSm. For Sensitivity, the best model is Ev, with no statistical difference with AF, EvSm, and StEv but presenting significant statistical difference with St and StSm.

**Response to RQ3:** Our approach presented the best performance for the models that include evolutionary metrics (AF, Ev, EvSm, and StEv). There is no statistical



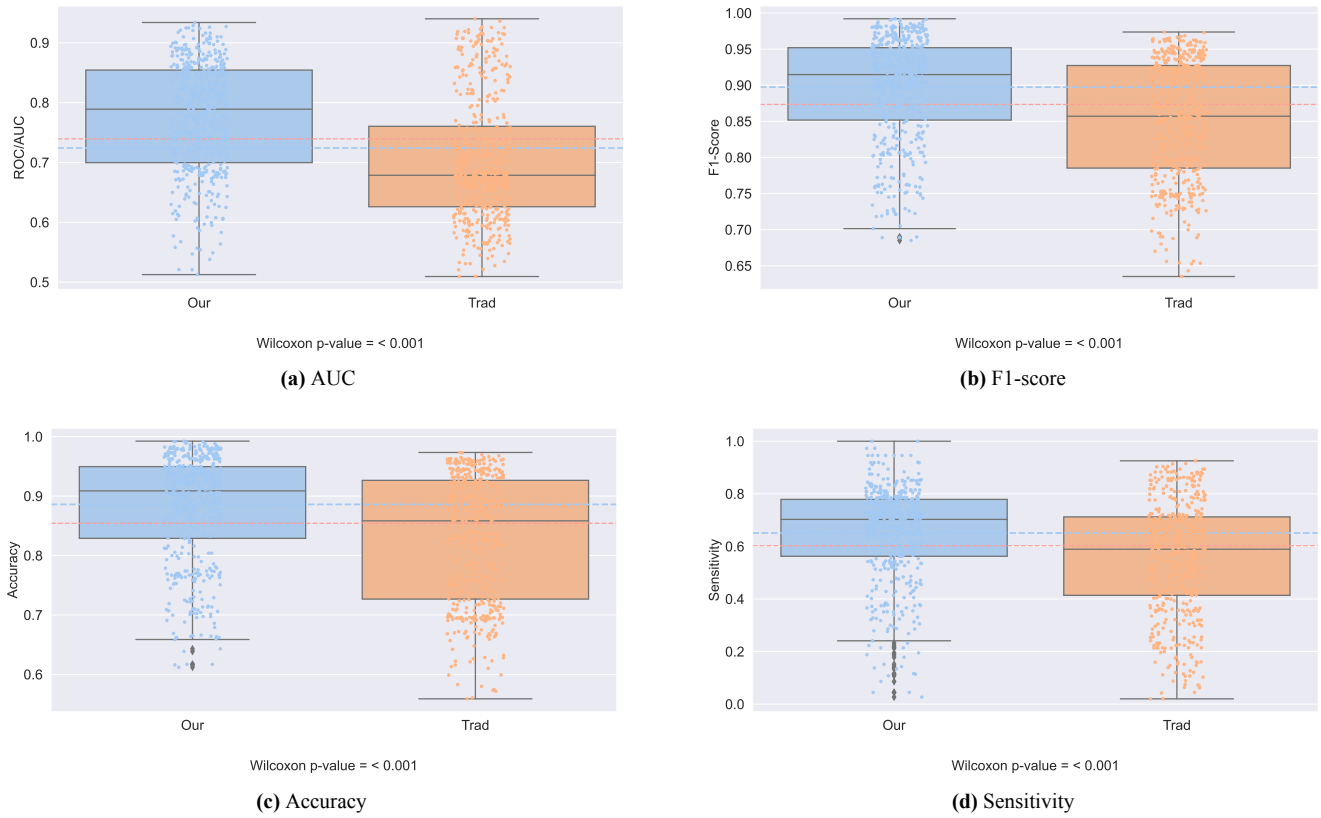


Figure 8. Boxplots comparing both approaches.

Table 10. Effect size results obtained comparing the models

Indicator	Model	Avg	Std	Magnitude
AUC	AF	0.8469	± 0.055	negligible
	Ev	0.8328	± 0.074	negligible
	EvSm	0.8369	± 0.066	negligible
	St	0.7228	± 0.090	large
	StEv	0.8482	± 0.055	best
	StSm	0.7287	± 0.087	large
Accuracy	AF	0.9115	± 0.076	negligible
	Ev	0.8754	± 0.101	small
	EvSm	0.8853	± 0.097	small
	St	0.8366	± 0.092	large
	StEv	0.9126	± 0.076	best
	StSm	0.8395	± 0.090	large
F1-score	AF	0.9218	± 0.062	negligible
	Ev	0.8944	± 0.081	small
	EvSm	0.9018	± 0.078	small
	St	0.8620	± 0.070	large
	StEv	0.9228	± 0.062	best
	StSm	0.8645	± 0.070	large
Sensitivity	AF	0.7701	± 0.073	negligible
	Ev	0.7830	± 0.094	best
	EvSm	0.7801	± 0.076	negligible
	St	0.5889	± 0.203	large
	StEv	0.7714	± 0.071	negligible
	StSm	0.5982	± 0.187	large

difference between these models, but the best result was obtained by StEv, i.e., combining structural and evolutionary metrics. The results show a large statistical difference between the best model and models combining structural and smell-based metrics (StSm) or structural metrics alone (St). These models lead to lower performance, considering all indicators.

#### 5.4 RQ4: Which algorithm leads our approach to produce the best results?

Figure 11 shows the results comparing the algorithms for each indicator considering all models and using the Kruskal-Wallis test. We can observe that, except for Sensitivity, the p-value is less than 0.001, which means a statistical difference between the algorithms. The RF algorithm presents the best performance for AUC, F1-score, and Accuracy indicators. The LR is far from the other algorithms. For Sensitivity, LR is the best algorithm but with no statistical difference (p-value = 0.241).

Table 11 shows the effect size. For the AUC, RF is the best algorithm but there is no statistical difference from MLP or DT. There is a statistical difference with LR of medium magnitude. Regarding Accuracy, RF has a statistically significant difference from MLP and LR. For the F1-score, this difference is medium from MLP and large from LR. These results indicate that RF has the best overall performance, followed by DT and MLP. The LR is only a good algorithm

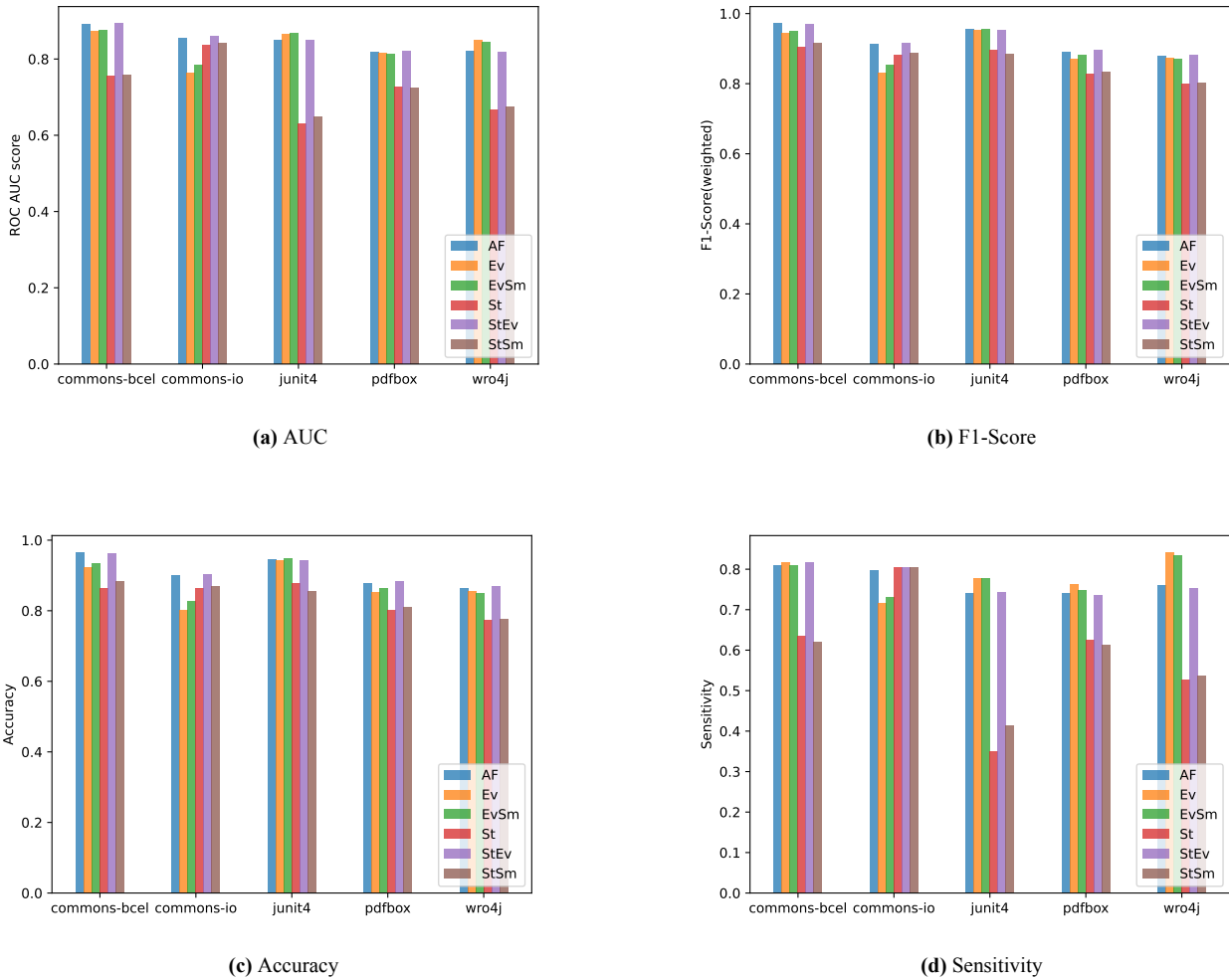


Figure 9. Average performance of the different kinds of models per indicator

to ensure more correct classifications due to its good performance for Sensitivity.

Table 11. Effect size results obtained by comparing the algorithms

Indicator	Algorithm	Avg	Std	Magnitude
AUC	DT	0.8063	± 0.088	small
	LR	0.7565	± 0.078	medium
	MLP	0.8184	± 0.068	small
	RF	0.8296	± 0.106	best
Accuracy	DT	0.9160	± 0.058	medium
	LR	0.7602	± 0.079	large
	MLP	0.8875	± 0.061	large
	RF	0.9436	± 0.038	best
F1-score	DT	0.9222	± 0.051	small
	LR	0.8090	± 0.068	large
	MLP	0.9029	± 0.050	medium
	RF	0.9442	± 0.038	best
Sensitivity	DT	0.6765	± 0.150	negligible
	LR	0.7522	± 0.104	best
	MLP	0.7373	± 0.126	negligible
	RF	0.6951	± 0.209	negligible

We also analyze the performance of the algorithms by using each set of features. Figure 12 shows the results for AUC.

The best performance of RF is for AF, Ev, EvSm, and StEv models. For the StSm and St models, MLP performs better. However, there is a statistical difference only for the models EvSm and EV. The RF algorithm presents a significant difference in comparison with the other algorithms in most cases, as we can see in Table 12. A complementary analysis of the results of the other indicators can be found in our repository.

Table 12. Effect size of the models and algorithms

Indicator	Algorithm	Avg	Std	Effect
Ev	DT	0.8254	± 0.034	large
	LR	0.7630	± 0.098	large
	MLP	0.8542	± 0.060	medium
	RF	0.8884	± 0.029	best
EvSm	DT	0.8327	± 0.036	large
	LR	0.7706	± 0.085	large
	MLP	0.8460	± 0.031	large
	RF	0.8983	± 0.030	best

We also analyzed the use of the resampling techniques. Based on Table 6, we present Table 13 to show the number of times each algorithm uses the resampling techniques

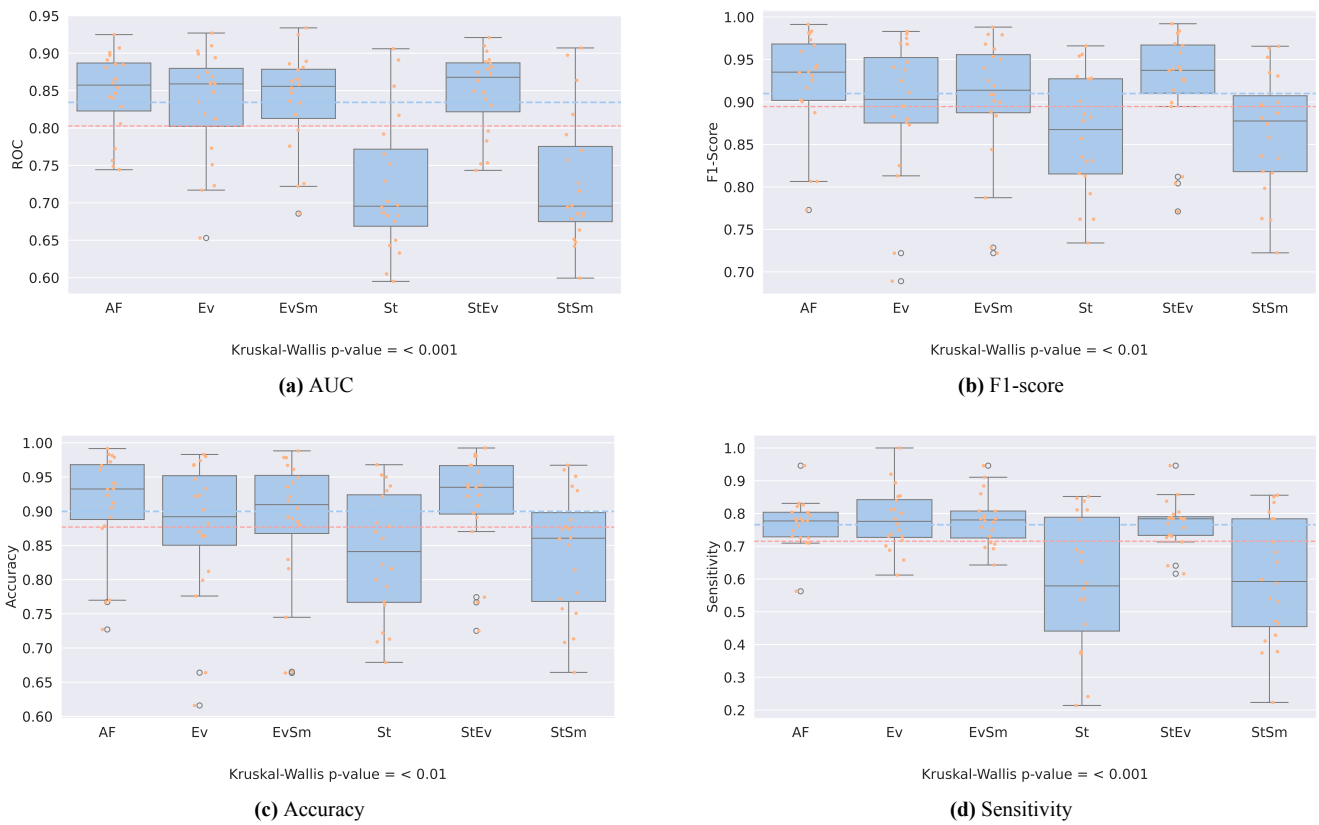


Figure 10. Boxplots comparing the kinds of models.

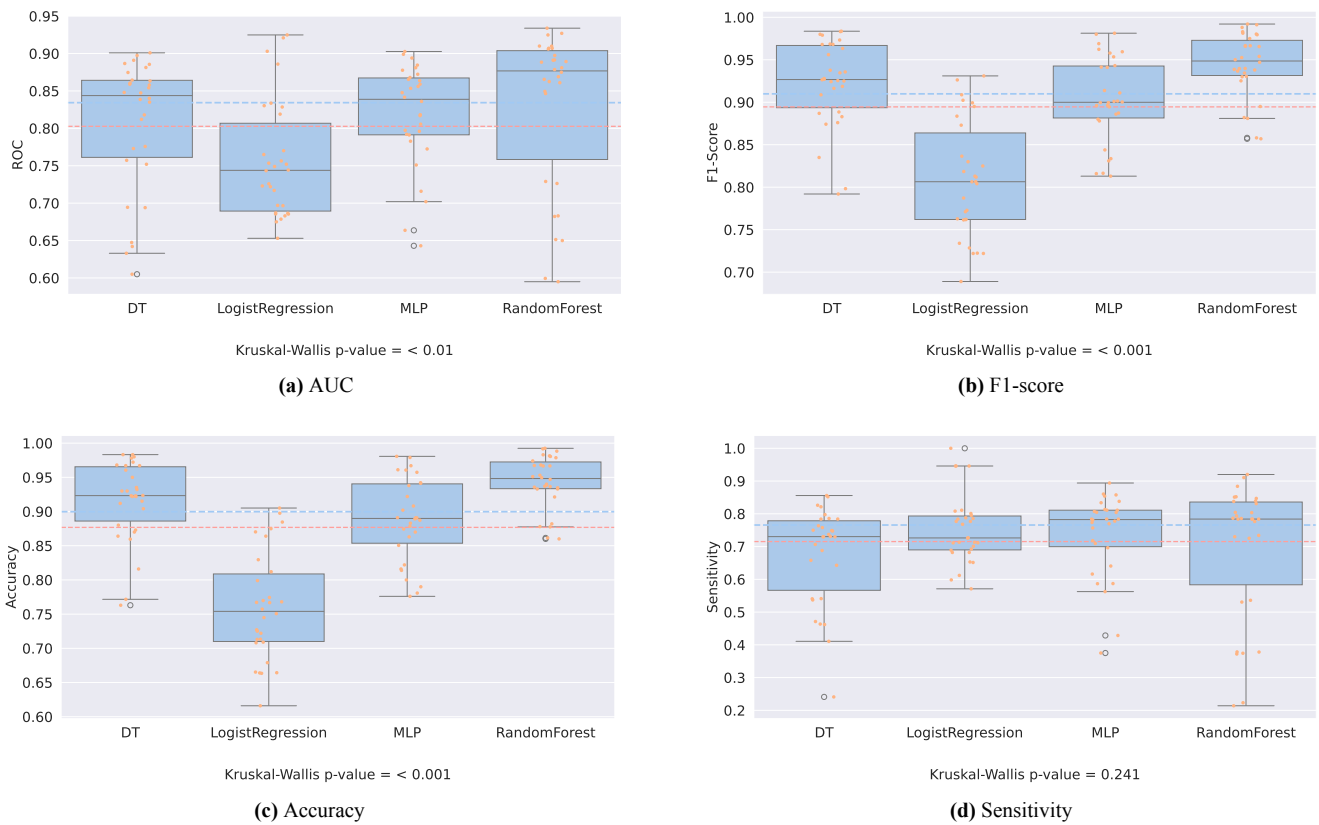
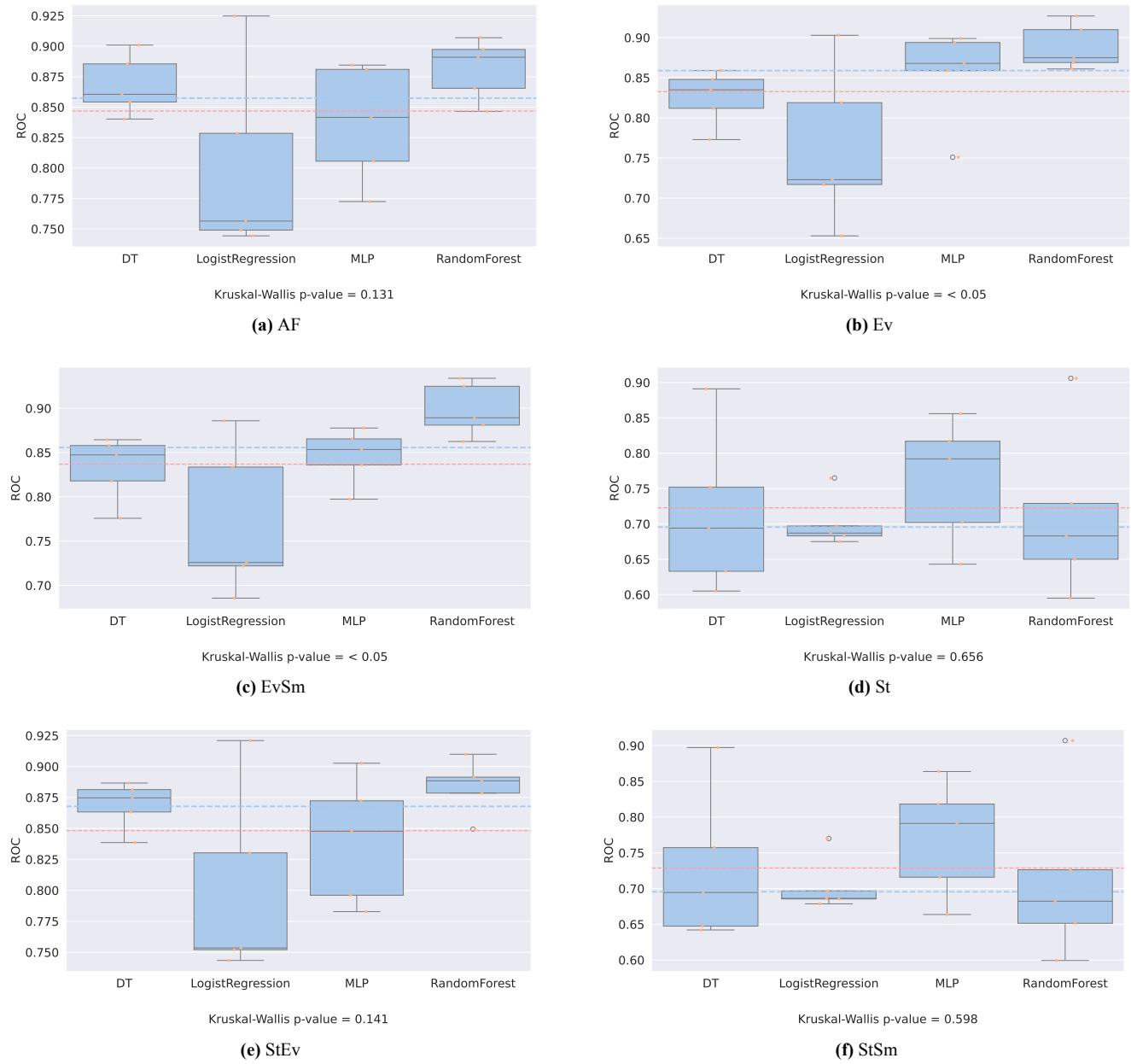


Figure 11. Boxplots comparing the algorithms per indicator

to generate the best models in both approaches, according to AUC. For our approach, we observe that, in all 30 cases (5

systems and 6 models), the RF algorithm obtained the best results using ADA (17 times). The same occurred for the tra-



**Figure 12.** Boxplots comparing the AUC values of the algorithms per kinds of models

ditional approach, using ADA 15 times. In both approaches, the performance of the DT algorithm is better using ADA (14 times). For the LR algorithm, SMOTE appears 10 times in both approaches. The MLP algorithm performed better in both approaches using ROS, appearing 16 times in our approach and 15 times in the traditional one. We can conclude that the ADA presents the better performance for the RF and DT algorithms.

**Response to RQ4:** The analysis indicates that our approach achieved the best results with RF for the AUC, F1-score, and Accuracy indicators, followed by DT and MLP. LR presents the best values of Sensitivity. We can highlight that ADA is the resampling technique with the best general performance for RF and DT. The RF presents the best results for Ev, EvSm, StEv, and AF

**Table 13.** Comparing resampling methods

Algorithm	ADA	NONE	ROS	SMOTE
Our				
DT	14	2	4	10
LR	9	1	10	10
MLP	9	0	16	5
RF	<b>17</b>	0	5	8
Traditional				
DT	14	1	6	9
LR	3	8	9	10
MLP	5	0	15	10
RF	<b>15</b>	0	4	11

models. However, MLP is the best for St the and StSm.

## 5.5 RQ5: Which metrics are the most important for change-proneness class prediction?

We evaluated the set containing all features (AF) with MDI and Information Gain methods to determine the most important features for change-proneness class prediction. The results of both methods for each dataset are in our repository.

To evaluate the information gain of each feature, we used the `mutual_info_classif` method of scikit-learn with default settings. First, we ranked the top 10 features for each dataset of each system individually. Then, we created a unique set of 50 features, which corresponds to the top 10 of each dataset. In this set of 50 features, we intersected the 10 most repeated ones. Table 14 presents this ranking result for Information Gain. The best performance in all datasets are FRCH (FrequencyOfChanges), TACH (TotalAmountOfChanges), CHD (ChangeDensity), and `sumCyclomaticModified` features. The BOC (BirthOfClass), `finalFieldsQty`, FCH (FirstChange), LCH (LastChange), and WFR (WeightedFrequencyOfChanges) appear in 4 out of 5 datasets. The CHO (ChangeOccurred) is the last one in the ranking, placed among the top 10 in just 2 datasets. We can observe that the evolutionary metrics performed better. There are only two structural metrics, `SumCyclomaticModified` and `finalFieldsQty`.

Similarly, we ranked with MDI the 10 most important features for each dataset using `ExtraTreesClassifier` method of scikit-learn. Then, we created a set of 50 features containing the top 10 features of each dataset. In this set of 50 features, we intersected the 10 most repeated ones. Table 14 also presents the result of this ranking for MDI. In all datasets, LCH (LastChange), FRCH (FrequencyOfChanges) and WFR (WeightedFrequencyOfChanges) appear in the top 10. The ACDF (AgregatedChangeDensityFrequency), ATAF (AgregatedChangeSizeNormalizedbyFrequencyOfChange), FANIN, FANOUT, CHG (ChangeDensity) and CSB (ChangeSinceBirth) are at the top 10 in 4 out of 5 datasets, followed by WCD (WeightedChangeDensity) in 3 out of 5. Only two of the top 10 are structural metrics (FANIN and FANOUT), which are associated with class coupling.

The LCH (LastChange), FRCH (FrequencyOfChanges), CHD (ChangeDensity), and WFR (WeightedFrequencyOfChanges) features appeared in both ranks. Density and diversity of do not appear between the top 10 features using both methods, even when any dataset is considered individually. These results reinforce RQ3 findings, which state that these metrics do not affect the change-proneness class prediction and that the evolutionary metrics are important for CPCP problems.

We also performed a correlation analysis between the features. The results are available in our repository. In summary, in all systems, we observed correlations between the following structural metrics: WMC (WeightedMethodsper-

Class) and `MethodsQty`, WMC and LOC (LinesOfCode), WMC and `SumCyclomatic`, RFC (ResponseForaClass) and LOC, RFC and `assignmentsQty`, RFC and `variablesQty`, RFC and `SumCyclomatic`, `MethodsQty` and LOC, `MethodsQty` and `SumCyclomatic`, LOC and `AssignmentsQty`, LOC and `VariablesQty`, LOC and `SumCyclomatic`, DIT (DepthofInheritanceTree) and `MaxInheritanceTree`. In summary, we can observe that the cyclomatic complexity is correlated to some of the other structural metrics. Future studies should better investigate the impact of removing these correlated metrics. There are no strong correlations related to other structural metrics such as `TotalFields`, TCC and LCC, for instance, and evolutionary features. This reinforces their importance.

**Response to RQ5:** The results obtained by MDI and Information Gain methods reinforce the importance of the evolutionary metrics for the CPCP problem. Evolutionary metrics are 8 out of the top 10 features in the rank of both methods, and the other two are structural. The `LastChange`, `FrequencyOfChanges`, `ChangeDensity`, and `WeightedFrequencyOfChanges` features appeared in both ranks.

In the next section, we discuss the implications of the results.

## 6 Implications

The findings obtained in the analysis of our RQs have some implications that help developers in practice and also point out some research opportunities.

### 6.1 Practical Implications

Our approach improves performance in ~80% of the cases compared to the traditional approach, considering all systems, indicators, models, and algorithms. This means that it generates better class change-proneness prediction models and is the best option for developers. This allows developers to plan preventive maintenance operations and allocate resources more efficiently. For instance, developers should pay more attention to change-prone classes in the next release, spending more time coding, refactoring, reviewing, and testing these classes. This could lead to a lower failure rate, reduced technical debt, and maintenance costs.

Consider the following scenario to illustrate a practical implication. A developer has added a new feature for a system. However, this feature was implemented hastily because other priorities arose, such as fixing a security bug. Because the new functionality was not programmed consistently, technical debt was created. This debt may be related to quality indicators, such as lack of cohesion or high coupling. These indicators are captured by the metrics used as independent variables in our approach. Then, this class will be predicted as change-prone, leading to fast identification of the debt by developers and reducing future costs.



**Table 14.** Top 10 features for all datasets

# Datasets	Information Gain	MDI
5	SumCyclomaticModified	LCH (LastChange)
	FRCH (FrequencyOfChanges)	FRCH (FrequencyOfChanges)
	TACH (TotalAmountOfChanges)	WFR (WeightedFrequencyOfChanges)
	CHD (ChangeDensity)	
4	BOC (BirthOfClass)	ACDF (AgregatedChangeDensityFrequency)
	finalFieldsQty	ATAF (AgregatedChangeSizeNormalized byFrequencyOfChanges), FANIN, FANOUT
	FCH (FirstChange)	CHD (ChangeDensity)
	LCH (LastChange)	CSB (ChangeSinceBirth)
3	-	WCD (WeightedChangeDensity)
2	CHO (ChangeOccurred)	-

According to RQ1, our approach does not require a long change history. A period of 3 releases may be enough, regardless of the set of features used. However, metrics need to be collected. We think that the software industry has already adopted this practice, and many tools that can be employed. Moreover, RQ3 and RQ5 findings show that not all the metrics evaluated in our study as independent variable needs to be used. Evolutionary metrics should be prioritized to reduce collection costs. For developers, the use of a lower number of features can be advantageous since it makes models simpler. Our results show that adding density and diversity of smells does not improve the performance of our approach. According to RQ4, regardless of the set of features used, RF is the best option for the developer. However, our approach achieves good performance with simpler algorithms, such as LR, with values of AUC greater than 0.7. The MLP is a good choice when using structural metrics.

## 6.2 Implication for research

We observe that larger window sizes do not improve the results. The larger the window size, the greater the class history required, thus excluding classes added in more recent versions. This reduces the training data and can generate bias towards older classes. However, further studies should be conducted to better characterize the life of a class along the project and its relationship with changes. It is expected that newborn classes are more change-prone, whereas more mature classes are not. Testing other window thresholds would be interesting to investigate the boundaries between small and large windows. Another possible research opportunity is to study the capacity of our approach in generating change-proneness prediction models considering a number  $x$  of releases ahead.

The RQ3 shows that the models presenting the best performance in our approach are based on evolutionary metrics. Our findings are different from the studies using the traditional approach. For instance, Catolino *et al.* [2020] shows that adopting code smell-related information improves the performance of different categories of models obtained with the traditional approach. Moreover, studies do not show a consensus on using structural metrics [Lu *et al.*, 2012; Zhou *et al.*, 2009; Tsantalis *et al.*, 2005]. New studies should evaluate the impact of the features according to the domain and kind of system being developed. In such studies, other kinds

of smells can be considered, as well as other smell-based metrics.

According to RQ4, our approach obtained similar results to the literature when applying the traditional approach. The Random Forest algorithm presents the best performance for the CPCP problem [Malhotra and Khanna, 2019]. Thus, smarter algorithms lead our approach to produce better results. We also observe that the performance of resampling techniques can vary according to the algorithms. The Random Forest and Decision Tree algorithms performed better with ADA. This preprocessing technique and others should be further investigated. A research opportunity is to investigate other algorithms, for instance, Deep Learning neural networks. They naturally can analyze the state of variables in different versions, e.g. long short-term memory (LSTM) networks can be used for speech recognition and time series prediction, among others. For this, the datasets should also be expanded since these networks demand a large amount of data.

Krüger *et al.* [2024] analyzed studies on Software-Change Intentions (SCIs) by using five dimensions: 1) Goals: purpose for which developers perform a change; 2) Actions: concrete activities developers perform to achieve their goals, which can be simple, or compound. This last one combines multiple simple actions); 3) Objects: artifacts manipulated by an action; 4) Customer: individual or entity interested in the change; and 5) Lifecycle: related to the development phase.

In our study the Object/Artifact is always the code, and we only focus on the dimension Actions. The activities performed are simple changes pointed out by the tool *ChangeDistiller*, as adopted in the literature. However, another way to define the dependent variable could be used, considering, e.g., only compound and more complex changes. Another research direction is to classify the changes according to the intentions. A database should be generated that relates changes with the developers' goals, or other dimensions proposed by Krüger *et al.* [2024]. The different intentions of the developers mentioned in their work could be predicted such as bug fixing, refactoring, testing, and resource configuration.

## 7 Threats to Validity

This section presents some limitations and threats to the validity of our results, according to the taxonomy of Wohlin *et al.* [2000].

**Internal Validity:** Possible threats are related to mistakes in data extraction, selection, and preparation and preprocessing steps in our approach. To minimize them, we followed a methodology commonly reported in the literature and provided the data for a possible replication study. The ML algorithms were configured with standard parameters. Then, tuning the parameters can lead to better results. The number of instances used for training the algorithms varies between 5383 and 111.917, but this may be a threat.

**Construct validity:** The selection of the features and how they are combined to compose the kinds of models is possibly a threat. For instance, the use of other smells or other smell-based metrics may lead to other findings. The good performance of the evolutionary metrics may be related to how we calculated the dependent variable, which does not consider a simple class change. The independent and dependent variables were selected based on the literature to mitigate this. We also did not analyze the independent variables to select the projects. However, other metrics and tools used to collect them may impact the results.

**External Validity:** A threat is related to the systems used. We analyze 5 systems with different sizes, diverse application domains, and from distinct developers. However, all systems were developed in Java. Therefore, we cannot generalize the obtained results to a different context.

**Conclusion Validity:** The conclusions may depend on the indicators used in our analysis. To minimize this threat, we employed different indicators and statistical tests. Our findings depend on the formulated RQs. Other RQs could lead to different implications.

## 8 Conclusions

Change-prone classes are more likely to change due to software development, refactoring, and maintenance activities. The early identification of these classes is useful for the software development team to reduce the cost and resource consumption [Elish and Al-Khiaty, 2013; Catolino *et al.*, 2017]. The approach introduced in this study adopts the sliding window method to incorporate the change history of the class to solve the CPCP problem. It is more realistic since it considers the structure of the problem and the temporal dependency between the instances obtained from different releases. Each learning instance is not limited to a fixed release, but it contains the total or partial change history of the class.

The performance of the approach was evaluated to derive six kinds of models, by combining structural, evolutionary, and smell-based metrics as predictors. We analyzed three window sizes (2, 3, and 4). The results point out that a change-prone class prediction does not necessarily require a long history. In the systems and kinds of models evaluated, three releases were enough. We compared our approach with the traditional approach. The results show that our approach,

regardless of the algorithm and set of features, improves the performance in most cases.

The Random Forest algorithm presented the best individual performance. Our approach works well with other tested algorithms, e.g., MLP and Decision Trees. We achieved the best results when applying the resampling techniques, e.g. ADA. We also evaluated the importance of features in the change-proneness prediction. 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 do not appear among the most important ones. The models including evolution-based metrics derived from our approach are the ones that reach the best performance for all systems, indicators, and algorithms.

This study can lead researchers to understand the aspects of history-based representation and the role of evolutionary, structural, and smell-related metrics in change-prone class prediction. Researchers can develop more precise tools to recommend change-prone classes. In the practical field, our approach can help developers to better focus on change-prone classes to minimize the number of future changes and guide maintenance team and resource distributions to reduce future efforts and costs.

In this study, we used some features from open-source Java applications or those that can be extracted for Object-Oriented software. However, our approach is language-independent and can be used to obtain models adopting other features not explored here, which are specific to other languages and contexts, such as Test classes and Highly-Configurable Software. In future work, we intend to evaluate other languages and sets of applications from the same domain to create more specific models for these problems. We analyzed the features but did not generate a model with the best features. This would be interesting in future studies to compare and investigate the possibility of adopting a general model for all systems. Other sets of metrics, such as the ones captured dynamically should be investigated, as well as the use of deep learning.

## Declarations

### Funding

This research was funded by CAPES (Coordination for the Improvement of Higher Education Personnel) and CNPq (National Council of Scientific and Technological Development) (grant: 310034/2022-1).

### Authors' Contributions

All authors contributed to writing and editing the text. Rogério and Paulo contributed to the work conception, methodology, implementation of the algorithms, conduction of the experiments, and analysis of the results. Silvia contributed to the conception, supervision, and analysis of the results. All authors read and approved the final manuscript.

### Competing interests

The authors declare no conflict of interest.

## Availability of data and materials

The datasets generated and/or analyzed during the present study are available at <https://github.com/carvalho7976/Change-History-Data-to-Enhance-Class-Change-Proneness-Prediction-Models>

## 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). Available at: <https://ccis2k.org/iajit/PDF/vol.1%2014,%20no%206/10840.pdf>.
- Aniche, M. (2015). *Java code metrics calculator (CK)*. Available at: <https://github.com/mauricioaniche/ck/>.
- Arcuri, A. and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE'11*, page 1–10, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1985793.1985795.
- Arisholm, E., Briand, L., and Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506. DOI: 10.1109/TSE.2004.41.
- Bansiya, J. and Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17. DOI: 10.1109/32.979986.
- 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. DOI: 10.1145/1007730.1007735.
- 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. DOI: 10.1109/METRIC.2003.1232454.
- Breiman, L. (2001). Random forests. *Machine learning*, 45:5–32. DOI: 10.1023/A:1010933404324.
- Brown, W., Malveau, R., Brown, W., McCormick, H. I., and Mowbray, T. (1999). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Addison-Wesley. Book.
- Brownlee, J. (2020). *Data Preparation for Machine Learning: Data Cleaning, Feature Selection, and Data Transforms in Python*. Machine Learning Mastery. Available at: <https://books.google.com.br/books?id=uAPuDwAAQBAJ>.
- 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*, pages 77–83. Available at: [https://www.researchgate.net/publication/246793187\\_Measuring\\_and\\_Predicting\\_the\\_Linux\\_Kernel\\_Evolution](https://www.researchgate.net/publication/246793187_Measuring_and_Predicting_the_Linux_Kernel_Evolution).
- 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. DOI: 10.1109/MALTESQUE.2018.8368455.
- 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. DOI: 10.1002/smr.2156.
- 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. DOI: 10.1109/ICPC.2017.19.
- 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. DOI: 10.1016/j.jss.2018.05.003.
- 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. DOI: 10.1007/s10664-019-09739-0.
- 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. DOI: 10.1613/jair.953.
- Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493. DOI: 10.1109/32.295895.
- Dietterich, T. G. (2002). Machine learning for sequential data: A review. In *Structural, Syntactic, and Statistical Pattern Recognition*, pages 15–30, Berlin, Heidelberg. Springer Berlin Heidelberg. DOI: 10.1007/3-540-70659-32.
- Elish, M. O. and Al-Khiaty, M. A. (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. DOI: 10.1002/smr.1549.
- 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. DOI: 10.1109/ICSTW.2011.43.
- 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. DOI: 10.1109/TSE.2007.70731.
- Fowler, M. (1999). *Refactoring – Improving the Design of Existing Code*. Addison-Wesley. Available at: <http://martinfowler.com/books/refactoring.html>.
- Gall, H. C., Fluri, B., and Pinzger, M. (2009). Change anal-

- ysis with Evolizer and ChangeDistiller. *IEEE software*, 26(1):26–33. DOI: 10.1109/MS.2009.6.
- 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. DOI: 10.1109/MSR.2012.6224284.
- 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. Available at: <https://ijcaonline.org/archives/volume105/number3/18361-9502/>.
- Han, J., Pei, J., and Tong, H. (2022). *Data mining: concepts and techniques*. Morgan kaufmann. DOI: 10.1016/C2009-0-61819-5.
- 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. DOI: 10.1109/IJCNN.2008.4633969.
- Ilyas, I. and Chu, X. (2019). *Data Cleaning*. ACM Collection II Series. Association for Computing Machinery. DOI: 10.1145/3310205.
- 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. DOI: 10.1007/978-981-10-3153-3\_56.
- 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. DOI: 10.1007/978-3-030-75251-4\_7.
- 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. DOI: 10.1109/WCRE.2009.28.
- Khomh, F., Di Penta, M., 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. DOI: 10.1007/s10664-011-9171-y.
- 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. DOI: 10.1109/TSE.2005.89.
- Krüger, J., Li, Y., Lossev, K., Zhu, C., Chechik, M., Berger, T., and Rubin, J. (2024). A meta-study of software-change intentions. *ACM Comput. Surv.* DOI: 10.1145/3661484.
- 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. DOI: 10.2307/2280779.
- Lanza, M., Marinescu, R., and Ducasse, S. (2010). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer. DOI: 10.1007/3-540-39538-5.
- Lindvall, M. (1998). Are large C++ classes change-prone? An empirical investigation. *Journal of Software: Practice and Experience*, 28(15):1551–1558. DOI: 10.1002/(SICI)1097-024X(19981225)28:15<1551::AID-SPE212>3.0.CO;2-0.
- 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. DOI: 10.1007/s10664-011-9170-z.
- Malhotra, R. and Bansal, A. (2015). Predicting change using software metrics: A review. In *IEEE International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 1–6. DOI: 10.1109/ICRITO.2015.7359253.
- Malhotra, R., Kapoor, R., Aggarwal, D., and Garg, P. (2021a). 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. DOI: 10.1109/MSR52588.2021.00015.
- Malhotra, R., Kapoor, R., Aggarwal, D., and Garg, P. (2021b). 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. DOI: 10.1109/MSR52588.2021.00015.
- 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. DOI: 10.1007/s13042-012-0095-7.
- Malhotra, R. and Khanna, M. (2018a). Particle swarm optimization-based ensemble learning for software change prediction. *Information and Software Technology*, 102:65–84. DOI: 10.1016/j.infsof.2018.05.007.
- 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. DOI: 10.3233/JIFS-169468.
- Malhotra, R. and Khanna, M. (2019). Software change prediction: A systematic review and future guidelines. *e-Informatica Software Engineering Journal*, 13(1):227–259. DOI: 10.37190/e-inf.
- 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*. DOI: 10.1007/s13198-021-01099-7.
- Malhotra, R. and Lata, K. (2020). An empirical study on predictability of software maintainability using imbalanced data. *Software Quality Journal*, 28. DOI: 10.1007/s11219-020-09525-y.
- Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60. DOI: 10.1214/aoms/1177730491.
- 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. Available at: <https://www.scitepress.org/PublishedPapers/2020/94106/94106.pdf>.
- Massey, F. J. (1951). The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78. DOI: doi/10.2307/2280095.
- 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. DOI: 10.1109/ICICV50876.2021.9388622.
- Melo, C. S., da Cruz, M. M. L., Martins, A. D. F., da Silva Monteiro Filho, J. M., and de Castro Machado, J. (2020). Time-series approaches to change-prone class prediction problem. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 122–132. Available at: <https://www.scitepress.org/Papers/2020/93971/93971.pdf>.
- Metz, C. E. (1978). Basic principles of roc analysis. *Seminars in Nuclear Medicine*, 8(4):283–298. DOI: 10.1016/S0001-2998(78)80014-2.
- Mitchell, T. M. (1997). *Machine learning*, volume 1. McGraw-hill New York. Available at: <https://www.cin.ufpe.br/~cavmj/Machine%20-%20Learning%20-%20Tom%20Mitchell1.pdf>.
- Nielsen, A. (2019). *Practical Time Series Analysis: Prediction with Statistics and Machine Learning*. O’Reilly Media, 1 edition. Available at: Book.
- 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. Available at: <http://jmlr.org/papers/v12/pedregosa11a.html>.
- Pritam, N., Khari, M., Hoang Son, L., Kumar, R., Jha, S., Priyadarshini, I., Abdel-Basset, M., and Viet Long, H. (2019). Assessment of code smell for predicting class change proneness using machine learning. *IEEE Access*, 7:37414–37425. DOI: 10.1109/ACCESS.2019.2905133.
- 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. DOI: 10.1109/ICSM.2011.6080797.
- Silva, R. d. C., Farah, P. R., and Vergilio, S. R. (2022). Machine learning for change-prone class prediction: A history-based approach. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering, SBES ’22*, page 289–298, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3555228.3555249.
- Silva, R. d. C., Farah, P. R., and Vergilio, S. R. (2024). Supplementary Material - On the use of Change History Data to Enhance Class Change-Proneness Prediction Models. Available at: <https://github.com/carvalho797/6/Change-History-Data-to-Enhance-Class-Change-Proneness-Prediction-Models>.
- Stone, M. (1974). Cross-validated choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133. DOI: 10.1111/j.2517-6161.1974.tb00994.x.
- Sultana, K. Z., Anu, V., and Chong, T.-Y. (2021). Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach. *Journal of Software: Evolution and Process*, 33(3):e2303. DOI: 10.1002/smr.2303.
- 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. DOI: 10.1109/TSE.2005.83.
- 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. DOI: 10.1016/j.jss.2020.110777.
- Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132. DOI: 10.3102/10769986025002101.
- Witten, I. H., Frank, E., Hall, M. A., Pal, C. J., and Data, M. (2005). *Practical machine learning tools and techniques*, volume 2. Available at: <https://researchcommons.waikato.ac.nz/server/api/core/bitstreams/b693de20-a3ff-4025-acdb-9e4568d8ac23/content>.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers. DOI: 10.1007/978-3-642-29044-2.
- 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. DOI: 10.1109/TSE.2009.32.
- 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. DOI: 10.1002/smr.2111.