Enhancing Change Prediction Models using Developer-Related Factors

Gemma Catolino1, Fabio Palomba2, Andrea De Lucia1, Filomena Ferrucci1, Andy Zaidman3

1 University of Salerno, Italy — 2 University of Zurich, Switzerland — 3 Delft University of Technology, The Netherlands

gcatolino@unisa.it, palomba@ifi.uzh.ch, adelucia@unisa.it, fferrucci@unisa.it, a.e.zaidman@tudelft.nl

Abstract

Continuous changes applied during software maintenance risk to deteriorate the structure of a system and threat its maintainability. In this context, predicting the portions of source code where specific maintenance operations should be focused on may be crucial for developers to prevent maintainability issues. Researchers proposed change prediction models based on product metrics, while recent papers have shown the adaptability of process metrics to the same context. However, we believe that existing approaches still miss an important information, i.e., developer-related factors that are able to capture how complex is the development process under different perspectives. In this paper, we firstly investigate three change prediction models that exploit developer-related factors (e.g., number of developers working on a class) as predictors of change-proneness of classes and then we compare them with existing models. Our findings reveal that these factors might improve in some cases the capabilities of change prediction models. Moreover, we observed interesting complementarities among the prediction models. For this reason, we devised a novel change prediction model exploiting the combination of developer-related factors and product and evolution metrics. The results show that such model is up to 20% more effective than the single models in the identification of change-prone classes.

Keywords: Change Prediction, Mining Software Repositories, Empirical Study

1. Introduction

Software systems are subject to continuous evolution, driven by changes in the requirements imposed by the stakeholders on the one hand and by the resolution of bugs threatening their reliability on the other hand [42]. Unfortunately, the more changes developers apply to the software system the more complex the system is likely to become, thereby eroding the original design and possibly reducing the overall maintainability [55]. While change is unavoidable, it needs to be controlled by developers. In this context, the up front identification of code elements potentially exhibiting a higher change-proneness may be important for developers for two main reasons: on the one hand, change-proneness can be considered as a quality indicator that can be used to warn developers when touching code that should be refactored [69]; on the other hand, developers can plan preventive maintenance operations, such as refactoring [27], peer-code review [3, 9], and testing [66], aimed at increasing the quality of the code and reducing future maintenance effort and costs [27].

Change prediction is the branch of software engineering aimed at identifying the entities more prone to be modified in the future, helping developers in both planning preventive maintenance actions and keeping the complexity of source code under control [39]. Previous research focused on (i) the analysis of the factors influencing the change-proneness of classes [11, 22, 37, 51, 66] and (ii) the definition of prediction models able to support developers by recommending the classes on which preventive maintenance actions should be performed [64, 33, 65, 34].

An important body of previous work has explored the possibility to use product metrics (e.g., the Chidamber and Kemerer Object Oriented metric suite [18]) as indicators of the change-proneness of classes. In this case, the underlying assumption is that classes having low code quality are more prone to be modified in the future. As an example, Zhou et al. [69] proposed a change prediction model relying on cohesion, coupling, and inheritance metrics, finding that code metrics can be exploited for predicting change-prone classes, while the number of lines of code often represents a confounding effect worsening the performance of prediction models.

In the recent past, Elish et al. [24] investigated the role of process metrics in the context of change prediction models. More specifically, they devised the so-called evolution metrics, i.e., metrics characterizing the history of a class under different perspectives (e.g., the number of past modifications of a class in a certain time window). Afterwards, they found that a change prediction model based on such evolution metrics performs better than the one built using code metrics.

Despite the effort devoted by the research community over the years, we believe that current approaches missed an important piece of information, i.e., they do not consider developer-related factors, which can provide information on how developers perform modifications and how complex is the development process. In our previous pa-
per [15] we conjectured that such aspects can be a useful source of information to predict classes more likely to be changed in the future. To verify the conjecture, we empirically evaluated the performance of three prediction models based on developer-related factors previously defined in literature. Specifically, we experimented the (i) Basic Code Change Model (BCCM) proposed by Hassan [35] which relies on the entropy of changes applied by developers, (ii) the Developer Changes Based Model (DCBM) devised by Di Nucci et al. [21] that considers to what extent developers apply scattered changes in the system, and (iii) the Developer Model (DM) proposed by Bell et al. [8] which analyzes how many developers touched a code element over time. Although such models were originally proposed in the context of bug prediction, we selected them since they are based on metrics possibly influencing the change-proneness of classes as well. For instance, the lack of coordination between multiple developers working on the same code element may lead to the introduction of design pitfalls that negatively influence the maintainability of source code [40], possibly making it more change-prone. Furthermore, to have a comprehensive view of the usefulness of developer-related factors in change prediction, we also compared the performance of the experimented models with the ones proposed by Elish et al. [24] and Zhou et al. [69].

The results demonstrated that the experimented prediction reached an overall F-Measure ranging between 57% and 68% and an Area Under the ROC Curve (AUC-ROC) ranging between 56% and 70%. Among them, the DCBM model was the one obtaining the highest accuracy. When compared to the model exploiting the evolution metrics devised by Elish et al. [24], we found that the developer-based prediction models improved the F-Measure up to 12% and the AUC-ROC up to 15%. More importantly, all the investigated prediction models showed interesting complementarities in the set of change-prone classes correctly predicted. Indeed, we discovered that different models capture different change-prone instances, paving the way for new prediction models exploiting a combination of the predictors used by the investigated models.

In this paper, we extend our previous work [15] with the aim of (i) designing a combined change prediction model that exploits the complementarities among the investigated product, process, and developer-based models and (ii) increasing the generalizability of our findings by considering a larger dataset. More specifically, we:

1. Devise and evaluate the performance of a new change prediction model based on a combination of the metrics used by the previously investigated models. On the basis of the complementarities discovered among the investigated change prediction models, we performed a detailed study—exploiting the Information Gain algorithm [58]—with the aim of finding the subset of predictors more relevant for the identification of change-prone classes. Then, we exploited them to build and evaluate a combined change prediction model.

2. Extend the empirical evaluation of developer-based change prediction models and their comparison with the state of the art. While we previously analyzed 197 releases of 10 software systems having a total of 105,693 commits and 358 developers, this study considers 192,274 commits made by 657 developers over 408 releases of 20 software systems having different size and scope.

On the one hand, the results of the study confirm our previous findings showing the usefulness of developer-related factors in change prediction. On the other hand, we found that the novel combined change prediction model clearly outperforms the baseline models, being more accurate in the predictions by up to 23% in terms of F-Measure.

Structure of the paper. In Section 2 we discuss background and related literature on change prediction. In Section 3 the design of the empirical study is described, while Section 4 reports the results achieved when evaluating the performance of the experimented change prediction models. Section 5 discusses the threats that could affect the validity of our study. Finally, Section 6 concludes the paper.

2. Background and Related Work

In this section we firstly present a background on the problem of change prediction and how it can be used to improve the quality of source code; then, we overview the related literature.

2.1. The problem of predicting change-prone classes

Change-prone classes represent pieces of code that, for different reasons, tend to change more often: this may be due to the importance of a class for the business logic of the system or because it is not properly designed by developers (e.g., in presence of code smells [37, 54]). Keeping track of these classes can be relevant to create awareness among developers about the fact that these classes tend to change frequently, possibly hiding design issues that should be solved.

It is important to note that this type of classes must not be confused with bug-prone code elements. The two sets of classes might have some relationships but they still remain conceptually disjoint. In the first place, bug-proneness indicates source code that is more likely to have bugs in the near future, thus, the fact that a class has bugs does not imply that it changes more often. Secondly, bug-prone classes might also be change-prone (changes are made to correct faults), but corrections are not the only reason for changes, as classes might change due to software evolution. Thus, change- and bug-proneness of classes might have some relation, but are not the same.
Change prediction models represent an established way to identify change-prone classes [69]. In this context, a supervised technique is exploited, where a set of independent variables (i.e., metrics characterizing a class) are used by a Machine Learning classifier (e.g., Logistic Regression [41]) to predict a dependent variable (i.e., the change-proneness of classes). In a real-case scenario, change prediction models might be directly integrated in developers software analytics dashboards (e.g., BITERGIA\(^1\)), thus continuously providing feedback on the source code classes that are more likely to change in the future. Such feedback can be used by developers as input for performing preventive maintenance activities before putting the code into production: for instance, in a continuous integration (CI) scenario, developers might want to refactor the code before the CI pipeline starts to avoid warnings given by static analysis tools [10, 68]. Similarly, change prediction models might be useful for project managers in order to properly schedule maintenance operations.

2.2. Related Work

The change-proneness of classes has been frequently investigated in the last decade by the research community under two main perspectives. On the one hand empirical studies were conducted with the aim of analyzing the factors influencing the phenomenon [11, 22, 37, 51, 66], while other studies investigated the role of product and evolution metrics as predictors of the future code-proneness of classes [8, 24, 67]. In the following we summarize the related literature on previous research.

2.2.1. Factors Influencing Change-proneness of Classes

Di Penta et al. [22] firstly investigated the relation between classes involved in design patterns [29] and change-proneness, finding that three design patterns, i.e., Adapter, Abstract Factory, and Command, tend to make classes more change-prone with respect to the other classes of a software system. These findings were subsequently confirmed by Bieman et al. [11] in the context of an industrial case study involving three proprietary systems. Also Posnett et al. [57] analyzed the influence of pattern roles on change-proneness, showing that the size seems to be a stronger determinant of change-proneness than design patterns.

Specularly, Khomh et al. [37] analyzed how bad design patterns (a.k.a., bad code smells [27]) affect the change- and bug-proneness of classes. They showed that smelly classes are significantly more likely to be the subject of changes and bugs than other classes. Moreover, systems containing a high number of smells are likely to be more change prone. On the other hand, Kim et al. [51] showed that refactoring is a key activity to reduce the change-proneness of classes.

Finally, Lindvall [43] found that the size of a class can influence the propensity to change, in fact large classes are statistically more change-prone than classes having a small size, and that developers tend to apply more changes to such classes during maintenance and evolution [44]. Further studies showed that coupling metrics are relevant measures to estimate the changeability of source code [1, 2, 13].

Our findings provide additional insights into the factors influencing the change-proneness of classes, since we show that developer-related factors play a relevant role in change prediction.

2.2.2. Predicting Change-Prone Classes

Most of the work conducted with the aim of predicting classes that are more likely to change in future releases of a software system refers to the usage of the structural information extracted from source code [47].

Chaumun et al. [17] and Tsantalis et al. [67] provided evidence of the usefulness of change prediction. The statistical analyses conducted by Lu et al. [46] and Malhotra et al. [48] clarified which Object Oriented metrics are better suited for change prediction, reporting a set of cohesion, coupling, and inheritance metrics that should be used in this context. On the basis of these results, several prediction models based on product metrics have been devised. For instance, Romano et al. [59] relied on code metrics for predicting change-prone fat interfaces, while Eski et al. [25] proposed a model based on both CK and QMOOD metrics [6] to estimate change-prone classes and to determine parts which should be tested first and more deeply.

Other previous research tried to estimate the change-proneness of classes using alternative methodologies. For instance, the combination between dependencies mined from UML diagrams [62] and code metrics has been proposed [64, 33, 65, 34]. Also genetic and learning algorithms have been proposed in this context [49, 50, 56]. Specifically, Malhotra et al. [49] validated the CK metrics suite for building an efficient software quality model which predict change prone classes with the help of Gene Expression Programming. Marinescu [50] reported the goodness of GAs for both change- and bug-prediction, while Peer et al. [56] devised the use of adaptive neuro-fuzzy inference system (ANFIS) to estimate the change-proneness of classes. Later on, Zhou et al. [69] showed that size metrics may lead to multi-collinearity [53] when mixed together with other cohesion and coupling metrics. As a result, they suggested to avoid using the LOC metric in product-based change prediction models [69].

The studies by Elish et al. [24] and Girba et al. [30] are the closest to our work. Elish et al. [24] reported the potential usefulness of evolution metrics for change prediction. Girba et al. [30] defined a tool that suggests change-prone code elements by summarizing previous changes. In a small-scale empirical study involving two systems, they

\(^1\)https://www.bitergia.com
observed that previous changes can effectively predict future modifications.

Besides the evolution metrics defined by Elish et al. [24] and Girba et al. [30], in this paper we also analyzed the role of developer-related factors that have shown to be relevant for prediction purpose in other contexts [21]. More importantly, we show that the combination of product, process, and developer-related metrics may provide better performance when predicting change-prone classes.

3. Empirical Study Design

The goal of the empirical study is to evaluate the usefulness of metrics capturing the complexity of the development process for predicting change-prone classes, with the purpose of improving the allocation of resources in preventive maintenance activities (e.g., refactoring, code review, etc.) by focusing the attention on such classes. The quality focus is on the prediction accuracy and completeness of the investigated approaches, while the perspective is that of both researchers and practitioners: the former are interested in evaluating the effectiveness of using developer-related factors within change prediction model; the latter are interested in understanding the actual applicability of change prediction models.

The context of the study consists of twenty open source software systems having different size and scope. Specifically, starting from the list of open source projects available on GitHub, we randomly selected the systems among those having more than 500 commits and more than 10 developers: in this way, we selected projects having enough change history and developer-related information for our study. Table 1 shows the characteristics of the considered systems, in particular (i) the software system’s evolution that we took into account, (ii) the average percentage of change-prone classes identified among all the time windows analyzed (more details later in this section), and (iii) the size in terms of number of commits, average number of developers in the considered time windows, classes, and KLOCs. Overall, our study considers 408 releases, 192,274 commits, and 657 developers.

3.1. Research Questions

The study addresses the following research questions:

RQ1: To what extent are developer-based prediction models able to correctly estimate the change-proneness of classes?

RQ2: How does the performance of developer-based prediction models differ from the ones of existing change prediction models?

RQ3: What is the complementarity between the developer-based models and the existing change prediction models?

RQ4: Is a combined change prediction model able to boost the performance of existing models?

The first research question (RQ1) aims at measuring the extent to which change prediction models built using developer-related factors can be useful when employed in the prediction of change-prone classes. With RQ2 our goal is to compare the performance of the experimented developer-based models with the ones previously defined in literature, while in RQ3 we measure the complementarity between the developer-based and existing change prediction models. Once we have assessed the performance of the individual change prediction techniques, in RQ4 we aim at investigating the performances of a combined model built taking into account the metrics used by the different investigated models.

3.2. Experimental Setup

To perform the study we set up an experimental environment to run both the developer-based and the baseline change prediction models. This lead to (i) the selection of the developer-based models, (ii) the identification of the baseline techniques, (iii) the definition of the dependent variable that the models need to classify, i.e., the change-proneness of the classes in our dataset, (iv) the machine learning technique to use for classifying the change-proneness of classes, and (v) the validation strategy to assess the performance of the models.

Developer-based Change Prediction Models. To understand the predictive power of developer-related factors in change prediction, we decided to test the performance of three prediction models (we refer to them as developer-based models since they rely on developer-related factors):

1. The Basic Code Change Model (BCCM) defined by Hassan [35], based on the entropy of changes applied by developers in a given time period. More in detail, the model construction comprises two steps. Firstly the so-called feature introduction modifications, i.e., changes applied in the time window with the aim of introducing new or enhancing existing features, are identified. To this aim, for each analyzed commit, it runs a keyword-based technique able to distinguish three types of modifications, i.e., (i) Fault Repairing modifications (FR), (ii) General Maintenance operations (GM), and (iii) Feature Introduction modifications (FI), based on the analysis of the commit message:

- FR modifications are identified as those whose commit message contains references to an issue, i.e., “fix”, “bug fix”, “#ID”.
- GM operations do not reflect the implementation of features and are identified using
keywords like “copyright”, “re-indent”, and “cleanup”.

- All the other modifications are marked as FI.

Once the FI modifications are identified, the entropy of the changes on a certain class $c_i$ in the time period $\alpha$ is computed exploiting the concept of Shannon entropy [63] as in the following equation:

$$entropy(c_i, \alpha) = -(p_k \cdot \log_2 p_k)$$  \hspace{1cm} (1)

where $p_k$ indicates the probability with which $c_i$ was changed with respect to feature introduction modifications in the considered time period. Such probability is simply computed considering the fraction between the number of feature introduction modifications applied on $c_i$ in the time period $\alpha$ over the total number of feature introduction modifications in $\alpha$. For instance, suppose that the class $c_i$ underwent one FI change in the time window, while the total amount of FI changes applied in the same window are four. The entropy of $c_i$ will then be $-(1/4 \cdot \log_2 1/4)$. It is important to note that this model can be considered developer-based since the entropy metric not only filters the types of changes performed on a certain class in a time period, but it also estimates how difficult its development was through the analysis of the entropy of the changes performed by developers.

2. The Developer Changes Based Model (DCBM) proposed by Di Nucci et al. [21]. It uses the structural and semantic scattering of the developers that worked on a code element in given time period $\alpha$ as predictors. The scattering metrics are computed for each class $c$ as in the following equations:

$$StrScatPred_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} StrScat_{d,\alpha}$$  \hspace{1cm} (2)

$$SemScatPred_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} SemScat_{d,\alpha}$$  \hspace{1cm} (3)

where $developers_{c,\alpha}$ represents the set of developers that worked on the class $c$ during the time period $\alpha$, and the functions $StrScat_{d,\alpha}$ and $SemScat_{d,\alpha}$ return the structural and semantic scattering, respectively, of a developer $d$ in the time window $\alpha$. Given the set $CH_{d,\alpha}$ of classes changed by a developer $d$ during a time period $\alpha$, the structural scattering of a developer is computed as follow:

$$StrScat_{d,\alpha} = |CH_{d,\alpha}| \times \text{average}_{c_i, c_j \in CH_{d,\alpha}} \left[\text{dist}(c_i, c_j)\right]$$  \hspace{1cm} (4)

where $\text{dist}$ is the number of packages to traverse in order to go from class $c_i$ to class $c_j$. More specifically, it is computed by applying the shortest path algorithm on the graph representing the systems package structure. As for the semantic scattering of a developer, it is based on the textual similarity of the classes changed by a developer in the time period $\alpha$ and it is computed as:

$$SemScat_{d,\alpha} = |CH_{d,\alpha}| \times \text{average}_{c_i, c_j \in CH_{d,\alpha}} \left[\text{sim}(c_i, c_j)\right]$$  \hspace{1cm} (5)

where the $\text{sim}$ function returns the textual similarity between the classes $c_i$ and $c_j$ according to the measurement performed using the Vector Space Model.
(VSM) [4]. The metric ranges between zero (no textual similarity) and one (the textual content of the two classes is identical).

3. The Developer Model (DM) devised by Bell et al. [8] relies on the number of developers that worked on a specific component of source code in a given time period $\alpha$. In the first place, the set $\text{Devs}(c, \alpha)$, which composed of the developers that committed at least one change to a certain class $c$, during the time period $\alpha$, is computed. Then, the number of developers for $c$ is given by the cardinality of the set $|\text{Devs}(c, \alpha)|$.

While the selected models have originally been defined in the context of bug prediction, the choice to use them for change prediction was guided by the will to explore the role of different aspects of the development process on the change-proneness of classes. For instance, having a high entropy of changes might indicate the presence of a complex development process where developers apply changes in an undisciplined manner that lead to source code that is less maintainable and possibly more change-prone in the future.

**Baseline Change Prediction Models.** In our work, we identified two main baseline techniques for the prediction of change-prone classes. The first one is a product-based prediction model. Among all the models relying on code metrics as predictors [47], we used as baseline the model by Zhou et al. [69], which relies on a set of cohesion (i.e., the Lack of Cohesion of Method — LCOM), coupling (i.e., the Coupling Between Objects — CBO — and the Response for a Class — RFC), and inheritance metrics (i.e., the Depth of Inheritance Tree — DIT). We also added the Lines of Code (LOC) metric as an additional independent variable with the aim of evaluating whether this metric actually represents a confounding factor [69], or rather if larger classes are more likely to be modified. In the following, we refer to this model as CM, i.e., Code Metrics Model.

In the second place, we selected the Evolution Model (EM) proposed by Elish et al. [24], which relies on the set of metrics shown in Table 2; these metrics capture different aspects of the evolution of classes, e.g., the change density or the date of birth of a class. Moreover, this model directly uses the number of previous changes of a class to predict its future change-proneness: basically, it exploits the concept of “change-caching”, i.e., classes that underwent more changes in the past will likely undergo more changes in the future since they encapsulate most of the complexity of the system. It is worth noting that the total amount of changes metric differs from the entropy measure proposed by Hassan since (i) it has no filter on the types of changes applied on the class, while the entropy metric only considers modifications aimed at adding or enhancing features and (ii) it does not capture the development complexity aspect, as opposite to the entropy metric that is able to estimate how difficult was the development of a class $c$ in a certain time period [35].

Being composed of several metrics, each of the baseline models might potentially suffer of multi-collinearity [53], which arises when two or more independent variables are highly correlated with each other, possibly causing a decrease in the overall performance of the model. To ensure a fair comparison with the developer-based models, we performed a feature selection preprocessing [45] with the goal of discarding the non-relevant features.

Specifically, for each baseline we exploited the information gain algorithm [58], a function able to quantify the gain provided by including each metric in a prediction model. In our context this algorithm is able to rank the metrics according to their ability to predict the change-proneness of classes. More formally, let $M$ be the combined change prediction model, let $F = \{f_1, \ldots, f_n\}$ be the set of features composing $M$, the algorithm [58] measures the difference in terms of entropy from before to after the set $F$ is split on a variable $f_1$ using the following formula:

$$\text{InfoGain}(M, f_1) = E(M) - E(M|f_1)$$  \hspace{1cm} (6)

where the function $E(M)$ represents the entropy of $M$ when it includes the feature $f_1$, and the function $E(M|f_1)$ represents the entropy of $M$ when it does not include $f_1$ as a feature. The entropy is computed as reported in the following equation:

$$E(M) = -\sum_{i=1}^{n} \text{prob}(f_i) \log_2 \text{prob}(f_i)$$  \hspace{1cm} (7)

The algorithm quantifies how much uncertainty in $M$ was reduced after splitting $M$ on predictor $f_1$. In the context of

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOC</td>
<td>Birth of a Class</td>
</tr>
<tr>
<td>FCH</td>
<td>First Time Changes Introduced to a Class</td>
</tr>
<tr>
<td>FRCH</td>
<td>Frequency of Changes</td>
</tr>
<tr>
<td>LCH</td>
<td>Last Time Changes Introduced to a Class</td>
</tr>
<tr>
<td>WCD</td>
<td>Weighted Change Density</td>
</tr>
<tr>
<td>WFR</td>
<td>Weighted Frequency of Changes</td>
</tr>
<tr>
<td>TACH</td>
<td>Total Amount of Changes</td>
</tr>
<tr>
<td>ATAF</td>
<td>Aggregated Change Size Normalized by Frequency of Change</td>
</tr>
<tr>
<td>CHD</td>
<td>Change Density</td>
</tr>
<tr>
<td>LCA</td>
<td>Last Change Amount</td>
</tr>
<tr>
<td>LCD</td>
<td>Last Change Density</td>
</tr>
<tr>
<td>CSB</td>
<td>Changes since the Birth</td>
</tr>
<tr>
<td>CSBS</td>
<td>Changes since the Birth Normalized by Size</td>
</tr>
<tr>
<td>ACDF</td>
<td>Aggregated Change Density Normalized by Frequency of Change</td>
</tr>
<tr>
<td>CHO</td>
<td>Change Occurred</td>
</tr>
</tbody>
</table>

Table 2: Independent variables considered by Elish et al. [24].
our work, we applied the Gain Ratio Feature Evaluation algorithm implemented in the WEKA toolkit [32] which ranks \( f_1, \ldots, f_n \) in descending order based on the contribution provided by \( f_i \) to the decisions made by \( M \).

The output of the algorithm is represented by a ranked list where the more relevant features, i.e., the ones having the higher expected reduction in entropy are placed at the top. We set the cut-off point of the ranked list equal to 0.1, as suggested by Quinlan [58]. As a result, we excluded FCH, LCH, WFR, ATAF, CHD, LCD, CSBS, and ACDF from the Evolution Model. In the case of CM, instead, none of the metrics were filtered out. However, we found the LOC metric to be the less important metric for the model, with an expected reduction of 0.11: this somehow confirms the findings by Zhou et al. [69] on the relatively low power of this metric in capturing change-prone classes.

For sake of readability, in Table 3 we report the (i) abbreviations used over all the paper, (ii) the names, and (iii) a brief description of the investigated models.

**Computing the Change-Proneness of Classes.** To evaluate the performance of the change prediction models, we needed an oracle reporting the actual change-prone classes. To the best of our knowledge, a public oracle reporting the ground-truth for the phenomenon taken into account is not available in literature. Thus, we had to build our own oracle. To this aim, we followed the guidelines provided by Romano et al. [59], which considered a class as change-prone if, in a given time period \( TW \), it underwent a number of changes higher than the median of the distribution of the number of changes experienced by all the classes of the system. In particular, for each pair of commits \( (c_i, c_i+1) \) of \( TW \) we run ChangeDistiller [26], a tree differencing algorithm able to extract the fine-grained code changes between \( c_i \) and \( c_i + 1 \). The complete list of change types identified by ChangeDistiller is provided in Table 4: as shown, we considered all of them while computing the number of changes. It is worth noting that the tool ignores white space-related differences and documentation-related updates, so that only the changes applied on the source code are considered. Moreover, ChangeDistiller is also able to identify renaming operations: in this way, we could handle cases where a class was modified during the change history, thus not biasing the counting of the number of changes.

We made the oracle reporting the change-prone classes of all the twenty considered systems publicly available in the online appendix [16].

**Classifier Selection.** The next step concerned the identification of the machine learning technique to use to classify the change-proneness of classes. The related literature proposes several alternatives (e.g., Tsantalis et al. [67] relied on Logistic Regression [41], while Romano and Pinzger [59] suggested the use of Support Vector Machine [12]), however it is still unclear which classifier is able to give the best overall performance.

For this reason, we experimented with several classifiers previously used for prediction purposes from the research community, i.e., ADTree [28], Decision Table Majority [38], Logistic Regression [41], Multilayer Perceptron [60], Support Vector Machine [12], and Naïve Bayes [36]. We empirically compared the results achieved when applying each classifier on each experimented baseline model on the software systems in our study (more details on the adopted procedure can be found in Section 3.3), and Logistic Regression [41] provided the best performance for all the tested prediction models. Thus, in this paper we report the results of the models built with this classifier. A comprehensive report of the comparison of the different classifiers is included in the online appendix [16].

**Validation Strategy.** To assess the performance of the experimented prediction models we split the evolution history of the subject systems into three-month time periods and we adopted a three-month sliding window to train and test the change prediction models. Specifically, starting from the first time window \( TW_1 \) (i.e., the one starting from the first commit), we trained each model on it, and tested its performances on the time window \( TW_2 \) (i.e., the subsequent three-month period). Then, we moved three months forward to the next time window, training the classifier using the data available in \( TW_2 \) and testing the model on \( TW_3 \). This process has been repeated until the end of the evolution history of the subject systems.

The choice of the validation methodology was based on three aspects. Firstly, all the models refer to a specific time window of size \( \alpha \) in which their own predictors have to be computed. Therefore, this validation technique better fits the characteristics of the experimented models. Secondly, developer-related metrics aim at capturing the dynamics among developers in a given period of time: considering larger time windows (e.g., entire releases) would be not conceptually correct, as such metrics would put together things happened far in the time. Thirdly, this methodology has been widely used in recent years to test the performance of prediction models [21, 35]. Moreover, the choice of considering three-month periods is based on (i) the results of previous work, such as the ones by Hassan [35] and Di Nucci et al. [21], and (ii) the findings of the empirical assessment we performed on such a parameter, which showed that the best results for all experimented techniques are achieved when using three-month periods. In particular, we tested time windows of size \( \alpha = 1, 2, 3, 6 \) months. A report of the results is available in the replication package [16].

### 3.3. Analysis Method

To answer RQ1, we firstly ran the previously selected developer-based prediction models, i.e., BCCM, DCBM, and DM, on every three-month window of the change history of the systems considered. Then, we computed three well-known Information Retrieval metrics, namely accuracy, precision and recall [4], defined as follow:
The Matthews Correlation Coefficient (MCC) [5], a regression coefficient that combines all four quadrants of a confusion matrix, thus also considering true negatives. Its formula is:

\[
MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}
\]

where \(TP\), \(TN\), \(FP\), and \(FN\) represent the number of (i) true positives, (ii) true negatives, and (iii) false positives, respectively, while \(FN\) is the number of false negatives. Its value ranges between 1 and +1. A coefficient equal to +1 indicates a perfect prediction; 0 suggests that the model is no better than a random one; and 1 indicates total disagreement between prediction and observation.

In addition, we computed three more metrics. In the first place, we considered the Area Under the ROC Curve (AUC-ROC): this metric quantifies the overall ability of a prediction model to discriminate between change-prone and non-change-prone classes. The closer the AUC-ROC to 1, the higher the ability of the classifier to discriminate classes that will change less or more in the future. On the other hand, the closer the AUC-ROC to 0.5, the lower the accuracy of the classifier.

Then, we computed the Matthews Correlation Coefficient (MCC) [5], a regression coefficient that combines all four quadrants of a confusion matrix, thus also considering true negatives. Its formula is:

\[
MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}
\]

where \(TP\), \(TN\), and \(FP\) represent the number of (i) true positives, (ii) true negatives, and (iii) false positives, respectively, while \(FN\) is the number of false negatives. Its value ranges between 1 and +1. A coefficient equal to +1 indicates a perfect prediction; 0 suggests that the model is no better than a random one; and 1 indicates total disagreement between prediction and observation.

Finally, we also statistically compared the AUC-ROC achieved by the experimented prediction models. To this aim, we exploited the Mann-Whitney test [19] (results are intended as statistically significant at \(\alpha=0.05\)). Furthermore, we estimated the magnitude of the measured differences by using Cliff’s Delta (or \(d\)), a non-parametric effect size measure [31] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for \(|d| < 0.10\), small for \(|d| < 0.33\), medium for \(0.33 \leq |d| < 0.474\), and large for \(|d| \geq 0.474\) [31].

To answer RQ2 and compare the developer-based models with those previously defined in literature, we ran the baseline change prediction models, i.e., EM and CM, over the three-month windows of the change history of the systems in the dataset. Subsequently, we compared the experimented models using the same procedures and metrics used in the context of RQ1. Moreover, we statistically compared the AUC-ROC achieved by such models.

To answer RQ3 and analyze the complementarity between the exploited models, we investigated to what extent different models correctly classify the change-proneness of different classes. To this aim, we exploited the overlap metrics. Specifically, for each pair \(m_i\) and \(m_j\) of the experimented prediction models, we computed the overlap between the sets of true positives correctly identified by both models (denoted by \(corr_{m_i \cap m_j}\)) and the percentage
Table 4: Change types extracted by ChangeDistiller. ✓ symbols indicate the types we considered when computing the change-proneness of classes.

<table>
<thead>
<tr>
<th>ChangeDistiller</th>
<th>Our Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement-level changes</td>
<td>✓</td>
</tr>
<tr>
<td>Statement Ordering Change</td>
<td>✓</td>
</tr>
<tr>
<td>Statement Parent Change</td>
<td>✓</td>
</tr>
<tr>
<td>Statement Insert</td>
<td>✓</td>
</tr>
<tr>
<td>Statement Delete</td>
<td>✓</td>
</tr>
<tr>
<td>Statement Update</td>
<td>✓</td>
</tr>
<tr>
<td>Class-body changes</td>
<td>✓</td>
</tr>
<tr>
<td>Insert attribute</td>
<td>✓</td>
</tr>
<tr>
<td>Delete attribute</td>
<td>✓</td>
</tr>
<tr>
<td>Declaration-part changes</td>
<td>✓</td>
</tr>
<tr>
<td>Access modifier update</td>
<td>✓</td>
</tr>
<tr>
<td>Final modifier update</td>
<td>✓</td>
</tr>
<tr>
<td>Declaration-part changes</td>
<td>✓</td>
</tr>
<tr>
<td>Increasing accessibility change</td>
<td>✓</td>
</tr>
<tr>
<td>Decreasing accessibility change</td>
<td>✓</td>
</tr>
<tr>
<td>Final Modified Insert</td>
<td>✓</td>
</tr>
<tr>
<td>Final Modified Delete</td>
<td>✓</td>
</tr>
<tr>
<td>Attribute declaration changes</td>
<td>✓</td>
</tr>
<tr>
<td>Attribute type change</td>
<td>✓</td>
</tr>
<tr>
<td>Attribute renaming change</td>
<td>✓</td>
</tr>
<tr>
<td>Method declaration changes</td>
<td>✓</td>
</tr>
<tr>
<td>Return type insert</td>
<td>✓</td>
</tr>
<tr>
<td>Return type delete</td>
<td>✓</td>
</tr>
<tr>
<td>Return type update</td>
<td>✓</td>
</tr>
<tr>
<td>Method renaming</td>
<td>✓</td>
</tr>
<tr>
<td>Parameter insert</td>
<td>✓</td>
</tr>
<tr>
<td>Parameter delete</td>
<td>✓</td>
</tr>
<tr>
<td>Parameter ordering change</td>
<td>✓</td>
</tr>
<tr>
<td>Parameter renaming</td>
<td>✓</td>
</tr>
<tr>
<td>Class declaration changes</td>
<td>✓</td>
</tr>
<tr>
<td>Class renaming</td>
<td>✓</td>
</tr>
<tr>
<td>Parent class insert</td>
<td>✓</td>
</tr>
<tr>
<td>Parent class delete</td>
<td>✓</td>
</tr>
<tr>
<td>Parent class update</td>
<td>✓</td>
</tr>
</tbody>
</table>

of change-prone classes correctly classified by $m_i$ only and missed by $m_j$ (denoted by $corr_{m_i \cap m_j}$) defined as follows:

\[
corr_{m_i \cap m_j} = \left( \frac{|corr_{m_i \cap m_j}|}{|corr_{m_i} \cup corr_{m_j}|} \right) \times 100 \%
\]  \hspace{1cm} (14)

\[
corr_{m_i \setminus m_j} = \left( \frac{|corr_{m_i \setminus m_j}|}{|corr_{m_i} \cup corr_{m_j}|} \right) \times 100 \%
\]  \hspace{1cm} (15)

where $corr_{m_i}$ represents the set of change-prone classes correctly classified by the prediction model $m_i$.

As for RQ4, we aimed at devising a combined model using a mix of the features exploited by the prediction models investigated in the previous research questions. It is important to note that a simple combination obtained by featuring together all the predictors used by the five models might lead to sub-optimal results because of overfitting [52]. To avoid this issue, we identified the subset of predictors actually leading to the best prediction performance. Thus, we re-applied the Gain Ratio Feature Evaluation algorithm [58] as done in RQ1, using 0.1 as cut-off point. As a result, only the relevant features were considered when building the combined model.

To ensure a fair comparison with the stand-alone prediction models, i.e., the ones exploiting the considered predictors in isolation, we measured the performance of the combined model using the same set of metrics previously exploited and computed the statistical difference of AUC-ROC between the combined and the stand-alone models.

4. Empirical Study Results

In this section we report the results achieved in the study. Tables 5 and 6 report the performance of the experimented change prediction models over the twenty considered subject systems. For sake of clarity, in the following we discuss each research question independently.

4.1. RQ1: Performance of Developer-based Models

Looking at Table 5, we can immediately provide quantitative answers to our first research question. In the first place, while developer-based models tend to perform well, it is worth noting that none of them achieves an overall accuracy higher than 77%. Even if this value is still quite positive, it is also important to highlight that a notable percentage of classes (at least 23%) is not correctly classified while using the models independently. Thus, the problem of identifying the change-proneness of classes seems to be not easily addressable by employing models based on single aspects of the development process.

Among the three developer-based models investigated, DCBM [21] tends to perform better than the others, achieving the best scores in term of all the quality metrics computed, i.e., accuracy=77%, precision=65%, recall=72%, F-Measure=68%, AUC-ROC=70%, MCC=65%, BS=38%. Based on these results, we can claim that the way developers apply changes in the system has an influence of the likelihood to make the touched classes more change-prone. The superiority of DCBM is particularly evident in the comparison with the DM model (i.e., the model based on the number of developers), where the F-Measure is 9% higher and the Brier Score is 20% lower. This result highlights that it is not simply the number of developers working on a class that influences the change-proneness, but rather the way developers apply (scattered) changes to the system. Our findings confirm, in the context of change prediction, previous findings achieved by Di Nucci et al. [21], which showed the superiority of the DCBM model in predicting bugs. For instance, consider the case of the class org.gjt.sp.BufferHistory

of the JEDIT system. Between August and October 2009 (i.e., one of the three-month periods considered in our
study) the class was modified 19 times by one developer, being a change-prone class since its number of changes was higher than the median number of changes of the time window. The DM model predicted the class as non-change-prone. However, in the same time period such developer performed 36 modifications spread over five different packages, thus accumulating a high level of both semantic and structural scattering. The scattered changes applied by the developer led to a decreasing of the cohesion of the modified classes (i.e., overall, the LCOM3 increases 16% in such classes): interestingly, the LCOM of the class org.gjt.sp.BufferHistory is the one increasing more (from 3 to 12). This made it more prone to be changed in the future. For instance, the CBO of the chartMeter.Legend instance, the CBO of the charts (i.e., chartVerticalBarRenderer class) were modified 16% higher than the alternative model, with an MCC 2% higher and a reduction of the Brier Score of 5%. Once again, the improvement is statistically significant (α < 0.01) with a large effect size (d = 0.71). The gain provided by DCBM is also visible when considering the other evaluation metrics: for instance, the accuracy is about 6% higher, while the recall 9% and the AUC-ROC 2%. From a practical point of view, this result indicates that the scattering metrics can capture the change-proneness of classes with a higher accuracy than the entropy of changes. This is due to the fact that DCBM works at a higher level of abstraction than BCCM [35]. Specifically, it considers the way developers apply changes rather than the changes themselves, allowing the model to be more efficient when the change process is not chaotic, but developers continuously perform modifications over different parts of the system. To better understand the reasons behind the different performance of these models, let us consider the case of the class chartMeter.Legend belonging to the JFreeChart system. Between April and June 2005, the class underwent 10 of the total 16 changes applied in that time window and was, therefore, considered as change-prone (the median of the time window was 3). In this case, the entropy of changes involving this class is low (α < 0.01), since most of the effort has been devoted to maintain it. However, the two developers performing modifications in the time window not only apply changes to the chartMeter.Legend class, but also to other classes involving 3 different packages. All these modifications were related to the visualization of chart legends, and indeed different other classes related to visualization components (e.g., the chart.VerticalBarRenderer class) were modified. However, the changes applied by developers had the effect of reducing the overall quality of such classes, making them more prone to be changed in the future. For instance, the CBO of chartMeter.Legend reached 8 (+3 with respect to the previous version). This example seems to confirm the hypothesis behind the good performance of the DCBM, namely the negative effect that scattering changes have on the maintainability of classes.

Another interesting example is related to the performance achieved by the two models on aTunes. As shown

<table>
<thead>
<tr>
<th>Project</th>
<th>BCCM</th>
<th>DCBM</th>
<th>DM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P</td>
<td>R</td>
<td>F-M</td>
</tr>
<tr>
<td>Ant</td>
<td>72</td>
<td>65</td>
<td>79</td>
</tr>
<tr>
<td>Cassandra</td>
<td>77</td>
<td>84</td>
<td>90</td>
</tr>
<tr>
<td>Lucene</td>
<td>60</td>
<td>58</td>
<td>71</td>
</tr>
<tr>
<td>Poi</td>
<td>79</td>
<td>70</td>
<td>85</td>
</tr>
<tr>
<td>Synapse</td>
<td>59</td>
<td>69</td>
<td>57</td>
</tr>
<tr>
<td>Velocity</td>
<td>75</td>
<td>62</td>
<td>58</td>
</tr>
<tr>
<td>Xalan</td>
<td>75</td>
<td>66</td>
<td>67</td>
</tr>
<tr>
<td>Xerxes</td>
<td>87</td>
<td>69</td>
<td>71</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>89</td>
<td>87</td>
<td>88</td>
</tr>
<tr>
<td>aTunes</td>
<td>63</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>FreeMind</td>
<td>35</td>
<td>35</td>
<td>38</td>
</tr>
<tr>
<td>JEdit</td>
<td>78</td>
<td>42</td>
<td>74</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>71</td>
<td>45</td>
<td>67</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>97</td>
<td>77</td>
<td>59</td>
</tr>
<tr>
<td>jVLT</td>
<td>80</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>pBeans</td>
<td>70</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>pdfTranslator</td>
<td>75</td>
<td>75</td>
<td>57</td>
</tr>
<tr>
<td>Redaktor</td>
<td>70</td>
<td>73</td>
<td>64</td>
</tr>
<tr>
<td>Servapion</td>
<td>68</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td>Zmel</td>
<td>55</td>
<td>72</td>
<td>55</td>
</tr>
<tr>
<td>Overall</td>
<td>71</td>
<td>63</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 5: Performance (in percentage) achieved by the investigated change prediction models.

A=Accuracy; P=Precision; R=Recall; F-M=F-Measure; AR=AUC-ROC; MCC=Matthews Correlation Coefficient; BS=Brier-Score.
in Table 5, BCCM has a very low precision of 6%: from a deeper analysis into the likely causes that have lead to this result, we discovered that in this system there are only five active developers that performed on average 10 changes per period to the core classes of the system (i.e., the 13 classes contained in the com.fleax.atunes.gui).

At the same time, other 11 changes have been performed on average on the other classes of the system. The entropy of changes computed on the core classes has been generally higher than the one of other classes, leading BCCM to identify all these classes as change-prone. However, in different time periods only a subset of such core classes have been actually change-prone, meaning that BCCM identified a high number of false positives (thus, having a low precision). On the other hand, the usage of scattering metrics allows the DCBM model to be more precise when detecting change-prone classes: indeed, in aTunes the code components changing more often are characterized by a higher scattering than non-change-prone classes (+13% on average), thus making DCBM to be more effective.

To broaden the scope of the discussion, we can generally observe that models previously used in the context of bug prediction achieve good performance also when employed in the identification of change-prone classes. This is somehow unexpected and seems to delineate a direct relationship between the complexity of the development process and several maintainability issues, including the change- and bug-proneness of classes. We plan to perform an extensive analysis of the impact of developer-related factors on a wider range of maintainability problems, as well as of the interplay between change- and bug-proneness of classes as part of our future research.

### Summary for RQ1

The investigated developer-based models achieve quite positive results. Among them, the DCBM prediction model obtains the highest performance, having an overall F-Measure equals to 68% and an accuracy equals to 77%. The superiority of DCBM is statistically significant and has a large effect size when compared to the other two models.

#### 4.2. RQ2: Comparison between Developer-based and State-of-the-art Models

The results achieved by the baseline change prediction models investigated in this study (i.e., EM and CM) are reported in Table 6. As it is possible to see, the EM model achieves a similar overall F-Measure as the DM and BCCM models (i.e., 59%) but worse than the DCBM model (-9% in terms of F-Measure and +11% considering the Brier Score). This is confirmed by the analysis reported in Table 7: indeed, the differences between DCBM and EM are statistically significant (α < 0.01) and the magnitude is large (d = 0.77).

Generally, it is important to remark that EM is the only model that directly measures the previous number of
changes of a class to predict its future change-proneness: our results indicate that this feature is not able to characterize the future change-proneness of classes better than other predictors. This result confirms previous findings by Ekanayake et al. [23] on the variability of the change-proneness of classes during different stages of software evolution. As a consequence, the previous knowledge about the number of changes a class underwent is not always suitable to correctly identify change-prone classes in future versions of a software system. Further analyzing the predictions provided by EM, we discovered that it is generally effective when a class has a central role in the architecture of a system and, as such, usually undergoes a high number of changes. For example, in the JHotDraw system, the class \texttt{svg.io.SVGFigureFactory} is responsible for performing the main functionality of the entire project, \textit{i.e.}, it manages the graph creation. This class is present in the system since its first commit and it was frequently modified by developers among all the time windows analyzed. In this case, the predictors used by the EM model (\textit{e.g.}, previous changes and birth date) are particularly effective since they characterize well the change-proneness of the class. On the other hand, the performance decreases in cases where a significant restructuring of the system’s architecture is applied, since the responsibilities of several code artifacts are modified and, therefore, predictors such as the birth date or the previous changes are less meaningful. For instance, in the time window ranging between December and February 2006 the Apache \texttt{ANT} developers performed an entire restructuring of the system, which led to the removal of some old classes as well as the re-distribution of the responsibilities of several code artifacts\footnote{As indicated in the release notes of the version 1.7.1, which correspond to that time period: http://tinyurl.com/hqwazgg}. As a consequence, the data considered by the EM model was not sufficient to correctly predict the change-proneness of classes: in fact, the accuracy achieved by the model in that time window was 43%. Noticeably, in the same time period the DCBM and DM models reached an accuracy equal to 87% and 83%, respectively. As expected, in the considered period the developers were busy modifying the source code and, thus, models relying on such information were performing better.

On the one hand, our results confirm previous findings on the potential usefulness of the evolution metrics in the context of change prediction [24]. On the other hand, we also found how the “change-caching” concept exploited by this model is valid for classes having a central role in the system, while it has less effect in other classes. At the same time, we showed that (i) other metrics based on developers can be effectively used for prediction purpose, and (ii) they seem to capture information orthogonal with respect to the EM model, \textit{i.e.}, they capture characteristics of the phenomenon that other metrics are not able to identify.

Switching the attention to the results obtained by the model relying on code metrics, we can observe that it constantly performs worse than the developer-based prediction models. Indeed, the CM model has an overall F-Measure always lower than BCCM, DCBM, and DM.

For instance, DCBM achieves an average F-Measure 10% higher than the model based on code metrics (68% vs 58%). The superiority of DCBM is also confirmed when considering all the other evaluation metrics, \textit{i.e.}, accuracy=+15\%, precision=+7\%, recall=+14\%, AUC-ROC=+9\%, MCC=+13\%, BS=+17\%. This result contradicts previous findings [46, 69], demonstrating that the use of code metrics is not enough to efficiently predict change-prone classes. A clear example is the class \texttt{xerces.dom.ElementImpl} of the Apache XERCES project. During the time window between May and July 2007, the class experienced only three changes (\textit{i.e.}, it is non-change-prone because the median was 9) applied by two different developers, who focused all their activities on the maintenance of classes belonging to the \texttt{xerces.dom} package. As a consequence, the value of their scattering metrics is zero, since they never performed modifications outside the scope of the package [21]. Thus, the DCBM model correctly marked this class as non-change-prone. At the same time, the class has an LCOM=28 and a CBO=7. Both the metrics are higher than the average metric values of the other classes composing the system, and for this reason the CM model wrongly marked the class as change-prone.

\begin{center}
\textbf{Summary for RQ2.} Developer-based prediction models generally perform better than the existing models. This is particularly true when considering the DCBM model, which has an overall F-Measure that is 10\% higher than the CM model and 9\% higher than the EM model.
\end{center}

4.3. RQ3: Complementarity of the Investigated Models

Table 8 reports the complementarity between each pair of prediction models. For sake of readability, the results have been aggregated by considering the overall overlap between the models, taking into account all the systems. It is worth remarking that in this analysis we only considered the set of correct instances predicted by each model (\textit{i.e.}, the true positive instances). A complete report of the findings on each system is available in the online appendix [16].

The results in Table 8 highlight a reasonable level of complementarity between all the investigated prediction models, meaning that they are able to correctly identify different sets of change-prone classes. To better understand the reasons behind such complementarity, we analyzed the predictions provided by the different models. Firstly, it is worth discussing the complementarity between DCBM and the other models. When considering the relationship between scattering and code metrics, we observed
Table 8: Overlap among the experimented change prediction models.

<table>
<thead>
<tr>
<th>A=BCCM</th>
<th>A=DCBM</th>
<th>A=DM</th>
<th>A=EM</th>
<th>A=CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>B=BCCM</td>
<td>16/16</td>
<td>51/20</td>
<td>55/20</td>
<td>-/16</td>
</tr>
<tr>
<td>B=DCBM</td>
<td>60/16</td>
<td>56/20</td>
<td>53/20</td>
<td>60/16</td>
</tr>
<tr>
<td>B=DM</td>
<td>56/24</td>
<td>51/19</td>
<td>43/23</td>
<td>57/19</td>
</tr>
<tr>
<td>B=EM</td>
<td>56/26</td>
<td>60/20</td>
<td>47/32</td>
<td>-/16</td>
</tr>
<tr>
<td>B=CM</td>
<td>55/27</td>
<td>57/24</td>
<td>47/32</td>
<td>-/16</td>
</tr>
</tbody>
</table>

A consistent set of change-prone classes (i.e., 43%) classified by both the prediction models, but at the same time in almost 35% of the cases the only model able to correctly predict the change-proneness is the DCBM model. Finally, 23% of change-prone classes have been identified using only code metrics. This result highlights the high complementarity between the two models, showing that different predictors work well on different sets of classes.

As for the comparison between DCBM and DM, we observed that 51% of the predicted change-prone classes are in the intersection, while 30% of change-prone classes are detected correctly only by the DCBM model. Finally, the change-proneness of a smaller percentage of classes (19%) can be detected solely using the DM model. Thus, the two models partially complement each other, making prediction improvements conceivable. An interesting case explaining when the DM model is able to outperform the DCBM model can be found in the FreeMind project (the smallest one of our dataset). Here the seven developers of the system often perform changes to a few classes located in the two core packages. Due to the small structure of the system, the scattering metrics cannot correctly capture the developers’ activities and, thus, they always have low values. In such case, the DM model produces more reliable predictions: indeed, it is worth noting that this project is the only one where the DM model performs better than the DCBM one (see Table 5).

The discussion is similar when comparing the DCBM and BCCM models. Even if the model based on scattering metrics generally achieved better performance than the BCCM model (Table 5), we observed an interesting complementarity that may lead to an additional improvement in the prediction through a combination. In fact, Table 8 shows that the change-proneness of almost 40% of classes can be correctly detected by only one of the two models (i.e., 24% of correct prediction have been made only by DCBM, 16% only by BCCM). Moreover, it is worth noting that the complementarity between BCCM and the other models is high as well. For instance, when compared to the CM model, we found 27% of correct predictions performed by the BCCM only and a further 18% of classes for which the change-proneness has been identified using code metrics. An interesting example is represented by the class thrift.CassandraServer which had a value of LCOM=44 and an RFC=23 in the time window between March and May 2010. In that period, this class has been changed 13 times, being classified as an actual change-prone class since the median number of changes was 10. However, the BCCM model was not able to correctly mark this class as change-prone, as it always changed together with a few other classes of the system (on average, 2 classes). As a consequence, the entropy of changes is low. On the other hand, the poor quality of the class was a relevant indicator of the change-proneness.

Furthermore, it is important to note that also the evolution metrics have nice complementarity to the other models. For instance, when comparing EM and BCCM, we observed that in 18% of the cases the change-proneness of classes can be correctly identified by the EM model only. At the same time, the contribution provided by the EM model is still more valuable in comparison to the CM model, where 32% of the change-prone classes are identified by using only the evolution metrics. An interesting example of a change-prone class correctly classified by EM and missed by CM is present in the Argouml project. During the time period between October and December 2006, the class ui.ProjectBrowser underwent 19 changes (more than the median of the period), while it has been introduced at the beginning of the project. Even though the structural metrics do not indicate issues in the maintainability of this class (i.e., LCOM=6, CBO=2, DIT=2, RFC=4), it tends to change frequently. In this case, the CM model does not recognize the change-proneness of the class, while the evolution metrics are able to classify it as change-prone. Conversely, an example of a class identified by CM and missed by EM in the same Argouml project is generator.GeneratorJava. This class has been introduced during the time window between March and May 2006 (i.e., in the middle of the observed history), where it underwent 10 changes—two more than the median. Since the class has not been introduced in the early stages of software development, the EM model was not able to correctly mark this class as change-prone. On the other hand, the class contains a well-known design issue, i.e., it is affected by a Complex Class code smell. Thus, the code metrics are particularly high (e.g., LCOM=49) and effective in capturing the change-proneness of the class.

All in all, the analyses conducted show that the problem of change prediction cannot be solved by only relying on a subset of metrics considered. More importantly, different models are able to capture different change-prone classes: from a practical point of view, this means that the investigated developer-based metrics can nicely complement evolution metrics, possibly providing additional
performance improvements when combined. At the same
time, the CM model can provide further insights, being
able to correctly recognize the change-proneness of a good
portion of classes missed by other models (e.g., CM iden-
tified 21% of classes that the EM model was not able to
identify).

**Summary for RQ3.** All the investigated models show
nice complementarities, being able to correctly capture
the change-proneness of the different classes. As a con-
sequence, our findings reveal the possibility to achieve
a better level of performance when considering a com-
bination of the predictors considered in this study.

### 4.4. RQ4: Performance of the Combined Model

The first step in the definition of a combined change
prediction model consisted of pruning the non-relevant
metrics through the application of the *Gain Ratio Fea-
ture Evaluation* algorithm [58], whose results are reported
in Table 9. As it is possible to see, both scattering metrics
included in the DCBM model provide a significant con-
tribution to the overall reduction of the entropy of the model:
in particular, the gain provided by the structural scattering
is quantifiable at 0.44, while the semantic scattering
provides a gain of around 0.37. Also the entropy of changes
and the frequency of modifications metrics exploited by
the BCCM and EM, respectively, provide a strong con-
tribution to the model (i.e., 0.41 for the former, 0.33 for
the latter). The results also show that other process met-
crics such as weighted change density and birth date of a
class (used by the EM model) still help in the reduction of
the uncertainty of the model during the classification of
change-prone classes. Interestingly, the algorithm found
that the CBO and RFC metrics, measuring coupling and
complexity of a class, respectively, might contribute to the
improvement of the prediction accuracy.

Conversely, the other metrics exploited by the con-
sidered models provide an information gain lower than
the threshold of 0.10, thus their usefulness can be con-
sidered limited. It is important to point out that also
the LOC metric gives a limited contribution when predict-
ing change-prone classes (expected reduction=0.08): this
means that the mere analysis of the size of a class cannot
characterize the number of changes that developers will
apply on it.

Based on these results, we devised a combined change
prediction model that only exploits the metrics reported
in bold in Table 9.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Info Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Struct. scattering</td>
<td>0.44</td>
</tr>
<tr>
<td>Change entropy</td>
<td>0.41</td>
</tr>
<tr>
<td>Semant. scattering</td>
<td>0.37</td>
</tr>
<tr>
<td>Change frequency</td>
<td>0.33</td>
</tr>
<tr>
<td>CBO</td>
<td>0.27</td>
</tr>
<tr>
<td>RFC</td>
<td>0.19</td>
</tr>
<tr>
<td>Weighted-Change-Density</td>
<td>0.15</td>
</tr>
<tr>
<td>Birth-Date</td>
<td>0.11</td>
</tr>
<tr>
<td>LCOM</td>
<td>0.08</td>
</tr>
<tr>
<td>LOC</td>
<td>0.08</td>
</tr>
<tr>
<td>Number of developers</td>
<td>0.08</td>
</tr>
<tr>
<td>Last-Change-Amount</td>
<td>0.08</td>
</tr>
<tr>
<td>Total-Amount-Changes</td>
<td>0.07</td>
</tr>
<tr>
<td>Changes-Since-Its-Birth</td>
<td>0.06</td>
</tr>
<tr>
<td>Change-Occurred</td>
<td>0.05</td>
</tr>
<tr>
<td>DIT</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 9: Gain provided by each feature to the prediction of change-
prone classes.

The results achieved by such a combined model are
reported in Table 6 and clearly show its superiority with
respect to the stand-alone models experimented in the pre-
vious research questions. Indeed, the overall F-Measure,
AUC-ROC, and MCC of the combined model are 12%,
10%, and 9% higher than DCBM (i.e., the best model
resulting from RQ2), respectively. Also the Brier Score
is 7% lower, meaning that the independent variables ex-
pected by the combined model are better related to the
dependent variable.

When compared to the other stand-alone models, the
results are still more evident, since the overall F-measure
of the combined model is 18%, 21%, 23%, and 22% higher
than the one of BCCM, EM, DM, and CM, re-
spectively. These results confirm that different stand-
alone models complement each other leading to higher ac-
curacy of the classification of change-prone classes. We
found an example of the capabilities of the combined
model in the classification of the change-proneness of the
class `synapse.ServerManager` belonging to the SYNAPSE
project. In the time window between October and De-
cember 2008 the class was change-prone since it changed
23 times while the median was 12. In this case, the only
stand-alone model able to correctly classify the class as
change-prone is the CM one: the values of CBO and RFC
were higher than the average of the period, while the
only developer committing changes to this class performed
focused modifications, thus leading the developer-based
models to fail in the prediction. The combined model has
instead been able to correctly classify this instance based
on the structural information considered. Similarly, the
combined model has been able to properly classify the
change-proneness of all the classes mentioned in the ex-
amples presented in the previous sections. For example,
both the classes `svg.io.SVGFigureFactory` (contained
in JHotDraw) and `chartMeter.Legend` (JFreeChart)
have been correctly predicted as change-prone, likely be-
cause of the presence of change frequency and scattering
metrics as independent variables of the combined model.

The statistical tests confirm the quantitative results
described so far. Indeed, as shown in Table 10 all the
comparisons reveal that the combined model performs sta-
In our study, we considered state-of-the-art change prediction models relying on their exact definitions. Some of them might account for approximations when computing the independent variables: for instance, the BCCM model defined by Hassan [35] uses a keyword-based approach to distinguish the feature introduction modifications adopted for the computation of the entropy of changes. We are aware that more sophisticated approaches might be more precise in the classification of feature introduction and enhancements, however the improvement of existing techniques is out of the scope of this paper.

As for the construction of the combined model, we firstly controlled for multi-collinearity [52] applying an effective feature selection algorithm such as information gain [58].

**Internal Validity.** Threats to internal validity concern factors that might have influenced our results. As for the evaluation procedure, we had the need to exploit change history information to compute the metrics composing the experimented developer-based models. Thus, the evaluation design adopted in our study is different from the ten-fold validation [20] generally exploited in the context of change prediction. In particular, we split the change history of the object systems into three-month time periods and we adopted a three-month sliding window to train and test the experimented bug prediction models. This type of validation is typically adopted when using process metrics as predictors [35], although it might be penalizing when using code metrics [21].

Furthermore, other temporal-based validation strategies, i.e., (i) using the entire change history accumulated until a release \( R_i \) as training set and data of the release \( R_{i+1} \) as test set or (ii) using the data coming from a release \( R_i \) and tested with the data of \( R_{i+1} \), could have been considered. However, we believe that these strategies do not fit well for prediction models relying on developer-related metrics: indeed, these metrics measure the dynamics among developers in a certain period of time, and thus taking into account time windows larger than 3 months would have been conceptually wrong, since developer-based metrics would have put together information happened far in the time.

This observation is also supported by experimental results. Specifically, we evaluated the performance of the experimented models when considering the two alternative validation strategies. In the first case, we trained the models using the entire change history accumulated until a release \( R_i \), while we tested it on the data of the release \( R_{i+1} \). Then, we adopted this procedure for all the system releases. As a result, we noticed that the performance of all the models decreased by up to 13% with respect to models trained using only the information of the previous three months. In the second case, we tested the models using a release-by-release scenario, finding that all the models tend to perform notably worse (on average, -6% in terms of F-Measure) than the three-month sliding window case. A complete report of these additional analysis is available in our online appendix [16].

Another threat is related to the use of developer-based and evolution metrics as predictors of the change-proneness of classes. Indeed, they somehow encapsulate the concept of change, possibly producing an “interplay” between independent and dependent variables of a prediction model. While the model proposed by Elish et al. [24] directly uses the number of changes a class in a previous
time window as predictor of the future change-proneness of that class, we carefully verified whether this possible interplay produced unreliable results, finding that the usefulness of the model is limited to the cases where a class has a central role in the system. As for the BCCM, DDBM, and DM models, it is important to note that all of them rely on metrics able to capture how complex is the development process under different perspectives (e.g., the number of developers who worked on a code component). Thus, they provide a higher abstraction level and do not directly measure the change-proneness of a class.

**Conclusion Validity.** Threats to conclusion validity refer to the relation between treatment and outcome. In order to evaluate the change prediction models we used metrics such as accuracy, precision, recall, F-Measure, and AUC-ROC, which are widely used in the evaluation of the performance of prediction models. Moreover, we interpreted the results also looking at the indications given by other metrics, i.e., MCC and Brier Score, and applying appropriate statistical procedures, i.e., the Wilcoxon [19] and the Cliff’s tests [31], to understand whether the differences in the performance of the experimental models were significant.

**External Validity.** As for the generalizability of the results, we analyzed twenty different systems from different application domains and having different characteristics (size, number of classes, etc.). However, we are aware that our study is based on systems developed in Java only, and therefore future investigations aimed at corroborating our findings on a different set of systems would be worthwhile.

6. Conclusion

Predicting the classes more likely to change in the future is an effective way to focus preventive maintenance activities on specific parts of a software system. While several researchers relied on code or evolution metrics to build change prediction models, little knowledge is available on the actual usefulness of developer-related factors in this context. This paper aims at bridging this gap, by providing (i) an empirical analysis of the performance achieved by three developer-based change prediction models on a set of twenty software systems and (ii) a comprehensive combined change prediction model exploiting developer-, process-, and product-metrics able to consider different aspects of software development. Specifically, the contributions made by this paper are:

1. **A large scale empirical investigation into the role of developer-related factors in change prediction.** To this aim, we analyzed the performance attained by three prediction models relying on metrics able to capture how complex is the development process under different perspectives [7, 21, 35].

2. **A comparison between developer-based and other state-of-the-art change prediction models.** We compared the prediction capabilities of models based on developer-related factors with two baseline approaches, i.e., the Evolution Model [24] and the Code Metric model [69].

3. **An analysis of the complementarity between the investigated models.** We evaluated the orthogonality of the different experimented models by computing overlap metrics and providing qualitative examples to understand under which situations a given model performs better than others.

4. **A combined change prediction model.** We exploited the complementaries between the standalone models to derive the set of metrics mostly connected with the phenomenon of interest, i.e., the change-proneness of classes, and devised a novel hybrid change prediction model.

The achieved results provide several findings:

- **Developer-based change prediction models generally show good performance.** Among them, the DDBM proposed by Di Nucci et al. [21] in the context of change prediction shows the best performance, reaching an overall F-Measure of 68% and an accuracy equals to 77%.
- **Developer-based change prediction models work better than a model built using code metrics.** In particular, when developers apply focused modifications in a given time period they are able to keep the complexity of the source code under control even in the cases where the code metrics highlight design issues.
- **The studied models show interesting complementaries, indicating that different metrics are suitable for predicting the change-proneness of different classes.**
- **The devised combined change prediction model has an overall F-Measure of 77% and outperforms the standalone baselines by up to 23%.**

Our future research agenda includes a deeper investigation into the factors leading classes to be more change-prone. At the same time, we envision to perform an extensive analysis of a wide range of maintainability problems and how they are impacted by developer-related factors. Part of this analysis is to study the relationship between these developer-related factors and the interplay between change-proneness and bug-proneness. Finally, part of our future research agenda includes (i) the analysis of the performance of the considered models in a cross-project setting and (ii) the definition of a new metrics of change-proneness that can effectively take into account the type and severity of changes, as well as the effort spent by developers when performing changes on the source code, and (iii) the replication of the study in an industrial context.