Enhancing Change Prediction Models using Developer-Related Factors

Gemma Catolino[1], Fabio Palomba[2], Andrea De Lucia[1], Filomena Ferrucci[1], Andy Zaidman[3]

[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 paper [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 change-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 CK metrics [18] for 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[2], 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?*

[2]https://github.com

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 α. More in detail, the model construction comprises two steps. Firstly the so-called *feature introduction* modifications, *i.e.,* changes applied in the time window α 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

Table 1: Characteristics of the Software Projects in Our Dataset

System	Period	% Change-prone Classes	#Releases	#Commits	#Dev.	#Classes	KLOCs
Apache Ant	Jan 2000-Jul 2014	35%	22	13,054	55	83-813	20-204
Apache Cassandra	Mar 2007-Jan 2012	22%	13	20,026	128	305-586	70-111
Apache Lucene	May 2004-Mar 2013	33%	44	13,784	62	376 - 5,506	102 - 142
Apache Poi	Jun 2003-Nov 2012	34%	25	5,472	21	763 - 2,854	154 - 542
Apache Synapse	Sep 2004-Oct 2010	22%	14	2,432	24	141 - 826	182 - 372
Apache Velocity	Nov 2000-Aug 2012	19%	24	12,924	22	188 - 872	238 - 527
Apache Xalan	Jan 1999-Dec 2011	22%	28	10,489	38	214 - 663	142 - 231
Apache Xerces	Nov 1999-Feb 2014	19%	16	5,471	34	162-736	62-201
ArgoUML	Oct 2002-Dec 2012	28%	16	19,961	31	777-1,415	147-249
aTunes	Aug 2005-Apr 2010	31%	31	6,276	21	141-655	20-106
FreeMind	Jun 2000-Feb 2012	28%	16	722	13	25-509	4-103
JEdit	Jan 2005-Jun 2012	24%	29	24,340	18	228-520	39-166
JFreeChart	Feb 1999-Jul 2013	33%	23	14,099	15	86-775	15-231
JHotDraw	Jan 2001-Dec 2012	23%	16	1,121	27	159-679	18-135
JVLT	Jan 2007-Dec 2012	29%	15	623	16	164-221	18-29
pBeans	Sep 2010-Sep 2015	28%	15	3,894	31	187 - 520	51 - 112
pdfTranslator	Oct 2008-Apr 2014	25%	21	1,038	18	383 - 471	16 - 48
Redaktor	Aug 2006-Dec 2011	31%	13	16,287	17	284 - 729	31 - 139
Serapion	May 2009-Mar 2013	26%	9	11,982	35	55 - 398	23 - 211
Zuzel	Nov 2004-Jan 2010	31%	18	8,279	31	119 - 392	119 - 293
Overall	**-**	**-**	**408**	**192,274**	**657**	**25-1,415**	**4-542**

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 α is computed exploiting the concept of Shannon entropy [63] as in the following equation:

$$entropy(c_i, \alpha) = -(p_k \cdot \log_2 p_k) \quad (1)$$

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 α over the total number of *feature introduction* modifications in α. 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 α as predictors. The scattering metrics are computed for each class c as in the following equations:

$$\text{StrScatPred}_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} \text{StrScat}_{d,\alpha} \quad (2)$$

$$\text{SemScatPred}_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} \text{SemScat}_{d,\alpha} \quad (3)$$

where $developers_{c,\alpha}$ represents the set of developers that worked on the class c during the time period α, 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 α. Given the set $CH_{d,\alpha}$ of classes changed by a developer d during a time period α, the structural scattering of a developer is computed as follow:

$$\text{StrScat}_{d,\alpha} = |CH_{d,\alpha}| \times \underset{\forall c_i, c_j \in CH_{d,\alpha}}{average} [dist(c_i, c_j)] \quad (4)$$

where $dist$ 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 α and it is computed as:

$$\text{SemScat}_{d,\alpha} = |CH_{d,\alpha}| \times \frac{1}{\underset{\forall c_i, c_j \in CH_{d,\alpha}}{average} [sim(c_i, c_j)]} \quad (5)$$

where the *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 α. In the first place, the set $Devs(c_i, \alpha)$, which composed of the developers that committed at least one change to a certain class c_i during the time period α, is computed. Then, the number of developers for c_i is given by the cardinality of the set $|Devs(c_i, \alpha)|$.

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

Table 2: Independent variables considered by Elish *et al.* [24].

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

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_i using the following formula:

$$InfoGain(M, f_i) = E(M) - E(M|f_i) \quad (6)$$

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

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

The algorithm quantifies how much uncertainty in M was reduced after splitting M on predictor f_1. In the context of 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 needed 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 , $ci + 1$) of TW we run ChangeDistiller [26], a tree differencing algorithm able to extract the fine-grained code changes between c_i and $ci + 1$. 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 Naive 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 α 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:

Table 3: Summary of the five investigated change prediction models

Abbreviation	Name	Description
BCCM [35]	Basic Code Change Model	It is based on the entropy of changes applied by developers in a given time period.
DCBM [21]	Developer Changes Based Model	It takes into account the developers structural and semantic scattering. The first measures how "structurally" far the code components modified by a developer in a given time period are. The second capture how much spread in terms of implemented responsibilities the code components modified by a developer in a given time period are.
DM [8]	Developer Model	It relies on the number of developers who modified a code component in a give time period.
EM [24]	Evolution Model	Based on a set of historical metrics shown in Table 2.
CM [69]	Code Metrics Model	It relies on a set of cohesion (*i.e.,* LCOM), coupling (*i.e.,* CBO and RFC), and inheritance metrics (*i.e.,* DIT).

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{8}$$

$$precision = \frac{TP}{TP + FP} \tag{9}$$

$$recall = \frac{TP}{TP + FN} \tag{10}$$

where TP is the number of change-prone classes classified as such by a prediction model; TN denotes the number of non-change-prone classes correctly classified by the model; FP and FN measure the number of classes for which a prediction model fails in identifying the change-proneness of classes by declaring these classes as change-prone (FP) or non-change-prone (FN). As an aggregate indicator of precision and recall, we also reported the *F-Measure*, a metric defined as the harmonic mean of precision and recall [4]:

$$F\text{-}Measure = 2 * \frac{precision * recall}{precision + recall} \tag{11}$$

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:

$$MCC = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{12}$$

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.

Thirdly, we computed the Brier score [14, 61], which measures the distance between the probabilities predicted by a model and the actual outcome. Formally, the Brier score is computed as follows:

$$Brier\text{-}score = \frac{1}{N}\sum_{i=1}^{N}(p_c - o_c) \tag{13}$$

where p_c is the probability predicted by the model on a class c, o_c is the actual outcome for class c, and N is the cardinality of the dataset. Lower Brier scores indicate better classifier performance, while higher scores indicate lower performance.

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 α=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, '–' .

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

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

$$corr_{m_i \cap m_j} = \frac{|corr_{m_i} \cap corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \tag{14}$$

$$corr_{m_i \setminus m_j} = \frac{|corr_{m_i} \setminus corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \tag{15}$$

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 investigfated 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

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

Project	BCCM							DCBM							DM						
	A	P	R	F-M	AR	MCC	BS	A	P	R	F-M	AR	MCC	BS	A	P	R	F-M	AR	MCC	BS
Ant	72	65	79	72	82	62	43	71	63	71	67	66	72	34	48	51	57	55	51	55	57
Cassandra	77	84	90	87	84	59	39	65	67	68	67	64	51	53	71	49	60	54	62	58	48
Lucene	60	58	71	64	60	59	47	75	73	64	68	71	72	37	49	54	58	56	58	53	47
Poi	79	70	85	77	77	66	34	50	55	59	57	55	53	46	75	46	49	47	60	59	48
Synapse	59	69	57	62	56	60	40	72	75	78	76	75	63	38	62	57	52	54	62	55	48
Velocity	75	62	58	60	64	65	41	79	61	64	62	74	69	36	56	61	72	66	61	50	51
Xalan	75	66	67	66	67	60	44	84	77	74	75	74	60	36	52	47	62	53	53	53	52
Xerces	87	69	71	70	62	64	35	69	66	75	71	52	55	62	76	73	66	69	62	54	51
ArgoUML	89	87	88	87	93	67	37	87	93	81	86	93	64	40	62	57	61	58	52	53	54
aTunes	63	58	62	60	67	66	39	61	52	55	54	53	54	52	66	75	50	60	63	54	46
FreeMind	35	35	28	31	59	68	40	68	42	45	44	70	59	42	63	62	64	63	63	53	49
JEdit	78	42	74	58	62	69	41	55	48	52	50	52	52	46	62	70	35	47	59	55	55
JFreeChart	71	45	67	56	55	67	45	73	42	62	52	55	68	35	64	45	61	53	57	54	67
JHotDraw	97	77	59	67	79	62	44	97	66	72	69	75	66	35	62	61	69	65	61	52	45
JVLT	80	50	50	50	50	61	41	81	51	76	62	59	63	38	49	44	48	46	52	54	56
pBeans	70	56	56	56	77	63	44	73	62	75	68	64	62	45	53	49	55	52	52	51	51
pdfTranslator	75	75	57	65	71	60	39	69	65	70	67	65	66	43	63	51	61	56	61	53	55
Redaktor	70	73	64	68	64	63	43	82	68	67	67	67	67	43	54	47	60	53	58	54	58
Serapion	68	63	63	63	72	68	48	71	70	86	77	73	68	37	61	60	56	58	57	53	54
Zuzel	55	72	65	68	66	66	49	84	71	73	72	67	67	39	55	51	55	53	55	50	58
Overall	**71**	**61**	**63**	**62**	**68**	**63**	**43**	**77**	**65**	**72**	**68**	**70**	**65**	**38**	**58**	**54**	**60**	**57**	**56**	**53**	**52**

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 LCOM[3] 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 since they encapsulated different responsibilities. Due to the high scattering of the developer, DCBM correctly predicts the change-proneness of the class. Thus, the results seem to delineate that the scattered changes applied by developers can produce some forms of software degradation that have effects on the change-proneness of classes. The statistical analyses conducted (see Table 7) confirm the superiority of DCBM with respect to DM ($\alpha < 0.01$, $d = 0.83$).

A similar discussion can be held when comparing the DCBM and BCCM models. From Table 5 we can observe that DCBM is able to obtain an F-Measure almost 6% 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 ($\alpha < 0.01$) 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 (*i.e.,* -0.13), 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

[3]It is worth remarking that the lower the LCOM the higher the cohesiveness of a class.

Table 6: 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 ; BC = Brier-Score

Project	EM							CM							Combined Model						
	A	P	R	F-M	AR	MCC	BS	A	P	R	F-M	AR	MCC	BS	A	P	R	F-M	AR	MCC	BS
Apache Ant	68	65	49	56	59	57	53	56	55	61	58	52	51	51	95	95	88	91	97	71	35
Apache Cassandra	88	79	85	82	91	65	40	61	61	65	63	59	55	52	83	87	93	90	92	81	31
Apache Lucene	79	69	72	70	63	55	49	58	55	66	60	62	50	49	82	78	68	73	75	76	32
Apache Poi	60	64	62	63	68	60	45	68	70	61	65	58	52	53	84	74	89	81	82	74	27
Apache Synapse	82	68	75	71	59	53	50	70	72	71	71	58	52	49	75	82	83	82	81	83	28
Apache Velocity	65	46	66	54	63	54	55	69	58	58	58	66	53	54	83	66	70	68	78	74	32
Apache Xalan	59	69	49	57	60	55	53	58	57	61	59	63	52	62	88	81	76	78	79	80	34
Apache Xerces	76	69	73	72	65	68	46	81	69	71	70	61	55	33	91	76	81	78	66	73	35
ArgoUML	67	39	82	53	61	56	48	73	73	45	55	76	54	60	87	72	75	73	78	82	27
aTunes	62	6	50	10	50	60	39	56	42	48	45	52	55	57	82	69	73	71	83	79	29
FreeMind	57	75	44	56	59	56	50	52	55	36	44	75	52	62	79	65	64	64	74	71	39
JEdit	75	48	53	51	63	60	37	54	42	50	45	50	51	59	81	69	72	70	68	70	31
JFreeChart	79	69	51	59	62	58	43	58	48	55	52	67	52	54	81	69	72	70	68	70	31
JHotDraw	60	38	78	51	60	53	48	42	46	27	34	50	52	56	98	75	85	80	84	64	28
JVLT	76	74	75	74	63	56	46	41	37	43	40	50	52	48	88	69	83	75	66	81	33
pBeans	71	64	70	67	60	54	50	59	67	69	68	64	54	50	79	67	79	73	71	78	27
pdfTranslator	58	58	49	53	62	59	54	63	71	70	70	59	55	58	74	73	75	74	79	73	28
Redaktor	56	41	59	48	61	59	47	72	59	71	64	68	53	50	86	72	79	75	72	69	29
Serapion	65	43	76	55	62	56	52	71	55	61	58	64	53	57	77	77	92	84	79	69	32
Zuzel	59	36	47	41	59	54	44	71	59	64	61	68	53	56	91	78	77	77	78	72	34
Overall	**68**	**58**	**60**	**59**	**61**	**56**	**49**	**71**	**63**	**64**	**64**	**70**	**64**	**43**	**77**	**70**	**74**	**72**	**70**	**64**	**42**

Table 7: Wilcoxon's t-test p-values of the hypothesis F-Measure achieved by a model is > than the compared model. Statistically significant results are reported in bold face. Cliff Delta d values are also shown.

Compared models	p-value	Cliff Delta	Magnitude
DCBM - BCCM	< **0.01**	0.71	large
DCBM - DM	< **0.01**	0.83	large
DCBM - EM	< **0.01**	0.73	large
DCBM - CM	< **0.01**	0.82	large
BCCM - DM	**0.04**	0.41	medium
BCCM - EM	**0.02**	0.21	small
BCCM - CM	< **0.01**	0.68	large
DM - EM	0.95	0.07	negligible
DM - CM	**0.02**	0.46	medium
EM - CM	0.43	0.02	negligible

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 semantic 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 ($\alpha < 0.01$) 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 `svg.io.SVGFigureFactory` is responsible for performing the main functionality of the entire project, *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 (*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 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[4]. 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 cases. 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, *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, *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 `xerces.dom.ElementImpl` of the Apache Xerces project. During the time window between May and July 2007, the class experienced only three changes (*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 `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.

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.

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 (*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

[4]As indicated in the release notes of the version 1.7.1, which correspond to that time period: http://tinyurl.com/hqwazgg

Table 8: Overlap among the experimented change prediction models.

	A=BCCM			A=DCBM			A=DM			A=EM			A=CM		
	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A	A∩B	A-B	B-A
B=BCCM	-	-	-	60	24	16	56	20	24	56	18	26	55	18	27
B=DCBM	60	16	24	-	-	-	51	19	30	53	20	27	43	23	35
B=DM	56	24	20	51	30	19	-	-	-	60	20	20	57	19	24
B=EM	56	26	18	53	27	20	60	20	20	-	-	-	47	32	21
B=CM	55	27	18	43	35	23	57	24	19	47	21	32	-	-	-

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 identified 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 consequence, our findings reveal the possibility to achieve a better level of performance when considering a combination 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 Feature 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 contribution 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 contribution to the model (*i.e.,* 0.41 for the former, 0.33 for the latter). The results also show that other process metrics 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 considered models provide an information gain lower than the threshold of 0.10, thus their usefulness can be considered limited. It is important to point out that also the LOC metric gives a limited contribution when predicting 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.

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 previous 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 exploited by the combined model are better related to the dependent variable.

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

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

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, respectively. These results confirm that different stand-alone models complement each other leading to higher accuracy 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 December 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 examples 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 because 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-

Table 10: Wilcoxon's t-test p-values of the hypothesis F-Measure achieved by a model is $>$ than the compared model. Statistically significant results are reported in bold face. Cliff Delta d values are also shown.

Compared models	p-value	Cliff Delta	Magnitude
Combined Model - DCBM	< **0.01**	0.66	large
Combined Model - BCCM	< **0.01**	0.73	large
Combined Model - DM	< **0.01**	0.77	large
Combined Model - EM	< **0.01**	0.78	large
Combined Model - CM	< **0.01**	0.81	large

tistically better than all the baselines. Thus, to conclude the discussion we can claim that an effective combination of metrics covering different aspects related to complexity of software development, previous history, and structural characteristics of classes can be significantly more effective in the identification of the classes more likely to be changed in the future. As a consequence, the model might better support developers interested in planning preventive maintenance activities.

Summary for RQ4. The combined model has an overall F-measure equals to 77%, while the overall accuracy is 85%. When comparing the performance of the combined model with the ones achieved by the standalone ones, we found that the accuracy improves by up to 23% in terms of the F-Measure.

5. Threats to Validity

This section describes the threats that can affect the validity of our study.

Construct Validity. Threats to *construct validity* concern the relationship between theory and observation. We exploited the guidelines provided by Romano *et al.* [59] in order to build a *golden set* reporting the actual change-prone classes present in each of the analyzed time windows. This strategy has been widely used in the past to assess the change-proneness of classes [24, 25, 69], and it is recognized as an efficient way to distinguish change and non-change prone classes [59]. However, we are aware that this definition of change-proneness does not take into account the type and severity of changes, as well as the effort spent by developers when changing the source code: we still preferred relying on the definition by Romano *et al.* [59] because of the lack of other established definitions that characterize the change-proneness of classes.

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, DCBM, 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 experimented 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 DCBM 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.

References

[1] Abdi, M., Lounis, H., and Sahraoui, H. 2006. Analyzing change impact in object-oriented systems. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*. IEEE, 310–319.

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

[3] Bacchelli, A. and Bird, C. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[4] Baeza-Yates, R., Ribeiro-Neto, B., et al. 1999. *Modern information retrieval.* Vol. 463. ACM press New York.

[5] Baldi, P., Brunak, S., Chauvin, Y., Andersen, C. A., and Nielsen, H. 2000. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics 16,* 5, 412–424.

[6] Bansiya, J. and Davis, C. G. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng. 28,* 1, 4–17.

[7] Bell, R. M., Ostrand, T. J., and Weyuker, E. J. 2011. Does measuring code change improve fault prediction? In *Proc. Int'l Conf. on Predictive Models in Software Engineering (PROMISE)*. ACM, 2.

[8] Bell, R. M., Ostrand, T. J., and Weyuker, E. J. 2013. The limited impact of individual developer data on software defect prediction. *Empirical Softw. Engg. 18,* 3, 478–505.

[9] Beller, M., Bacchelli, A., Zaidman, A., and Jürgens, E. 2014. Modern code reviews in open-source projects: which problems do they fix? In *11th Working Conference on Mining Software Repositories (MSR)*. ACM, 202–211.

[10] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, 470–481.

[11] Bieman, J. M., Straw, G., Wang, H., Munger, P. W., and Alexander, R. T. 2003. Design patterns and change proneness: an examination of five evolving systems. In *Proc. Int'l Workshop on Enterprise Networking and Computing in Healthcare Industry*. 40–49.

[12] Bottou, L. and Vapnik, V. 1992. Local learning algorithms. *Neural Comput. 4,* 6, 888–900.

[13] Briand, L. C., Wust, J., and Lounis, H. 1999. Using coupling measurement for impact analysis in object-oriented systems. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 475–482.

[14] Brier, G. W. 1950. Verification of Forecasts expressed in terms of probability. *Monthly Weather Review 78,* 1, 1–3.

[15] Catolino, G., Palomba, F., De Lucia, A., Ferrucci, F., and Zaidman, A. 2017a. Developer-related factors in change prediction: An empirical assessment. In *Proceedings of the 25th International Conference on Program Comprehension*. ICPC '17. IEEE Press, Piscataway, NJ, USA, 186–195.

[16] Catolino, G., Palomba, F., De Lucia, A., Ferrucci, F., and Zaidman, A. 2017b. Developer-related factors in change prediction: An empirical assessment - replication package - `http://tinyurl.com/y79ttbwa`.

[17] Chaumun, M. A., Kabaili, H., Keller, R. K., and Lustman, F. 1999. A change impact model for changeability assessment in object-oriented software systems. In *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 130–138.

[18] Chidamber, S. and Kemerer, C. 1994. A metrics suite for object oriented design. *IEEE Trans. on Softw. Engineering 20,* 6, 476–493.

[19] Conover, W. J. 1998. *Practical Nonparametric Statistics* 3rd Edition Ed. Wiley.

[20] Devijver, P. A. and Kittler, J. 1982. *Pattern Recognition: A Statistical Approach.*

[21] Di Nucci, D., Palomba, F., De Rosa, G., Bavota, G., Oliveto, R., and De Lucia, A. 2017. A developer centered bug prediction model. *IEEE Trans. on Softw. Engineering*, to appear.

[22] Di Penta, M., Cerulo, L., Gueheneuc, Y. G., and Antoniol, G. 2008. An empirical study of the relationships between design pattern roles and class change proneness. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 217–226.

[23] Ekanayake, J., Tappolet, J., Gall, H. C., and Bernstein, A. 2012. Time variance and defect prediction in software projects. *Empirical Software Engineering 17,* 4-5, 348–389.

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

[25] 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 *Proc. Int'l Conf Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 566–571.

[26] Fluri, B., Wuersch, 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.

[27] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[28] Freund, Y. and Mason, L. 1999. The alternating decision tree learning algorithm. In *icml.* Vol. 99. 124–133.

[29] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[30] Girba, T., Ducasse, S., and Lanza, M. 2004. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proc. Int'l Conf. Softw. Maintenance (ICSM)*. IEEE, 40–49.

[31] Grissom, R. J. and Kim, J. J. 2005. *Effect sizes for research: A broad practical approach* 2nd Edition Ed. Lawrence Earlbaum Associates.

[32] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. 2009. The weka data mining software: an update. *ACM SIGKDD explorations newsletter 11,* 1, 10–18.

[33] Han, A.-R., Jeon, S.-U., Bae, D.-H., and Hong, J.-E. 2008. Behavioral dependency measurement for change-proneness prediction in uml 2.0 design models. In *32nd Annual IEEE International Computer Software and Applications Conference*. IEEE, 76–83.

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

[35] Hassan, A. E. 2009. Predicting faults using the complexity of code changes. In *Int'l Conf. Software Engineering (ICSE)*. IEEE, 78–88.

[36] John, G. H. and Langley, P. 1995. Estimating continuous distributions in bayesian classifiers. In *Proc. Conf. on Uncertainty in artificial intelligence.* Morgan Kaufmann, 338–345.

[37] Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., and Antoniol, G. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Softw. Engg. 17,* 3, 243–275.

[38] Kohavi, R. 1995. The power of decision tables. In *European conference on machine learning.* Springer, 174–189.

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

[40] Kraut, R. E. and Streeter, L. A. 1995. Coordination in software development. *Commun. ACM 38,* 3, 69–81.

[41] Le Cessie, S. and Van Houwelingen, J. C. 1992. Ridge estimators in logistic regression. *Applied statistics*, 191–201.

[42] Lehman, M. M. and Belady, L. A., Eds. 1985. *Program Evolution: Processes of Software Change.* Academic Press Professional, Inc.

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

[44] Lindvall, M. 1999. Measurement of change: stable and change-prone constructs in a commercial c++ system. In *Proc. Int'l Software Metrics Symposium.* IEEE, 40–49.

[45] Liu, H. and Motoda, H. 1998. *Feature Selection for Knowledge Discovery and Data Mining.* Kluwer Academic Publishers, Norwell, MA, USA.

[46] Lu, H., Zhou, Y., Xu, B., Leung, H., and Chen, L. 2012. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical software engineering 17,* 3, 200–242.

[47] Malhotra, R. and Bansal, A. 2015. Predicting change using software metrics: A review. In *Int'l Conf. on Reliability, Infocom Technologies and Optimization (ICRITO).* IEEE, 1–6.

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

[49] Malhotra, R. and Khanna, M. 2014. A new metric for predicting software change using gene expression programming. In *Proc. Int'l Workshop on Emerging Trends in Software Metrics.* ACM, 8–14.

[50] Marinescu, C. 2014. How good is genetic programming at predicting changes and defects? In *Int'l Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC).* IEEE, 544–548.

[51] Miryung Kim, Tom Zimmermann, N. N. 2014. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering 40.*

[52] O'Brien, R. 2007. A caution regarding rules of thumb for variance inflation factors. *Quality & Quantity 41,* 5, 673.

[53] O'brien, R. M. 2007. A caution regarding rules of thumb for variance inflation factors. *Quality & Quantity 41,* 5, 673–690.

[54] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., and De Lucia, A. 2017. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 1–34.

[55] Parnas, D. L. 1994. Software aging. In *Proc. of the International Conference on Software Engineering (ICSE).* IEEE, 279–287.

[56] Peer, A. and Malhotra, R. 2013. Application of adaptive neuro-fuzzy inference system for predicting software change proneness. In *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on.* IEEE, 2026–2031.

[57] Posnett, D., Bird, C., and Dévanbu, P. 2011. An empirical study on the influence of pattern roles on change-proneness. *Empirical Software Engineering 16,* 3, 396–423.

[58] Quinlan, J. R. 1986. Induction of decision trees. *Mach. Learn. 1,* 1, 81–106.

[59] Romano, D. and Pinzger, M. 2011. Using source code metrics to predict change-prone java interfaces. In *Proc. Int'l Conf. Software Maintenance (ICSM).* IEEE, 303–312.

[60] Rosenblatt, F. 1961. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Tech. rep., DTIC Document.

[61] Rufibach, K. 2010. Use of Brier score to assess binary predictions. *Journal of Clinical Epidemiology 63,* 8, 938–939.

[62] Rumbaugh, J., Jacobson, I., and Booch, G. 2004. *Unified Modeling Language Reference Manual, The (2Nd Edition).* Pearson Higher Education.

[63] Shannon, C. E. 1948. A mathematical theory of communication. *Bell system technical journal 27.*

[64] Sharafat, A. R. and Tahvildari, L. 2007. A probabilistic approach to predict changes in object-oriented software systems. In *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR).* IEEE, 27–38.

[65] Sharafat, A. R. and Tahvildari, L. 2008. Change prediction in object-oriented software systems: A probabilistic approach. *Journal of Software 3,* 5, 26–39.

[66] Soetens, Q. D., Demeyer, S., Zaidman, A., and Pérez, J. 2016. Change-based test selection: An empirical evaluation. *Empirical Softw. Engg. 21,* 5, 1990–2032.

[67] 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.

[68] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. Context is king: The developer perspective on the usage of static analysis tools. to appear.

[69] 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.