A Developer Centered Bug Prediction Model

Dario Di Nucci[1], Fabio Palomba[1], Giuseppe De Rosa[1]
Gabriele Bavota[2], Rocco Oliveto[3], Andrea De Lucia[1]

[1]University of Salerno, Fisciano (SA), Italy, [2]Università della Svizzera italiana (USI), Switzerland,
[3]University of Molise, Pesche (IS), Italy

ddinucci@unisa.it, fpalomba@unisa.it, giuderos@gmail.com
gabriele.bavota@usi.ch, rocco.oliveto@unimol.it, adelucia@unisa.it

Abstract—Several techniques have been proposed to accurately predict software defects. These techniques generally exploit characteristics of the code artefacts (e.g., size, complexity, etc.) and/or of the process adopted during their development and maintenance (e.g., the number of developers working on a component) to spot out components likely containing bugs. While these bug prediction models achieve good levels of accuracy, they mostly ignore the major role played by human-related factors in the introduction of bugs. Previous studies have demonstrated that focused developers are less prone to introduce defects than non-focused developers. According to this observation, software components changed by focused developers should also be less error prone than components changed by less focused developers. We capture this observation by measuring the scattering of changes performed by developers working on a component and use this information to build a bug prediction model. Such a model has been evaluated on 26 systems and compared with four competitive techniques. The achieved results show the superiority of our model, and its high complementarity with respect to predictors commonly used in the literature. Based on this result, we also show the results of a "hybrid" prediction model combining our predictors with the existing ones.

Index Terms—Scattering Metrics, Bug Prediction, Empirical Study, Mining Software Repositories

1 Introduction

Bug prediction techniques are used to identify areas of software systems that are more likely to contain bugs. These prediction models represent an important aid when the resources available for testing are scarce, since they can indicate *where* to invest such resources. The scientific community has developed several bug prediction models that can be roughly classified into two families, based on the information they exploit to discriminate between "buggy" and "clean" code components. The first set of techniques exploits *product metrics* (*i.e.*, metrics capturing intrinsic characteristics of the code components, like their size and complexity) [1], [2], [3], [4], [5], while the second one focuses on *process metrics* (*i.e.*, metrics capturing specific aspects of the development process, like the frequency of changes performed to code components) [6], [7], [8], [9], [10], [11], [12]. While some studies highlighted the superiority of these latter with respect to the *product metric based* techniques [7], [13], [11] there is a general consensus on the fact that no technique is the best in all contexts [14], [15]. For this reason, the research community is still spending effort in investigating under which circumstances and during which coding activities developers tend to introduce bugs (see *e.g.*, [16], [17], [18], [19], [20], [21], [22]).

Some of these studies have highlighted the central role played by developer-related factors in the introduction of bugs.

In particular, Eyolfson *et al.* [17] showed that more experienced developers tend to introduce less faults in software systems. Rahman and Devanbu [18] partly contradicted the study by Eyolfson *et al.* by showing that the experience of a developer has no clear link with the bug introduction. Bird *et al.* [20] found that high levels of ownership are associated with fewer bugs. Finally, Posnett *et al.* [22] showed that focused developers (*i.e.*, developers focusing their attention on a specific part of the system) introduce fewer bugs than unfocused developers.

Although such studies showed the potential of human-related factors in bug prediction, this information is not captured in state-of-the-art bug prediction models based on process metrics extracted from version history. Indeed, previous bug prediction models exploit predictors based on (i) the number of developers working on a code component [9] [10]; (ii) the analysis of change-proneness [13] [11] [12]; and (iii) the entropy of changes [8]. Thus, despite the previously discussed finding by Posnett *et al.* [22], none of the proposed bug prediction models considers how focused the developers performing changes are and how scattered these changes are. In our previous work [23] we studied the role played by *scattered changes* in bug prediction. We defined two measures, namely the developer's *structural* and *semantic scattering*. The first assesses how "structurally far" in the software project the code components modified by a developer in a given time period are.

The "structural distance" between two code components is measured as the number of subsystems one needs to cross in order to reach one component from the other.

The second measure (*i.e.*, the *semantic scattering*) is instead meant to capture how much spread in terms of implemented responsibilities the code components modified by a developer in a given time period are. The conjecture behind the proposed metrics is that high levels of *structural* and *semantic scattering* make the developer more error-prone. To verify this conjecture, we built two predictors exploiting the proposed measures, and we used them in a bug prediction model. The results achieved on five software systems showed the superiority of our model with respect to (i) the Basic Code Change Model (BCCM) built using the entropy of changes [8] and (ii) a model using the number of developers working on a code component as predictor [9] [10]. Most importantly, the two scattering measures showed a high degree of complementarity with the measures exploited by the baseline prediction models.

In this paper, we extend our previous work [23] to further investigate the role played by scattered changes in bug prediction. In particular we:

1) Extend the empirical evaluation of our bug prediction model by considering a set of 26 systems.
2) Compare our model with two additional competitive approaches, *i.e.*, a prediction model based on the focus metrics proposed by Posnett *et al.* [22] and a prediction model based on structural code metrics [24], that together with the previously considered models, *i.e.*, the BCCM proposed by Hassan [8] and the one proposed by Ostrand *et al.* [9] [10], lead to a total of four different baselines considered in our study.
3) Devise and discuss the results of a *hybrid* bug prediction model, based on the best combination of predictors exploited by the five prediction models experimented in the paper.
4) Provide a comprehensive replication package [25] including all the raw data and working data sets of our studies.

The achieved results confirm the superiority of our model, achieving a F-Measure 10.3% higher, on average, than the change entropy model [8], 53.7% higher, on average, with respect to what achieved by exploiting the number of developers working on a code component as predictor [9], 13.3% higher, on average, than the F-Measure obtained by using the developers' focus metric by Posnett *et al.* [22] as predictor, and 29.3% higher, on average, with respect to the prediction model built on top of product metrics [1]. The two scattering measures confirmed their complementarity with the metrics used by the alternative prediction models. Thus, we devised a "hybrid" model providing an average boost in prediction accuracy (*i.e.*, F-Measure) of +5% with respect to the best performing model (*i.e.*, the one proposed in this paper).

Structure of the paper. Section 2 discusses the related literature, while Section 3 presents the proposed scattering measures. Section 4 presents the design of our empirical study and provides details about the data extraction process and analysis method. Section 5 reports the results of the study, while Section 6 discusses the threats that could affect their validity. Section 7 concludes the paper.

2 Related Work

Many bug prediction techniques have been proposed in the literature in the last decade. Such techniques mainly differ for the specific predictors they use, and can roughly be classified in those exploiting *product metrics* (*e.g.*, lines of code, code complexity, etc), those relying on *process metrics* (*e.g.*, change- and fault-proneness of code components), and those exploiting a mix of the two. Table 1 summarizes the related literature, by grouping the proposed techniques on the basis of the metrics they exploit as predictors.

The Chidamber and Kemerer (CK) metrics [36] have been widely used in the context of bug prediction. Basili *et al.* [1] investigated the usefulness of the CK suite for predicting the probability of detecting faulty classes. They showed that five of the experimented metrics are actually useful in characterizing the bug-proneness of classes. The same set of metrics has been successfully exploited in the context of bug prediction by El Emam *et al.* [26] and Subramanyam *et al.* [27]. Both works reported the ability of the CK metrics in predicting buggy code components, regardless of the size of the system under analysis.

Still in terms of product metrics, Nikora *et al.* [28] showed that measuring the evolution of structural attributes (*e.g.*, number of executable statements, number of nodes in the control flow graph, etc.) it is possible to predict the number of bugs introduced during the system development. Later, Gyimothy *et al.* [2] performed a new investigation on the relationship between CK metrics and bug proneness. Their results showed that the *Coupling Between Object* metric is the best in predicting the bug-proneness of classes, while other CK metrics are untrustworthy.

Ohlsson *et al.* [3] focused the attention on the use of design metrics to identify bug-prone modules. They performed a study on an Ericsson industrial system showing that at least four different design metrics can be used with equivalent results. The metrics performance are not statistically worse than those achieved using a model based on the project size. Zhou *et al.* [29] confirmed their results showing that size-based models seem to perform as well as those based on CK metrics except than the *Weighted Method per Class* on some releases of the Eclipse system. Thus, although Bell *et al.* [35] showed that more complex metric-based models have more predictive power with respect to size-based models, the latter seem to be generally useful for bug prediction.

TABLE 1
Prediction models proposed in literature

Type of Information Exploited	Prediction Model	Predictors
Product metrics	Basili *et al.* [1]	CK metrics
	El Emam *et al.* [26]	CK metrics
	Subramanyam *et al.* [27]	CK metrics
	Nikora *et al.* [28]	CFG metrics
	Gyimothy *et al.* [2]	CK metrics, LOC
	Ohlsson *et al.* [3]	CFG metrics, complexity metrics, LOC
	Zhou *et al.* [29]	CK metrics, OO metrics, complexity metrics, LOC
	Nagappan *et al.* [14]	CK metrics, CFG metrics, complexity metrics
Process metrics	Khoshgoftaar *et al.* [6]	debug churn
	Nagappan *et al.* [30]	relative code churn
	Hassan and Holt [31]	entropy of changes
	Hassan and Holt [32]	entropy of changes
	Kim *et al.* [33]	previous fault location
	Hassan [8]	entropy of changes
	Ostrand *et al.* [10]	number of developers
	Nagappan *et al.* [34]	consecutive changes
	Bird *et al.* [20]	social network analysis on developers' activities
	Ostrand *et al.* [9]	number of developers
	Posnett *et al.* [22]	module activity focus, developer attention focus
Product and process metrics	Graves *et al.* [7]	various code and change metrics
	Nagappan and Ball [4]	LOC, past defects
	Bell *et al.* [35]	LOC, age of files, number of changes, program type
	Zimmerman *et al.* [5]	complexity metrics, CFG metrics, past defects
	Moser *et al.* [13]	various code and change metrics
	Moser *et al.* [11]	various code and change metrics
	Bell *et al.* [12]	various code and change metrics
	D'Ambros *et al.* [15]	various code and change metrics

Nagappan and Ball [4] exploited two static analysis tools to early predict the pre-release bug density. The results of their study, conducted on the Windows Server system, show that it is possible to perform a coarse grained classification between high and low quality components with a high level of accuracy. Nagappan *et al.* [14] analyzed several complexity measures on five Microsoft software systems, showing that there is no evidence that a single set of measures can act universally as bug predictor. They also showed how to methodically build regression models based on similar projects in order to achieve better results. Complexity metrics in the context of bug prediction are also the focus of the work by Zimmerman *et al.* [5]. Their study reports a positive correlation between code complexity and bugs.

Differently from the previous discussed techniques, other approaches try to predict bugs by exploiting process metrics. Khoshgoftaar *et al.* [6] analyzed the contribution of debug churns (defined as the number of lines of code added or changed to fix bugs) to a model based on product metrics in the identification of bug-prone modules. Their study, conducted on two subsequent releases of a large legacy system, shows that modules exceeding a defined threshold of debug churns are often bug-prone. The reported results show a misclassification rate of just 21%.

Nagappan *et al.* [30] proposed a technique for early bug prediction based on the use of relative code churn measures. These metrics relate the number of churns to other factors such as LOC or file count. An experiment performed on the Windows Server system showed that relative churns are better than absolute value.

Hassan and Holt [31] conjectured that a chaotic development process has bad effects on source code quality and introduced the concept of entropy of changes. Later they also presented the top-10 list [32], a methodology to highlight to managers the top ten subsystems more likely to present bugs. The set of heuristics behind their approach includes a number of process metrics, such as considering the *most recently modified*, the *most frequently modified*, the *most recently fixed* and the *most frequently fixed* subsystems.

Bell *et al.* [12] pointed out that although code churns are very effective bug predictors, they cannot improve a simpler model based on the code components' change-proneness. Kim *et al.* [33] presumed that faults do not occur in isolation but in burst of related faults. They proposed the bug cache algorithm that predicts future faults considering the location of previous faults. Similarly, Nagappan *et al.* [34] defined change burst as a set of *consecutive changes over a period of time* and proposed new metrics based on change burst. The evaluation of the prediction capabilities of the models was performed on Windows Vista, achieving high accuracy.

Graves *et al.* [7] experimented both product and process metrics for bug prediction. They observed that history-based metrics are more powerful than product metrics (*i.e.*, change-proneness is a better indicator than LOC). Their best results were achieved using a combination of module's age and number of changes, while combining product metrics had no positive effect on the bug prediction. They also saw no benefits provided by the inclusion of a metric based on the number of developers modifying a code component.

Moser *et al.* [13] performed a comparative study between product-based and process-based predictors. Their study, performed on Eclipse, highlights the superiority of process metrics in predicting buggy code components. Later, they performed a deeper study [11] on the bug prediction accuracy of process metrics, reporting that the *past number of bug-fixes performed on a file* (*i.e.*, bug-proneness), the *maximum changeset size occurred in a given period*, and the *number of changes involving a file in a given period* (*i.e.*, change-proneness) are the process metrics ensuring the best performances in bug prediction.

D'Ambros *et al.* [15] performed an extensive comparison of bug prediction approaches relying on process and product metrics, showing that no technique based on a single metric works better in all contexts.

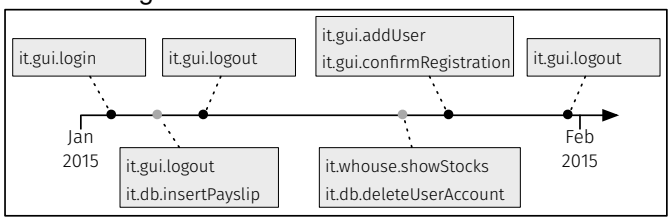
Hassan [8] analyzed the complexity of the development process. In particular he defined the entropy of changes as the scattering of code changes across time. He proposed three bug prediction models, namely Basic Code Change Model (BCCM), Extended Code Change Model (ECCM), and File Code Change Model (FCCM). These models mainly differ for the choice of the temporal interval where the bug proneness of components is studied. The reported study indicates that the proposed techniques have a stronger prediction capability than a model purely based on the amount of changes applied to code components or on the number of prior faults. Differently from our work, all these predictors do not consider the number of developers who performed changes to a component, neither how many components they changed at the same time.

Ostrand *et al.* [9], [10] proposed the use of the *number of developers who modified a code component in a give time period* as a bug predictor. Their results show that combining developers' information poorly, but positively, impact the detection accuracy of a prediction model. Our work does not use a simple count information of developers who worked on a file, but also takes in consideration the change activities they carry out.

Bird *et al.* [20] investigated the relationship between different ownership measures and pre- and post-releases failures. Specifically, they analyzed the developers' contribution network by means of social network analysis metrics, finding that developers having low levels of ownership tend to increase the likelihood of introducing defects. Our scattering metrics are not based on code ownership, but on the "distance" between the code components modified by a developer in a given time period.

Posnett *et al.* [22] investigated factors related to the one we aim at capturing in this paper, *i.e.*, the developer's scattering. In particular, the "focus" metrics presented by Posnett *et al.* [22] are based on the idea that a developer performing most of her activities on a single module (a module could be a method, a class, etc.) has a higher focus on the activities she is performing and is less likely to introduce bugs.

Fig. 1. Example of two developers having different levels of "scattering"



Following this conjecture, they defined two symmetric metrics, namely the *Module Activity Focus* metric (shortly, *MAF*), and the *Developer Attention Focus* metric (shortly, *DAF*) [22]. The former is a metric which captures to what extent a module receives focused attention by developers. The latter measures how focused are the activities of a specific developer. As it will be clearer later, our scattering measures not only take into account the frequency of changes made by developers over the different system's modules, but also considers the "distance" between the modified modules. This means that, for example, the contribution of a developer working on a high number of files all closely related to a specific responsibility might not be as much "scattered" as the contribution of a developer working on few *unrelated* files.

3 Computing Developer's Scattering Changes

We conjecture that the developer's effort in performing maintenance and evolution tasks is proportional to the number of involved components and their spread across different subsystems. In other words, we believe that a developer working on different components scatters her attention due to continuous changes of context. This might lead to an increase of the developer's *"scattering"* with a consequent higher chance of introducing bugs.

To get a better idea of our conjecture, consider the situation depicted in Figure 1, where two developers, d_1 (black point) and d_2 (grey point) are working on the same system, during the same time period, but on different code components. The tasks performed by d_1 are very focused on a specific part of the system (she mainly works on the system's GUI) and on a very targeted topic (she is mainly in charge of working on GUIs related to the users' registration and login features). On the contrary, d_2 performs tasks scattered across different parts of the system (from GUIs to database management) and on different topics (users' accounts, payslips, warehouse stocks).

Our conjecture is that during the time period shown in Figure 1, the contribution of d_2 might have been more "scattered" than the contribution of d_1, thus having a higher likelihood of introducing bugs in the system.

To verify our conjecture we define two measures, named the *structural* and the *semantic scattering* measures, aimed at assessing the scattering of a developer d in a given time period p. Note that both measures are meant to work in object oriented systems at the class level granularity. In other words, we measure how scattered are the changes performed by developer d during the time period p across the different classes of the system. However, our measures can be easily adapted to work at other granularity levels.

3.1 Structural scattering

Let $CH_{d,p}$ be the set of classes changed by a developer d during a time period p. We define the *structural scattering* measure as:

$$\text{StrScat}_{d,p} = |CH_{d,p}| \times \underset{\forall c_i, c_j \in CH_{d,p}}{average} [dist(c_i, c_j)] \quad (1)$$

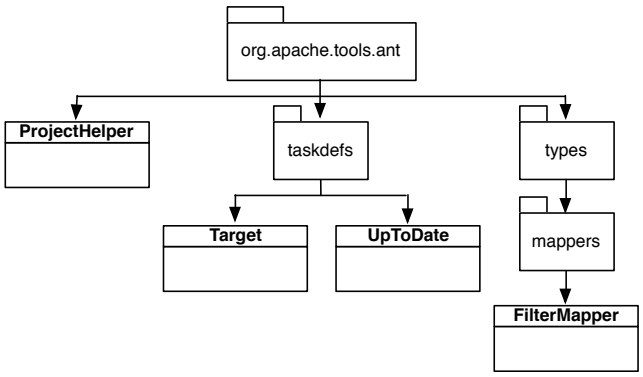
where $dist$ is the number of packages to traverse in order to go from class c_i to class c_j; $dist$ is computed by applying the shortest path algorithm on the graph representing the system's package structure. For example, the $dist$ between two classes `it.user.gui.`c_1 and `it.user.business.db.`c_2 is three, since in order to reach c_1 from c_2 we need to traverse `it.user.business.db`, `it.user.business`, and `it.user.gui`. We (i) use the $average$ operator for normalizing the distances between the code components modified by the developer during the time period p and (ii) assign a higher scattering to developers working on a higher number of code components in the given time period (see $|CH_{d,p}|$). Note the the choice to use the average to normalize the distances is driven by the fact that other central operators, such as the median, are not affected by the outliers. Indeed, suppose that a developer performs a change (*i.e.,* commit) C, modifying four files F1, F2, F3, and F4. The first three files are in the same package, while the fourth one (F4) is in a different subsystem. When computing the structural scattering for C, the median would not reflect the scattering of the change performed on F4, since half of the six pairs of files involved in the change (and in particular, F1-F2, F1-F3, F2-F3) have zero as structural distance (*i.e.*, they are in the same package). Thus, the median would not capture the fact that C was, at least in part, a scattered change. This is instead captured by the mean that is influenced by outliers.

To better understand how the *structural scattering* measure is computed and how it is possible to use it in order to estimate the developer's scattering in a time period, Figure 2 provides a running example based on a real scenario we found in *Apache Ant*[1], a tool to automate the building of software projects.

The tree shown in Figure 2 depicts the activity of a single developer in the time period between 2012-03-01 and 2012-04-30.

1. http://ant.apache.org/

Fig. 2. Example of structural scattering



In particular, the leafs of the tree represent the classes modified by the developer in the considered time period, while the internal nodes (as well as the root node) illustrate the package structure of the system. In this example, the developer worked on the classes `Target` and `UpToDate`, both contained in the package `org.apache.tools.ant.taskdefs` grouping together classes managing the definition of new commands that the *Ant*'s user can create for customizing her own building process. In addition, the developer also modified `FilterMapper`, a class containing utility methods (*e.g.*, map a java String into an array), and the class `ProjectHelper` responsible for parsing the build file and creating java instances representing the build workflow. To compute the *structural scattering* we compute the distance between every pair of classes modified by the developer. If two classes are in the same package, as in the case of the classes `Target` and `UpToDate`, then the distance between them will be zero. Instead, if they are in different packages, like in the case of `ProjectHelper` and `Target`, their distance is the minimum number of packages one needs to traverse to reach one class from the other. For example, the distance is one between `ProjectHelper` and `Target` (we need to traverse the package `taskdefs`), and three between `UpToDate` and `FilterMapper` (we need to traverse the packages `taskdefs`, `types` and `mappers`).

After computing the distance between every pair of classes, we can compute the *structural scattering*. Table 2 shows the structural distances between every pair of classes involved in our example as well as the value for the *structural scattering*. Note that, if the developer had modified only the `Target` and `UpToDate` classes in the considered time period, then her *structural scattering* would have been zero (the lowest possible), since her changes were focused on just one package. By also considering the change performed to `ProjectHelper`, the *structural scattering* raises to 2.01. This is due to the number of classes involved in change set (3) and the average of the distance among them (0.67).

TABLE 2
Example of structural scattering computation

Changed components		Distance
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.taskdefs.Target	1
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.taskdefs.UpToDate	1
org.apache.tools.ant.ProjectHelper	org.apache.tools.ant.types.mappers.FilterMapper	2
org.apache.tools.ant.taskdefs.Target	org.apache.tools.ant.taskdefs.UpToDate	0
org.apache.tools.ant.taskdefs.Target	org.apache.tools.ant.types.mappers.FilterMapper	3
org.apache.tools.ant.taskdefs.UpToDate	org.apache.tools.ant.types.mappers.FilterMapper	3
Structural Developer scattering		**6.67**

Finally the *structural scattering* reaches the value of 6.67 when also considering the change to the `FilterMapper` class. In this case the change set is composed of 4 classes and the average of the distances among them is 1.67. Note that the *structural scattering* is a direct scattering measure: the higher the measure, the higher the *estimated* developer's scattering.

3.2 Semantic scattering

Considering the package structure might not be an effective way of assessing the similarity of the classes (*i.e.*, to what extent the modified classes implement similar responsibilities). Because of the software "aging" or wrong design decisions, classes grouped in the same package may have completely different responsibilities [37]. In such cases, the *structural scattering* measure might provide a wrong assessment of the level of developer's scattering, by considering classes implementing different responsibilities as similar only because grouped inside the same package. For this reason, we propose the *semantic scattering* measure, based on the textual similarity of the changed software components. Textual similarity between documents is computed using the *Vector Space Model* (VSM) [38]. In our application of VSM we (i) used *tf-idf* weighting scheme [38], (ii) normalized the text by splitting the identifiers (we also have maintained the original identifiers), (iii) applied a stop word removal, and (iv) stemmed the words to their root (using the well known Porter stemmer). The semantic scattering measure is computed as:

$$\text{SemScat}_{d,p} = |CH_{d,p}| \times \frac{1}{\underset{\forall c_i, c_j \in CH_{d,p}}{average} \; [sim(c_i, c_j)]} \qquad (2)$$

where the sim function returns the textual similarity between the classes c_i and c_j as a value between zero (no textual similarity) and one (the textual content of the two classes is identical). Note that, as for the *structural scattering*, we adopt the $average$ operator and assign a higher scattering to developers working on a higher number of code components in the given time period.

Figure 3 shows an example of computation for the *semantic scattering* measure. Also in this case the figure depicts a real scenario we identified in *Apache Ant* of a single developer in the time period between 2004-04-01 and 2004-06-30. The developer worked on the classes `Path`, `Resource` and `ZipScanner`, all contained in the package `org.apache.tools.ant.types`.

Fig. 3. Example of semantic scattering measure

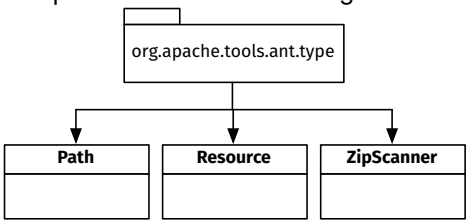


TABLE 3
Example of semantic scattering computation

Changed components		Text. sim.
org.apache.tools.ant.type.Path	org.apache.tools.ant.type.Resource	0.22
org.apache.tools.ant.type.Path	org.apache.tools.ant.type.ZipScanner	0.05
org.apache.tools.ant.type.Resource	org.apache.tools.ant.type.ZipScanner	0.10
Semantic Developer scattering		**24.32**

`Path` and `Resource` are two data types and have some code in common, while `ZipScanner` is an archives scanner. While the *structural scattering* is zero for the example depicted in Figure 3 (all classes are from the same package), the *semantic scattering* is quite high (24.32) due to the low textual similarity between the pairs of classes contained in the package (see Table 3). To compute the *semantic scattering* we firstly calculate the textual similarity between every pair of classes modified by the developer, as reported in Table 3. Then we calculate the average of the textual similarities (≈ 0.12) and we apply the inverse operator (≈ 8.11). Finally the *semantic scattering* is calculated multiplying the obtained value by the number of elements in the change set, that is 3, achieving the final result of ≈ 24.32.

3.3 Applications of Scattering Measures

The scattering measures defined above could be adopted in different areas concerned with monitoring maintenance and evolution activities. As an example, a project manager could use the scattering measures to estimate the workload of a developer, as well as to re-allocate resources. In the context of this paper, we propose the use of the defined measures for class-level bug prediction (**i.e.,** to predict which classes are more likely to be buggy). The basic conjecture is that *developers having a high scattering are more likely to introduce bugs during code change activities*.

To exploit the defined scattering measures in the context of bug prediction, we built a new prediction model called *Developer Changes Based Model* (DCBM) that analyzes the components modified by developers in a given time period. The model exploits a machine learning algorithm built on top of two predictors. The first, called *structural scattering predictor*, is defined starting from the structural scattering measure, while the second one, called *semantic scattering predictor*, is based on the semantic scattering measure.

The predictors are defined as follow:

$$\text{StrScatPred}_{c,p} = \sum_{d \in developers_{c,p}} \text{StrScat}_{d,p} \quad (3)$$

$$\text{SemScatPred}_{c,p} = \sum_{d \in developers_{c,p}} \text{SemScat}_{d,p} \quad (4)$$

where the $developers_{c,p}$ is the set of developers that worked on the component c during the time period p.

4 Evaluating Scattering Metrics in the Context of Bug Prediction

The *goal* of the study is to evaluate the usefulness of the developer's scattering measures in the prediction of bug-prone components, with the *purpose* of improving the allocation of resources in the verification & validation activities focusing on components having a higher bug-proneness. The *quality focus* is on the detection accuracy and completeness of the proposed technique as compared to competitive approaches. The *perspective* is of researchers, who want to evaluate the effectiveness of using information about developer scattered changes in identifying bug-prone components.

The *context* of the study consists of 26 Apache software projects having different size and scope. Table 4 reports the characteristics of the analyzed systems, and in particular (i) the software history we investigated, (ii) the mined number of commits, (iii) the size of the active developers base (those who performed at least one commit in the analyzed time period), (iv) the system's size in terms of KLOC and number of classes, and (v) the percentage of buggy files identified (as detailed later) during the entire change history. All data used in our study are publicly available [25].

4.1 Research Questions and Baseline Selection

In the context of the study, we formulated the following research questions:

- **RQ$_1$:** *What are the performances of a bug prediction model based on developer's scattering measures and how it compares to baseline techniques proposed in literature?*
- **RQ$_2$:** *What is the complementarity between the proposed bug prediction model and the baseline techniques?*
- **RQ$_3$:** *What are the performances of a "hybrid" model built by combining developer's scattering measures with baseline predictors?*

In the first research question we quantify the performances of a prediction model based on developer's scattering measures (DCBM). Then, we compare its performances with respect to four baseline prediction models, one based on product metrics and the other three based on process metrics.

The first model exploits as predictor variables the CK metrics [1], and in particular size metrics (*i.e.*, the Lines of Code—LOC—and the Number of Methods—NOM), cohesion metrics (*i.e.*, the Lack of Cohesion of Method—LCOM), coupling metrics (*i.e.*, the Coupling Between Objects—CBO—and the Response for a Class—RFC), and complexity metrics (*i.e.*, the Weighted Methods per Class—WMC). We refer to this model as CM.

We also compared our approach with three prediction models based on process metrics. The first is the one based on the work by Ostrand *et al.* [10], and exploiting the number of developers that work on a code component in a specific time period as predictor variable (from now on, we refer to this model as DM).

The second is the Basic Code Change Model (BCCM) proposed by Hassan and using code change entropy information [8]. This choice is justified by the superiority of this model with respect to other techniques exploiting change-proneness information [11], [12], [13]. While such a superiority has been already demonstrated by Hassan [8], we also compared these techniques before choosing BCCM as one of the baselines for evaluating our approach. We found that the BCCM works better with respect to a model that simply counts the number of changes. This is because it filters the changes that differ from the code change process (i.e., fault repairing and general maintenance modifications) considering only the *Feature Introduction modifications* (FI), namely the changes related to adding or enhancing features. However, we observed a high overlap between the BCCM and the model that use the number of changes as predictor (almost 84%) on the dataset used for the comparison, probably due to the fact that the nature of the information exploited by the two models is similar. The interested reader can find the comparison between these two models in our online appendix [25].

Finally, the third baseline is a prediction model based on the *Module Activity Focus* metric proposed by Posnett *et al.* [22]. It relies on the concept of predator-prey food web existing in ecology (from now on, we refer to this model as MAF). The metric is based on the measurement of the degree to which a code component receives focused attention by developers. It can be considered as a form of ownership metric of the developers on the component. It is worth noting that we do not consider the other *Developer Attention Focus* metric proposed by Posnett *et al.*, since (i) the two metrics are symmetric, and (ii) in order to provide a probability that a component is buggy, we need to qualify to what extent the activities on a file are focused, rather than measuring how are developers' activities focused. Even if Posnett *et al.* have not proposed a prediction model based on their metric, the results of this comparison will provide insights on the usefulness of developer's scattering measures for detecting bug-prone components.

Note that our choice of the baselines is motived by the will of: (i) considering both models based on product and process metrics, and (ii) covering a good number of different process metrics (since our model exploits process metrics), including approaches exploiting information similar to the ones used by our scattering metrics.

TABLE 4
Characteristics of the software systems used in the study

Project	Period	#Commits	#Dev.	#Classes	KLOC	% buggy classes
AMQ	Dec 2005 - Sep 2015	8,577	64	2,528	949	54
Ant	Jan 2000 - Jul 2014	13,054	55	1,215	266	72
Aries	Sep 2009 - Sep 2,015	2,349	24	1,866	343	40
Camel	Mar 2007 - Sep 2015	17,767	128	12,617	1,552	30
CXF	Apr 2008 - Sep 2015	10,217	55	6,466	1,232	26
Drill	Sep 2012 - Sep 2015	1,720	62	1,951	535	63
Falcon	Nov 2011 - Sep 2015	1,193	26	581	201	25
Felix	May 2007 - May 2015	11,015	41	5,055	1,070	18
JMeter	Sep 1998 - Apr 2014	10,440	34	1,054	192	37
JS2	Feb 2008 - May 2015	1,353	7	1,679	566	34
Log4j	Nov 2000 - Feb 2014	3,274	21	309	59	58
Lucene	Mar 2010 - May 2015	13,169	48	5,506	2,108	12
Oak	Mar 2012 - Sep 2015	8,678	19	2,316	481	43
OpenEJB	Oct 2011 - Jan 2013	9,574	35	4,671	823	36
OpenJPA	Jun 2007 - Sep 2015	3,984	25	4,554	822	38
Pig	Oct 2010 - Sep 2015	1,982	21	81,230	48,360	16
Pivot	Jan 2010 - Sep 2015	1,488	8	11,339	7,809	22
Poi	Jan 2002 - Aug 2014	5,742	35	2,854	542	62
Ranger	Aug 2014 - Sep 2015	622	18	826	443	37
Shindig	Feb 2010 - Jul 2015	2,000	27	1,019	311	14
Sling	Jun 2009 - May 2015	9,848	29	3,951	1,007	29
Sqoop	Jun 2011 - Sep 2015	699	22	667	134	14
Sshd	Dec 2008 - Sep 2015	629	8	658	96	33
Synapse	Aug 2005 - Sep 2015	2,432	24	1,062	527	13
Whirr	Jun 2010 - Apr 2015	569	17	275	50	21
Xerces-J	Nov 1999 - Feb 2014	5,471	34	833	260	6

In the second research question we aim at evaluating the complementarity of the different models, while in the third one we build and evaluate a "hybrid" prediction model exploiting as predictor variables the scattering measures we propose as well as the measures used by the four experimented competitive techniques (*i.e.*, DM, BCCM, MAF, and CM). Note that we do not limit our analysis to the experimentation of a model including all predictor variables, but we exercise all 2,036 possible combinations of predictor variables to understand which is the one achieving the best performances.

4.2 Experimental process and oracle definition

To evaluate the performances of the experimented bug prediction models we need to define the machine learning classifier to use. For each prediction technique, we experimented several classifiers, namely ADTree [39], Decision Table Majority [40], Logistic Regression [41], Multilayer Perceptron [42] and Naive Bayes [43]. We empirically compared the results achieved by the five different models on the software systems used in our study (more details on the adopted procedure later in this section). For all the prediction models the best results were obtained using the Majority Decision Table (the comparison among the classifiers can be found in our online appendix [25]). Thus, we exploit it in the implementation of the five models. This classifier can be viewed as an extension of one-valued decision trees [40]. It is a rectangular table where the columns are labeled with predictors and rows are sets of decision rules.

Each decision rule of a decision table is composed of (i) a pool of conditions, linked through and/or logical operators which are used to reflect the structure of the if-then rules; and (ii) an outcome which mirrors the classification of a software entity respecting the corresponding rule as bug-prone or non bug-prone. Majority Decision Table uses an attribute reduction algorithm to find a good subset of predictors with the goal of eliminating equivalent rules and reducing the likelihood of overfitting the data.

To assess the performance of the five models, we split the change-history of the object systems into three-month time periods and we adopt a three-month sliding window to *train* and *test* the bug prediction models. Starting from the first time period TP_1 (*i.e.*, the one starting at the first commit), we train each model on it, and test its ability in predicting buggy classes on TP_2 (*i.e.*, the subsequent three-month time period). Then, we move three months forward the sliding window, training the classifiers on TP_2 and testing their accuracy on TP_3. This process is repeated until the end of the analyzed change history (see Table 4) is reached. Note that our choice of considering three-month periods is based on: (i) choices made in previous work, like the one by Hassan *et al.* [8]; and (ii) the results of an empirical assessment we performed on such a parameter showing that the best results for all experimented techniques are achieved by using three-month periods. In particular, we experimented with time windows of one, two, three, and six months. The complete results are available in our replication package [25].

Finally, to evaluate the performances of the five experimented models we need an oracle reporting the presence of bugs in the source code.

Although the PROMISE repository collects a large dataset of bugs in open source systems [44], it provides oracles at release-level. Since the proposed measures work at time period-level, we had to build our own oracle. Firstly, we identified bug fixing commits happened during the change history of each object system by mining regular expressions containing issue IDs in the change log of the versioning system (*e.g.*, *"fixed issue #ID"* or *"issue ID"*). After that, for each identified issue ID, we downloaded the corresponding issue report from the issue tracking system and extracted the following information: *product name*; *issue's type* (*i.e.*, whether an issue is a bug, enhancement request, etc); *issue's status* (*i.e.*, whether an issue was closed or not); *issue's resolution* (*i.e.*, whether an issue was resolved by fixing it, or it was a duplicate bug report, or a "works for me" case); *issue's opening date*; *issue's closing date*, if available.

Then, we checked each issue's report to be correctly downloaded (*e.g.*, the issue's ID identified from the versioning system commit note could be a false positive). After that, we used the issue type field to classify the issue and distinguish bug fixes from other issue types (*e.g.*, enhancements). Finally, we only considered bugs having *Closed* status and *Fixed* resolution. Basically, we restricted our attention to (i) issues that were related to bugs as we used them as a measure of fault-proneness, and (ii) issues that were neither duplicate reports nor false alarms.

Once collected the set of bugs fixed in the change history of each system, we used the SZZ algorithm [45] to identify when each fixed bug was introduced. The SZZ algorithm relies on the annotation/blame feature of versioning systems. In essence, given a bug-fix identified by the bug ID, k, the approach works as follows:

1) For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).
2) starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of *Git* is used to identify the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [46]. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

By adopting the process described above we are able to approximate the periods of time where each class of the subject systems was affected by one or more bugs (*i.e.*, was a buggy class). In particular, given a bug-fix BF_k performed on a class c_i, we consider c_i buggy from the date in which the bug fixed in BF_k was introduced (as indicated by the SZZ algorithm) to the date in which BF_k (*i.e.*, the patch) was committed in the repository.

4.3 Metrics and Data Analysis

Once defined the oracle and obtained the predicted buggy classes for every three-month period, we answer $\mathbf{RQ}_1$ by using three widely-adopted metrics, namely accuracy, precision and recall [38]:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5}$$

$$precision = \frac{TP}{TP + FP} \tag{6}$$

$$recall = \frac{TP}{TP + FN} \tag{7}$$

where TP is the number of classes containing bugs that are correctly classified as bug-prone; TN denotes the number of bug-free classes classified as non bug-prone classes; FP and FN measure the number of classes for which a prediction model fails to identify bug-prone classes by declaring bug-free classes as bug-prone (FP) or identifying actually buggy classes as non buggy ones (FN). As an aggregate indicator of precision and recall, we also report the F-measure, defined as the harmonic mean of precision and recall:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall} \tag{8}$$

Finally, we also report the Area Under the Curve (AUC) obtained by the prediction model. The AUC quantifies the overall ability of a prediction model to discriminate between buggy and non-buggy classes. The closer the AUC to 1, the higher the ability of the classifier to discriminate classes affected and not by a bug. On the other hand, the closer the AUC to 0.5, the lower the accuracy of the classifier. To compare the performances obtained by DCBM with the competitive techniques, we performed the bug prediction using the four baseline models BCCM, DM, MAF, and CM on the same systems and the same periods on which we ran DCBM.

To answer $\mathbf{RQ}_2$, we analyzed the orthogonality of the different measures used by the five experimented bug prediction models using Principal Component Analysis (PCA). PCA is a statistical technique able to identify various orthogonal dimensions (principal components) from a set of data. It can be used to evaluate the contribution of each variable to the identified components. Through the analysis of the principal components and the contributions (scores) of each predictor to such components, it is possible to understand whether different predictors contribute to the same principal components. Two models are complementary if the predictors they exploit contribute to capture different principal components. Hence, the analysis of the principal components provides insights on the complementarity between models.

Such an analysis is necessary to assess whether the exploited predictors assign the same bug-proneness to the same set of classes.

However, PCA does not tell the whole story. Indeed, using PCA it is not possible to identify to what extent a prediction model complements another and *vice versa*. This is the reason why we complemented the PCA by analyzing the overlap of the five prediction models. Specifically, given two prediction models m_i and m_j, we computed:

$$corr_{m_i \cap m_j} = \frac{|corr_{m_i} \cap corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \quad (9)$$

$$corr_{m_i \setminus m_j} = \frac{|corr_{m_i} \setminus corr_{m_j}|}{|corr_{m_i} \cup corr_{m_j}|}\% \quad (10)$$

where $corr_{m_i}$ represents the set of bug-prone classes correctly classified by the prediction model m_i, $corr_{m_i \cap m_j}$ measures the overlap between the sets of true positives correctly identified by both models m_i and m_j, $corr_{m_i \setminus m_j}$ measures the percentage of bug-prone classes correctly classified by m_i only and missed by m_j. Clearly, the overlap metrics are computed by considering each combination of the five experimented detection techniques (e.g., we compute $corr_{BCCM \cap DM}$, $corr_{BCCM \cap DCBM}$, $corr_{BCCM \cap CM}$, $corr_{DM \cap DCBM}$, etc.). In addition, given the five experimented prediction models m_i, m_j, m_k, m_p, m_z, we computed:

$$corr_{m_i \setminus (m_j \cup m_k \cup m_p \cup m_z)} = \frac{|corr_{m_i} \setminus (corr_{m_j} \cup corr_{m_k} \cup corr_{m_p} \cup corr_{m_z})|}{|corr_{m_i} \cup corr_{m_j} \cup corr_{m_k} \cup corr_{m_p} \cup corr_{m_z}|}\% \quad (11)$$

that represents the percentage of bug-prone classes correctly identified only by the prediction model m_i. In the paper, we discuss the results obtained when analyzing the complementarity between our model and the baseline ones. The other results concerning the complementarity between the baseline approaches are available in our online appendix [25].

Finally, to answer $\mathbf{RQ}_3$ we build and assess the performances of a "hybrid" bug prediction model exploiting different combinations of the predictors used by the five experimented models (*i.e.*, DCBM, BCCM, DM, MAF, and CM). Firstly, we assess the boost in performances (if any) provided by our scattering metrics when plugged-in the four competitive models, similarly to what has been done by Bird *et al.* [20], who explained the relationship between ownership metrics and bugs building regression models in which the metrics are added incrementally in order to evaluate their impact on increasing/decreasing the likelihood of developers to introduce bugs.

Then, we create a "comprehensive baseline model" featuring all predictors exploited by the four competitive models and again, we assess the possible boost in performances provided by our two scattering metrics when added to such a comprehensive model. Clearly, simply combining together the predictors used by the five models could lead to sub-optimal results, due for example to model overfitting.

Thus, we also investigate the subset of predictors actually leading to the best prediction accuracy. To this aim, we use the wrapper approach proposed by Kohavi and John [47]. Given a training set built using all the features available, the approach systematically exercises all the possible subsets of features against a test set, thus assessing their accuracy. Also in this case we used the Majority Decision Table [40] as machine learner.

In our study, we considered as training set the penultimate three-month period of each subject system, and as test set the last three-month period of each system. Note that this analysis has not been run on the whole change history of the software systems due to its high computational cost. Indeed, experimenting all possible combinations of the eleven predictors means the run of 2,036 different prediction models across each of the 26 systems (52,936 overall runs). This required approximately eight weeks on four Linux laptops having two dual-core 3.10 GHz CPU and 4 Gb of RAM.

Once obtained all the accuracy metrics for each combination, we analyzed these data in two steps. Firstly, we plot the distribution of the average F-measure obtained by the 2,036 different combinations over the 26 software systems. Then we discuss the performances of the top five configurations comparing the results with the ones achieved by (i) each of the five experimented models, (ii) the models built plugging-in the scattering metrics as additional features in the four baseline models, and (iii) the comprehensive prediction models that include all the metrics exploited by the four baseline models plus our scattering metrics.

5 Analysis of the Results

In this section we discuss the results achieved aiming at answering the formulated research questions.

5.1 $\mathbf{RQ}_1$: On the Performances of DCBM and Its Comparison with the Baseline Techniques

Table 5 reports the results—in terms of AUC-ROC, accuracy, precision, recall, and F-measure—achieved by the five experimented bug prediction models, *i.e.*, our model, exploiting the developer's scattering metrics (DCBM), the BCCM proposed by Hassan [8], a prediction model that uses as predictor the number of developers that work on a code component (DM) [9], [10], the prediction model based on the degree to which a module receives focused attention by developers (MAF) [22], and a prediction model exploiting product metrics capturing size, cohesion, coupling, and complexity of code components (CM) [1].

The achieved results indicate that the proposed prediction model (*i.e.*, DCBM) ensures better prediction accuracy as compared to the competitive techniques. Indeed, the area under the ROC curve of DCBM ranges between 62% and 91%, outperforming the competitive models.

TABLE 5
AUC-ROC, Accuracy, Precision, Recall, and F-Measure of the five bug prediction models

System	DCBM					DM					BCCM				
	AUC-ROC	Accuracy	Precision	Recall	F-measure	AUC-ROC	Accuracy	Precision	Recall	F-measure	AUC-ROC	Accuracy	Precision	Recall	F-measure
AMQ	83%	53%	42%	53%	47%	58%	24%	18%	19%	19%	61%	52%	33%	49%	39%
Ant	88%	69%	66%	72%	69%	69%	26%	28%	37%	31%	67%	63%	67%	68%	68%
Aries	86%	56%	51%	54%	52%	65%	23%	23%	25%	24%	58%	50%	34%	45%	39%
Camel	81%	55%	51%	55%	53%	51%	27%	18%	28%	22%	50%	39%	39%	46%	42%
CFX	79%	94%	88%	94%	91%	54%	25%	19%	25%	21%	71%	79%	86%	84%	85%
Drill	63%	53%	45%	48%	46%	58%	23%	22%	39%	28%	52%	39%	14%	25%	18%
Falcon	75%	98%	96%	98%	97%	50%	25%	20%	21%	21%	75%	89%	86%	90%	88%
Felix	88%	70%	69%	67%	68%	50%	25%	17%	30%	22%	59%	61%	60%	65%	63%
JMeter	91%	77%	72%	68%	70%	50%	29%	24%	53%	33%	69%	65%	65%	63%	64%
JS2	62%	87%	83%	86%	84%	50%	26%	22%	17%	19%	58%	81%	70%	74%	72%
Log4j	89%	71%	62%	66%	64%	50%	19%	13%	26%	17%	52%	43%	36%	78%	49%
Lucene	77%	84%	79%	83%	81%	54%	27%	22%	30%	26%	63%	72%	61%	86%	71%
Oak	67%	97%	95%	97%	96%	52%	27%	15%	29%	19%	66%	95%	92%	80%	86%
OpenEJB	82%	98%	97%	98%	98%	50%	22%	25%	20%	22%	78%	95%	81%	91%	85%
OpenJPA	83%	79%	71%	77%	74%	51%	20%	20%	38%	26%	78%	72%	61%	68%	64%
Pig	79%	89%	79%	89%	84%	50%	22%	21%	37%	27%	73%	71%	64%	75%	69%
Pivot	78%	86%	75%	86%	80%	53%	26%	19%	24%	21%	68%	69%	71%	79%	75%
Poi	87%	68%	88%	59%	71%	50%	25%	34%	16%	22%	66%	60%	74%	49%	59%
Ranger	77%	95%	90%	95%	93%	50%	28%	18%	19%	19%	76%	92%	83%	91%	87%
Shindig	73%	66%	50%	65%	56%	50%	24%	23%	23%	23%	58%	58%	43%	61%	50%
Sling	62%	85%	76%	84%	80%	57%	21%	17%	18%	18%	61%	80%	62%	68%	65%
Sqoop	78%	98%	96%	98%	97%	55%	26%	19%	32%	23%	77%	97%	90%	89%	90%
Sshd	86%	70%	59%	70%	64%	55%	24%	19%	36%	25%	69%	52%	49%	54%	52%
Synapse	67%	62%	50%	62%	56%	53%	23%	17%	24%	20%	64%	49%	48%	56%	52%
Whirr	76%	98%	95%	98%	97%	52%	26%	20%	24%	21%	74%	96%	84%	88%	86%
Xerces-J	83%	94%	94%	88%	91%	52%	49%	28%	35%	31%	71%	74%	59%	80%	68%

System	CM					MAF				
	AUC-ROC	Accuracy	Precision	Recall	F-measure	AUC-ROC	Accuracy	Precision	Recall	F-measure
AMQ	55%	43%	37%	41%	39%	56%	56%	38%	45%	41%
Ant	58%	38%	28%	33%	30%	60%	59%	60%	62%	61%
Aries	56%	38%	28%	33%	30%	51%	45%	30%	43%	35%
Camel	41%	42%	44%	41%	42%	50%	38%	35%	38%	36%
CFX	53%	52%	55%	46%	50%	76%	75%	82%	73%	77%
Drill	50%	34%	26%	32%	29%	52%	32%	22%	29%	25%
Falcon	51%	52%	45%	54%	49%	71%	81%	70%	81%	75%
Felix	53%	55%	53%	51%	52%	67%	56%	62%	65%	63%
JMeter	50%	43%	44%	43%	43%	68%	58%	61%	59%	60%
JS2	50%	43%	44%	43%	43%	62%	80%	72%	78%	75%
Log4j	50%	35%	38%	31%	34%	52%	51%	44%	58%	52%
Lucene	50%	35%	38%	31%	34%	65%	66%	66%	76%	70%
Oak	52%	46%	54%	55%	54%	64%	88%	89%	78%	83%
OpenEJB	61%	62%	66%	57%	61%	80%	78%	77%	79%	78%
OpenJPA	51%	55%	59%	45%	51%	67%	70%	57%	67%	62%
Pig	57%	62%	58%	52%	55%	69%	68%	62%	68%	65%
Pivot	50%	41%	47%	40%	43%	66%	64%	65%	69%	67%
Poi	58%	65%	61%	65%	63%	61%	55%	58%	56%	57%
Ranger	60%	65%	61%	65%	63%	76%	81%	77%	82%	79%
Shindig	52%	48%	36%	46%	40%	54%	55%	39%	59%	47%
Sling	55%	38%	35%	41%	38%	61%	76%	59%	63%	61%
Sqoop	53%	59%	59%	64%	61%	78%	92%	89%	84%	87%
Sshd	50%	28%	26%	31%	28%	67%	48%	46%	52%	49%
Synapse	56%	43%	47%	52%	49%	61%	47%	47%	53%	50%
Whirr	54%	61%	55%	63%	59%	69%	82%	82%	82%	82%
Xerces-J	58%	61%	55%	63%	59%	65%	71%	68%	75%	72%

In particular, the Developer Model achieves an AUC between 50% and 69%, the Basic Code Change Model between 50% and 78%, the MAF model between 50% and 78%, and the CM model between 41% and 61%. Also in terms of accuracy, precision and recall (and, consequently, of F-measure) DCBM achieves better results. In particular, across all the different object systems, DCBM achieves a higher F-measure with respect to DM (mean=+53.7%), BCCM (mean=+10.3%), MAF (mean=+13.3%), and CM (mean=+29.3%). The higher values achieved for precision and recall indicates that DCBM provides less false positives (*i.e.*, non-buggy classes indicated as buggy ones) while also being able to identify more classes actually affected by a bug as compared to the competitive models. Moreover, when considering the AUC, we observed that DCBM reaches higher values with respect the competitive bug prediction approaches. This result highlights how the proposed model performs better in discriminating between buggy and non-buggy classes.

Interesting is the case of `Xerces-J` where DCBM is able to identify buggy classes with 94% of accuracy (see Table 5), as compared to the 74% achieved by BCCM, 49% of DM, 71% of MAF, and 59% of CM. We looked into this project to understand the reasons behind such a strong result. We found that the `Xerces-J`'s buggy classes are often modified by few developers that, on average, perform a small number of changes on them. As an example, the class `XSSimpleTypeDecl` of the package `org.apache.xerces.impl.dv.xs` has been modified only twice between May 2008 and July 2008 (one of the three-month periods considered in our study) by two developers. However, the sum of their structural and semantic scattering in that period was very high (*161* and *1,932*, respectively). It is worth noting that if a low number of developers work on a file, they have higher chances to be considered as the owner of that file. This means that, in the case of the MAF model, the probability that the class is bug-prone decreases. At the same time, models based on the change entropy (BCCM) or on the number of developers modifying a class (DM) experience difficulties in identifying this class as buggy due to the low number of changes it underwent and to the low number of involved developers, respectively. Conversely, our model does not suffer of such a limitation thanks to the exploited developers' scattering information.

Finally, the CM model relying on product metrics fails in the prediction since the class has code metrics comparable with the average metrics of the system (*e.g.*, the CBO of the class 12, while the average CBO of the system is 14).

Looking at the other prediction models, we can observe that the model based only on the number of developers working on a code component never achieves an accuracy higher than 49%. This result confirms what previously demonstrated by Ostrand *et al.* [10], [9] on the limited impact of individual developer data on bug prediction.

Regarding the other models, we observe that the information about the ownership of a class as well as the code metrics and the entropy of changes have a stronger predictive power compared to number of developers. However, they still exhibit a lower prediction accuracy with respect to what allowed by the developer scattering information.

In particular, we observed that the MAF model has good performances when it is adopted on well-modularized systems, *i.e.*, systems grouping in the same package classes implementing related responsibilities. Indeed, MAF achieved the highest accuracy on the `Apache CFX`, `Apache OpenEJB`, and `Apache Sqoop` systems, where the average modularization quality (MQ) [48] is of 0.84, 0.79, and 0.88, respectively. The reason behind this result is that a high modularization quality often correspond to a good distribution of developers activities. For instance, the average number of developers per package working on `Apache CFX` is 5. As a consequence, the focus of developers on specific code entities is high. The same happens on `Apache OpenEJB` and `Apache Sqoop`, where the average number of developers per package is 3 and 7, respectively. However, even if the developers mainly focus their attention on few packages, in some cases they also apply changes to classes contained in other packages, increasing their chances of introducing bugs. This is the reason why our prediction model still continue to work better in such cases. A good example is the one of the class `HBaseImportJob`, contained in the package `org.apache.sqoop.mapreduce` of the project `Apache Sqoop`. Only two developers worked on this class over the time period between July 2013 and September 2013, however the same developers have been involved in the maintenance of the class `HiveImport` of the package `com.cloudera.sqoop.hive`. Even if the two classes shared the goal to import data from other projects into `Sqoop`, they implement significantly different mechanisms for importing data. This results in a higher proneness of introducing bugs. The sum of the structural and semantic scattering in that period for the two developers reached *86* and *92*, respectively, causing the correct prediction of the buggy file for our model, and an error in the prediction of the MAF model.

The BCCM [8] often achieves a good prediction accuracy. This is due to the higher change-proneness of components being affected by bugs. As an example, in the `JS2` project, the class `PortalAdministrationImpl` of the package `org.apache.jetspeed.administration` has been modified 19 times between January and March 2010. Such a high change frequency led to the introduction of a bug. However, not always such a conjecture is valid. Let us consider the *Apache Aries* project, in which BCCM obtained a low accuracy (recall=45%, precision=34%). Here we found several classes with high change-proneness that were not subject to any bug. For instance,

TABLE 6
Wilcoxon's t-test p-values of the hypothesis F-Measure achieved by DCBM $>$ 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 - CM	$<$ **0.01**	0.81	large
DCBM - BCCM	0.07	0.29	small
DCBM - DM	$<$ **0.01**	0.96	large
DCBM - MAF	$<$ **0.01**	0.44	medium

the class `AriesApplicationResolver` of the package `org.apache.aries.application.managament` has been changed 27 times between November 2011 and January 2012.

It was the class with the higher change-proneness in that time period, but this never led to the introduction of a bug. It is worth noting that all the changes to the class were applied by only one developer.

The model based on structural code metrics (CM) obtains fluctuating performance, with quite low F-measure achieved on some of the systems, like the `Sshd` project (28%). Looking more in depth into such results, we observed that the structural metrics achieve good performances in systems where the developers tend to repeatedly perform evolution activities to the same subset of classes. Such a subset of classes generally centralizes the system behavior, is composed of complex classes, and exhibits a high fault-proneness. As an example, in the `AMQ` project the class `activecluster.impl.StateServiceImpl` controls the state of the services provided by the system and it experienced five changes during the time period between September 2009 and November 2009. In this period, developers heavily worked on this class increasing its size from 40 to 265 lines of code. This sudden growth of the class size resulted in the introduction of a bug, correctly predicted by the CM model.

We also statistically compare the F-measure achieved by the five experimented prediction models. To this aim, we exploited the Mann-Whitney test [49] (results are intended as statistically significant at $\alpha = 0.05$). We also estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [50] 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 $|d| \geq 0.474$ [50]. Table 6 reports the results of this analysis. The proposed DCBM model obtains a significant higher F-measure with respect to the other baselines (p-value$<$0.05), with the only exception of the model proposed by Hassan [8], for which the p-value is partially significant (p-value=0.07). At the same time, the magnitude of the the differences is large in the comparison with the model proposed by Ostrand *et al.* [9] and the one based on product metrics [24], medium in the comparison with the model based on the Posnett *et al.* metric [22], and small when our model is compared with the model based on the entropy of changes [8].

Summary for RQ$_1$. Our approach showed quite high accuracy in identifying buggy classes. Among the 26 object systems its accuracy ranges between 53% and 98%, while the F-measure between 47% and 98%. Moreover, DCBM performs better than the baseline approaches, demonstrating its superiority in correctly predicting buggy classes.

5.2 RQ$_2$: On the Complementarity between DCBM and Baseline Techniques

Table 7 reports the results of the Principal Component Analysis (PCA), aimed at investigating the complementarity between the predictors exploited by the different models. The different columns (PC1 to PC11) represent the components identified by the PCA as those describing the phenomenon of interest (in our case, bug-proneness). The first row (*i.e.*, the proportion of variance) indicates on a scale between zero and one how much each component contributes to the phenomenon description (the higher the proportion of variance, the higher the component's contribution). The identified components are sorted on the basis of their "importance" in describing the phenomenon (*e.g.*, the PC1 in Table 7 is the most important, capturing 39% of the phenomenon as compared to the 2% brought by PC11). Finally, the values reported at row i and column j indicate how much the predictor i contributes in capturing the PC j (*e.g.*, structural scattering captures 69% of PC1). The *structural scattering* predictor is mostly orthogonal with respect to the other ten, since it is the one capturing most of PC1, the most important component. As for the other predictors, the semantic scattering and the change entropy information seem to be quite related by capturing the same components (*i.e.*, PC2 and PC3), while the MAF predictor is the one better capturing PC4 and PC5. The number of developers is only able to partially capture PC5, while the product metrics are the most important to capture the remaining components (PC6 to PC11). From these results, we can firstly conclude that the information captured by our predictors is strongly orthogonal with respect to the competitive ones. Secondly, we also observe a high complementarity between the MAF predictor and the others, while the predictor based on the number of developers working on a code component only partially capture the phenomenon, demonstrating again its limited impact in the context of bug prediction. Finally, the code metrics capture portions of the phenomenon that none of the other (process) metrics is able to capture. Such results highlight the possibility to achieve even better bug prediction models by combining predictors capturing orthogonal information (we investigate this possibility in **RQ**$_3$).

As a next step toward understanding the complementarity of the five prediction models, Tables 8, 9, 10, and

TABLE 7
Results achieved applying the Principal Component Analysis

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
Proportion of Variance	0.39	0.16	0.11	0.10	0.06	0.05	0.03	0.03	0.03	0.02	0.02
Cumulative Variance	0.39	0.55	0.66	0.76	0.82	0.87	0.90	0.92	0.95	0.97	1.00
Structural scattering predictor	**0.69**	-	-	0.08	0.04	-	-	-	-	-	-
Semantic scattering predictor	-	**0.51**	0.33	0.16	0.03	-	-	-	-	-	-
Change entropy	0.07	0.34	**0.45**	0.25	0.11	0.22	-	0.01	-	-	-
Number of Developers	-	-	0.05	0.02	0.29	-	0.04	0.05	0.01	-	0.07
MAF	0.04	0.11	-	**0.38**	**0.45**	-	0.21	0.04	0.06	-	0.1
LOC	0.04	-	0.01	-	0.03	0.07	0.18	0.21	0.11	0.09	**0.33**
CBO	0.1	0.04	0.05	0.07	-	**0.56**	0.2	**0.33**	0.21	**0.44**	0.12
LCOM	0.01	-	0.04	-	0.01	-	**0.24**	0.1	0.06	0.09	0.05
NOM	0.03	-	0.01	0.01	-	0.11	-	0.12	**0.43**	0.22	0.1
RFC	0.01	-	0.04	0.01	0.03	-	0.13	0.06	0.12	0.1	0.09
WMC	0.01	-	0.02	0.02	0.01	0.04	-	0.08	-	0.06	0.14

TABLE 8
Overlap analysis between DCBM and DM

System	DCBM ∩ DM%	DCBM \ DM%	DM \ DCBM%
AMQ	14	81	5
Ant	9	74	17
Aries	12	65	23
Camel	16	67	17
CXF	12	66	22
Drill	27	72	1
Falcon	12	84	4
Felix	14	65	21
JMeter	8	89	3
JS2	22	75	3
Log4j	13	75	12
Lucene	18	75	7
Oak	19	81	0
OpenEJB	17	80	3
OpenJPA	22	71	7
Pig	16	74	10
Pivot	18	80	2
Poi	11	72	17
Ranger	11	76	13
Shindig	20	61	18
Sling	16	62	21
Sqoop	19	71	10
Sshd	22	64	14
Synapse	12	79	9
Whirr	19	66	15
Xerces	32	55	13
Overall	**14**	**73**	**13**

11 report the overlap metrics computed between DCBM-DM, DCBM-BCCM, DCBM-CM, and DCBM-MAF, respectively.

In addition, Table 12 shows the percentage of buggy classes correctly identified only by each of the single bug prediction models (*e.g.*, identified by DCBM and not by DM, BCCM, CM and MAF). While in this paper we only discuss in details the overlap between our model and the alternative ones, the interested readers can find the analysis of the overlap among the other models in our online appendix [25].

Regarding the overlap between our predictor (DCBM) and the one built using the number of developers (DM), it is interesting to observe that there is high complementarity between the two models, with an overall 73% of buggy classes correctly identified only by our model, 13% only by DM, and 14% of instances correctly classified by both models. This result is consistent on all the object systems (see Table 8).

An example of buggy class identified only by our model is represented by `LuceneIndexer` contained in the package `org.apache.camel.component.lucene` of the `Apache Lucene` project. This class, between February 2012 and April 2012, has been modified by one developer that in the same time period worked on five other classes (the sum of structural and semantic scattering reached 138 and 192, respectively). This is the reason why our model correctly identified this class as buggy, while DM was not able to detect it due to the single developer who worked on the class. On the other side, DM was able to detect few instances of buggy classes not identified by DCBM. This generally happens when developers working on a code component apply less scattered changes over the other parts of the system, as in the case of the `Apache Sling` project, where the class `AbstractSlingRepository` of the package `org.apache.sling.jrc.base` was modified by four developers between March 2011 and May 2011. Such developers did not apply changes to other classes, thus having a low structural and semantic scattering. DM was instead able to correctly classify the class as buggy.

A similar trend is shown in Table 9, when analyzing the overlap between our model and BCCM. In this case, our model correctly classified 42% of buggy classes that are not identified by BCCM that is, however, able to capture 29% of buggy classes missed by our approach (the remaining 29% of buggy classes are correctly identified by both models). Such complementarity is mainly due to the fact that the change-proneness of a class does not always correctly suggest buggy classes, even if it is a good indicator. Often it is important to discriminate in which situations such changes are done. For example, the class `PropertyIndexLookup` of the package `oak.plugins.index.property` in the `Apache Oak` project, during the time period between April 2013 and June 2013, has been changed 4 times by 4 developers that worked, in the same period, on other 6 classes. This

caused a high scattering (both structural and semantic) for all the developers, and our model correctly marked the class as buggy.

Instead, BCCM did not classify the component as buggy since the number of changes applied on it is not high enough to allow the model to predict a bug. However, the model proposed by Hassan [8] is able to capture several buggy files that our model does not identify. For example, in the `Apache Pig` project the class `SenderHome` contained in the package `com.panacya.platform.service.bus.sender` experienced 27 changes between December 2011 and February 2012. Such changes were made by two developers that touched a limited number of related classes of the same package. Indeed, the sum of structural and semantic scattering was quite low (13 and 9, respectively) thus not allowing our model to classify the class as buggy. Instead, in this case the number of changes represent a good predictor.

Regarding the overlap between our model and the code metrics-based model (Table 10), also in this case the set of code components correctly predicted by both the models represents only a small percentage (13% on average). This means that the two models are able to predict the bug-proneness of different code components. Moreover, the DCBM model captures 78% of buggy classes missed by the code metrics model that is able to correctly predict 9% of code components missed by our model. For example, the DCBM model is able to correctly classify the `pivot.serialization.JSONSerializer` class of the `Apache Pivot` project, having low (good) values of size, complexity, and coupling, but modified by four developers in the quarter going from January 2013 to March 2013.

As for the overlap between MAF and our model, DCBM was able to capture 45% of buggy classes not identified by MAF. On the other hand, MAF correctly captured 29% of buggy classes missed by DCBM, while 26% of the buggy classes were correctly classified by both models. An example of class correctly classified by DCBM and missed by MAF can be found in the package `org.apache.drill.common.config` of the `Apache Drill` project, where the class `DrillConfig` was changed by three developers during the time period between November 2014 and January 2015. Such developers mainly worked on this and other classes of the same package (they can be considered as owners of the `DrillConfig` class), but they also applied changes to components structurally distant from it. For this reason, the sum of structural and semantic scattering increased and our model was able to correctly classify `DrillConfig` as buggy. On the other hand, an example of class correctly classified by MAF and missed by DCBM is `LogManager` of the package `org.apache.log4j` from the `Log4j` project. Here the two developers working on the component between March 2006 and May 2006 applied several changes to this class, as well as related classes belonging to different packages. Such related updates decreased the semantic scattering accumulated by developers.

TABLE 9
Overlap Analysis between DCBM and BCCM

System	DCBM ∩ BCCM %	DCBM \ BCCM %	BCCM \ DCBM %
AMQ	23	32	45
Ant	39	37	24
Aries	24	39	37
Camel	19	43	38
CXF	20	44	36
Drill	27	47	26
Falcon	34	40	26
Felix	29	38	34
JMeter	28	45	27
JS2	21	40	39
Log4j	16	67	17
Lucene	16	45	39
Oak	29	37	34
OpenEJB	36	35	28
OpenJPA	19	36	45
Pig	31	39	30
Pivot	34	46	20
Poi	37	33	30
Ranger	40	44	16
Shindig	31	33	36
Sling	16	31	53
Sqoop	32	49	19
Sshd	18	36	46
Synapse	20	31	49
Whirr	40	48	12
Xerces	22	43	35
Overall	**29**	**42**	**29**

Thus, DCBM did not classify the instance as buggy, while MAF correctly detect less focused attention on the class and marked the class as buggy.

Finally, looking at Table 12, we can see that our approach identifies 43% of buggy classes missed by the other four techniques, as compared to 24% of BCCM, 8% of DM, 18% of MAF, and 7% of CM. This confirms that (i) our model captures something missed by the competitive models, and (ii) by combining our model with BCCM/DM/MAF/CM ($\mathbf{RQ}_3$) we could further improve the detection accuracy of our technique. An example of a buggy class detected only by DCBM can be found in the *Apache Ant* system. The class `Exit` belonging to the package `org.apache.tools.ant.taskdefs` has been modified just once by a single developer in the time period going from January 2004 to April 2004. However, the sum of the structural and semantic scattering in that period was very high for the involved developer (*461.61* and *5,603.19*, respectively), who modified a total of 38 classes spread over 6 subsystems. In the considered time period the DM does not identify `Exit` as buggy given the single developer who worked on it, and the BCCM fails too due to the single change `Exit` underwent between January and April 2004. Similarly, the CM model is not able to identify this class as buggy due to its low complexity and small size.

Conversely, an example of buggy class not detected by DCBM is represented by the class

TABLE 10
Overlap Analysis between DCBM and CM

System	DCBM ∩ CM %	DCBM \ CM %	CM \ DCBM %
AMQ	10	65	25
Ant	8	68	24
Aries	12	58	30
Camel	22	53	25
CXF	7	84	9
Drill	5	73	22
Falcon	18	79	3
Felix	15	68	17
JMeter	15	78	7
JS2	6	88	6
Log4j	11	87	2
Lucene	11	77	12
Oak	14	83	3
OpenEJB	6	88	6
OpenJPA	18	67	15
Pig	16	75	9
Pivot	13	78	9
Poi	14	75	11
Ranger	21	75	5
Shindig	7	82	11
Sling	7	82	11
Sqoop	9	86	5
Sshd	15	72	13
Synapse	18	63	19
Whirr	8	85	7
Xerces2-j	39	59	2
Overall	**13**	**78**	**9**

TABLE 11
Overlap Analysis between DCBM and MAF

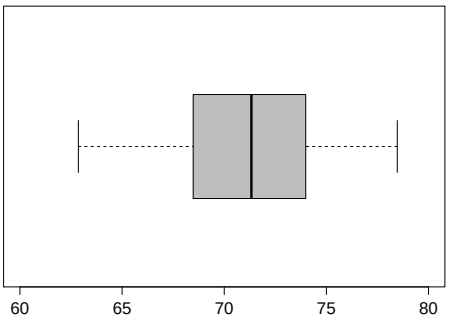
System	DCBM ∩ MAF %	DCBM \ MAF %	MAF \ DCBM %
AMQ	24	47	29
Ant	23	46	31
Aries	32	47	21
Camel	19	51	29
CXF	20	50	31
Drill	23	43	34
Falcon	19	42	39
Felix	24	56	20
JMeter	25	53	22
JS2	23	40	37
Log4j	26	45	30
Lucene	31	41	28
Oak	26	46	28
OpenEJB	28	49	24
OpenJPA	22	46	32
Pig	25	44	31
Pivot	26	55	19
Poi	27	44	29
Ranger	27	41	32
Shindig	27	46	27
Sling	21	37	42
Sqoop	33	43	24
Sshd	19	40	41
Synapse	21	56	23
Whirr	27	53	20
Xerces2-j	30	42	28
Overall	**26**	**45**	**29**

`AbstractEntityManager` belonging to the package `org.apache.ivory.resource` of the *Apache Falcon* project.

Here we found 49 changes occurring on the class on the time period going from October 2012 to January 2013 applied by two developers. The sum of the structural and semantic scattering metrics in this time period was very low for both the involved developers (*14.77* is the sum for the first developer, *18.19* for the second one). Indeed, the developers in that period only apply changes to another subsystem. This is the reason why our prediction model is not able to mark this class as buggy. On the other hand, BCCM and MAF prediction models successfully identify the buggyness of the class exploiting the information about the number of changes and ownership, respectively. DM fails due to the low number of developers involved in the change process of the class. Finally, CM is not able to correctly classify this class as buggy because of the low complexity of the class.

Summary for RQ$_2$. The analysis of the complementarity between our approach and the four competitive techniques showed that the proposed scattering metrics are highly complementary with respect to the metrics exploited by the baseline approaches, paving the way to "hybrid" models combining multiple predictors.

Fig. 4. Boxplot of the average F-Measure achieved by the 2,036 combinations of predictors experimented in our study.



5.3 RQ$_3$: A "Hybrid" Prediction Model

Table 13 shows the results obtained while investigating the creation of a "hybrid" bug prediction model, exploiting a combination of predictors used by the five experimented models.

The top part of Table 13 (*i.e.*, Performances of each experimented model) reports the average performances—in terms of AUC-ROC, accuracy, precision, recall, and F-measure—achieved by each of the five experimented bug prediction models. As already discussed in the context of RQ$_1$, our DCBM model substantially outperforms the competitive ones. Such values only serve as a reference to better interpret the results of the different hybrid

TABLE 12
Overlap Analysis considering each Model independently

System	DCBM \ (BCCM ∪ DM ∪ CM ∪ MAF) %	BCCM \ (DCBM ∪ DM ∪ CM ∪ MAF) %	DM \ (DCBM ∪ BCCM ∪ CM ∪ MAF) %	MAF \ (CM ∪ DCBM ∪ BCCM ∪ DM) %	CM \ (DCBM ∪ BCCM ∪ CM ∪ DM) %
AMQ	44	24	9	17	6
Ant	40	25	8	20	7
Aries	41	22	10	19	8
Camel	39	21	6	22	12
CXF	45	25	9	14	7
Drill	44	25	8	18	5
Falcon	46	27	8	18	2
Felix	43	21	5	19	12
JMeter	42	23	7	17	11
JS2	45	26	10	15	4
Log4j	43	20	8	19	10
Lucene	44	23	8	20	5
Oak	39	26	9	19	7
OpenEJB	43	24	8	16	9
OpenJPA	41	26	9	18	6
Pig	44	25	9	20	2
Pivot	45	25	8	19	3
Poi	39	23	9	17	12
Ranger	48	19	10	14	9
Shindig	46	24	6	17	7
Sling	41	25	9	16	9
Sqoop	41	26	7	19	7
Sshd	44	22	10	19	5
Synapse	41	22	7	20	10
Whirr	40	23	8	18	11
Xerces	47	23	9	12	9
Overall	**43**	**24**	**8**	**18**	**7**

models we discuss in the following.

The second part of Table 13 (*i.e.*, Boost provided by our scattering metrics to each baseline model), reports the performances of the four competitive bug prediction models when augmented with our predictors.

The boost provided by our metrics is evident in all the baseline models. Such a boost goes from a minimum of +8% in terms of F-Measure (for the model based on change entropy) up to +49% for the model exploiting the number of developers as predictor. However, it is worth noting that the combined models do not seem to improve the performances of our DBCM model.

The third part of Table 13 (*i.e.*, Boost provided by our scattering metrics to a comprehensive baseline model) seems to tell a different story. In this case, we combined all predictors belonging to the four baseline models into a single, comprehensive, bug prediction model, and assessed its performances. Then, we added our scattering metrics to such a comprehensive baseline model and assessed again its performances. As it can be seen from Table 13, the performances of the two models (*i.e.*, the one with and the one without our scattering metrics) are almost the same (F-measure=71% for both of them). This suggests the absence of any type of impact (positive or negative) of our metrics on the model's performances, which is something unexpected considered the previously performed analyses.

Such a result might be due to the high number of predictor variables exploited by the model (eleven in this case), possibly causing model overfitting on the training sets with consequent bad performances on the test set. Again, the combination of predictors does no seem to improve the performances of our DBCM model. Thus, as explained in Section 4.3, to verify the possibility to build an effective hybrid model we investigated in an exhaustive way the combination of predictors that leads to the best prediction accuracy by using the wrapper approach proposed by Kohavi and John [47].

Figure 4 plots the average F-measure obtained by each of the 2,036 combinations of predictors experimented. The first thing that leaps to the eyes is the very high variability of performances obtained by the different combinations of predictors, ranging between a minimum of 62% and a maximum of 79% (mean=70%, median=71%). The bottom part of Table 13 (*i.e.*, Top-5 predictors combinations obtained from the wrapper selection algorithm) reports the performances of the top five predictors combinations. The best configuration, achieving an average F-Measure of 79% exploits as predictors the CBO coupling metric [1], the change entropy by Hassan [8], the structural and semantic scattering defined in this paper, and the module activity focus by Posnett et al. [22]. Such a configuration also exhibits a very high AUC (90%) and represents a substantial improvement in prediction accuracy over the best model used in isolation (*i.e.*, DCBM with an average F-Measure of 74% and an AUC=76%) as well as over the comprehensive model exploiting all the baselines' predictors

TABLE 13
RQ_3: Performances of "hybrid" prediction models

Performances of each experimented model	Avg. AUC-ROC	Avg. Accuracy	Avg. Precision	Avg. Recall	Avg. F-measure
DM	51	24	19	25	21
BCCM	63	70	61	69	64
CM	52	46	44	45	44
MAF	62	65	59	64	61
DCBM	76	77	72	77	74
Boost provided by our scattering metrics to each baseline model					
DM + Struct-scattering + Seman-scattering	78	71	73	68	70
BCCM + Struct-scattering + Seman-scattering	77	70	76	69	72
CM + Struct-scattering + Seman-scattering	76	70	73	70	71
MAF + Struct-scattering + Seman-scattering	77	70	73	70	71
Boost provided by our scattering metrics to a comprehensive baseline model					
# Developers, Entropy, LOC, CBO, LCOM, NOM, RFC, WMC, MAF	78	69	73	68	71
# Developers, Entropy, LOC, CBO, LCOM, NOM, RFC, WMC, MAF, Struct-scattering, Seman-scattering	76	71	72	71	71
Top-5 predictors combinations obtained from the wrapper selection algorithm					
CBO, Change Entropy, Struct-scattering, Seman-scattering, MAF	90	85	77	81	79
LOC, LCOM, Change Entropy, Seman-scattering, # Developers, MAF	78	72	77	77	77
LOC, NOM, WMC, Change Entropy, Struct-scattering	78	70	77	75	76
LOC, LCOM, NOM, Seman-scattering	77	70	75	75	75
LOC, CBO, LCOM, NOM, RFC, Struct-scattering, Seman-scattering	77	71	76	73	75

in combinations (+8% in terms of F-Measure). Such a result supports our conjecture that blindly combining predictors (as we did in the comprehensive model) could result in sub-optimal performances likely due to model overfitting.

Interestingly, the best combination of baselines' predictors (*i.e.*, all predictors from the four competitive models) obtained as result of the wrapper approach is composed of BCCM (i.e., entropy of changes), MAF, and the RFC and WMC metrics from the CM model, and achieves 70% in terms of F-Measure (9% less with respect to the best combination of predictors which also exploits our scattering metrics).

We also statistically compare the prediction accuracy obtained across the 26 subject systems by the best-performing "hybrid" configuration and the best performing model. Also in this case, we exploited the Mann-Whitney test [49] for this statistical test, as well as the Cliff's Delta [50] to estimate the magnitude of the measured differences. We observed a statistically significant difference (p-value=0.03) with a medium effect size ($d = 0.36$).

Looking at the predictors more frequently exploited in the five most accurate prediction models, we found that:

1) *Semantic-scattering, LOC*. Our semantic predictor and the LOC are present in 4 out of the 5 most accurate prediction models. This confirms the well-known bug prediction power of the size metrics (LOC) and suggests the importance for developers to work on semantically related code components in the context of a given maintenance/evolution activity.
2) *Change entropy, LCOM, Structural-scattering*. These predictors are present in 3 out of the 5 most accurate prediction models. This confirms that (i) the change entropy is a good predictor for buggy code components [8]), (ii) classes exhibiting low cohesion can be challenging to maintain for developers [1], and (iii) scattered changes performed across different subsystems can increase the chances of introducing bugs.

In general, the results of all our three research questions seem to confirm the observations made D'Ambros *et al.* [15]: no technique based on a single metric works better in all contexts. This is why the combination of multiple predictors can provide better results. We are confident that plugging other orthogonal predictors in the "hybrid" prediction model could further increase the prediction accuracy.

Summary for RQ_3. By combining the eleven predictors exploited by the five prediction models subject of our study it is possible to obtain a boost of prediction accuracy up to +5% with respect to the best performing model (*i.e.*, DCBM) and +9% with respect to the best combination of baseline predictors. Also, the top five "hybrid" prediction models include at least one of the predictors proposed in this work (*i.e.*, the structural and semantic scattering of changes) and the best model includes both.

6 Threats to Validity

This section describes the threats that can affect the validity our study. Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important type of threat for our study and it is related to:

- *Missing or wrong links between bug tracking systems and versioning systems [51]*: although not much can be done for missing links, as explained in the design we verified that links between commit notes and issues were correct;
- *Imprecision due to tangled code changes [52]*. We cannot exclude that some commits we identified as bug-fixes grouped together tangled code changes, of which just a subset represented the committed patch.

- *Imprecision in issue classification made by issue-tracking systems [19]*: while we cannot exclude misclassification of issues (*e.g.*, an enhancement classified as a bug), at least all the systems considered in our study used Bugzilla as issue tracking system, explicitly pointing to bugs in the issue type field;
- *Undocumented bugs present in the system*: while we relied on the issue tracker to identify the bugs fixed during the change history of the object systems, it is possible that undocumented bugs were present in some classes, leading to wrong classifications of buggy classes as "clean" ones.
- *Approximations due to identifying fix-inducing changes using the SZZ algorithm [45]*: at least we used heuristics to limit the number of false positives, for example excluding blank and comment lines from the set of fix-inducing changes.

Threats to *internal validity* concern external factors we did not consider that could affect the variables being investigated. We computed the developer's scattering measures by analyzing the developers' activity on a single software system. However, it is well known that, especially in open source communities and ecosystems, developers contribute to multiple projects in parallel [53]. This might negatively influence the "developer's scattering" assessment made by our metrics. Still, the results of our approach can only improve by considering more sophisticated ways of computing our metrics.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. The metrics used in order to evaluate our defect prediction approach (*i.e.*, accuracy, precision, recall, F-Measure, and AUC), are widely used in the evaluation of the performances of defect prediction techniques [15]. Moreover, we used appropriate statistical procedures, (*i.e.*, PCA [54]), and the computation of overlap metrics to study the orthogonality between our model and the competitive ones.

Since we had the necessity to exploit change-history information to compute the scattering metrics we proposed, the evaluation design adopted in our study is different from the k-fold cross validation [55] generally exploited while evaluating bug prediction techniques. 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 [8], although it might be penalizing when using product metrics, which are typically assessed using a ten-fold cross validation. Furthermore, although we selected a model exploiting a set of product metrics previously shown to be effective in the context of bug prediction [1], the poor performances of the CM model might be due to the fact that the model relies on too many predictors, resulting in a model overfitting. This conjecture is supported by the results achieved in the context of $\mathbf{RQ}_3$, where we found that the top five "hybrid" prediction models include only a subset of code metrics.

Threats to *external validity* concern the generalization of results. We analyzed 26 Apache systems from different application domains and with different characteristics (number of developers, size, number of classes, etc).

However, systems from different ecosystems should be analyzed to corroborate our findings.

7 Conclusion and Future Work

A lot of effort has been devoted in the last decade to analyze the influence of the development process on the likelihood of introducing bugs. Several empirical studies have been carried out to assess under which circumstances and during which coding activities developers tend to introduce bugs. In addition, bug prediction techniques built on top of process metrics have been proposed. However, changes in source code are made by developers that often work under stressing conditions due to the need of delivering their work as soon as possible.

The role of developer-related factors in the bug prediction field is still a partially explored area. This paper makes a further step ahead, by studying the role played by the *developer's scattering* in bug prediction. Specifically, we defined two measures that consider the amount of code components a developer modifies in a given time period and how these components are spread structurally (*structural scattering*) and in terms of the responsibilities they implement (*semantic scattering*). The defined measures have been evaluated as bug predictors in an empirical study performed on 26 open source systems. In particular, we built a prediction model exploiting our measures and compared its prediction accuracy with four baseline techniques exploiting process metrics as predictors. The achieved results showed the superiority of our model and its high level of complementarity with respect to the considered competitive techniques. We also built and experimented a "hybrid" prediction model on top of the eleven predictors exploited by the five competitive techniques. The achieved results show that (i) the "hybrid" is able to achieve a higher accuracy with respect to each of the five models taken in isolation, and (ii) the predictors proposed in this paper play a major role in the best performing "hybrid" prediction models.

Our future research agenda includes a deeper investigation of the factors causing scattering to developers, and negatively impacting their ability of dealing with code change tasks. We plan to reach such an objective by performing a large survey with industrial and open source developers. We also plan to apply our technique at different levels of granularity, to verify if we can point out buggy code components at a finer granularity level (*e.g.*, methods).

REFERENCES

[1] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, Oct 1996.

[2] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 10, pp. 897–910, 2005.

[3] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switchess," *Software Engineering, IEEE Transactions on*, vol. 22, no. 12, p. 886894, 1996.

[4] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 580–586. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062558

[5] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/PROMISE.2007.10

[6] A. N. Taghi M. Khoshgoftaar, Nishith Goel and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Software Reliability Engineering*. IEEE, 1996, pp. 364–371.

[7] J. S. M. Todd L. Graves, Alan F. Karr and H. P. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, no. 7, pp. 653–661, 2000.

[8] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*. Vancouver, Canada: IEEE Press, 2009, pp. 78–88.

[9] R. Bell, T. Ostrand, and E. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9178-4

[10] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 19:1–19:10. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868357

[11] R. Moser, W. Pedrycz, and G. Succi, "Analysis of the reliability of a subset of change metrics for defect prediction," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 309–311. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414063

[12] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?" in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, ser. Promise '11. New York, NY, USA: ACM, 2011, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2020390.2020392

[13] W. P. Raimund Moser and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *International Conference on Software Engineering (ICSE)*, ser. ICSE '08, 2008, pp. 181–190.

[14] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[15] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, p. 531577, 2012.

[16] J. Sliwerski, T. Zimmermann, and A. Zeller, "Don't program on fridays! how to locate fix-inducing changes," in *Proceedings of the 7th Workshop Software Reengineering*, May 2005.

[17] L. T. Jon Eyolfso and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 153–162.

[18] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 491–500.

[19] E. J. W. J. Sunghun Kim and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[20] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 4–14.

[21] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12, 2012, pp. 104–113.

[22] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 452–461.

[23] D. D. Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. D. Lucia, "On the role of developer's scattered changes in bug prediction," in *Proceedings of the 31st International Conference on Software Maintenance and Evolution, ICSME '15, Bremen, Germany*, 2015, pp. 241–250.

[24] V. Basili, G. Caldiera, and D. H. Rombach, *The Goal Question Metric Paradigm*. John Wiley and Sons, 1994.

[25] D. D. Nucci, F. Palomba, G. D. Rosa, G. Bavota, R. Oliveto, and A. D. Lucia. (2016) A developer centered bug prediction model - replication package - https://figshare.com/articles/A_Developer_Centered_Bug_Prediction_Model/3435299.

[26] W. M. Khaled El Emam and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, p. 6375, 2001.

[27] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, p. 297310, 2003.

[28] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," in *Proceedings of the 9th IEEE International Symposium on Software Metrics*. IEEE CS Press, 2003, pp. 338–349.

[29] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems," *Journal of Systems and Software*, vol. 83, no. 4, pp. 660–674, 2010.

[30] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 284–292.

[31] A. E. Hassan and R. C. Holt, "Studying the chaos of code development," in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[32] ——, "The top ten list: dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005*, ser. ICSM '05. IEEE Computer Society, 2005, pp. 263–272.

[33] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.

[34] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 309–318.

[35] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for bugs in all the right places," in *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 2006, pp. 61–72.

[36] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, June 1994.

[37] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[38] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[39] L. M. Y. Freund, "The alternating decision tree learning algorithm," in *Proceeding of the Sixteenth International Conference on Machine Learning*, 1999, pp. 124–133.

[40] R. Kohavi, "The power of decision tables," in *8th European Conference on Machine Learning*. Springer, 1995, pp. 174–189.

[41] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[42] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1961.

[43] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Eleventh Conference on Uncertainty in Artificial Intelligence*. San Mateo: Morgan Kaufmann, 1995, pp. 338–345.

[44] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: http://promisedata.googlecode.com

[45] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*. ACM, 2005.

[46] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 2001, pp. 13–22.

[47] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artif. Intell.*, vol. 97, no. 1-2, pp. 273–324, Dec. 1997. [Online]. Available: http://dx.doi.org/10.1016/S0004-3702(97)00043-X

[48] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proccedings of 6th International Workshop on Program Comprehension*. Ischia, Italy: IEEE CS Press, 1998.

[49] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

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

[51] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716

[52] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 121–130.

[53] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 280–289.

[54] I. Jolliffe, *Principal Component Analysis*. John Wiley & Sons, Ltd, 2005. [Online]. Available: http://dx.doi.org/10.1002/0470013192.bsa501

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