CLASS CHANGE PREDICTION BY INCORPORATING COMMUNITY SMELL: AN EMPIRICAL STUDY

Qingyuan Dou

Department of Computer Science and Technology, Shanghai Normal University, 100 Haisi Road, Fengxian District
Shanghai, 201418, China
Dqy060628@163.com

Junhua Chen

Department of Computer Science and Technology, Shanghai Normal University, 100 Haisi Road, Fengxian District
Shanghai, 201418, China
chenjh@shnu.edu.cn

Jianhua Gao

Department of Computer Science and Technology, Shanghai Normal University, 100 Haisi Road, Fengxian District
Shanghai, 201418, China
jhgao@shnu.edu.cn

Zijie Huang

Department of Computer Science and Technology, East China University of Science and Technology,130 Meilong Road, Xuhui District
Shanghai,200237,China
hzj@mail.ecust.edu.cn

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

To adapt to changing software requirements, developers need to maintain and modify software through code changes. Predicting change-prone code can help developers to reduce the cost of software maintenance in advance. Prior work confirmed code smell intensity is a reliable metric for predicting change-prone classes. Community smell is a derivation of the concept of code smell in open-source software development community, it refers to poor communication and collaboration problems among developers. We add community smell to existing change prediction models, and propose a software class change prediction model integrating process metrics, code smell intensity metrics, anti-pattern metrics, and community smell metrics, which takes into account the technicality and organisational of software development. Experimental results demonstrate that when Multilayer Perceptron is used to build a change prediction model, community smell improves the baseline model by 4.4% and 31.5% in terms of F-Measure and Recall. In addition, community smell improves baseline model performance to a greater extent in

terms of Recall and Precision than code smell-related information.

Keywords: Community Smell; Change Prediction; Code Smell; Empirical Software Engineering.

1. Introduction

During the maintenance and development of software, software systems are constantly being modified to match the growing needs of society to improve their performance or eliminate potential threats. The original software system becomes more complex and the program gradually suffers from design degradation, thus making the software system progressively less maintainable. Software maintenance is a technical and socially relevant activity for developers. Identifying change-prone classes in software systems as early as possible allows more maintenance resources to be allocated to them in advance, to improve software maintenance efficiency. Therefore, the prediction of change-prone classes is a crucial research content.

Change prediction could help arrange software quality assurance resources by identifying code components that are prone to change, helping developers plan preventive maintenance actions, and controlling code complexity. To achieve effective context-aware change prediction, researchers incorporate various information extracted from software artifacts. In terms of the goal and description of code changes, Vonken et al. [1] incorporated code refactoring, and Pascarella et al. [2] involved code review. In terms of the adaptivity of the model, Catolino et al. [3] considered developer-related factors to construct a model for predicting change-prone classes by defining developer-related factors. Kim et al. [4] and Soetens et al. [5] have studied the factors that cause code classes in software systems to be change-prone.

In the refactoring of software code, code smell is defined by Fowler [6] as suboptimal design and choices applied by programmers during the development of software projects. In earlier studies, Khomh et al. [7] and Palomba et al. [8] demonstrated that classes associated with code smell in software systems are more likely to be changed than other classes. Ambros et al. [9] confirmed in a study that classes affected by code smell had a higher tendency to change compared to classes not affected by code smell. Later, Catolino et al. [10] added the code smell intensity metric, which describes the degree of smell-related information, to the prediction model and found that the addition of the code smell intensity metric significantly improved the performance of the change prediction model. Current change prediction only examines source code or the change to the software by developers to predict software systems with change-prone classes without taking into account the organization and communication patterns between developers.

During the software development process, the accessibility of developers to excellent communication and exchange is crucial to ensure that the software is developed correctly. Tamburri et al. [11] defined a method for describing the organizational and social structure of software engineering and investigate the impact of developer organization and social structure on development problem solving and community

welfare. In comparison, Caglayan et al. [12] proposed a method for modeling with a collaborative approach among developers, which can assign issues arising from development to relevant developers. Tamburri et al. [13] defined a term called 'community smell' to explain social and organizational anti-pattern and suggest how these can be mitigated based on observations of the software development industry.

As software development is a technical and organizational activity, to address the lack of existing research which does not take into account the organizational structure and collaboration patterns between developers, we propose a change prediction model that adds community smells to existing change prediction models for change-prone classes.

The main contributions of this experiment include:

1. A new change prediction model was designed and evaluated based on the combination of Metrics used in the previous researcher's model. Based on the best performing baseline model from existing studies, we assess the contribution of the community smell metric to the baseline model for change prediction in our study. The results show that the community smell metric improves the performance of the baseline model for predicting software class changes, as well as the community smell metric has a significant impact on the prediction of change-prone classes.
2. The feature importance of community smells in the change prediction model is assessed. In the change prediction baseline model, we add three community smells and calculate the strength of information gain provided by different community smells to different models. Our results show that the information gain provided by the Radio Silence community smell to the baseline model performance is higher than that provided by both Organizational Silo and Missing Links community smells.
3. We assess the performance of a combined model of community smell, code smell intensity, and anti-pattern metrics in change prediction, in addition to empirically discussing the contribution of the combined smell metrics to the baseline model.

The rest of this paper is organized as follows. Section 2 discusses research findings on change prediction models and community smells. Section 3 examines and designs predictive models for software class changes; Section 4 reports the experimental results and analysis. Section 5 discusses the threats to the validity of empirical research. Section 6 concludes this experimental work and outlines the way forward.

2. Related Work

2.1. Definition and impact of community smell

Developers with different cultures and development habits are often required to join in the development of software systems, and Gren et al. [14] argues that social interaction among developers is often influenced by physical and cultural distance

and expertise. In research, suboptimal organization and poor interaction between developers are referred to as community smells. The nine community smells defined in the study by Tamburri et al. [13] are shown in Table 1, which explain the poor organizational structure and communication patterns among developers from different aspects.

Table 1. The name and description of community smell proposed by Tamburri

Name	Acronym	Description
Organizational Silo	O.Silo	A software development pattern in an organization where the developers collaboratewith others but do not communicate with in the analyzed communication channel.
Black Cloud	BC	This reflects an information overload due to lackof structured communications or cooperation governance.
Lone Wolf	LW	This smell appears in cases where the development community presents unsanctioned or defiant contributors who carry out their work with little consideration of their peers, their decisions and communication.
Radio Silence	RS	A pattern such as there is unique knowledge and information brokers toward different sub-communities.
Missing Links	ML	A pattern where there is a collaboration between developers and there is a lack of communication between collaborators.
Prima Donnas	PD	One or several members repeatedly display condescension and superiority and the developer has persistently divisive, uncooperative behaviour.
Architecture Hood	AH	Developers blaming architectural decisions for any technical issues that arise during development, leading to uncooperative behaviour across the development community.
Sharing Villainy	SV	The lack of knowledge exchange incentives or face-to-face meetings limits the value of developers in sharing their knowledgeand experience.
Organizational Skirmish	OS	Operations and development units are misaligned in their organizational culture, in their communication habits and in their expertise levels.

Developer-related metrics can provide more insight into the impact of personnel factors on the performance of change prediction models; for example, Ostrand et al. [15] studied the number of developers contributing to software modules in software system development, Nucci et al. [16] studied developer attention to software modules, and Calikli et al. [17] studied the level of confirmation bias of developers to software system development. In a recent study, Eken et al. [18] investigated personalised models for developers, training a dedicated model for each developer through the historical development activities of developers in a software system. Soltanifar et al. [19] provide a detailed study of the review activities of developers and the review process in which they are involved. Catolino et al. [20] propose ways to use community smell to reconfigure development techniques, for example by reconstructing development teams and developing appropriate communication

plans for software development team members. Palomba et al. [21] further investigated the relationship between community smell, socio-technical consistency and code smell intensity by investigating whether developers could perceive the relationship between code smell in projects and inter-organizational community smell, and performing fine-grained analysis of the dataset to measure the extent to which community smell influences code smell intensity, concluding that the presence of community smell leads to a maintainability to be reduced. Palomba's et al. [21] study of codes and community smell inspired this experiment to further investigate community smell and the extent to which community smell contributes to change prediction.

2.2. Significance of Class Change Forecasts and Influencing Factors

Software class change is a continuous change in the class level of the software due to the changing demand of the software users and the modifications made to the software system by the developers. Lehman's et al. [22] research points out that although software changes cannot be avoided, the prediction of changes allows software changes to be controlled by the developer. Predicting classes in a software system that have a propensity to change is critical for developers, in that software class change prediction can alert developers to when to refactor code and developers can plan preventive maintenance operations to reduce change costs before software class changes.

The change-prone classes are those classes in a software system that have a tendency to change, and research has demonstrated that factors that influence the occurrence of change in classes include design patterns, software size, code smell, class size and coupling metrics. Penta et al. [23] first investigated the relationship between design patterns and change-prone classes in software systems and found that classes designed by applying the three design patterns ADAPTER, ABSTRACT FACTORY and COMMAND in software systems were more prone to change compared to other classes.

In addition, Posnett et al. [24] showed by analysing the effect of pattern roles on propensity to change that software size appears to have a more significant impact than design patterns in changes to software systems. On the other hand, Khomh et al. [7] investigated how poor design patterns, prior to software development, influenced class change and bug propensity. Their study showed that classes with off-flavours are more susceptible to changes and bugs than other classes. In addition, systems that contain a lot of off-flavours may be more prone to change, on the other hand Kim et al. [4] pointed out that refactoring is a key activity to reduce the tendency of classes to change. Lindvall et al. [25] found that class size affects the propensity for class change and noted that developers tend to make more improvements to large classes during maintenance and evolution. Finally, Kavitha et al. [26] showed that the coupling metric is a relevant metric for estimating source code variability.

2.3. Class Change Prediction Methods

Within the system of knowledge of change prediction techniques, structural and process metrics provide a good description of the propensity of software system components to change. Specifically, Romano et al. [27] relied on code metrics to predict the change-prone so-called fat interfaces, i.e. Java interfaces with poor cohesion, while Eski et al. [28] proposed a model based on CK and QMOOD metrics to predict classes with a propensity to change. The potential usefulness of process metrics for change prediction is reported by Elish et al. [29], who defined a set of evolutionary metrics that describe the historical characteristics of software classes, for example, they defined metrics such as the date of birth of a class or the total number of class changes applied in the past. The results of Elish et al. [29] show that a predictive model based on process metrics can predict change classes in terms of class evolution. Girba et al. [30] make suggestions for subsequent code elements that have a propensity to change by summarising information from previous change histories. In a small-scale empirical study involving two systems, they observed that prior changes were effective in predicting future modifications.

Catolino et al. [3] empirically assessed the role of developer-related factors in change prediction. They investigated the performance of three developer-based prediction models that rely on (1) the entropy of the development process , (2) the number of developers working on a class as proposed by Bell et al. [31] and (3) the changing structure and semantic dispersion proposed by Nucci et al. [16], which showed that they can be more accurate than models based on structural or process metrics. In a recent study, Catolino et al. [10] added a metric indicating code smell intensity to a change prediction model and compared it with the anti-pattern and code smell metrics, which indicated that the code smell intensity metric is a good metric for change prediction.

Given that suboptimal organizational patterns and poor communication among developers can have an impact on software quality, we integrated community smell to the change prediction model to explore the impact of community smell on model performance.

3. Construction of Change Prediction Models

By combining structural, process, and developer-related metrics, the software class change prediction model identifies change-prone classes in software systems. Based on previous work, we integrate community smell as a feature metric in our software class change-prone prediction model, as shown in Figure 1.

3.1. Dataset Selection

The dataset we used is based on the JAVA open-source project dataset used in Catolino's [10] study. Some of the projects in Catolino's [10] dataset are early versions of projects completed between 2000 and 2008. Because there is insufficient

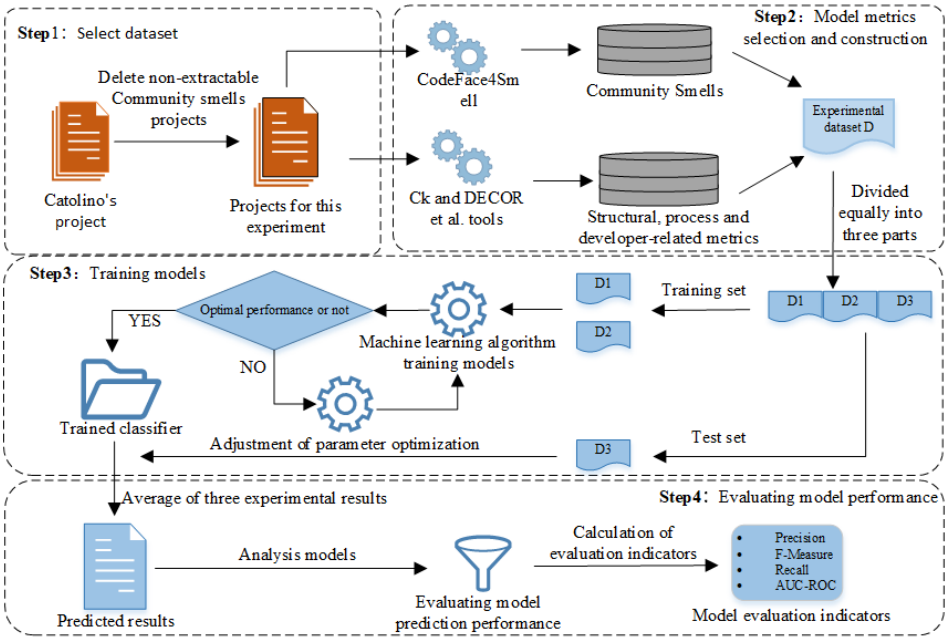

Fig. 1. Change prediction models built using community smells

history of collaboration and communication between developers, the tool for extracting community smell fails to extract community smell in these projects. We chose some of these items after selection, and the dataset presently contains 16 versions of seven items. Table 2 reports more details on the names of the projects used in the experiment, the number of releases for each projects, the number of smells and the number of change-prone classes.

Table 2. Characteristics of the software project in the dataset

System	Releases	Classes	Change-Prone Class	Community Smell Classes
Apache Ant	4	124-350	29-93	5-32
Apache Log4j	2	104-194	24-56	3-5
Apache ivy	1	352	97	15
Apache Lucene	3	186-330	46-88	3-14
Apache Synapse	1	251	64	14
Apache Velocity	2	214-229	58	4
Apache Xerces	3	331-452	77-122	40-52

3.2. Feature Selection and Dataset Construction

To explore whether community smell performs well in change prediction models, we select three baseline models based on structural, process, and developer-related metrics, all of which have been demonstrated to be excellent models for change prediction in prior reserch. Furthermore, the best performing code smell intensity and anti-pattern metric from the current study results are chosen for comparison, in order to further investigate the amount to which community smell contributes to change prediction. The model metrics were extracted as shown in Figure 2.

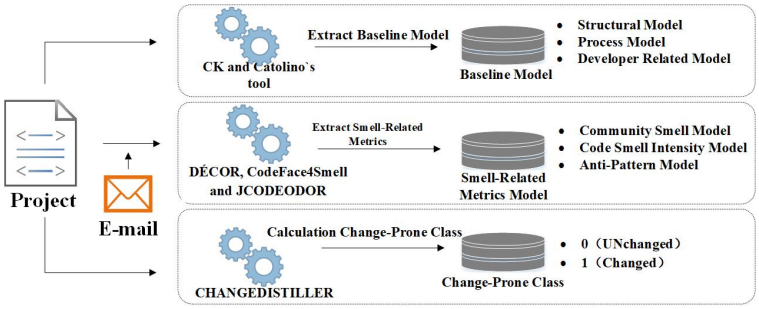

Fig. 2. Extracting features of change prediction models

3.2.1. Baseline Model Selection

1. Models based on structural metrics

   The change prediction model based on structural metrics was originally proposed by Zhou et al. [32] The model reveals the robustness of project development by calculating the structural properties of the project source code. Metrics including code cohesion and coupling, responsiveness to classes, and inheritance metrics are all part of the source code structure property metric. We use a publically available tool developed by Spinellis et al. [33], which has been widely used and empirically demonstrated to be reliable by many researchers, to extract structural metrics of code classes in software systems. The model constructed using the structural metric is denoted as SM in the experimental results, i.e. the structural model.
2. Models based on process metrics

   The study by Elish et al. [29] proposes an Evolution Model (EM) model that extracts metrics for different aspects of class evolution, such as metrics like the density of changes to the class. The study by Elish et al. [29] suggests that a class that has experienced more changes in the past, this class will experience more changes than other classes afterwards, and the EM model directly uses the number of previous changes to the class to predict the propensity of the class to

change in the future. The metric employed in this model differs from Hassan's [34] entropy metric since it does not filter for the sort of changes made to the class as well as does not focus on software development difficulty. To calculate these metrics, we use the tool developed by Catolino et al. [3] et al. We removed some of the metrics since a significant degree of correlation between them would impair the model's accuracy. The model constructed using the process metric is denoted as PM in the experimental results, i.e. the process model.

3. Models based on developer-related metrics

In a previous study, Catolino et al. [3] added developer-related metrics to a class change prediction model and achieved significant effectiveness. Developer-related metrics reveal how developers work on modifications in software changes and the complexity of the development process. The Developer-based change prediction model(DCBM) designed by Nucci et al. [16] has been confirmed to be the most effective approach in change prediction, which uses the structural and semantic dispersion of developers working on code components within a given time period α as predictors. The scattering metric for each class c is calculated according to Eq.(1) and (2), where $developers_{c,\alpha}$ denotes the set of developers for class c within a time period, with $\text{StrScat}_{c,\alpha}$ and $\text{SemScat}_{c,\alpha}$ denotint the structural and semantic scattering of developer d within the time period α.

$$\text{StrScatPred}_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} StrScat_{d,\alpha} \tag{1}$$

$$\text{SemScatPred}_{c,\alpha} = \sum_{d \in developers_{c,\alpha}} SemScat_{d,\alpha} \tag{2}$$

Given a set of developers changing classes over a period of time α, the $CH_{d,\alpha}$ developer's structure dispersion is calculated as shown in Eq. (3), where the '$dist$' function returns the number of packages to be traversed from class c_i to class c_j, calculated by using the shortest path algorithm on a graph representing the structure of the system packages. The higher the measurement, the higher the developer scatter value, i.e. the more changes a developer makes in different packages in a given period of time, the higher the structure scatter value will be.

$$\text{StrScat}_{c,\alpha} = |CH_{d,\alpha}| \times \frac{1}{average_{\forall c_i, c_j \in CH_{d,\alpha}}[dist(c_i, c_j)]} \tag{3}$$

The semantic dispersion of developers based on the textual similarity of developers changing classes in time period α is shown in Eq. (4), where the "sim" function returns the textual similarity between classes c_i to c_j in the system, which is measured according to the vector space model (VSM) [35].

$$\text{SemScat}_{c,\alpha} = |CH_{d,\alpha}| \times \frac{1}{average_{\forall c_i, c_j \in CH_{d,\alpha}}[sim(c_i, c_j)]} \tag{4}$$

The model constructed using the developer-related metric is denoted as the DCBM in the experimental results, i. e. the developer model.

Given that the model's performance suffers when two or more metrics are significantly correlated, we performed a feature selection of the metrics utilized in the model and excluded some of the metrics with high covariance. Tables 3 illustrate the final metrics incorporated in each model.

Table 3. Metrics and descriptions used in the baseline model

Baseline Model	Indicators	Description	Deleted Metrics
Structural Model (SM)	wmc, dit, noc, cbo, ce, npm, lcom3, loc, dam, moa, mfa, cam, ic, cbm, amc, avg_cc	Source code structure properties such as lines of code, cohesive coupling, class inheritance metric, etc.	rfc, lcom, ca, max_cc
Process Model (PM)	boc, frch, wcd, tach, lca, csb, cho	Different aspects of the evolution of a class such as frequency of change or date of birth of the class	fch, lch, wfr, ataf, chd, lcd, csbs, acdf
Developer-Related Model (DCBM)	fi.change, ostrand, scattering	Number of developers, structural, and semantic dispersion, and change complexity	None

3.2.2. Selection of Smell-Related Metrics

Community smell reveals suboptimal organizational structures and poor communication patterns in software development communities. To extract community smells from open-source projects, we use publicly available tool CodeFace4Smell developed by Tamburri et al. [36], which generates collaboration and communication graphs between software project developers via email lists and source code repositories. Palomba et al. [37] used the CodeFace4Smell tool to extract community smells in the development community, meanwhile interviewing developers if they noticed the presence of community smells to test the tool's dependability. The tool's dependability is determined by comparing the outcomes of the two studies, thus we believe that applying the CodeFace4Smell tool to extract community smells in projects is accurate and dependable. According to the study in section 2.1 of the paper, we use the tool to extract O.Silo(Organizational Silo), RS(Radio Silence), and ML(Missing Links) three community smells.

Code smell intensity indicates the severity of code smell in a software project, and this experiment uses JCODEODOR, an automated detection tool developed by Fontana et al. [38] to calculate the intensity metric, the reliability of which was confirmed in a study by Palomba et al. [39] The six code smells calculation software items in Table 4 were selected for this experiment to calculate the code smell intensity.

Anti-patterns are a reoccurring bad design problem in software projects that explain that developers design classes poorly in software projects. Although the

Table 4. Calculates the selected code smell for code smell intensity

Code Smells	Acronym	Description
Gold Class	GC	A poorly cohesive class that implements different responsibilities.
Class Data Should be Private	CDP	CDP occurs when a class has more than 10 public variables, it is considered to be involved in CDP smell.
Complex Class	CC	If the McCabe Cyclomatic Complexity of a class is above 200, the class is considered to be involved in CC smell.
Functional Decomposition	FD	FD may occur when a class is intended to be object-oriented and implemented by non-object-oriented developers.
Spagheeti Code	SC	SC may occur when a class is implemented in the way of procedural thinking.
Long Method	LM	If a class has one or more methods with more than 100 lines and more than two input parameters, it is considered to be involved in LM smell.

presence of anti-patterns in a software project does not preclude it from operating, they indicate design flaws. In our project, we employ the DECOR approach described by Moha et al. [40], which has been proved to obtain 100% Recall in Moha's [40] study, therefore we consider the application of this method for detecting anti-patterns to be reliable.

3.2.3. Calculation of Software Class Changes

We refers to experimental studies by Catolino et al. [10], Elish et al. [29], and Zhou et al. [32] who used an within-project strategy, i.e. counting changes in classes across multiple releases of the project. Romano et al. [27] proposed that a class is considered change-prone if the number of changes it undergoes in a given time period Tw is higher than the median number of changes experienced by all classes in the system.

In addition to this, we ran the tool CHANGEDISTILLER proposed by Fluri et al. [41] for each project, which extracts the fine granularity of code changes between c_i to $c_{(i+1)}$ in a tree difference algorithm. CHANGEDISTILLER recognises renaming operations and can recognise when classes have been modified during the change history and therefore does not bias the correct calculation of the number of changes. Given that the model's performance suffers when two or more metrics are significantly correlated, we performed a feature selection of the metrics utilized in the model and excluded some of the metrics with high covariance. Tables 5 illustrates specific information about the community smell metric and the two other code smell intensity metrics and anti-pattern metrics used for comparison

Table 5. Description and acronym of smell-related metrics

Metrics	Indicators	Description	Deleted Metrics
Community Smell (Comm)	Org.silo, Radio Silence, Missing Links	Suboptimal organization and poor interaction among developers	None
Code Smell Intensity (CodeInt)	intensity	The severity of the code smell	None
Anti-Pattern (Anti)	ana, acm, arl, acpd	Poor design patterns in software development	None

3.2.4. Classifier Selection and Validation

Different machine learning classifiers have been proposed in previous studies to distinguish change-prone and non-change-prone classes. Romano et al. [28] used Support Vector Machines in his study, while Tsantalis et al. [42] used Logistic Regression. Based on previous results, no researcher has explicitly stated which machine learning algorithm showed the best performance in change prediction. For this reason, we utilised each of the six machine learning algorithms in Table 6 - Decision Tree, Naive Bayes, Random Forest, Logistic Regression, Multilayer Perceptron and Support Vector Machine - to train and test the models.

Table 6. Machine learning algorithm name and acronyms

Algorithms	Acronyms	Algorithms	Acronym
Random Forest	RF	Naive Bayes	NB
Logistic Regression	LR	Decision Tree	ADT
Support Vector Machine	SVM	Multilayer Perceptron	MLP

During the performance validation of the prediction model, we employ ten repetitions of the 10-fold cross-validation approach as the validation strategy to assure the reliability of the research results. The approach can partition the Dataset into 10 randomly sized copies for each item being assessed, with nine copies serving as the training set and one serving as the test set. The procedure is performed ten times, each time with a different test set, before the results of the ten tests are averaged to confirm the experiment's dependability.

3.2.5. Building the Model

Eight change-prone class prediction models are constructed: a baseline model using each of the three structural, process, and developer-related metrics, three combination models using a combination of community smell, code smell intensity, and anti-pattern with the baseline model, another three models using two of the three smell metrics with the baseline, and a final model using a combination of the three

smell metrics with the baseline. Table 7 demonstrates the eight models we have constructed.

Table 7. All change prediction models used in the research

Models	Metrics Included in the Model
Baseline	SM [32], PM [3], DCBM [16]
Base+Comm	Baseline, Community Smell
Base+CodeInt	Baseline, Code Intensity
Base+Anti	Baseline, Anti
Base+Comm+CodeInt	Baseline, Community Smell, Code Intensity
Base+Comm+Anti	Baseline, Community Smell, Anti
Base+CodeInt +Anti	Baseline, Code Intensity, Anti
Base+Comm+CodeInt+Anti	Baseline, Community Smell, Code Intensity, Anti

3.3. Evaluation Metrics

In order to measure and evaluate the performance of the model, we use Precision, Recall, F-Measure and AUC-ROC to evaluate the model. Eq. (5) and (6) are the Precision and Recall calculated from the confusion matrix of the predicted results.

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

Where TP is the number of true positives, TN is the number of true negatives and FP is the number of false positives. In order to have a unique value to represent how good the model is, Eq. (7) calculates F-Measure, the harmonic mean of Precision and Recall.

$$F - Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{7}$$

In addition, we considers another metric: the area under the characteristic operating curve, ROC, which is a graph showing the performance of the classification model at all classification thresholds, and AUC, which measures the entire two-dimensional area below the entire ROC curve. This metric quantifies the overall ability of the change prediction model to distinguish between change-prone and non-change-prone classes, and can quantify the robustness of the model in distinguishing between the two binary classes.

4. Results and Discussion

We evaluate the effectiveness of metrics measuring development process complexity for predicting classes of variability in software systems, with the purpose of

improving resource allocation in preventive software system maintenance activities by focusing on classes of variability to reduce the cost of change. To this end, we propose the following questions.

- Q1: Does community smell improve the performance of baseline models?
- Q2: What is the importance of community smell as a feature in different baseline models?
- Q3: What is the performance of change prediction models for combinations of smell-related information?

Q1 aims to investigate the contribution of community smell to the performance of three baseline change prediction models constructed based on structural, process and developer-related metrics. In Q2, we compare the importance of the three community smell characteristics in change prediction models constructed using different baselines. Q3 assesses the combined ability to use smell-related information in change prediction.

In this section, we respond to the proposed Q1, Q2, and Q3 by giving the experimental findings of the change prediction models associated with each question in terms of Precision, Recall, F-Measure, and AUC-ROC. Six machine learning methods are employed to train and validate each model in order to evaluate its performance, and the final test results show that MLP works best. We only show the experimental results obtained by the best performing MLP.

4.1. Q1: Does community smell improve the performance of baseline models?

We start with training tests of the change prediction models using the three baseline models and community smell, and Table 8 depicts the performance of the change prediction models with community smell added to three of the baseline models, SM(Structural Model), PM(Process Podel) and DCBM(Developer-Related Metrics).

Table 8. Baseline and Base+Comm model performance

Models	Precision	Recall	F-Measure	AUC-ROC
SM	0.761	0.801	0.734	0.795
SM+Comm	0.760	0.794	**0.753**	0.797
PM	0.753	0.793	0.744	0.774
PM+Comm	**0.774**	**0.803**	**0.754**	**0.789**
DCBM	0.651	0.670	0.636	0.713
DCBM+Comm	0.646	**0.678**	**0.643**	0.713

As shown in the experimental results, community smell improved the performance of the SM baseline model by 1.9% in terms of F-Measure and 0.2% in terms of AUC-ROC, improved the performance of the PM baseline model by 2.1%, 1%, 1%, and 1.5% in terms of Precision, Recall, F-Measure and AUC-ROC, while improving

the performance of the DCBM baseline model in terms of Recall and F-Measure by 0.8%, and 0.7%. In general, the performance of the models is improved by adding community smell to the three baseline models, with community smell contributing more to PM than SM and DCBM. The experimental results did not observe a significant contribution of community smell to the performance of the baseline model. It is speculated that the ratio of the classes containing community smell to the total class in the dataset is too low. After statistical analysis of the data set, it was found that 5% of the classes contained at least one community smell, the proportion of the change-prone class was 26%, and the proportion of the change-prone class containing the community smell in the data set was only 1%.

Afterwards, we considers that there is a difference in performance between the models constructed by applying different machine learning algorithms in the experiment, and since the models constructed using the process metric outperform the other metrics, Table 9 gives the model performance for PM and PM+Comm constructed using different algorithms.

Table 9. Performance of models build by different algorithms

Models	Evaluation Metrics	RF	ADT	NB	LR	MLP	SVM
PM	Precision	0.742	0.697	0.619	0.539	0.753	0.811
	Recall	0.818	0.745	0.642	0.566	0.793	0.890
	F-Measure	0.747	0.702	0.620	0.539	0.744	0.810
	AUC-ROC	0.789	0.682	0.648	0.559	0.774	0.862
PM+Comm	Precision	0.732	0.677	0.554	**0.561**	**0.774**	0.763
	Recall	0.802	0.712	**0.957**	**0.599**	**0.803**	0.849
	F-Measure	0.728	0.680	**0.664**	**0.568**	**0.754**	0.762
	AUC-ROC	**0.798**	0.670	**0.653**	**0.574**	**0.789**	0.814

The experimental results show that comparing the training results of six machine learning algorithms on the model, both SVM and MLP performed the best, with LR being the least effective. The results of the models constructed using NB showed that community smell improved PM's model performance by 31.5% in terms of Recall and 4.4% in terms of F-Measure.

4.2. Q2: What is the importance of community smell as a feature in different baseline models?

To further observe the impact of community smells on the change prediction models, this experiment evaluated the importance of the selected three community smells in the selected change prediction models constructed using different baseline models in Q2 and ranked the importance of the community smells. The characteristic importance of the three community smells in the change prediction models constructed using the structural metric, the process metric and the developer-related metric, respectively, is presented in Table 10 and the three community smells are ranked

in terms of strong, medium and weak.

Table 10. Feature importance of community smell in different baseline models

Models	Importance of features		
	Strong	Medium	Weak
SM+Comm	ML(0.016)	RS(0.014)	O.Silo(0.003)
PM+Comm	RS(0.017)	ML(0.008)	O.Silo(0.004)
DCBM+Comm	RS(0.041)	ML(0.039)	O.Silo(0.027)

This experiment quantifies the gain provided by three community smells to different baseline models by means of the information gain method, an algorithm that ranks model features according to their ability to predict the propensity for class change. The information gain method was calculated as shown in Eq. (8) as follows.

$$Gain\,(D,a) = Ent\,(D) - \sum_{v=1}^{V} \frac{|D^v|}{|D|} Ent\,(D^v) \tag{8}$$

This formula represents the information gain $Gain(D,\alpha)$ obtained by dividing the Dataset D by the feature a. Assuming that the proportion of the kth class samples in the dataset D is p_k, the information entropy of the Dataset D is $Ent(D)$ as shown in Eqs. (9) as follows.

$$Ent\,(D) = -\sum_{k=1}^{|y|} p_k \log_2 p_k \tag{9}$$

The experimental results show that O. Silo(Organizational Silo) provides the weakest information gain across the different baseline models and that this smell indicates the presence of isolated areas of non-communication in the development community. For change prediction models constructed using structural metrics, ML(Missing Links) have the strongest feature importance, i.e. the lack of communication between developers have a higher degree of impact on class change prediction than the two smells O. Silo and RS(Radio Silence). When the change prediction models were constructed using both process metrics and developer-related metrics, RS showed strong feature importance, and RS showed higher feature importance in the change prediction models constructed using developer-related metrics than in the change prediction models constructed using structural metrics. Overall, in the software class-level change prediction model, community smell plays a significant role in software class change prediction by representing the suboptimal organizational model of the development community, with O.Silo having a weaker impact than ML and RS.

4.3. Q3: What is the performance of change prediction models for combinations of smell-related information?

In Q3 we aimed to verify whether a combination of smell metrics would improve the performance of the baseline model and to compare the difference in performance between change prediction models constructed using multiple smell-related information. The experimental results are presented in Table 11 where the performance of the combined smell on the three baseline models SM, PM, and DCBM are presented and compared to the performance of the baseline models.

Table 11. Performances of models build by using combinations of smell-related information

Models	Precision	Recall	F-Measure	AUC-ROC
SM	0.761	0.801	0.734	0.795
SM+Anti+Comm	0.794	0.864	0.790	0.807
SM+CodeInt+Comm	0.757	0.803	0.765	0.810
SM+CodeInt+Anti+Comm	**0.807**	**0.864**	**0.794**	**0.819**
PM	0.753	0.793	0.744	0.774
PM+Anti+Comm	**0.811**	0.855	**0.797**	0.826
PM+CodeInt+Comm	0.769	0.818	0.778	0.787
PM+CodeInt+Anti+Comm	0.808	**0.859**	0.787	**0.827**
DCBM	0.651	0.670	0.636	0.713
DCBM+Anti+Comm	0.769	0.822	0.760	0.807
DCBM+CodeInt+Comm	0.667	0.686	0.656	0.731
DCBM+CodeInt+Anti+Comm	**0.781**	**0.832**	**0.780**	**0.811**

In terms of Precision, Recall, F-measure, and AUC-ROC, the SM+Anti+Comm performed well when the SM was used for the baseline model. More specifically, performance of the SM was improved by 3.3%, 6.3%, 5.6%, and 1.2%. The SM+CodeInt+Comm improved the performance of the baseline model in terms of Recall, F-measure, and AUC-ROC by 0.2%, 3.1%, and 1.5% respectively, while the SM+CodeInt+Anti+Comm improved the performance of the baseline model in terms of the four metrics by 2.4% - 6.3%.

When the baseline model used process metrics, the performance of the PM is improved in various magnitudes by different combinations of smell metrics for all four metrics. The PM+Anti+Comm improved the performance of the PM by 5.8%, 6.2%, 5.3%, and 5.2% in terms of Precision, Recall, F-measure, and AUC-ROC respectively, The performance of the PM+CodeInt+Anti+Comm improves the performance of the PM by 4.3% to 6.6% in four metrics. Among all the models, the model achieved the worst AUC-ROC value of 77.4%, while the best model was the PM+CodeInt+Anti+Comm, which achieved a performance value of 82.7% in terms of AUC-ROC.

When the baseline model used the DCBM, the change prediction model using the three combinations of smell-related information performed the best of all models in terms of Precision, Recall, F-measure, and AUC-ROC, improving the

performance of the DCBM by 13%, 16.2%, 14.4%, and 9.8% respectively. Performance of the DCBM was also improved by 1.6%-2% in the four metrics, although the DCBM+CodeInt+Comm showed the least improvement in performance.

Combining the experimental results of all the models, the community smell in change prediction model proposed in this experiment greatly improves the performance of the model. In particular, when the baseline model was constructed using the process metric, the PM+ CodeInt+Anti+Comm performed best, with model performance reaching 80.8%, 85.9%, 78.7%, and 82.7% in terms of Precision, Recall, F-Measure, and AUC-ROC.

5. Threats to Validity

This section will address the threats that may affect the validity of this experimental study in terms of structural validity, conclusion validity, and external validity.

The threat to structural validity is the reliability of the Dataset used in this experiment. We select a part of the dataset items from Catolino's [10] study from which community smells can be extracted. The available Dataset contains 16 versions of dataset files from seven open-source projects, and this experimental study was confirmed to be reliable in terms of the source of the Dataset. The extraction tools used for the extraction of structural, process, and developer-related metrics as well as smell-related information have been tested and proven reliable by several researchers. For the use of the metric extraction tools, the guidelines for installation, configuration, and use of the tools were strictly followed.

The threat to conclusion validity is the result of model training and validation. This experiment used Precision, F-Measure, Recall, and AUC-ROC metrics to evaluate model performance for all class change prediction models, which are widely used to assess the performance of classification models. When validating the model, this experiment used a tri-fold cross-validation method with 10 replications, with the Dataset being randomised before each replication to avoid data order bias.

The threat of external validity is related to the generalisation of results. The projects considered in this experiment differ in terms of application area, sizes, and number of classes. The projects considered in this experiment differ in terms of application area, sizes, and number of classes. The construction of change prediction models covers a wide range of metrics in use, with structural metrics, process metrics, and developer-related metrics revealing different aspects of the software project's characteristics.

6. Conclusion

Software changes was an essential part of the development of a software system and it was vital for developers to identify in advance the code modules in the software system that has a higher propensity to change. Based on prior work of Palomba et al. [8] and Catolino et al. [10], we focused on the impact of community smell in change prediction models.

Our research revealed that integrating community smell to the baseline model improved the predictive performance of the model, when the baseline model was built using the process metric, community smell improved the performance of the change prediction model in terms of F-measure and Recall by 4.4% and 31.5% respectively. The information gain provided to the model by O. Silo was weaker than that provided by RS and ML in the three community smells used in this experiment, while RS and ML provided different strengths of information gain in different baseline models. The change prediction models constructed using the three smell-related information performed well, achieving a highest AUC-ROC value of 82.7% in terms of model performance.

Future work includes: First, extending the scope of measured community smells to explore the impact of more smells in change prediction; Second, extend the dataset to include projects developed in other programming languages; And third, contact developers on multiple projects by questionnaires to assess whether developers perceive the presence of change-prone classes to further explore the practicability of our model.

References

[1] F. Vonken and A. Zaidman, Refactoring with Unit Testing: A Match Made in Heaven?, Proceedings of 19th Working Conference on Reverse Engineering, Kingston, Canada, 15-18 Oct, 2012, pp. 29-38, doi: 10.1109/WCRE.2012.13.

[2] L. Pascarella, D. Spadini, and F. Palomba, et al. Information needs in contemporary code review, Proceedings of the ACM on Human-Computer Interaction, New York, United States, November, 2018, pp:1-27, doi: 10.1145/3274404.

[3] G. Catolino, F. Palomba, and A. Lucia, et al. Enhancing change prediction models using developer-related factors, ournal of Systems and Software, 143(2018): 14-28,2018.

[4] M. Kim, T. Zimmermann, and N. Nagappan. An Empirical Study of RefactoringChallenges and Benefits at Microsoft, IEEE Transactions on Software Engineering, 40(7), pp. 633-649, 2014, doi: 10.1109/TSE.2014.2318734.

[5] D. Q. Soetens, S. Demeyer, and A. Zaidman, et al. Change-based test selection: an empirical evaluation, Empirical Software Engineering, 21(5): 1990-2032, 2016.

[6] M. Fowler. Refactoring: improving the design of existing code. Addison-Wesley, 1999.

[7] F. Khomh, M. D. Penta, and Y. G . Gueheneuc, et al. An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering, 17(3), 243-275, 2012.

[8] F. Palomba, G. Bavota, and M. D. Penta, et al. On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation, Proceedings of IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May-3 June 2018, 23(3)pp.482-482, doi: 10.1145/3180155.3182532.

[9] M. D. Ambros, A. Bacchelli, and M. Lanza. On the Impact of Design Flaws on Software Defects, Proceedings of 10th International Conference on Quality Software, Zhangjiajie, China, 14-15 July, 2010, pp. 23-31, doi: 10.1109/QSIC, 2010.

[10] G. Catolino, F. Palomba, and F. A. Fontana, et al. Improving change prediction models with code smell-related information, Empir Software Engineering, 25(1), 49–95, 2020, doi: 10.1007/s10664-019-09739-0.

[11] D. A. Tamburri, P. Lago, and H. Vliet. Organizational social structures for software

engineering, Proceedings of IEEE/ACM Computing Surveys, Niagara Falls, Canada, 25-28 Aug, 2013, 46(1), 3.

[12] B. Caglayan. An issue recommender model using the developer collaboration network, Bogazici University, 2014.

[13] D. A. Tamburri, P. Kruchten, and P. Lago, et al. Social debt in software engineering: insights from industry, Journal of Internet Services and Applications, 6(1):1-17, 2015.

[14] L. Gren. On Gender, Ethnicity, and Culture in Empirical Software Engineering Research, Proceedings of IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), Gothenburg, Sweden, 27 May-3 June, 2018, pp. 77-78.

[15] J. T. Ostrand, J. E Weyuker, and M. R Bell. Programmer-based fault prediction, Proceedings of the 6th International Conference on Predictive Models in Software Engineering, Timişoara, Romania, September 12-13, 2010, 1-10.

[16] D. D. Nucci, F. Palomba, and G. De Rosa, et al. A Developer Centered Bug Prediction Model, IEEE Transactions on Software Engineering, 44(1), pp. 5-24, doi: 10.1109/TSE.2017.2659747, 2018.

[17] G. Calikli and A. Bener. Empirical analysis of factors affecting confirmation bias levels of software engineers, Software Quality Journal, 23(4): 695-722, 2015.

[18] B. Eken. Assessing Personalized Software Defect Predictors, Proceedings of IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 27 May-3 June, 2018, pp. 488-491.

[19] B. Soltanifar, A. Erdem, and A. A. Bener. Predicting defectiveness of software patches, Proceedings of the 10th IEEE/ACM International Symposium on Empirical Software Engineering and Measurement, Ciudad Real, Spain, September, 2016, pp.22:1-10, doi:10.1145/2961111.2962601.

[20] G. Catolino, F. Palomba, and D. A. Tamburri, et al. Refactoring Community Smells in the Wild: The Practitioner's Field Manual, Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), Seoul, Korea (South), 5-11 Oct, 2020, pp. 25-34.

[21] F. Palomba, D. A. Tamburri, and F. A. Fontana, et al. Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?, IEEE Transactions on Software Engineering, 2018:1-1, pp. 108-129, 1 Jan. 2021, doi: 10.1109/TSE.2018.2883603.

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

[23] M. D. Penta, L. Cerulo, and Y. G. Gueheneuc, et al. An empirical study of the relationships between design pattern roles and class change proneness, Proceedings of the IEEE International Conference on Software Maintenance, Beijing, China, 28 Sept-4 Oct, 2008, pp. 217-226, doi: 10.1109/ICSM.2008.4658070.

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

[25] M. Lindvall. Measurement of change: stable and change-prone constructs in a commercial C++ system, Proceedings of 6th International Software Metrics Symposium, Boca Raton, FL, United States, Nov4-6, 1999, pp. 40-49, doi: 10.1109/METRIC.1999.809724.

[26] A. Kavitha. A. Shanmugam, Dynamic Coupling Measurement of Object Oriented Software Using Trace Events, Proceedings of 6th International Symposium on Applied Machine Intelligence and Informatics, Herlany, Slovakia, 21-22 January, 2008, pp. 255-259, doi: 10.1109/SAMI.2008.4469179.

[27] D. Romano and M. Pinzger. Using source code metrics to predict change-prone Java

interfaces, Proceedings of 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA, Sept. 25 to Sept. 30, 2011, pp. 303-312, doi: 10.1109/ICSM.2011.6080797.

[28] S. Eski and F. Buzluca. An Empirical Study on Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes, Proceedings of IEEE 4th International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, March 21-25, 2011, pp. 566-571, doi: 10.1109/ICSTW.2011.43.

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

[30] T. Girba, S. Ducasse, and M. Lanza. Yesterday's Weather: guiding early reverse engineering efforts by summarizing the evolution of changes, Proceedings of 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11-14 Sept, 2004, pp. 40-49, doi: 10.1109/ICSM.1357788.

[31] M. R. Bell, J. T. Ostrand, and J. E. Weyuker. The limited impact of individual developer data on software defect prediction, Empirical Software Engineering, 18(3): 478-505, 2013.

[32] Y. Zhou, H. Leung, and B. Xu. Examining the Potentially Confounding Effect of Class Size on the Associations between Object-Oriented Metrics and Change-Proneness, IEEE Transactions on Software Engineering, 35(3), pp.607-623, Sept.-Oct, 2009, doi: 10.1109/TSE.2009.32.

[33] D. Spinellis. Tool writing: a forgotten art? (software tools), IEEE Software, 22(4), pp. 9-11,.2005, doi: 10.1109/MS.2005.111.

[34] A. E. Hassan. Predicting faults using the complexity of code changes, Proceedings of IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16-24 May 2009, pp. 78-88, doi: 10.1109/ICSE.2009.5070510.

[35] R. Y. Baeza and B. N. Ribeiro. Modern information retrieval, New York: ACM press, 1999.

[36] D. A. Tamburri, F. Palomba, and R. Kazman. Exploring community smells in open-source: An automated approach, IEEE Transactions on software Engineering, 2019:1-1, doi: 10.1109/TSE.2019.2901490.

[37] F. Palomba and D. A. Tamburri. Predicting the emergence of community smells using socio-technical metrics: a machine-learning approach, Journal of Systems and Software, 171(1):110847, 2021, doi:10.1016/j.jss.2020.110847.

[38] F. A. Fontana, V. Ferme and M. Zanoni, et al. Towards a prioritization of code debt: A code smell Intensity Index, Proceedings of IEEE 7th International Workshop on Managing Technical Debt, Bremen, Germany, 2-2 Oct, 2015, pp. 16-24, doi: 10.1109/MTD.2015.7332620.

[39] F. Palomba, M. Zanoni, and F. A. Fontana, et al. Toward a smell-aware bug prediction model, IEEE Transactions on Software Engineering, 45(2): 194-218, 2017.

[40] N. Moha, Y. Gueheneuc and L. Duchien, et al. DECOR: A Method for the Specification and Detection of Code and Design Smells, IEEE Transactions on Software Engineering, 36(1), pp. 20-36, 2009, doi: 10.1109/TSE.2009.50.

[41] B. Fluri, M. Wursch and M. PInzger, et al. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction, IEEE Transactions on Software Engineering, 33(11), pp. 725-743, 2007, doi: 10.1109/TSE.2007.70731.

[42] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, Predicting the probability of change in object-oriented systems, IEEE Transactions on Software Engineering, 31(7), pp. 601-614, July 2005, doi: 10.1109/TSE.2005.83.