SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2021, Vol. 64 132102:1–132102:17
https://doi.org/10.1007/s11432-019-2902-5

Toward accurate detection on change barriers

Tingting LV, Zhilei REN[*], Xiaochen LI, Guojun GAO & He JIANG

School of Software, Dalian University of Technology, Dalian 116024, *China*

Received 4 August 2019/Revised 19 November 2019/Accepted 3 April 2020/Published online 7 February 2021

Abstract In software development, it is easy to introduce code smells owing to the complexity of projects and the negligence of programmers. Code smells reduce code comprehensibility and maintainability, making programs error-prone. Hence, code smell detection is extremely important. Recently, machine learning-based technologies turn to be the mainstream detection approaches, which show promising performance. However, existing machine learning methods have two limitations: (1) most studies only focus on common smells, and (2) the proposed metrics are not effective when being used for uncommon code smell detection, e.g., change barrier based code smells. To overcome these limitations, this paper investigates the detection of uncommon change barrier based code smells. We study three typical code smells, i.e., Divergent Change, Shotgun Surgery, and Parallel Inheritance, which all belong to change barriers. We analyze the characteristics of change barriers and extract domain-specific metrics to train a Logistic Regression model for detection. The experimental results show that our method achieves 81.8%–100% precision and recall, outperforming existing algorithms by 10%–30%. In addition, we analyze the correlation and importance of the utilized metrics. We find our domain-specific metrics are important for the detection of change barriers. The results would help practitioners better design detection tools for such code smells.

Keywords code smells, change barrier, logical regression, machine learning, software development

Citation Lv T T, Ren Z L, Li X C, et al. Toward accurate detection on change barriers. Sci China Inf Sci, 2021, 64(3): 132102, https://doi.org/10.1007/s11432-019-2902-5

1 Introduction

Software development and maintenance are essential software engineering processes. During the software lifecycle, it is easy to introduce code smells owing to the negligence of developers, which has a negative impact on software maintenance and evolution. To address these problems, Fowler et al. [1] proposed the concept of code smell and defined 22 types of code smells. Code smell is a symptom of poorly designed or implemented code that makes programs difficult to understand or error-prone. It can hinder code understandability and maintainability. It also makes software difficult to develop and maintain. Hence, it is important to detect code smells and help developers identify design issues that may cause problems in future maintenance.

So far, there have been many techniques for code smell detection, such as machine learning-based methods, metric-based detection strategies [2], search-based algorithms [3], and text and structural analysis [4]. In recent years, more and more researchers focus on machine learning approaches [5,6]. Machine learning, as the most mainstream technology at present, also has a better detection effect [7].

However, we identify two limitations for machine learning based technologies: (1) most articles only focus on common code smells, and (2) the metrics used in the literature may not be sufficient to train an effective classifier for detecting uncommon code smells, e.g., change barrier code smells. Many studies prefer to identify common code smells with simple structure, e.g., Long Method [8,9] and God Class [10, 11]. The reason they pay little attention to other code smells may be that many code smells cannot be detected using a rule-based approach so that no valid metric[1)] can be found. Kreimer [10] and Fontana

[*] Corresponding author (email: zren@dlut.edu.cn)

1) "Metrics" are the features/attributes to measure code. Metrics can convert the code into feature vectors for training machine learning models. In this paper, the words "metric", "feature", and "attribute" are used according to the specific scene, and the meaning is the same.

et al. [8] focused on one or several bad smells (God Class, Data Class, Feature Envy, and Long Method), which are more common in software projects. The metrics applied in the literature can only work for the identification of code smells concerned in their papers, not other complex and uncommon code smells, e.g., change barriers. Therefore, this paper studies the efficient identification method on such code smells.

To overcome the limitations, we focus on three different types of change barriers: Divergent Change, Shotgun Surgery, and Parallel Inheritance. Linders [12] first proposed the concept of change barrier based code smells, which means that a change in code affects many classes and requires extensive modifications to the entire code. If a developer wants to implement a new requirement or adjust an existing module, it is inevitable that small modifications might lead to a big, bad change, which may result in an unpredictable workload. Therefore, it is necessary to detect this type of code smells. We consider metrics associated with the smell characteristics. We not only extract common object-oriented metrics, such as code size, encapsulation, and complexity, but also fully analyze the characteristics of these smells and apply some domain-specific metrics such as concern diffusion over components (CDC), number of children (NOC), the depth of inheritance tree (DIT), changing classes (ChC), and changing methods (CM). We integrate the proposed metrics into a Logistic Regression algorithm to detect change barriers. We compare the experimental results with existing metrics and classifiers including support vector machine (SMO), NaiveBayes, JRip, J48 Decision Tree, and RandomForest.

We find that the domain-specific metrics fully reflect the definition and characteristics of each smell. They have a positive effect on the recognition of smells. The experimental results show that our algorithm achieves 81.8%–100% precision and recall in the detection of change barriers, which are 10% to 30% higher than those of other algorithms. Our method is 20% to 40% higher in precision and recall than the algorithms trained with the metrics proposed in other literatures, which proves that our metrics are more beneficial to the recognition of change barriers. In addition, we conduct an experimental analysis of the correlation and importance of the metrics. We use the Spearman rank correlation test to calculate the correlation between the metrics and verify the effect of the correlation metric on the experimental results. We also use the correlation coefficient analysis on Logistic Regression and node analysis on the J48 Decision Tree to identify the importance of the metrics for smell detection. The analysis results confirm the importance of our domain-specific metrics used in this paper.

The main contributions of this paper are as follows.

- To the best of our knowledge, we are the first to systematically study the change barrier code smell, including its core characteristics and the role of different metrics in such bad smells. We collect domain-specific metrics based on the characteristics of change barriers, and compare the experimental results of the algorithms trained on the literature metrics. We find our solution is more beneficial to the detection of change barriers.
- We apply Logistic Regression to solve the problem of "change barriers" code smell detection. We compare our algorithm with state-of-the-art algorithms. The experimental result shows that our algorithm outperforms baseline algorithms in terms of precision, recall, and F-measure.
- In addition, we carry out a metric correlation experiment and a metric importance experiment to investigate the effects of metrics on the identification of code smells. To verify the effectiveness of our approach, we conduct a cross-project experiment. The cross-project experimental results show that our method can achieve better performance.

The remainder of this article is organized as follows. Section 2 describes the definition of change barriers and our motivation. Section 3 presents the framework of our approach. Section 4 describes the experimental setup. Section 5 shows the experimental results. Threats to validity of the study are described in Section 6. Section 7 introduces the related work. Section 8 concludes this paper.

2 Background

This section describes the definition of three change barrier based code smells and the motivation for change barrier detection.

2.1 Definition

Divergent Change is proposed by Fowler [1], which is produced if a class is subjected to different changes. It is that all corresponding modifications to the external changes occur in a single class. For example,

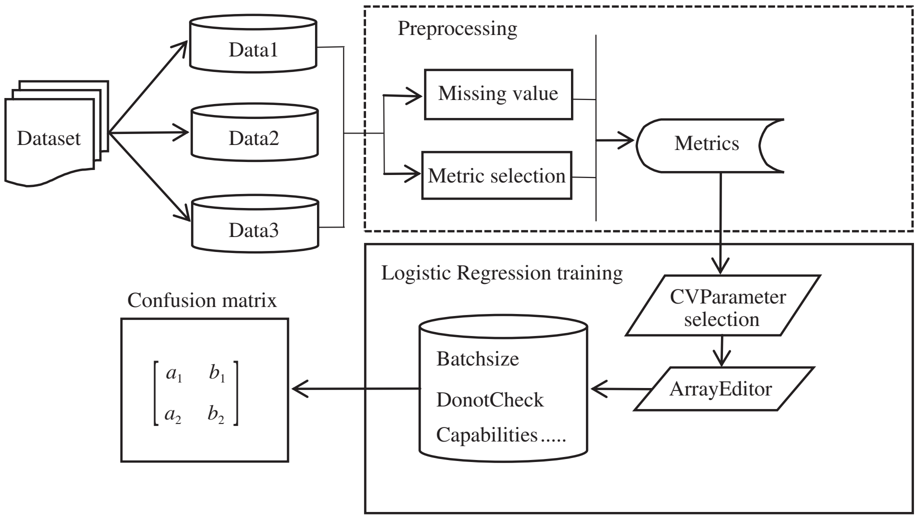

Figure 1 Approach overview.

when you want to add a function, you have to modify many unrelated functions in a single class at the same time. This requires changing other or even unrelated contents for some specific reasons.

Shotgun Surgery refers to the fact that whenever you make a change, it must make small modifications to many different classes [1]. However, if this change appears everywhere, it will be time-consuming to find then one by one. In this case, you may miss some information and cause the program to go wrong. The source of this error is hard to find, so you need to keep debugging to find bugs and fix them.

Parallel Inheritance means that when you create a subclass of a class, it must create a subclass of another class [1]. There is a strong coupling between the two classes. When modifying a class, it is necessary to find another class with the same prefix and modify it accordingly.

The relationship between these three code smells is interesting. They are similar in type, covering multiple levels, but slightly different. Divergent Change is a problem that a class has been impacted by multiple changes, meaning that a single class takes on multiple tasks. In contrast, Shotgun Surgery refers to a variation that causes multiple classes to be modified accordingly, indicating that there are too many couplings between classes. Parallel Inheritance is a subset of Shotgun Surgery, which is defined as a form of Shotgun Surgery.

2.2 Motivation

This paper focuses on three code smells: Divergent Change, Shotgun Surgery, and Parallel Inheritance. They all belong to change barrier based code smells. This type of code smell can turn small changes into big changes with knock-on effects. Developers constantly modify a software product, owing to a series of unpredictable, inevitable reasons such as the requirements of end-users and error debugging. The existence of change barriers makes these modifications become a tough task. If this smell is not recognized and eliminated, developers are about to face endless changes. Therefore, the identification of change barrier code smell has extremely practical significance. This paper applies Logistic Regression to detect change barrier based code smells.

3 Methodology

In this paper, we study the identification of change barrier code smell by integrating the object-oriented software quality metric to construct the Logistic Regression algorithm model. This section describes our approach in detail.

As shown in Figure 1, we extract source code metrics from the dataset, forming the Divergent Change dataset (Data1), the Shotgun Surgery dataset (Data2), and the Parallel Inheritance dataset (Data3) which are composed of metrics related to three code smells, respectively. The specific dataset information is explained in Subsection 4.1. Then, we perform data preprocessing on the three set of data, and take the metric set formed by them as the input of the Logistic Regression model. In the model construction phase,

Table 1 Terminological interpretation

Terminology	Description
CVParameterSelection	A meta-classifier for parameter optimization selection in Weka to find the optimal value for one or more parameters within a specified range.
Filter	A function used to manipulate attributes and instances in the data preprocessing phase.
ReplaceMissingValues	A function for replacing all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data.
AttributeSelection	A supervised attribute filter that can be used to select features.
InfoGainAttributeEval	An evaluator that evaluates the worth of an attribute by measuring the information gain with respect to the class.
Ranker	A search strategy that ranks attributes by their individual evaluations.
ArrayEditor	A module for inputting parameters during parameter optimization.
BatchSize	A parameter for the preferred number of instances to process when a batch prediction is being performed.
DonotCheckCapabilities	A module that decides whether to check the classifier function.
NominalToBinaryFitter	A module that converts the nominal attribute type to a numeric attribute.
Ridge Estimation	A biased estimation regression method for collinear data analysis.

we use CVParameterSelection to optimize the parameters of the model. The CVParameterSelection classifier uses the ArrayEditor module to perform the parameters to be optimized. We use the best parameters of the output to build a model to predict change barriers. Finally, the prediction results output the model's precision and recall, as well as the confusion matrix. Table 1 details the terminology explanations related to data preprocessing and parameter optimization in this section.

3.1 Metric extraction

We use Iplasma [13], Together [14], and Concerntagger [15] to calculate the 80 object-oriented quality metrics for each class as shown in Table 2. We list the information about metrics, including acronyms and their corresponding meanings. These categories of metrics are widely used in existing literature. These three tools obtain the structural attributes of software projects by parsing the source code. Unlike existing studies, we focus on the measurement of specific characteristics of each code smell, such as Concern Metric for Divergent Change, CM and ChC for Shotgun Surgery, and DIT and NOC for Parallel Inheritance.

CM refers to the number of different methods that call the method being measured, while ChC refers to the number of classes. The combination of CM and ChC can reflect the coupling between classes. These metrics present the nature of Shotgun Surgery because this code smell reflects severe coupling between multiple classes.

DIT is the maximum depth from the node to the root of the tree. The depth of a class determines its hierarchy, and the number of methods inherited determines its potential reusability. NOC refers to the number of direct subclasses of a class, which is used to measure how many subclasses inherit the parent class. The metrics suite [16] provided by DIT and NOC has proven to be effective for finding higher failure rates.

Concern Metric is designed to capture the module properties of concerns in the drive design [17]. It depends on the mapping of the element to concerns. Concerns are those important properties of software artifacts that are handled in a modular manner. Each concern belongs to a modular unit. The concern can be on classes, functions, methods, or even a member variable. Existing research has shown that concern metrics can be used not only for the implementation of modularity issues in software products but also for design flaws identification [17]. We expect to use this metric to improve the detection of Divergent Change. In the experimental section, we analyze and evaluate these metrics in detail. We experiment and demonstrate that these characteristic related metrics promote and support the identification of change barriers.

3.2 Data preprocessing

We use the open-source machine learning software Weka [18] to perform data preprocessing on Data1, Data2, and Data3. We handle missing values and select important metrics. In Weka, we use Filter to process the metrics and instances. Filter is a function that contains a variety of data processing

Table 2 Metric definition

Category	Name	Definition	Category	Name	Definition
Basic	Address	Address of class	Complexity	AMW	Average methods weight
	BUR	Usage ratio		AC	Attribute complexity
	Class_name	Name of class		CC	Cyclomatic complexity
	CRIX	Children ratio		EC	Essential complexity
	GREEDY	Raised exceptions		NORM	Number of remote methods
	is static	is static class		NOAV	Number of accessed variables
	is leafclass	is leafclass		NOLV	Number of local variable
	is interface	is interface		NOEU	Number of external variables
	is rootclass	is rootclass		NOP_M	Number of parents in the method
	is abstract	is abstract class		NOP_Pr	Number of parents in the project
	Package	Package of class		RFC	Response for class
	Project	Project of class		WOC	Weight of a class
	PNAS	Public number of accessors		WMC	Weighted methods count
	PS	Package size		WMPC1	Weighted methods per class 1
Code Size	LOC	lines of code in the class		WMPC2	Weighted methods per class 2
	LOCC	Lines of code classes	Coupling	AOFD	Access of foreign data
	LOC_M	lines of code in the method		ATFD	Access to foreign data
	LOC_Pa	lines of code in the packages		CE	Efferent coupling
	LOC_Pr	lines of code in the project		ChC	Changing classes
	NOCON	Number of constructors		CM	Changing methods
	NOM	Number of methods		CBO	Coupling between objects
	NOM_Pr	Project number of methods		DD	Dependency dispersion
	NOM_Pa	Package number of methods		DAC	Data abstraction coupling
	NOA	Number of ancestors		FDP	Foreign data providers
	NAM	Number of accessor methods		FANOUT	Number of classes referenced
	NOD	Number of descendants		FANIN	Number of classes that reference
	NOIS	Number of import statements		MIC	Method invocation coupling
	NOO	Number of operations		NOEC	Number of external clients
	NCC	Number of client classes		NRSS	Number of static calls
	NOIC	Number of internal clients		NOED	Number of external dependencies
	PProtM	Percentage of protected members		WCM	Weighted changing methods
	PPubM	Percentage of public members	Cohesion	ALD	Access of local data
	PPrivM	Percentage of private members		AID	Access of import data
Inheritance	DIT	Depth of inheritance tree		TCC	Tight class cohesion
	DOIH	Depth of inheritance hierarchy	Encapsulation	LAA	Locality of attribute accesses
	HIT	Height of inheritance tree		NOPA	Number of public attribute
	NOC	Number of children		NOAM	Number of added methods
	NOCC	Number of child classes		NOOM	Number of overridden methods
	NOC_Pr	Number of children in the project	Concern	CDC	Concern diffusion over components
	NOC_Pa	Number of children in the package		CDO	Concern diffusion over operations

operations. It is divided into two parts, supervised and unsupervised. Owing to missing value processing independent of category, we take the unsupervised filter to process the data.

First, we use the ReplaceMissingValues filter to solve the problem of the missing data. The ReplaceMissingValues function is used to replace the missing values with the average values. Next, for the problem of redundant metrics in the data, we adopt the AttributeSelection filter in Weka to select metrics and remove metrics unrelated to output variables. In the AttributeSelection function, we use the "InfoGainAttributeEval" as an evaluator and "Ranker" as a search strategy. They rank metrics by calculating the information gain value of metrics. Finally, we remove 10 metrics that are listed at the end with an information gain value of zero as they have no effect on the classification of smells. The deleted metrics include NOEC, NOIC, NRSS, AID, HIT, NCC, NOCP, PPrivM, PProtM, and PPubM. The remaining 70 metrics are used as experimental data to construct the Logistic Regression model.

3.3 Logistic Regression

There are many optional machine learning models such as Logistic Regression, NaiveBayes, and RandomForest. This study uses Logistic Regression to predict code smells. In the experiment, we compare the performance of different machine learning models. As a generalized linear analysis model, Logistic Regression is one of the most classical binary classification algorithms. The principle of Logistic Regression is to establish the cost function in the face of a regression or classification problem and then iteratively solve the optimal model parameters through the optimization method. The traditional method for solving Logistic Regression parameters is gradient descent. After the cost function is constructed, a small part of the partial derivative direction (the direction with the fastest descending speed) is stepped forward each time until the lowest point is reached after N iterations. The threshold value obtained by Logistic Regression can be mapped as a decision boundary of the plane. The greatest advantage of Logistic Regression is that its output can not only be used for classification but also represent its corresponding probability. It can map the original output from 0 to 1, thus completing the estimation of the probability.

Here, we consider the detection of code smell as a binary classification problem. There are only two output results of Logistic Regression, which respectively represent the positive and negative categories. The independent predictor of this experiment contains a series of extracted quality metrics, and the dependent variable is whether a class contains code smells. First, we build and use a polynomial Logistic Regression model with a ridge estimation. In order to improve estimation accuracy and make the model relatively stable, we introduce a ridge estimation. It adds a regularization term to the loss function and a coefficient adjusting the weight of linear regression term and the regularization term, which reduces the regression coefficient and the variance of the estimated value. The parameters of the ridge estimation are usually set by the gradient descent method. Common gradient descent methods include the conjugate gradient descent method and the quasi-Newton method. Here we use the conjugate gradient descent method. Because compared with the quasi-Newton method, the conjugate gradient descent method can converge faster with smaller memory. It is also more stable than the quasi-Newton method in calculating parameter problems. To determine the direction of the gradient descent, we set BatchSize to be processed to the full dataset pattern when performing the prediction. Since the full dataset can better represent the sample population, it is more accurate toward the direction of the extreme value and faster to process the data.

3.4 Parameter optimization

Machine learning algorithms usually have many parameters. These parameters have a huge effect on the classifier algorithm. However, manually finding the best parameter configuration is very difficult and inefficient. We solve the above problem with the CVParameterSelection classifier. CVParameterSelection [19] is a meta-classifier for parameter optimization selection. It finds the optimal values for one or more parameters within a specified range. It uses a cross-validation method to optimize the selection of any number of parameters and automatically run all possible combinations of parameters. When performing cross-validation, CVParameterSelection executes parameter composition using only the collapsed training dataset. We use Weka to implement this method and find the best parameter configuration for the experiment.

As shown in Figure 1, in the process of parameter optimization, we first enter the parameters to be optimized in the ArrayEditor module of the CVParameterSelection classifier, and set the numerical range and step size. The parameters to be tuned include the minimum number of bits, BatchSize, the number of iterations, the ridge value, DonotCheckCapabilities, and NominalToBinaryFitter. Detailed terminology explanations for parameters are listed in Table 1. In the visualization area, it outputs the parameters that work best within this test range after running. From the optimal parameter results, CVParameterSelection sets the minimum number of bits in the model for outputting numbers to four, and BatchSize is set to the full dataset. The maximum number of iterations to be executed and the ridge value are set to different standards depending on the smell. DonotCheckCapabilities is set to False to check the capabilities of the classifier before building it. The default value is replaced by ReplaceMissingVakuesFitter. NominalToBinaryFitter converts the nominal attribute type to a numeric attribute.

4 Experimentation

The experimental process is as follows: first, we collect the dataset. A total of 17 labeled software projects are collected. We define our dataset as

$$Q = \{X_1, X_2, X_3, \ldots, X_i\}, \quad i = 1, 2, 3, \ldots, \tag{1}$$

Table 3 Project information for the dataset

Project	KLOC	Packages	Classes	Divergent Change	Parallel Inheritance	Shotgun Surgery
Aardvark	25	11	103	0	0	0
And Engine	20	90	596	0	0	0
frameworks-base	770	253	2766	2	3	0
cassandra	117	43	826	3	0	0
commons-codec	23	7	103	0	0	0
commons-logging	23	17	61	1	3	0
derby	166	194	1746	0	0	0
Eclipse Core	162	843	1190	0	7	0
Google Guava	16	25	153	0	0	0
HealthWatcher	6	26	132	12	0	7
James Mime4j	280	26	250	1	0	0
MobileMedia	3	10	51	4	0	3
sdk	54	198	268	1	12	0
support	59	22	246	1	0	1
telephony	75	17	223	0	0	0
Tomcat	336	154	1284	1	10	1
tool-base	119	69	532	0	0	0

where Q is our training dataset, consisting of a series of class files, X_i represents a class which is marked as a code smell (or not).

Then, for each X_i in Q, its features are extracted as a set of metric values. We define these metrics as

$$M = \{(x_1, y_1), (x_2, y_2), \ldots, (x_i, y_i)\}, \quad i = 1, 2, 3, \ldots, \tag{2}$$

where M is the set of software quality metrics, x_i is the class in which the measures are calculated, and y_i marks whether the class is a bad smell, with a value of 0 or 1.

Next, we construct a Logistic Regression model for training. We perform parameter optimization to select the optimal parameters. We apply the optimized model on the testing dataset for evaluation.

4.1 Dataset collection

In our research, we consider the Landfill dataset labeled by Palomba et al. [20] and the HealthWatcher and MobileMedia projects collected by Padilha et al. [17]. The Landfill dataset has 20 projects. We remove five projects since the marked class files cannot be found in the snapshot versions of these projects. Together with the HealthWathcer and MobileMedia projects [17], we collect a total of 17 projects with 2254 KLOC and 22151 classes. The lines of codes (KLOC), the number of packages, the number of classes of these projects, and the number of classes affect by code smells are listed in Table 3.

In these projects, change barrier based code smells have been manually labeled by previous studies. The size of these projects varies from 3 to 770 KLOC, which reflects the characteristics of real-world industrial applications. We also adopt 7 projects that are not affected by code smells to ensure the integrity of experimental data. As can be seen from Table 3, instances affected by code smells are evenly distributed among projects from small to large. We mix all the labeled results in these unequal-sized projects together to form the final dataset, thus avoiding the bias owing to the size of the project. At last, we get a total of 296 change barrier related instances. The dataset of Divergent Change contains 27 positive instances and 81 negative instances. Parallel Inheritance's dataset contains 35 positive instances and 105 negative instances. Shotgun Surgery contains 12 positive instances and 36 negative instances. We divide these datasets into the training set, validation set, and testing set in a ratio of 4:2:4. The training and validation sets are used to train the Logistic Regression model and adjust parameters. The testing set is used to evaluate our model.

4.2 Comparison algorithms

In this experiment, we compare our algorithm with five classifiers commonly used in the literature [8], i.e., SMO, J48 Decision Tree, NaiveBayes, RandomForest, and JRip. For each classifier, we perform parameter optimization with CVParameterSelection for fair comparison.

For SMO, we select the best kernel configuration for classification from PolyKernel, Puk Kernel, and RBFKernel. Since SMO implements sequential minimal optimization, its output coefficient belongs to standardized data. Other SMO parameters include normalize training data, standardize training data, and no normalize/standardize. They represent converting data to normalized data, converting data to standardized data, and not converting data, respectively. For J48, both trimming and unpruning are considered. After pruning, we solve the zero-probability problem according to the number of Laplace smooth cotyledons. The pruned decision tree can reduce the complexity of the tree and prevent it from over-fitting. For the numerical properties in NaiveBayes, there are two implementations that use kernel estimation and discretization to convert them into nominal attributes. As a rule-based algorithm, JRip outputs a set of rules just like the decision tree algorithm. It is necessary to complete the four stages of construction, growth, pruning, and optimization. Finally, it summarizes detection rules by repeated incremental pruning to produce error reduction (RIPPER).

4.3 Evaluation metric

The results in all the experiments in this paper are evaluated using precision, recall, F-measure, and ROC area.

Precision is the ratio of correctly predicted bad smell instances to all predicted bad smell instances.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{3}$$

where TP is an instance of correctly predicted bad smell and FP is an instance of incorrectly predicted bad smell.

Recall is the ratio of correctly predicted bad smell instances to all actual bad smell instances.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{4}$$

where FN is an example of an incorrect prediction that is not a bad smell.

F-measure is the harmonic mean of precision and recall, which comprehensively considers the two evaluation metrics,

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{5}$$

The ROC curve area reflects a comprehensive measurement of the two indicators of specificity and sensitivity. The specificity is the proportion of instances that are incorrectly predicted as not bad smell to all instances that are not actually bad smell. The sensitivity is by definition the same as recall. In general, the larger the ROC curve area is, the better the performance of the model is.

4.4 Research questions

In order to verify the validity of our method, we investigate and answer the following research questions.

RQ1: What is the computation time of the Logistic Regression model and other machine learning algorithms?

RQ2: Is the precision, recall, F-measure and ROC area of the Logistic Regression algorithm superior to other machine learning algorithms in the detection of code smell?

RQ3: Is the performance of the algorithm built by existing literature metrics better than the performance of the algorithm built by our metrics?

RQ4: How much influence does the removal of correlation metrics have on the experimental results?

RQ5: For each bad smell, which metrics are the most important to the classifiers?

RQ6: How does our algorithm perform for cross-project detection?

5 Experimental result

This section analyzes and discusses our experimental results, and answers the six research questions.

Table 4 Comparison of the computation time between Logical Regression and other machine learning algorithms

Machine learning algorithm	Divergent Change		Shotgun Surgery		Parallel Inheritance	
	Training time (s)	Testing time (s)	Training time (s)	Testing time (s)	Training time (s)	Testing time (s)
SMO	0.08	0.05	0.04	0.02	0.48	0.14
NaiveBayes	0.10	0.05	0.04	0.03	0.17	0.13
J48	0.08	0.05	0.04	0.01	0.10	0.05
JRip	0.08	0.05	0.04	0.03	0.09	0.05
RandomForest	0.07	0.05	0.04	0.02	0.10	0.05
Logistic Regression	0.08	0.04	0.04	0.02	0.09	0.04

Table 5 Comparison of Logistic Regression with five other machine learning algorithms

Machine learning algorithm	Divergent Change				Shotgun Surgery				Parallel Inheritance			
	Precision (%)	Recall (%)	F-measure (%)	ROC	Precision (%)	Recall (%)	F-measure (%)	ROC	Precision (%)	Recall (%)	F-measure (%)	ROC
SMO	72.7	72.7	72.7	0.817	100	83.3	90.9	0.917	81.8	81.8	81.8	0.887
NaiveBayes	60.0	81.8	69.2	0.901	83.3	83.3	83.3	0.949	60.0	54.5	57.1	0.788
J48	60.0	81.8	69.2	0.838	71.4	83.3	76.9	0.821	69.2	81.8	75.0	0.885
JRip	64.3	81.8	72.0	0.885	75.0	50.0	60.0	0.712	88.9	72.7	80.0	0.853
RandomForest	77.8	63.6	70.0	0.903	100	75.0	85.7	0.817	77.8	63.6	70.0	0.937
Logistic Regression	81.8	81.8	81.8	0.895	100	83.3	90.9	1.000	81.8	81.8	81.8	0.935

5.1 RQ1: Computation time comparison

RQ1 discusses the computation time of different algorithms on code smell detection. The experiment is performed on Windows10 Intel(R) Core i7-7700 CPU@3.60 GHz processors. 296 data instances are adopted, and 70 metrics are used to build the model. Table 4 shows the training time and testing time for Logistic Regression and five other machine learning algorithms. The experimental result is the average of ten times running.

From Table 4, the computation time of Logistic Regression is 0.02–0.09 s, which is less than other algorithms. In the detection of Parallel Inheritance, the difference between Logical Regression and SMO in training time is 0.39 s at most, and the difference in testing time is 0.1 s. We can observe that the training time and testing time of Logistic Regression on the three change barrier code smells are slightly less than other algorithms, but the overall difference is not significant.

Answer to RQ1: Overall, all the algorithms can detect change barriers efficiently.

5.2 RQ2: Algorithm comparison

RQ2 evaluates the performance of different algorithms on change barrier detection. Table 5 shows precision, recall, and F-measure for Logistic Regression and five other machine learning algorithms. The precision of our model is between 81.8% and 100%, recall is between 81.8% and 83.3%, and F-measure is between 81.8% and 90.9%. For Divergent Change and Parallel Inheritance, precision, recall, and F-measure are both 81.8%. Precision of Shotgun Surgery is as high as 100%, recall is 83.3%, and F-measure is 90.9%. For ROC, ROC values of the three smells are all over 0.8, indicating the good performance of our model.

Compared with other machine learning algorithms, Logistic Regression performs better in detecting the three change barrier code smells on the four indicators. In order to improve the credibility of the results, we carry out T-test [21], a statistical test method, to verify the difference between five algorithms and Logistic Regression in precision, recall, F-measure, and ROC. The results of the T-test are shown in Table 6. In general, if the P-value is less than 0.05, it means the two samples are considered to have significant differences.

The results of Logistic Regression in the Divergent Change detection in Table 5 show a higher F-measure. In statistical testing, only the difference between SMO and Logistic Regression is significant. Since the other four algorithms are identical to the Logistic Regression in recall and ROC respectively, there is no statistically significant difference between them and Logistic Regression. For Shotgun Surgery, SMO and Logistic Regression achieve the same good effect, while the F-measure values of NaiveBayes and RandomForest are slightly lower. From the statistical test, Logistic Regression is significantly different

Table 6 The P-value of T-test of five other machine learning algorithms

Machine learning algorithm	Divergent Change	Shotgun Surgery	Parallel Inheritance
SMO	0.0251	0.7081	0.7359
NaiveBayes	0.2688	0.1895	0.0121
J48	0.1411	0.0209	0.2769
JRip	0.2585	0.0054	0.6659
RandomForest	0.7943	0.2768	0.2802

Table 7 Logistic Regression and comparison classifiers trained on metrics in the existing literature

Machine learning algorithm	Divergent Change				Shotgun Surgery				Parallel Inheritance			
	Precision (%)	Recall (%)	F-measure (%)	ROC	Precision (%)	Recall (%)	F-measure (%)	ROC	Precision (%)	Recall (%)	F-measure (%)	ROC
SMO-o	66.7	30.8	42.1	0.621	50.0	33.3	40.0	0.635	66.7	42.9	52.2	0.679
NaiveBayes-o	66.7	22.2	33.3	0.595	50.0	33.3	40.0	0.750	57.1	57.1	57.1	0.776
J48-o	50.0	44.4	47.1	0.513	12.5	33.3	18.2	0.375	66.7	42.9	52.2	0.679
JRip-o	66.7	22.2	33.3	0.596	12.5	33.3	18.2	0.375	62.5	35.7	45.5	0.643
RandomForest-o	66.7	44.4	53.3	0.809	40.0	66.7	50.0	0.646	66.7	28.6	40.0	0.619
Logistic Regression-o	50.0	33.3	40.0	0.552	40.0	66.7	50.0	0.646	56.3	64.3	60.0	0.757

from J48 and JRip. In this way, it is confirmed that Logistic Regression can achieve better detection effect than these two algorithms. The other three algorithms are not significantly different from Logistic Regression. As for Parallel Inheritance, apart from Logistic Regression, SMO and JRip with F-measure above 80% can also achieve better effects. The difference between them and Logistic Regression is not significant. NaiveBayes has a P-value of less than 0.05, which is not effective for Parallel Inheritance detection.

From the above analysis, Logistic Regression performs well in many cases. There are several reasons why we analyze its good performance. First, Logistic Regression is more suitable for this continuous and categorical independent variable, and the dependent variable obeys the binomial distribution. Secondly, compared with other machine learning methods, Logistic Regression does not as sensitive to the correlation of metrics as NaiveBayes does, and does not require the linear relationship between independent variables and dependent variables. Thirdly, Logistic Regression gets a probability feedback and is anomaly-tolerant, compared with Decision Tree and SMO. Therefore, Logistic Regression performs better on smell detection in this study. In our framework, the promising performance is achieved by both the domain-specific metrics and the Logistic Regression model. However, through experiments, we find that under the same metrics, the average performance of Logistic Regression on different indicators is better.

Answer to RQ2: In summary, Logistic Regression is superior to SMO, Decision Tree, NaiveBayes, JRip, and RandomForest in terms of precision, recall, and F-measure for change barrier detection.

5.3 RQ3: Metric comparison

RQ3 is designed to verify whether our proposed metrics have a positive effect on the detection of code smells. We use the metrics in Reshi et al.'s work [22] to construct machine learning models to compare with our proposed metrics. The literature applies the machine learning method to predict defects. They use code smell as the influencing factor, that is they use Iplasma to analyze code smells in software projects and extract software metrics for training machine learning algorithms. Hence, the defects and code smells here can be analogized. Code smells can lead to defects, which is one of the methods for discriminating defects. Many studies take advantage of code smells to analyze whether there are defects [23]. Therefore, we use the metrics in [22] for comparison.

Table 7 reports the precision, recall, F-measure, and ROC curve area of machine learning algorithms trained on the metrics in the existing literature. We use -o to represent these algorithms constructed with the metrics in reference [22]. Table 8 shows the P-value of the T-test between the corresponding algorithms constructed by different metrics. From the experimental results, the precision, recall, and F-measure achieved by Logistic Regression-o are always below 60% on Divergent Change, Shotgun Surgery, and Parallel Inheritance. It is far less effective at identifying change barriers than we are.

For the other five algorithms, the precisions of NaiveBayes-o and JRip-o increase by 6.7% and 2.4%, while the recall decreases, and the overall F-measure is lower than those of our method in the classification

Table 8 The P-value of T-test of classifiers trained on metrics in this paper and classifiers trained on metrics in the existing literature

Machine learning algorithm	Divergent Change	Shotgun Surgery	Parallel Inheritance
SMO-o	0.0309	0.0009	0.0059
NaiveBayes-o	0.0539	0.0088	0.9619
J48-o	0.0046	0.0002	0.0275
JRip-o	0.0387	0.0033	0.0085
RandomForest-o	0.2013	0.0098	0.0514
Logistic Regression-o	0.0003	0.0021	0.0069

of Divergent Change. For Shotgun Surgery, the precision and recall of the five algorithms are all below 50%. In Parallel Inheritance recognition, the precision of NaiveBayes-o decreases, and the recall increases. The overall F-measure does not change. The measurement values of the other four algorithms are reduced, which are significantly lower than our method. From a statistical point of view, in addition to the RandomForest-o and NaiveBayes-o in the Divergent Change and Parallel Inheritance detection, other algorithms present a significant difference. To sum up, our method as an integrated approach is superior to the algorithm model constructed following the existing study [22].

Answer to RQ3: Metrics in the existing literature are not effective to detect change barriers. The metrics related to bad smell characteristics proposed in this paper promote the detection results, which proves that our metrics play a very important role in change barrier detection.

5.4 RQ4: Metric correlation evaluation

RQ4 analyzes the correlation between metrics and their impact on experimental results. We analysis the pairwise correlation between measures using the Spearman rank correlation coefficient (ρ), which analyzes the closeness of the relationship between the two groups of variables with linear correlation analysis. It is calculated based on the rank. Its value is between -1 and 1. In the meantime, the closer the value is to 1 (or -1), the more positive/negative the correlation is. We respectively define two sets of metrics as X and Y, and the dataset size is n. The reordered positions of the elements x and y in the two sets are respectively denoted by x_i and y_i. ρ is calculated as

$$\rho = 1 - \frac{6\Sigma d_i^2}{n(n^2-1)}, \tag{6}$$

where d_i represents the ranking difference between elements x and y, $d_i = x_i - y_i, 1 \leqslant i \leqslant n$.

We divide the experiments into two groups according to the value of ρ. The first group of experiments removes one of a pair of metrics with $\rho > 0.9$, leaving only a single metric. The second group eliminates metrics with $\rho > 0.8$. We reconstruct the algorithm model for the experiment of code smell recognition after deleting correlation metrics and observe the effect of correlation metrics.

Figures 2–4 show the correlation analysis results for each bad smell. We keep the metrics that are easy for calculation in each correlative group. There are 45 pairs of correlation metrics over 0.8 in Divergent Change and 19 pairs over 0.9. Shotgun Surgery has 40 pairs of correlation metrics over 0.8 and 18 pairs over 0.9. In addition, there are 33 pairs of Parallel Inheritance with over 0.8 and 21 pairs with over 0.9. We remove 24, 23, 19 metrics with all correlations above 80% for each of the three code smells. For the 90% threshold, we delete 12 metrics for all three code smells.

Tables 9 and 10 show the experimental results after removing the 80% and 90% correlation metrics respectively. In Table 10, except for the ROC curve area floating above 0.01, there is no change in the precision, recall, and F-measure. After removing the 80% correlation metrics, Divergent Change experiences a decrease in precision and an increase in recall, but overall, its F-measure increases by 1.5%. The ROC curve area floats above 0.005. Shotgun Surgery and Parallel Inheritance remain unchanged in value. The P-value of both groups is greater than 0.05, which is not significantly different from our method. On the whole, the removal of the correlation metrics has little effect on the original experimental results, but the learning time of classifiers is shortened.

Answer to RQ4: For the three bad smells, there are 118 pairs of metrics with $\rho > 0.8$ and 58 pairs with $\rho > 0.9$. The deletion of the correlation metrics has no significant effect on the experimental results.

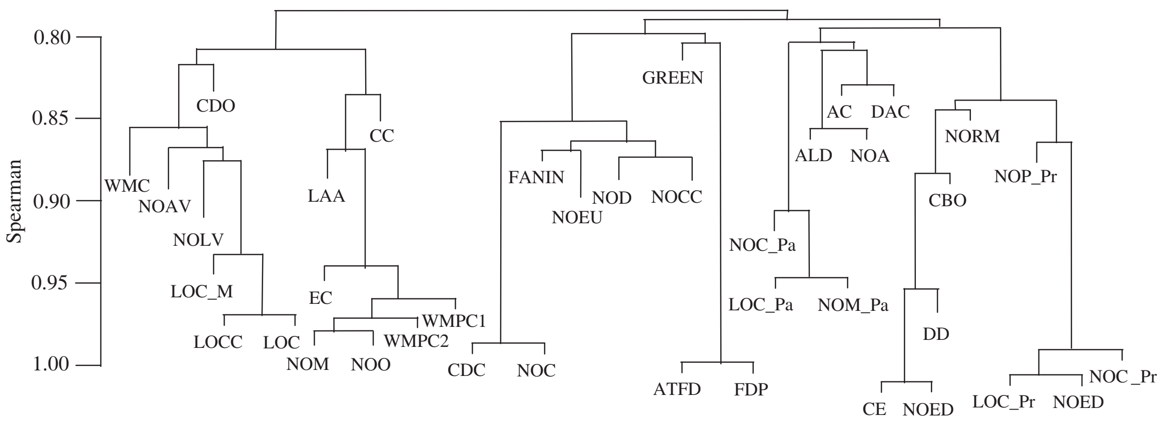

Figure 2 The correlation analysis results of Divergent Change.

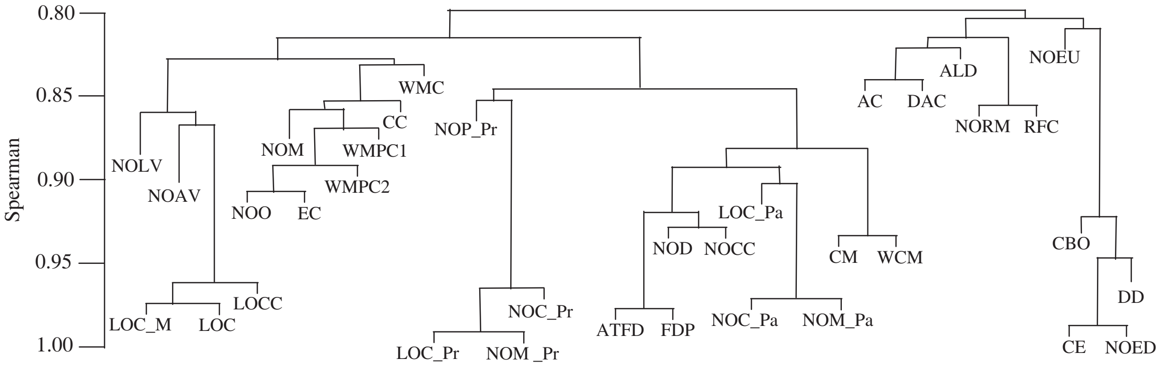

Figure 3 The correlation analysis results of Shotgun Surgery.

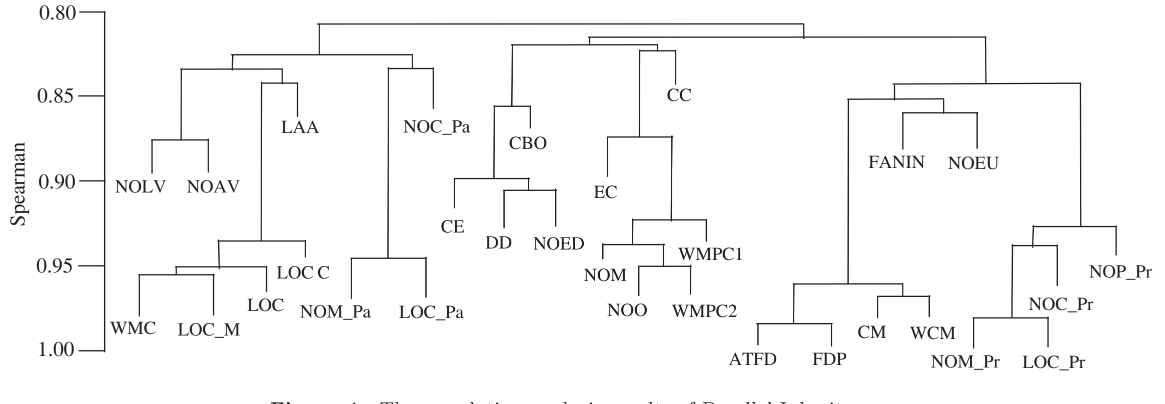

Figure 4 The correlation analysis results of Parallel Inheritance.

5.5 RQ5: Metric importance evaluation

RQ5 discusses the importance of the metrics for change barrier detection. To verify which metrics play a key role in the detection of each bad smell, we perform a metric importance verification experiment. In the output of the Logistic Regression model, we can get the correlation coefficient of the metric index, so as to judge the importance of the metric. In addition, the J48 algorithm constructs a decision tree in which top nodes in trees represent important metrics. We compare the importance analysis results of Logistic Regression and J48 to analyze their correlations and differences. Table 11 reports the top ten metric rankings for the three code smells in the Logistic Regression correlation coefficient. Figures 5–7 show the decision trees generated for the three code smells.

For the Divergent Change, the most important metrics are LOC_M, NOP, and CDC. In Figure 5, the root node of the decision tree is LOC_M. NOP is located on the third layer of the tree. This shows that the

Table 9 Results without the 0.8 correlation metrics

Code smell	Precision (%)	Recall (%)	F-measure (%)	ROC area	P-value
Divergent Change	76.9	90.9	83.3	0.903	0.6856
Shotgun Surgery	100	83.3	90.9	1.000	1.0000
Parallel Inheritance	81.8	81.8	81.8	0.943	0.9642

Table 10 Results without the 0.9 correlation metrics

Code smell	Precision (%)	Recall (%)	F-measure (%)	ROC area	P-value
Divergent Change	81.8	81.8	81.8	0.881	0.8927
Shotgun Surgery	100	83.3	90.9	1.000	1.0000
Parallel Inheritance	81.8	81.8	81.8	0.937	0.9908

Table 11 Results of the top ten metric coefficients

Number	Divergent Change		Shotgun Surgery		Parallel Inheritance	
	Metric	Coefficients	Metric	Coefficients	Metric	Coefficients
1	is interface	1.4067	TCC	46.5202	BUR	2.4688
2	is root	1.2052	is static	7.6223	is static	1.6495
3	PNAS	1.1199	DOIH	2.1524	is leaf class	1.3356
4	is leaf class	1.0874	DIT	1.6331	is interface	1.1756
5	BUR	1.0834	NAM	1.3989	MIC	1.1132
6	AMW	1.0525	NOC	1.3080	NOCON	1.1101
7	DAC	1.0422	FDP	1.1255	CRIX	1.0605
8	CDC	1.0333	AMW	1.1252	DAC	1.0536
9	CRIX	1.0265	ChC	1.0140	NAM	1.0448
10	MIC	1.0203	CM	1.0035	NOC_Pa	1.0007

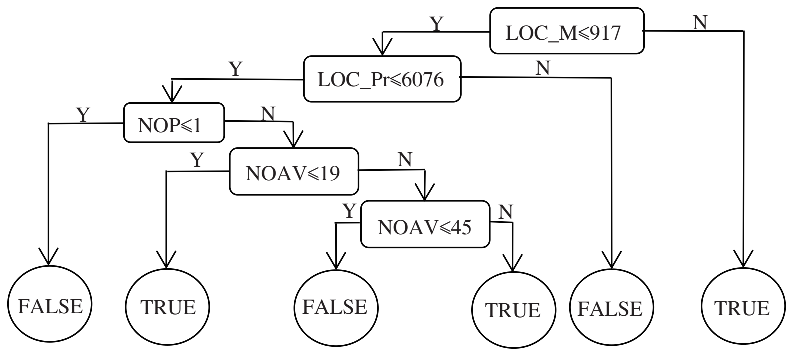

Figure 5 The decision tree generated by Divergent Change.

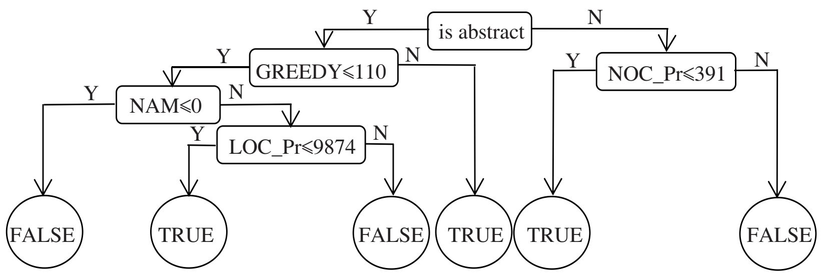

Figure 6 The decision tree generated by Shotgun Surgery.

larger the number of codes and project size, the more complex the project. Such projects are more likely to have the code smell of Divergent Change. As shown in Table 11, the top three most important metrics for Divergent Change detection are "is interface", "is root", and PNAS, with correlation coefficients above 1.1. Although CDC ranks the eighth in Table 11, its coefficient is not much different from the previous ones. It proves that the concern metric does play a significant role in Divergent Change detection and

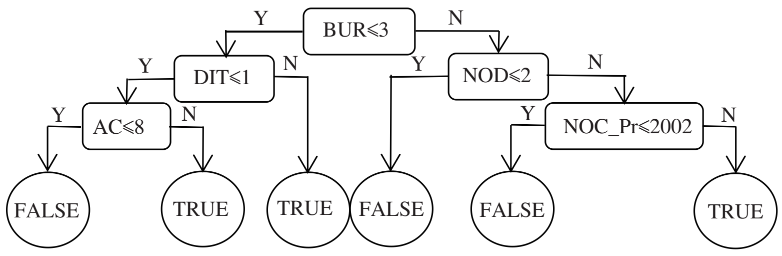

Figure 7 The decision tree generated by Parallel Inheritance.

Table 12 Results of the cross-project experiment

Code smell	Precision (%)	Recall (%)	F-measure (%)	ROC area	P-value
Divergent Change	100	50.0	66.7	0.833	0.4549
Shotgun Surgery	100	66.7	80.0	1.000	0.4786
Parallel Inheritance	60.0	30.0	40.0	0.603	0.0037

has a positive influence. The primary purpose of the CDC (concern diffusion over components) is to achieve the number of classes of concerns and the number of other classes accessing them. If CDC is spread widely enough, a project is easier to have Divergent Change.

For the Shotgun Surgery, TCC, NAM, ChC, and CM are critical for detection. The coefficient of TCC is 46.5202. The number of public method pairs of classes with common attribute usage affects the detection of Shotgun Surgery with a high probability. NAM ranks the fifth in the metric coefficient list, and is in the third level in the decision tree, which indicates the importance of NAM. In other words, if there is a declared non-inherited accessor method in the interface of a class, Shotgun Surgery is likely to occur. The number of different methods and classes that call the measured method can affect the classification of Shotgun Surgery. The more methods and classes that call the measured method, the more likely Shotgun Surgery exists.

The most important indicators for Parallel Inheritance are BUR, DIT, and NOC_Pa. BUR ranks first in the coefficient ranking and locates at the first level of J48 Decision Tree, which is the most critical metric. As the root node, BUR determines the branch structure of the decision tree. Figure 7 is a four-level tree structure rooted by BUR. 1-level judges whether BUR exceeds 3 or not. If the inheritance tree depth exceeds 1, the program will have Parallel Inheritance. This fully demonstrates that the deeper the Inheritance tree is, the more descendants it has, the more likely Parallel Inheritance will appear. NOC_Pa and NOC_Pr in Table 11 and Figure 7 show the number of subclasses in the entire project or package that can facilitate Parallel Inheritance detection. The more subclasses of a class, the more likely it is to have Parallel Inheritance in the project.

Taken together, some metrics are critical for two or more code smell detection, including "is interface", "is static", "is leaf class", BUR, AMW, DAC, CRIX, MIC, and NAM. These metrics have a positive impact on the three code smells. Many metrics reflect domain-specific such as CDC, DIT, NOC, CM, and ChC.

Answer to RQ5: In the Logistic Regression algorithm model and the J48 Decision Tree, the most important metrics with influence include LOC_M, CDC, NOP, TCC, NAM, ChC, CM, BUR, NOC_Pa, and DIT. These metrics play an important role in detecting code smells.

5.6 RQ6: Cross-project detection

In order to re-evaluate the performance of our algorithm, cross-project experiments are carried out. In this experiment, we conduct the leave-one-out evaluation; i.e., we train models with instances from 16 projects and detect change barriers in the remaining one project each time. In addition, other experimental settings are kept unchanged, and the optimal parameter configuration and metrics are still adopted.

Cross-project experimental results are reported in Table 12. As shown in Table 12, the precision of Divergent Change increases, the recall decreases, and the F-measure decreases by 15.1 percentage points overall. Shotgun Surgery's recall drops slightly, while the ROC curve area remains constant. The overall F-measure decreases by 10.9 percentage points. In the detection of Parallel Inheritance, the P-value is less than 0.05, which shows a statistically significant difference. The precision, recall, and F-measure of

Parallel Inheritance all decrease. On the whole, the detection results of these three bad smells are not much changed, and the effect is reduced within the reasonable range of the cross-project experiment. This indicates that the experimental method has good effectiveness and performance.

Answer to RQ6: The overall detection results of Divergent Change, Shotgun Surgery and Parallel Inheritance decrease within the normal range of cross-project experiments.

6 Threats to validity

Generalization. Our datasets are open source systems labeled in [17, 20]. Our experiment is only built on a single version of the system. The experimental results may be one-sided. If further research is needed in future work, we will consider using multiple versions of the system to validate our approach.

Representativeness. The representation of the dataset needs to be considered. We use the publicly available dataset that contains change barrier code smells for experiments. Although our dataset is relatively small, its projects cover a wide range of topics, largely including most categories of industrial environments. The 17 project systems we collect cover a wide variety of examples, from small to large and from simple to complex. They include Apache's open-source projects (Tomcat, Cassandra, Commons Codec, Derby, James Mime4j, and Commons Logging), Android source code (framework-opt-telephony, frameworks-base, frameworks-support, sdk, and tool-base), the Java development platform Eclipse kernel, Google's Java core dependency library Guava, massive information search engine Aardvark and game engine "And Engine", Health Watcher and Mobile Media system. They can be displayed in the vast majority of categories. In future work, we consider labeling more datasets to support our experiments.

Usability. Our algorithm models and measurement characteristics can be used to solve the problem of code smell detection in software projects. It is used to help maintenance personnel quickly find change barriers that appear in programs and restore good and clean code. As a result, the maintenance time and cost could be reduced.

7 Related work

This section introduces the related work on the development of code smell detection techniques and machine learning based approaches.

7.1 Development of code smell detection

In recent years, code smell detection techniques evolve rapidly. Since 2005, numerous automated detection tools have been proposed such as iPlasma [13] and DECOR [24]. Some tools like JDeodorant [25] even support the reconstruction of code smells. However, the precision of these tools is different, and many false positives are reported. Since 2008, many search-based techniques have been investigated like parallel evolutionary algorithm (P-EA) [3], competitive co-evolutionary algorithm (CCEA) [26], hybrid particle swarm optimisation with mutation (HPSOM) [27], and genetic programming (GP) [28]. They are widely used in code reconstruction and code smell detection. To some extent, it slightly compensates for the poor precision in previous work. In recent years, machine learning, as the main technology leading the trend of the times, has become one of the most important means of code smell detection. Khomh et al. [29] studied a Bayesian approach and verified that it can identify all bad smells faster than existing methods. Importantly, there are very few instances of false positives. Amorim et al. [21] evaluated the effectiveness of C5.0 in identifying code smells. Kaur et al. [9] researched SVM to detect some common smells.

Nowadays, more and more researchers pay attention to machine learning. They try to use machine learning to solve the problem of code smell detection, which has achieved great success. Subsection 7.2 focuses on the related work of machine learning in the field of code smell detection.

7.2 Machine learning based approaches

In 2005, Kreimer [10] first proposed a combination of machine learning mechanism and metric-based design defect detection method to solve the problem of design flaws in software development. Since then, a new field of machine learning-based code smell detection has been opened.

Fontana et al. [8] compared 16 different machine learning algorithms in order to find the best algorithm model for four common bad smells. They calculated six quality dimensions and custom metrics including size, complexity, cohesion, coupling, encapsulation, and inheritance. They built code smell detection rules that can evaluate new candidates through a sampling learning process. Dario et al. [30] conducted an empirical study. They compared different dataset settings with a previous study [8]. They discussed some potential limitations of current machine learning detection methods. Maneerat et al. [31] used seven machine learning algorithms to predict seven bad smells from a software design model. They extracted 27 different model metrics like Diagrams, Class Employment, Model size, Complexity, Relationship, Inheritance, MOOD, and others. They studied several statistical significance tests including the predictive value of tests and prediction accuracy. Khomh et al. [29] proposed a Bayesian belief network method based on the goal question metric (GQM) to detect anti-patterns. They presented three levels of the model, namely conceptual goals, operational problems, and quantitative measures, designed to retrieve anti-patterns that have symptoms of the problem in a given context.

Most previous work has focused on several smells that are structurally simple and repetitive in software projects. Kaur et al. [9] and Vaucher et al. [11] focused on one or several common bad smells: God Class, Data Class, Feature Envy, Long Method. Khomh et al. [29] and Hassaine et al. [32] studied the detection of anti-patterns, which means poor solutions occur repeatedly in design patterns. The difference between this paper and the existing research work is that we focus on the change barrier type of code smell, and detect the code smell with the object-oriented software quality metric and concern metrics. We evaluate the effectiveness of our approach in predicting this type of smell. On this basis, we also try to analyze the correlation and importance of metrics and observe the influence of metrics on smell recognition.

8 Conclusion

In this paper, we use Logistic Regression and metrics related to smell characteristics to detect three kinds of change barrier code smells on 17 projects. We conduct two comparative experiments. The first experiment compares the computation time and performance between our model and other machine learning algorithms. The second experiment compares the metrics in the literature with those in this paper. In addition, we systematically analyze the correlation and importance of metrics. We also observe the most important metrics for the detection of three code smells. Finally, we conduct a cross-project experimental evaluation to verify the effectiveness of our experiment.

The experimental results show that our method achieves higher performance in change barrier detection. Our experimental results are superior to other algorithms in terms of precision, recall, and F-measure. Moreover, our algorithm with domain-specific metrics is better than the algorithm trained on the existing metrics, which proves that our metrics are indeed effective for smell detection. Finally, we also evaluate our method with a cross-project experiment. The result of the cross-project experiment is lower within a reasonable range, indicating the future work for change barrier detection.

Acknowledgements This work was supported by National Key Research and Development Program of China (Grant No. 2018YFB1003900) and in part by National Natural Science Foundation of China (Grant Nos. 61722202, 61772107).

References

1 Fowler M. Refactoring: Improving the Design of Existing Code. Hoboken: Addison-Wesley Professional, 2018
2 Fontana F A, Ferme V, Zanoni M, et al. Automatic metric thresholds derivation for code smell detection. In: Proceedings of the 6th International Workshop on Emerging Trends in Software Metrics, Florence, 2015. 44–53
3 Ouni A, Kessentini M, Inoue K, et al. Search-based web service antipatterns detection. IEEE Trans Serv Comput, 2017, 10: 603–617
4 Palomba F. Textual analysis for code smell detection. In: Proceedings of the 37th International Conference on Software Engineering-Volume 2, 2015. 769–771
5 Deng C W, Huang G B, Xu J, et al. Extreme learning machines: new trends and applications. Sci China Inf Sci, 2015, 58: 020301
6 Zhou Z-H. Abductive learning: towards bridging machine learning and logical reasoning. Sci China Inf Sci, 2019, 62: 076101
7 Khomh F, Vaucher S, Guéhéneuc Y G, et al. A Bayesian approach for the detection of code and design smells. In: Proceedings of the 9th International Conference on Quality Software, 2009. 305–314
8 Fontana F A, Mäntylä M V, Zanoni M, et al. Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng, 2016, 21: 1143–1191
9 Kaur A, Jain S, Goel S. A support vector machine based approach for code smell detection. In: Proceedings of International Conference on Machine Learning and Data Science (MLDS), 2017. 9–14
10 Kreimer J. Adaptive detection of design flaws. Electron Notes Theor Comput Sci, 2005, 141: 117–136
11 Vaucher S, Khomh F, Moha N, et al. Tracking design smells: lessons from a study of God classes. In: Proceedings of the 16th Working Conference on Reverse Engineering, 2009. 145–154

12 Linders B. Refactoring and Code Smells — A Journey Toward Cleaner Code. 2016. https://www.infoq.com/news/2016/09/refactoring-code-smells/
13 Cristina M, Radu M, Mihancea F, et al. iPlasma: an integrated platform for quality assessment of object-oriented design. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. 77–80
14 Singh P, Singh H. DynaMetrics: a runtime metric-based analysis tool for object-oriented software systems. SIGSOFT Softw Eng Notes, 2008, 33: 1–6
15 Eaddy M, Aho A, Murphy G C. Identifying, assigning, and quantifying crosscutting concerns. In: Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques, 2007. 2
16 Chidamber S R, Kemerer C F. A metrics suite for object oriented design. IEEE Trans Softw Eng, 1994, 20: 476–493
17 Padilha J, Pereira J, Figueiredo E, et al. On the effectiveness of concern metrics to detect code smells: an empirical study. In: Proceedings of International Conference on Advanced Information Systems Engineering, 2014. 656–671
18 Witten I H, Frank E, Hall M A, et al. Data Mining: Practical Machine Learning Tools and Techniques. San Fransisco: Morgan Kaufmann, 2016
19 Staelin C. Parameter Selection for Support Vector Machines. Hewlett-Packard Company, Technical Report HPL-2002-354R1, 2003
20 Palomba F, Di Nucci D, Tufano M, et al. Landfill: an open dataset of code smells with public evaluation. In: Proceedings of 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015. 482–485
21 Amorim L, Costa E, Antunes N, et al. Experience report: evaluating the effectiveness of decision trees for detecting code smells. In: Proceedings of IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), 2015. 261–269
22 Reshi J A, Singh S. Predicting software defects through SVM: an empirical approach. 2018. ArXiv: 1803.03220
23 Soltanifar B, Akbarinasaji S, Caglayan B, et al. Software analytics in practice: a defect prediction model using code smells. In: Proceedings of the 20th International Database Engineering & Applications Symposium, 2016. 148–155
24 Moha N, Gueheneuc Y G, Duchien L, et al. DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng, 2010, 36: 20–36
25 Fokaefs M, Tsantalis N, Chatzigeorgiou A. Jdeodorant: identification and removal of feature envy bad smells. In: Proceedings of 2007 IEEE International Conference on Software Maintenance, 2007. 519–520
26 Boussaa M, Kessentini W, Kessentini M, et al. Competitive coevolutionary code-smells detection. In: Proceedings of International Symposium on Search Based Software Engineering. Berlin: Springer, 2013. 50–65
27 Saranya G, Nehemiah H K, Kannan A. Hybrid particle swarm optimisation with mutation for code smell detection. Int J Bio-Inspired Comput, 2018, 12: 186–195
28 Kessentini M, Kessentini W, Sahraoui H, et al. Design defects detection and correction by example. In: Proceedings of IEEE 19th International Conference on Program Comprehension, 2011. 81–90
29 Khomh F, Vaucher S, Guéhéneuc Y G, et al. BDTEX: a GQM-based Bayesian approach for the detection of antipatterns. J Syst Softw, 2011, 84: 559–572
30 Dario D N, Fabio P, Damian A T, et al. Detecting code smells using machine learning techniques: Are we there yet? In: Proceedings of IEEE 25th International Conference on Software Analysis Evolution and Reengineering (SANER), 2018. 612–621
31 Maneerat N, Muenchaisri P. Bad-smell prediction from software design model using machine learning techniques. In: Proceedings of the 8th International Joint Conference on Computer Science and Software Engineering (JCSSE), 2011. 331–336
32 Hassaine S, Khomh F, Guéhéneuc Y G, et al. IDS: an immune-inspired approach for the detection of software design smells. In: Proceedings of the 7th International Conference on the Quality of Information and Communications Technology, 2010. 343–348