Yet Another Model! A Study on Model's Similarities for Defect and Code Smells

Geanderson Santos[1 [0000−0002−7571−6578]], Amanda Santana[1 [0000−0003−1969−3460]], Gustavo Vale[2 [0000−0002−8879−5797]], and Eduardo Figueiredo[1 [0000−0002−6004−2718]]

[1] Federal University of Minas Gerais, Brazil
{geanderson,amandads,figueiredo}@dcc.ufmg.br
[2] Saarland University, Germany
vale@cs.uni-saarland.de

Abstract. Software defect and code smell prediction help developers identify problems in the code and fix them before they degrade the quality or the user experience. The prediction of software defects and code smells is challenging, since it involves many factors inherent to the development process. Many studies propose machine learning models for defects and code smells. However, we have not found studies that explore and compare these machine learning models, nor that focus on the explainability of the models. This analysis allows us to verify which features and quality attributes influence software defects and code smells. Hence, developers can use this information to predict if a class may be faulty or smelly through the evaluation of a few features and quality attributes. In this study, we fill this gap by comparing machine learning models for predicting defects and seven code smells. We trained in a dataset composed of 19,024 classes and 70 software features that range from different quality attributes extracted from 14 Java open-source projects. We then ensemble five machine learning models and employed explainability concepts to explore the redundancies in the models using the top-10 software features and quality attributes that are known to contribute to the defects and code smell predictions. Furthermore, we conclude that although the quality attributes vary among the models, the complexity, documentation, and size are the most relevant. More specifically, Nesting Level Else-If is the only software feature relevant to all models.

Keywords: Defect Prediction · Code Smells Detection · Explainable Machine Learning · Quality Attributes

1 Introduction

Software defects appear in different stages of the life-cycle of software systems degrading the software quality and hurting the user experience [26]. Sometimes, the damage caused by software defects are in-reversible [45]. As consequence, the software cost increases as developers need time to fix defects [44]. As a result, it is better to avoid them as much as possible. Several studies showed that the presence of code smells and anti-patterns, are normally related to defecting code [25, 35, 50, 52]. Code smells are symptoms of decisions on the implementation that may degrade the code quality [23]. Anti-patterns are the misuse of solutions to recurring problems [9]. For instance, Khomh et al. (2012) found that classes classified as God Classes are more defect-prone than classes that are not smelly. In this paper, we refer to code smells and anti-patterns as code smells.

One technique to mitigate the impact of defects and code smells is the application of strategies that anticipate problematic code [48], usually with the use of machine learning models that predict a defect or code smell [12–14, 27, 36, 46, 48, 53, 74]. Training and evaluating machine learning models is a hard task, since (i) it needs a large dataset, to avoid overfitting; (ii) the process of obtaining the labels and features to serve as input is costly, and it requires the use of different tools to support it; (iii) setting up the environment for training and evaluating models is time-consuming and computationally expensive, even though some tools help to automatize the process, and; (iv) understanding the importance of the features and how they affect the model is complex [40].

With these difficulties in mind, our goal is to identify a set of features that can be used by developers to simplify the process of defect and code smell prediction. To simplify, we aim at reducing the number of features that need to be collected to predict or identify possible candidates to present defects and code smells, through an analysis of model redundancies. To the best of our knowledge, no other studies have investigated similarities between the defect and code smells models. Instead, most studies focus on proposing and assessing the performance of models that predict defects or detect code smells [28, 36, 42, 45]. In this work, we fill this gap through an analysis of which features are redundant or different in models built for defects and seven code smells respectively. Even more, we highlight which quality attributes are relevant to their prediction. This analysis is possible by the use of the SHAP technique, which determines the contribution of each feature for the prediction. As a result, using SHAP allows the verification of the features that contributed the most to the prediction and whether the features had high or low values.

To achieve our goal, we use a subset of 14 open-source Java systems that had its metrics and defects annotated [16, 17]. We then employ the Organic tool [49] to detect nine code smells. We merged three of the smells due to similar definitions. After merging the data, we train and evaluate an ensemble machine learning model composed of five algorithms for each of our targets, i.e., defects and code smells. After evaluating the performance of our ensemble, we apply the SHAP technique to identify which features are relevant for each model. Finally, we analyze the results in terms of: (i) which features are relevant for each model; (ii) which features contribute the most for two or more models to identify redundancies in the models; (iii) which quality attributes are important to the defect and code smell prediction.

Our main findings are: (i) from the seven code smells evaluated, we identified that the most similar models to the Defect are the God Class, Refused Bequest, and Spaghetti Code; (ii) Nesting Level Else-If (NLE) and Comment Density

(CD) are the most important features; (iii) most features have high values, except on Refused Bequest; (iv) we identified sets of features that are common in trios of problems, such as API Documentation (AD), which is important for Defects, God Class, and Refused Bequest; (v) documentation, complexity, and size are the quality attributes that contribute the most for the prediction of defects and code smells; (vi) the intersection of features between the defects and code smells ranges from 40% for Refused Bequest to 60% of the God Class. We also contributed to the community by providing an extension of the previous dataset of defects [16, 17] through the addition of nine smells, available in our online appendix [65]. As a consequence of these analyses, we obtained a smaller set of features that contributes to the prediction of defects and code smells. Developers and researches may train machine learning models with less effort using these findings, or they may use these features to identify possible candidates for introducing defects and code smells.

We organize the remainder of this work as follows. Section 2 describes the background of our work. Section 3 shows how we structured the methodology. Then, Section 4 presents the results of our evaluation comparing the defect model with the code smells. Section 5 discusses the main threats to validity of our investigation. Section 6 presents the related work our investigation is based on. Finally, Section 7 concludes this paper with remarks for further explorations about the subject.

2 Background

2.1 Defects

A software defect represents an error, failure, or bug [1] in a software project, that harm the appearance, operation, functionality, or performance of the target software project [26]. Defects may appear on different development stages [72] and may interrupt the development progress and increase the planned budget of software projects [44]. Furthermore, a software team may discover software defects after code release, generating significantly effort to tackle defects in production [38]. To mitigate these defects in software development, defect prediction may find the defective classes [43, 44, 74] before system testing and release. For instance, if a software team has limited resources for software inspection, a defect predictor may indicate which modules are most likely to be defective.

2.2 Code Smells

Brown et al. [9] proposed a catalog of anti-patterns, that are solutions to recurring problems based on design patterns, but instead of providing reusable code, it impacts negatively on the source code. Later, Fowler [23] introduced the code smells as symptoms of sub-optimal decisions in the software implementation that leads to code quality degradation. Since our defect dataset is class-level, we only consider the problems related to classes. In our work, we considered the following smells: Refused Bequest (RB), Brain Class (BC), Class Data Should be Private (CDSBP), Complex Class (CC), Data Class (DaCl), God Class (GC), Lazy Class (LC), Spaghetti Code (SC), and Speculative Generality (SG). The definitions of the problems presented in this paper are: God Class is a large class that has too many responsibilities and centralizes the module functionality [62]. Refused Bequest is a class that does not use its parent behavior [23]. Spaghetti Code is a class that has methods with large and unique multistage process flow [9]. Due to space constraints, the definitions of all evaluated problems can be found in our replication package [65].

3 Study Design

3.1 Research Questions

In this paper, we investigate the similarities and redundancies between the software features used to predict defects and code smells. We can use this information to simplify the prediction model or identify possible candidates of introducing defects or smells. We employed data preparation to find the software features for the defect and code smell prediction models. Therefore, our main objective is to examine the software features applied for both predictions. Our paper investigates the following research questions.

RQ1. Are the defect and class-level code smell models explainable?
RQ2. Which software features are present in both defect and code smell models?
RQ3. Which software quality attributes are more relevant for the prediction of both defects and code smells?

3.2 Data

Predicting a defect or a code smell is a supervised learning problem that requires a dataset with the values of the independent and dependent variables for each sample. Many datasets were proposed in the literature [13, 32, 45]; however, in this work, the selected dataset portrays a joined version of several resources publicly available in the literature [16–18, 75]. In total, five data sources composes this dataset: PROMISE [66], Eclipse Bug Prediction [84], Bug Prediction Dataset [13], Bugcatchers Bug Dataset [25], and GitHub Bug Dataset [75][3]. The dataset has classes from 34 open source Java projects [78]. Furthermore, the data comprises 70 software features related to different aspects of the code. We can divide the features into seven quality attributes: documentation, coupling, cohesion, clone, size, complexity, and inheritance. We also highlight that the dataset is imbalanced. Only around 20% of the classes have a defect, and for the code smells, the range of classes they affect is between 4 to 16.2%. For these reasons, the dataset has a wide range of software features that may promote

[3] https://zenodo.org/record/3693686

interesting analysis over the defects and code smells. Finally, the open source data facilitates the collection of code smells.

Data Collection. The first step of our study is to collect the data about the code smells to merge with the defect data [16]. We applied the Organic tool [49] to detect the code smells. As all the projects are available on GitHub, we manually cloned the source code matching the project version included in the dataset. Since most of the systems in the original dataset have less than 1000 classes (20 systems), we collected data from the ones with more than 1000 classes (14 projects). We decided to focus on these projects because they represent 75% of the entire data and are readily available on GitHub. Additionally, we matched the name of the detected instances of code smells to the class name present in our defect dataset. Hence, independently of whether a class had a smell or not, we only consider it if the match was found in both datasets. In these cases, we do not consider the class for further investigation. We use this approach to avoid bias as it would be unfair to determine that a class that Organic could not find in the defect dataset is non-smelly.

Table 1. Summary of the data for each project.

Project	Version	Classes	CDSBP	DACL	GC	LC	RB	SC	SG	defects
Ant	1.7	1592	12	161	403	211	57	102	36	330
Broadleaf	3.0	1303	3	231	168	97	66	36	36	277
Camel	1.6	2456	7	115	198	519	53	7	87	550
Elasticsearch	0.9	2605	52	42	380	374	187	88	88	362
Hazelcast	3.3	1443	19	71	74	123	115	26	46	232
JDT	3.4	960	308	44	358	1	54	150	31	197
Jedit	4.3	1108	101	56	331	133	9	144	58	264
Lucene	2.4	500	51	13	96	67	66	36	15	208
Neo4J	1.9	1654	64	20	101	187	67	22	92	18
OrientDB	1.6	880	54	30	181	141	40	58	53	171
PDE	3.4	1130	5	34	206	0	22	56	84	167
POI	3.0	822	6	103	58	130	219	18	17	434
Titan	0.5	765	28	11	75	96	18	29	54	66
Xalan	2.7	1794	102	113	456	298	211	159	60	947
Total		**19012**	**812**	**1044**	**3085**	**2377**	**1184**	**931**	**757**	**4223**
Percentage		**100%**	**4.3%**	**5.5%**	**16.2%**	**12.5%**	**6.2%**	**4.9%**	**4%**	**22.2%**

***CDSBP**: Class Data Should be Private; **DACL**: Data Class; **GC**: God Class; **LC**: Lazy Class; **RB**: Refused Bequest; **SC**: Spaghetti Code; **SG**: Speculative Generality.*

Organic collects a wide range of code smells, including methods and classes ones. However, as the defect dataset is class-level, we only use the code smells found in classes. For this reason, we obtained the ground truth of nine smells, as described in Section 2.2. After collecting the data, we merged three code smells: Brain Class (BC), God Class (GC), and Complex Class (CC) into one code smell. Beyond the similar definitions, we merged the BC and CC to GC due to their low occurrence on the dataset. Hence, we named the code smell as

God Class (GC), since it is more used in the literature [67]. Consequently, we evaluate seven smells in total.

Table 1 shows a summary of the data for each project. The first column presents the project's name. The second column presents the project version included in the dataset. The third column shows the number of classes for each system. Columns 4 through 10 show the number of smells found. The last column presents the number of defects in the system. The Total row presents the absolute number of classes and smelly/defective classes. The Percentage row presents the percentage of classes affected by smell/defect. We can observe from Table 1 that the projects vary in size, with Lucene having the least (500), while Elasticsearch has the most (2605). We also observe that the number of smells and defects varies greatly for each system. For instance, the Xalan system had 456 instances of God Class and presented 947 defects. Meanwhile, even though the Neo4J is a large system, it had only 18 defects, i.e., 1% of its classes had defects.

Code Smells Validation. To validate the code smells collected with Organic, we conducted a manual validation with developers. First, we selected three of the most frequent code smells (GC, RB, and SC), since manual validation is costly and developers have to first understand the code. Then, we elaborate questions about each code smell based on the current literature: God Class (GC) [67], Refused Bequest (RB) [37] and Spaghetti Code (SC) [9]. We then produced a pilot study with four developers to improve the questions using classes that Organic classified as either one of the code smells. This allowed us to verify that the questions are suitable for our goals and whether the surveyed developer understood them. For each instance in our sample, we asked nine questions (3 for each smell). The developer was blind to which code smell they were evaluating and had four possible responses: “Yes”, “No”, “Don't Know”, and “NA” (Not Applicable). The questions and developers' answers can be found in our replication package [65].

To make our validation robust, we calculated the sample size based on the number of instances for each of the three smells in our dataset. We then set a confidence level of 90% and a margin error of 10%. As a result, the sample size should have at least eighteen classes of each target code smell. Furthermore, to avoid biasing the analysis, we determine that two developers should evaluate each instance in our sample. In this case, developers had to validate 108 software classes (54 unique). To validate the 108 software classes, we invited fifteen developers from different backgrounds, including three co-authors. One of the authors was the moderator of the analysis and did not participate in the validation. As there were three questions for each tool, in order to consider the instance as truly containing the smell, developers needed to reach an agreement on two out of three questions. In addition, if the two developers that evaluated a same instance disagree on the presence of the smell, a third and more experienced developer checked the instance to make the final decision. This tiebreaker evaluation was done by two software specialists that did not participate in the previous validation.

In the end, the developers agree that all GC classified by the tool were correct (i.e., 18 out of 18 responses). For RB, the developers agree in 14 out of the 18 software classes (meaning that approximately 77% of developers agree with the tool). Finally, SC is slightly worse, where the developers classified 13 out of the 18 classes as SC. Thus, SC classes achieved an agreement of 72% between the developers and the tool. The results demonstrate that Organic can identify code smells with an appropriate level of accuracy (around 84% of agreement between them). For this reason, we conclude that the Organic data is adequate to represent code smells.

3.3 Quality Attributes

Although the literature proposes many quality attributes to group software features [4, 8, 69], we focus on the quality attributes previously discussed in the selected dataset [16, 17]. These quality attributes cluster the entire collection of software features. Therefore, we separate the aforementioned software features into seven quality attributes: (i) *Complexity*, (ii) *Coupling*, (iii) *Size*, (iv) *Documentation*, (v) *Clone*, (vi) *Inheritance*, and (vii) *Cohesion*. Table 2 presents the quality attributes with their definition and reference. The complete list of software features (66 in total) and the quality attributes are available under the replication package of this study [65].

Table 2. Quality Attributes.

Class	Definition	Reference
Clone	Measure the code cloning. They may be a copy and paste of an existing piece of source code, and may present smaller modifications considering the original code.	[16, 75]
Cohesion	Measure to what extent the source code elements are coherent in the system.	[17, 75]
Complexity	Measure the complexity of source code elements (typically algorithms).	[8, 17, 69]
Coupling	Measure the amount of dependencies of source code elements.	[3, 17, 69]
Documentation	Measure the amount of comments and documentation of source code elements in the system.	[4, 17, 23]
Inheritance	Measure the different aspects of the inheritance hierarchy of the system.	[4, 17, 23]
Size	Measure the basic properties of the analyzed system in terms of different aspects (e.g., number of code lines, number of classes or methods).	[17, 23, 79]

3.4 Machine Learning

The predictive accuracy of machine learning classification models depends on the association between the structural software properties and a binary outcome.

In this case, the properties are the software features widely evaluated in the literature [16, 17], the binary outcome is the prediction if the class is defective or non-defective or if the class presents each of the evaluated smells. To compare the defect and code smell prediction models, we rely on the same set of software features, i.e., the models are trained with the same 66 measures, except on the target, the presence/absence of defect/code smell. We train each machine learning model for each target, each code smell and software defect having its own model. To build these models, we employ a tool known as PyCaret [6] to assist in the different parts of the process, described later. Finally, to test the capacity of the models, we apply five evaluation metrics: accuracy, recall, precision, F1, and AUC [11].

Data Preparation. To build our models, we follow these fundamental steps described in Figure 1. The three rounded rectangles indicate the steps and the actions we performed to build the models. Figure 1 exemplifies the three steps. First, we clean the data (i). Then, we explore the data identifying how better to represent them for our models (ii). After, we prepare the features to avoid overfitting (iii). The process repeats until no modifications on the data is needed.

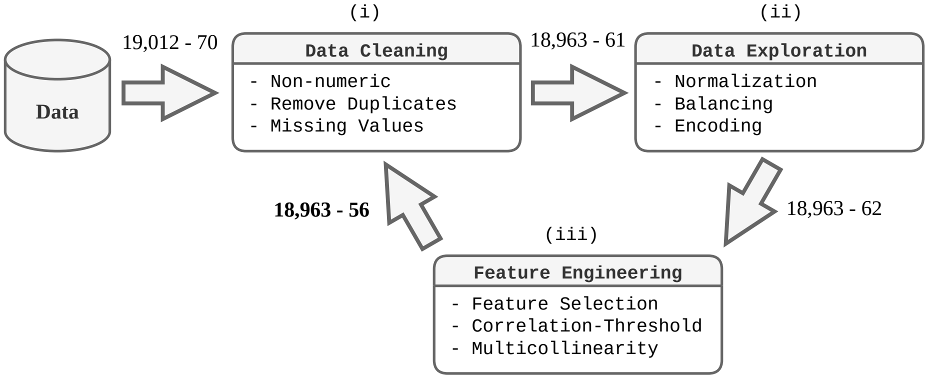


Fig. 1. Data Preparation Process Overview.

Data Cleaning. We first applied data cleaning to eliminate duplicated classes, non-numeric data, and missing values [57]. Hence, it was possible to vertically reduce the data as we removed a small chunk of repeated entries (61 classes). Further, we also reduced the horizontal dimension of the data from 70 to 65 features. We also removed four over-represented software features. These software features gathered information about the exact line and column of the source code a class started and ended. In the end, we executed data imputation to track the missing values, but the dataset had none.

Data Exploration. In the second step of the machine learning processing, we executed the data exploration. Therefore, we used one-hot encoding [39] to the *type* feature, that stores information about the class type. For instance, we created two new features for class and interface types. Subsequently, we applied data normalization using Standard Scaler [60]. Finally, we employed Synthetic Minority Oversampling Technique (SMOTE) [71] to deal with the imbalanced nature of the dataset. Table 1 summarizes the imbalanced nature of the targets compared to the data collection. For instance, from 19K classes, only 757 present Spaghetti Code (almost 4% of classes).

Feature Engineering. In the final step, we applied feature engineering to select the relevant software features. As a result, we executed feature selection, correlation analysis, and multicollinearity thresholds. First, the feature selection technique chooses a subset of software features from the combination of various permutation importance techniques, including Random Forest, Adaboost, and Linear correlation. Second, we checked the correlation between the subset of software features (99% of threshold). In doing so, we removed five software features (LLDC, TNLPA, TNA, TNPA, and TCLOC) because they were highly correlated with other software features (LDC, CLOC, NA, NLPA, and NPA). Additionally, we set the multicollinearity threshold to 85%, meaning that we remove software features with a correlation higher than the threshold. At the end, we ended up with 56 software features.

Training the Models. To build our classifier, we employ a technique known as the ensemble machine learning model [6]. This technique learns how to best combine the predictions from multiple machine learning models. Thus, we use a stronger machine learning model in terms of prediction, since the ensemble combines the prediction power of multiple models. To train the models, we divided the dataset into two sets: 70% of the data is used for training the models, and 30% for testing the models. To assess the performance of our models, we employed a method called k-fold cross-validation. This technique splits the data into K partitions. In our work, we used K=10 [11], and at each iteration we use nine folds for training and the remaining fold for validation. We then permute these partitions on each iteration. As a result, we use each fold as training and as validation set at least once. This method allow us to compare distinct models, helping us to avoid overfitting, as the training set varies on each iteration.

To identify which models are suitable to our goal, we evaluated 15 machine learning algorithms: CatBoost Classifier [6], Random Forest [24], Decision Tree [17], Extra Trees [6], Logistic Regression [30], K-Neighbors Classifier (KNN) [76], Gradient Boosting Machine [73], Extreme Gradient Boosting [64], Linear Discriminant Analysis [6], Ada Boost Classifier [56], Light Gradient Boosting Machine [33], Naive Bayes [81], Dummy Classifier [56], Quadratic Discriminant Analysis [6], and Support Vector Machines (SVM) [15]. Furthermore, to tune the hyper-parameters of each model, we apply a technique called Optuna [5]. Optuna uses Bayesian optimization to find the best hyper-parameters for each model. After experimenting with all the targets, we observed that four models are able of achieving good performance independently of the target (i.e., defects or code smells): Random Forest [24], LightGBM [33], Extra Trees [10], and Gradient Boosting Machine [73]. The data on the performance of the evaluated models can be found in our replication package [65]. To evaluate our models, we focus on the

F1 and AUC metrics. F1 represents the harmonic mean of precision and recall. Additionally, AUC is relevant because we are dealing with binary classification and this metric shows the performance of a model at all thresholds. For these reasons, both metrics are suitable for the imbalanced nature of data [11].

Explaining the Models. The current literature offers many possibilities to explain machine learning models in multiple problems. One of the most prominent techniques spread in the literature is the application of SHAP (SHapley Addictive exPlanation) values [40]. These values compute the importance of each feature in the prediction model. Therefore, we can reason why a machine learning model made such decisions about the specific domain. For this reason, SHAP is appropriate as machine learning models are hard to explain [70], because the features interact in complex patterns to create models that provide more accurate predictions. Consequently, knowing the logic behind a software class is a determinant factor that can help to tackle the reasons behind a defect or code smell in the target class.

4 Results

4.1 Predictive Capacity

Before explaining the models, we evaluate if they can effectively predict the code smells and defects. Even though we originally built models for the entire set of code smells, we observed that only three code smells (God Class, Refused Bequest, and Spaghetti Code) have comparable models to the defects. For this reason, we only present the results of these three code smells. We believe some code smells are not similar to the defect model because they indicate simple code with less chance of having a defect, for instance, Lazy Class and Data Class. As a result, these code smells seem to not have similarities with the defects. The remaining code smells results is available in the replication package [65].

Table 3. Performance of the Machine Learning Models.

Target	Accuracy	AUC	Recall	Precision	F1
God Class	0.944	0.973	0.801	0.844	0.823
Refused Bequest	0.976	0.951	0.645	0.939	0.765
Spaghetti Code	0.971	0.977	0.715	0.692	0.705
Defect	0.843	0.865	0.701	0.609	0.652

Table 3 shows the performance of each ensemble machine learning model with our four targets (i.e., defects and the three code smells). The values in the columns represent the mean of the 10-fold cross-validation. We present in each column the performance for the five evaluation metrics. We can observe from Table 3 that the performance of the ensemble model for the four targets is fairly acceptable, with models presenting an F1 score ranging from approximately 65%

(defect model) to 82% (God Class model). These numbers are similar to other studies with similar purposes [16, 17]. We conclude that the models can predict the targets with acceptable accuracy, as shown by the high AUC values in Table 3. For this reason, we may exploit these machine learning models to explain their prediction using the SHAP technique. In doing so, we can reason about the similarities of the software features that are associated with defects and code smells.

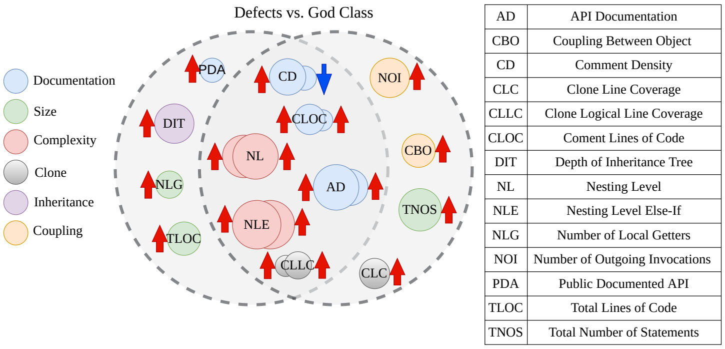
RQ1. The results show that the predictive accuracy of the defect and code smell models can be used to compare the models in terms of their features, with good F1 measures and high AUC. We also found that the class-level code smell models are slightly superior to the defect model in all five evaluation metrics.

4.2 Explaining the Models

This section discusses the explanation of each target model. We rely on SHAP to support the model explanation [40]. To simplify our analysis, we consider the top-10 software features that influence the most on the target in each prediction model. We then compare each code smell model with the defective one. Our goal is to find similarities and redundancies between the software features that help the machine learning model to predict the target code smells and defects. We extract these ten software features from the four target models (i.e., the defect model and the three code smell models).

To illustrate our results, we employ a Venn diagram to check the intersection of features between the four models (Figures 2, 3 and 4). The Venn diagram displays two outside circles, one for the code smell model and another for the defect model. Inside each big circle, we present the top-10 software features that contributed the most to the prediction of the target with inner circles. The color of these inner circles represents the feature's quality attribute. Likewise, the size of the inner circle represents the influence of the feature on the model, meaning that the bigger the size, the most it contributes to the target prediction. On each side of the inner circles, we have an arrow that indicates the direction of the feature value. For instance, a software feature with an arrow pointing up means that the software feature contributes to the prediction when its value is high. On the other hand, a software feature with an arrow pointing down means that the feature contributes to the prediction when its value is low. The software features on the intersection have two inner circles because they have a different impact on each target (i.e., defects and the three code smells). For a better understanding of the acronyms, we show on the right side of each diagram, a table with the acronym and the feature full name of all features that appears on the diagram.

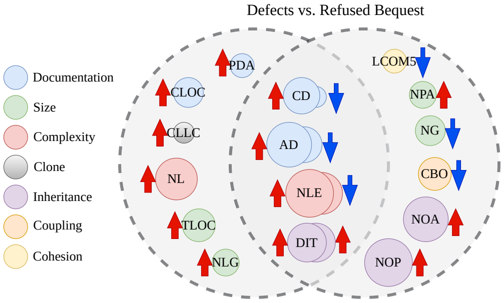
God Class. Figure 2 shows the top-10 features that contribute to the Defects and God Class models, and their feature intersection. We can observe from Figure 2 that the defect model has an intersection with God Class of 6 out of 10 features. This means that 60% of the top-10 software features that contribute the most to their models are the same for both models. These features are: CD, CLOC, AD, NL, NLE, and CLLC; and most of them are related to documentation (3 out of 6) and complexity (2 out of 6). The only difference is for the CD metric, in which it need to have low values to help in the God Class prediction. All remaining software features require a high value to predict a defect or a God Class (see arrows). Moreover, in terms of importance, for both models, the largest inner circles are for NLE, NL, and AD. For the AD feature, its importance is smaller than for the defect model. Meanwhile, for the NLE, the importance of God Class is a bit larger than for Defect. For the NL feature, their importance was equivalent.



AD	API Documentation
CBO	Coupling Between Object
CD	Comment Density
CLC	Clone Line Coverage
CLLC	Clone Logical Line Coverage
CLOC	Coment Lines of Code
DIT	Depth of Inheritance Tree
NL	Nesting Level
NLE	Nesting Level Else-If
NLG	Number of Local Getters
NOI	Number of Outgoing Invocations
PDA	Public Documented API
TLOC	Total Lines of Code
TNOS	Total Number of Statements

Fig. 2. top-10 Software Features for the Defect and God Class Models.

Refused Bequest. Figure 3 shows the top-10 features that contributes the most for the Defects and Refused Bequest models. We can observe from the Venn diagram on Figure 3 that the defect model has an intersection of 40% (4 out of 10 features) with the Refused Bequest model when considering their top-10 software features. The features that intersect are CD, AD, NLE, and DIT. It is interesting to notice that for 3 out of the 4 software features in the intersection, the values that helps to detect the Refused Bequest have to be low (see arrows pointing down), while for the defect model, all of them have to have high values. Furthermore, most of the Refused Bequest features have to be low (6 or 60%). In terms of importance, for the DIT and NLE feature, for both models their importance was similar. However, for both CD and AD, their contribution to the Refused Bequest model was smaller. Additionally, two features that highly contributed to the Refused Bequest are not in the intersection (NOP and NOA), while one (NL) is outside the intersection for the Defect. We also note that there are three features related to the inheritance quality attribute, but only one intersects for both models, the DIT one. We also observe that the size is relevant for both models; however, we do not have any of the features on the intersection of the models. The cohesion aspect was important only for the Refused Bequest model. The documentation attribute, which is relevant for the defect model (4 out of 10), have two of them with small importance (CLOC and PDA). The complexity attribute, mainly the NLE, is also very relevant for both models. CBO is the only coupling metric coupling metric in the Refused Bequest model.

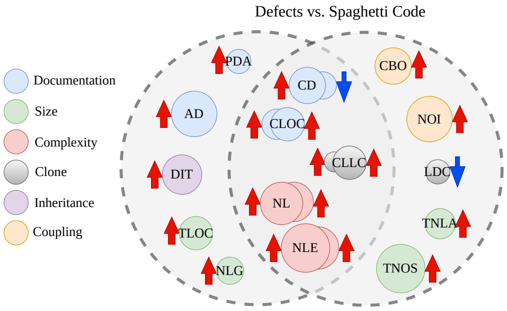


AD	API Documentation
CBO	Coupling Between Object
CD	Comment Density
CLLC	Clone Logical Line Coverage
CLOC	Coment Lines of Code
DIT	Depth of Inheritance Tree
LCOM5	Lack of Cohesion in Methods 5
NG	Number of Getters
NL	Nesting Level
NLE	Nesting Level Else-If
NLG	Number of Local Getters
NOA	Number of Ancestors
NOP	Number of Parents
NPA	Number of Public Attributes
PDA	Public Documented API
TLOC	Total Lines of Code

Fig. 3. top-10 Software Features for the Defect and Refused Bequest Models.

Spaghetti Code. Figure 4 presents the 10 features that are more important to the Defects and Spaghetti Code models. We observe in Figure 4 that the Spaghetti Code model has 50% of intersection with the Defect model. It intersects with the CD, CLOC, CLLC, NL, and NLE software features. For both models, most features have to have high values, except one for the Spaghetti Code, the CD. In terms of importance, the metrics NL, NLE and CLOC had similar results. However, the CD metric contributes less on the Spaghetti Code. Meanwhile, the CLLC feature contributes less to the Defect model. It is interesting to notice that most features that highly contributes to the Spaghetti Code prediction are outside the intersection (NOI, TNOS, and CBO). Furthermore, the complexity quality attribute intersects for both models (i.e., 2 out of 5). In addition, two of the documentation features on the Defect model are important for the Spaghetti Code model. In terms of clone duplication, it also intersects for half of the features of the Spaghetti Code model (CLLC). The size is relevant for both models, but none of the features intersects (2 out of 10 for both models). The features TLOC and NLG appear on the defect model, while the TNOS and TNLA on the Spaghetti Code model. The coupling is exclusive to the Spaghetti Code model, while the inheritance is exclusive to the defect model.

After observing the three figures (Figures 2, 3 and 4), we notice some intersections between the four models. For instance, CLOC was important for Defects, God Class, and Spaghetti Code models, even though the importance for the God Class was smaller (see inner circle sizes). For this trio, we also have that NL and CLLC were important for the three models, but the CLLC has a small contribution in comparison to other features. For the Defects, God Class, and Refused Bequest, we highlight that the AD feature has high importance for all three models. Meanwhile, we also have some intersections between smells models. For



AD	API Documentation
CBO	Coupling Between Object
CD	Comment Density
CLC	Clone Line Coverage
CLLC	Clone Logical Line Coverage
CLOC	Coment Lines of Code
DIT	Depth of Inheritance Tree
NL	Nesting Level
NLE	Nesting Level Else-If
NLG	Number of Local Getters
NOI	Number of Outgoing Invocations
PDA	Public Documented API
TLOC	Total Lines of Code
TLNA	Total Number of Local Attributes
TNOS	Total Number of Statements

Fig. 4. top-10 Software Features for the Defect and Spaghetti Code Models.

the God Class and Spaghetti Code pair, we note that both NOI and TNOS were highly relevant to the models. Finally, CBO was important for the God Class, Refused Bequest, and Spaghetti Code, having a moderate importance.

RQ2. There is a group of software features that intersect between the defect models and the three code smells. More importantly, Nesting Level Else-If (NLE) and Comment density (CD) appear in the four models, although the CD influence is considerably low for the evaluated code smells. Furthermore, CBO is important for all the code smells, but not the defect model.

Figure 5 presents the number of features that correspond to the evaluated quality attributes according to the top-10 features discovered by SHAP. We stack each quality attribute horizontally to facilitate the comparison between them. Hence, our results indicate that researchers do not need to concentrate on all software features to predict defects and the investigated code smells. A subset of features is enough to predict the targets. For instance, software features related to the documentation are the most relevant for the defect and God Class models, with 4 and 3 features on the top-10, respectively. The Refused Bequest model needs software features related to the inheritance (3 features), but size and documentation is also relevant with two features each. Meanwhile, the Spaghetti Code model is the most comprehensive, requiring features linked to documentation, size, complexity, coupling, and clone duplication, with all of them having two features.

Based on the results discussed, we conclude that the four ensemble machine learning models require at least one software feature related to documentation (CD) and complexity (NLE) to predict the target. Hence, future studies about defect and code smell prediction, independently of the dataset and domain, could focus on these two feature collections. Furthermore, as we can observe in Figure 5, considering all the machine learning models evaluated, the documentation, complexity, and size are the most important quality attributes that contribute to the detection of the defects and the code smells.

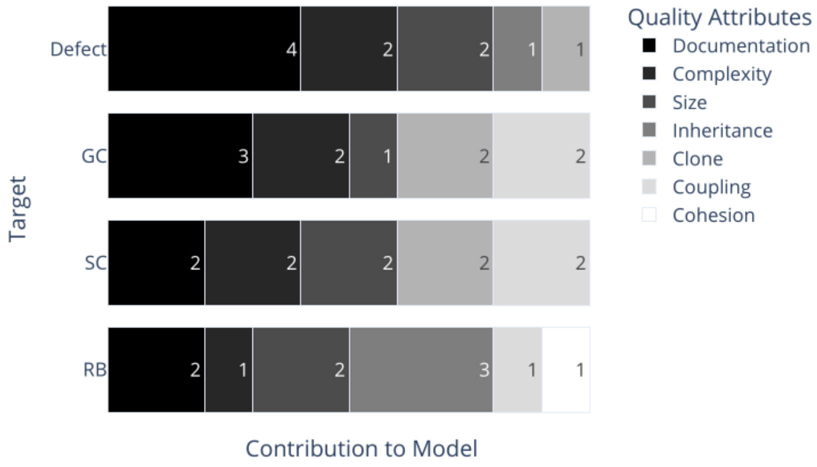


Fig. 5. Comparison between the top-10 features of each target.

RQ3. The most relevant quality attributes to predict defects and code smells vary greatly between each model. For instance, documentation is more important for the Defect and God Class models, while Spaghetti Code has all of its five quality attributes with the same importance, and Refused Bequest prioritizes the inheritance. In general, documentation, complexity, and size contribute more to the prediction of defects and the investigated code smells.

5 Threats to Validity

- **Internal Validity**: In our investigation, the chosen dataset is a potential threat to internal validity [80], as we employed the data documented in the current literature [16, 17]. For this reason, we cannot reason on data quality, as any storing process could insert erroneous data into the dataset, which is common in a complex context such as software development. Furthermore, the use of Organic is also a threat; however, we validated the outcome by asking developers for a statistical sample of the results. Finally, the limited number of projects evaluated may interfere with the model's generalization to other contexts, although we covered 75% of the defect data with the chosen projects.
- **External Validity**: In this study, the external threat to validity [80] connects to the limited number of programming languages we examined to compare the defects and code smell. In this case, we limit the scope to the Java programming language to make our analysis feasible. However, we selected relevant systems that vary in domains, maturity, and development practices. For this reason, we cannot guarantee that our results generalize to other programming languages.

- **Construct Validity**: The use of SHAP is a possible threat to construct validity [80]. There are other tools to explain a machine learning model in the literature, such as Lime [61]. However, we tested only SHAP for our experimentation. Further interactions of this data could compare to other tools that focus on model explainability.
- **Conclusion Validity**: Our study could only match a chunk of the data collected with Organic with the defect dataset. Even though we pulled the same version from GitHub, we could not find some matching classes within the dataset. One of the main reasons for unmatched software classes is probably the refactoring of the class name and dependencies. For this reason, we cannot guarantee how different the results would be if we could match more classes. Furthermore, our study focuses on a diverse set of domains, which is a potential issue for generalization.

6 Related Work

Defect Prediction. Several studies [43, 76] share the ability of applying code metrics for defect prediction. They vary in terms of accuracy, complexity, target programming language, input prediction density, and machine learning models. Menzies et al. [43] presented defect classifiers using code attributes defined by McCabe and Halstead metrics. They concluded that the choice of the learning method is more important than which subset of the available data we use for learning the software defects. In a similar approach, Turhan et al. [76] used cross-company data for building localized defect predictors. They used principles of analogy-based learning to cross-company data to fine-tune these models for localization and used static code features extracted from the source code, such as complex software features and Halstead metrics. They concluded that cross-company data are useful in extreme cases and when within-company data is not available [76].

In the same direction, the study of Turhan et al. [77] evaluate the effect of mixing data from different projects stages. In this case, the authors use within and cross-project data to improve the prediction performance. They show that mixing project data based on the same project stage does not significantly improve the model performance. Hence, they concluded that optimal data for defect prediction is still an open challenge for researchers [77]. Similarly, He at al. [28] investigate defect prediction based on data selection. The authors propose a brute force approach to select the most relevant data for learning the software defects. To do so, they experiment with three large-scale experiments on 34 datasets obtained from ten open source projects. They conclude that training data from the same project does not always help to improve the prediction performance [28]. On the other hand, we base our investigation on ensemble learning to improve the prediction performance and a wide set of software features.

Code Smells Prediction. Several automated detection strategies for code smells, and anti-patterns were proposed in the literature [19]. They also use diverse strategies in their identification. For instance, some methods are based on combination of metrics [49, 58]; refactoring opportunities [20]; textual information [55]; historical data [53]; and machine learning techniques [7, 12, 14, 21, 22, 36, 41, 42]. Khomh et al. [36] used Bayesian Belief Networks to detect three anti-patterns. They trained the models using two Java open-source systems. Maiga et al. [42] investigated the performance of the Support Vector Machine trained in three systems to predict four anti-patterns. Later, the authors introduced a feedback system to their model [41]. Amorim et al. [7] investigated the performance of Decision Trees to detect four code smells in one version of the Gantt project. Differently from these works, our dataset is composed of 14 systems, and we evaluate 9 code smells at the class level.

Cruz et al. [12] evaluated seven models to detect four code smells in 20 systems. The authors found that algorithms based on trees had a better F1 score than other models. Fontana et al. [21] evaluated six models to predict four smells. However, they have used the severity of the smells as the target. They reported high-performance numbers of the evaluated models. Later, Di Nucci et al. [14] replicated it [21] to address several limitations that potentially generated bias on the models' performance. Thus, the authors found out that the models' performance, when compared to the reference study, was 90% lower, indicating the need to further explore how to improve code smell prediction. Differently from previous work on code smell prediction, we are interested in exploring the similarities and differences between models for predicting code smells, in contrast with the models for defect prediction.

Defects and Code Smells. Several works tried to understand how code smells can affect software, evaluating different aspects of quality, such as maintainability [22, 68, 83], modularity [63], program comprehension [2], change-proneness [34, 35], and how developers perceive code smells [54, 82]. Other studies aim to evaluate how code smells impact the defect proneness [25, 29, 35, 50–52]. Olbrich et al. [50] evaluated the fault-proneness evolution of the God Class and Brain Class of three open-source systems. They discovered that classes with these two smells can be more faulty, however, this did not hold for all analyzed systems. Similarly, Khomh et al. [35] evaluated the impact on fault-proneness of 13 different smells in several versions of three large open-source systems. They report the existence of a relationship between some code smells with defects, but it is not consistent for all system versions. Openja et al. [51] evaluated how code smells can make the class more fault-prone in quantum projects. Differently from these studies, we aim to understand whether models build for defects and code smells are similar or not.

Hall et al. [25] investigated if files with smells present more defects than files that do not have them. They found that for most of these smells, there is no statistical difference between smelly and non-smelly classes. Palomba et al. [52] evaluated how 13 code smells affect the presence of defects using a dataset of 30 open-source java systems. They reported that classes with smells have more bug fixes than classes that do not have any smells. Jebnoun et al. [29] evaluated how Code Clones are related to defects in three different programming languages. They concluded that smelly classes are more defect prone, but it varies according

to the programming language. Differently from these three studies, we aim to understand how the prediction of defects differs from the models used to detect code smells, not on establishing a correlation between defect and code smell.
Explainable Machine Learning for Software Features. Software defect explainability is a relatively recent topic in the literature [31, 47, 59]. Mori and Uchihira [47] analyzed the trade-off between accuracy and interpretability of various models. The experimentation displays a comparison between the balanced output that satisfies both accuracy and interpretability criteria. Likewise, Jiarpakdee et al. [31] empirically evaluated two model-agnostic procedures, Local Interpretability Model-agnostic Explanations (LIME) [61] and BreakDown techniques. They improved the results obtained with LIME using hyperparameter optimization, which they called LIME-HPO. This work concludes that model-agnostic methods are necessary to explain individual predictions of defect models. Finally, Pornprasit et al. [59] proposed a tool that predicts defects for systems developed in Python. The input data consists of software commits, and the authors compare its performance with the LIME-HPO [31]. They conclude that the results are comparable to the state-of-the-art technology to explain models.

7 Conclusion

In this work, we investigated the relationship between defects and code smell machine learning models. To do so, we identified and validated the code smells collected with Organic. Then, we applied an extensive data processing step to clean the data and select the most relevant features for the prediction model. Subsequently, we trained and evaluated the models using an ensemble of models. In the end, as the models performed well, we employed an explainability technique to understand the models' decisions known as SHAP. We concluded that among the seven code smells initially collected, only three of them were similar to the defect model (Refused Bequest, God Class, and Spaghetti Code). In addition, we found that the features Nesting Level Else-If and Comment Density were relevant for the four models. Furthermore, most features require high values to predict defects and code smells, except for the Refused Bequest. Finally, we reported that the documentation, complexity, and size quality attributes are the most relevant for these models. In the future steps of this investigation, we can compare the SHAP results with other techniques (e.g., Lime) and employ white-box models to simplify the explainability. Another potential application of our study is the comparison between the reported code smells with other tools. We encourage the community to further investigate and replicate our results. For this reason, we made all data available under the replication package [65].

References

1. Ieee standard glossary of software engineering terminology. In: IEEE Std 610.12-1990 (1990). https://doi.org/10.1109/IEEESTD.1990.101064

2. Abbes, M., Khomh, F., Guéhéneuc, Y., Antoniol, G.: An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: European Conference on Software Maintenance and Reengineering (CSMR) (2011)
3. Abdullah AlOmar, E., Wiem Mkaouer, M., Ouni, A., Kessentini, M.: Do Design Metrics Capture Developers Perception of Quality? An Empirical Study on Self-Affirmed Refactoring Activities. In: International Symposium on Empirical Software Engineering and Measurement (ESEM) (2019)
4. Aghajani, E., Nagy, ., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., Shepherd, D.C.: Software documentation: The practitioners' perspective. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE) (2020)
5. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework (2019)
6. Ali, M.: PyCaret: An open source, low-code machine learning library in Python (April 2020), https://www.pycaret.org, pyCaret version 1.0.0
7. Amorim, L., Costa, E., Antunes, N., Fonseca, B., Ribeiro, M.: Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In: International Symposium on Software Reliability Engineering (ISSRE) (2015)
8. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering (1996)
9. Brown, W.H., Malveau, R.C., McCormick, H.W.S., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc. (1998)
10. Bui, X.N., Nguyen, H., Soukhanouvong, P.: Extra trees ensemble: A machine learning model for predicting blast-induced ground vibration based on the bagging and sibling of random forest algorithm. In: Proceedings of Geotechnical Challenges in Mining, Tunneling and Underground Infrastructures (2022)
11. Cawley, G.C., Talbot, N.L.: On over-fitting in model selection and subsequent selection bias in performance evaluation. Journal of Machine Learning Research (2010)
12. Cruz, D., Santana, A., Figueiredo, E.: Detecting bad smells with machine learning algorithms: an empirical study. In: International Conference on Technical Debt (TechDebt '20) (2020)
13. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 7th IEEE Working Conference on Mining Software Repositories (MSR) (2010)
14. Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A., De Lucia, A.: Detecting code smells using machine learning techniques: Are we there yet? In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2018)
15. Elish, K.O., Elish, M.O.: Predicting defect-prone software modules using support vector machines. Journal of Systems and Software (2008)
16. Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for java. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE) (2018)
17. Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for java and its assessment regarding metrics and bug prediction. In: Software Quality Journal (SQJ) (2020)
18. Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: Unified bug dataset (Mar 2020), https://doi.org/10.5281/zenodo.3693686

19. Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E.: A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE) (2016)
20. Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A.: Jdeodorant: identification and application of extract class refactorings. In: 2011 33rd International Conference on Software Engineering (ICSE) (2011)
21. Fontana, F.A., Mäntylä, M.V., Zanoni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. In: Empirical Software Engineering (2016)
22. Fontana, F.A., Zanoni, M., Marino, A., Mäntylä, M.V.: Code smell detection: Towards a machine learning-based approach (icsm). In: Int'l Conf. on Software Maintenance (2013)
23. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
24. Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N.: An empirical study of just-in-time defect prediction using cross-project models. In: Working Conference on Mining Software Repositories (MSR) (2014)
25. Hall, T., Zhang, M., Bowes, D., Sun, Y.: Some code smells have a significant but small effect on faults. In: Transactions on Software Engineering and Methodology (TOSEM) (2014)
26. Haskins, B., Stecklein, J., Dick, B., Moroney, G., Lovell, R., Dabney, J.: Error cost escalation through the project life cycle. In: INCOSE International Symposium (2004)
27. Hassan, A.E.: Predicting faults using the complexity of code changes. In: International Conference of Software Engineering (ICSE) (2009)
28. He, Z., Shu, F., Yang, Y., Li, M., Wang, Q.: An investigation on the feasibility of cross-project defect prediction. In: Automated Software Engineering (ASE) (2012)
29. Jebnoun, H., Rahman, M.S., Khomh, F., Muse, B.: Clones in deep learning code: What, where, and why? In: Empir Software Eng 27, 84 (2022) (2022)
30. Jiang, T., Tan, L., Kim, S.: Personalized defect prediction. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2013)
31. Jiarpakdee, J., Tantithamthavorn, C., Dam, H.K., Grundy, J.: An empirical study of model-agnostic techniques for defect prediction models. In: Transactions on Software Engineering (TSE) (2020)
32. Jureczko, M., D., S.D.: Using object-oriented design metrics to predict software defects. In: Models and Methods of System Dependability (MMSD) (2010)
33. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y.: Lightgbm: A highly efficient gradient boostingdecision tree. In: 31st Conference on Neural Information Processing System (2017)
34. Khomh, F., Di Penta, M., Gueheneuc, Y.: An exploratory study of the impact of code smells on software change-proneness. In: Proceedings of the 16th Working Conference on Reverse Engineering (2009)
35. Khomh, F., Di Penta, M., Guéhéneuc, Y., Antoniol, G.: An exploratory study of the impact of antipatterns on class change- and fault-proneness. In: Empirical Software Engineering (EMSE) (2012)
36. Khomh, F., Vaucher, S., Guéhéneuc, Y., Sahraoui, H.: Bdtex: A gqm-based bayesian approach for the detection of antipatterns. In: Journal of Systems and Software (2011)
37. Lanza, M., Marinescu, R., Ducasse, S.: Object-Oriented Metrics in Practice. Springer-Verlag (2005)

38. Levin, S., Yehudai, A.: Boosting automatic commit classification into maintenance activities by utilizing source code changes. In: Proceedings of the 13rd International Conference on Predictor Models in Software Engineering (PROMISE) (2017)
39. Lin, Z., Ding, G., Hu, M., Wang, J.: Multi-label classification via feature-aware implicit label space encoding. In: International Conference on International Conference on Machine Learning (ICML) (2014)
40. Lundberg, S.M., Lee, S.: A unified approach to interpreting model predictions. In: Conference on Neural Information Processing Systems (NIPS) (2017)
41. Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y., Aimeur, E.: Smurf: A svm-based incremental anti-pattern detection approach. In: Working Conference on Reverse Engineering (WCRE) (2012)
42. Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y., Antoniol, G., Aïmeur, E.: Support vector machines for anti-pattern detection. In: Proceedings of Int'l Conf. on Automated Software Engineering (ASE) (2012)
43. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. In: Transactions on Software Engineering (TSE) (2007)
44. Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: Defect prediction from static code features: current results, limitations, new approaches. In: Automated Software Engineering (ASE) (2010)
45. Menzies, T., Zimmermann, T.: Software analytics: So what? (2013)
46. Menzies, T., Distefano, J., Orrego, A., Chapman, R.: Assessing predictors of software defects. In: In Proceedings, Workshop on Predictive Software Models (PROMISE) (2004)
47. Mori, T., Uchihira, N.: Balancing the trade-off between accuracy and interpretability in software defect prediction. In: Empirical Software Engineering (EMSE) (2018)
48. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: International Conference on Software Engineering (ICSE) (2006)
49. Oizumi, W., Sousa, L., Oliveira, A., Garcia, A., Agbachi, A.B., Oliveira, R., Lucena, C.: On the identification of design problems in stinky code: experiences and tool support. In: Journal of the Brazilian Computer Society (2018)
50. Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I.K.: Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: 2010 IEEE Int. Conf. on Soft. Maintenance (2010)
51. Openja, M., Morovati, M.M., An, L., Khomh, F., Abidi, M.: Technical debts and faults in open-source quantum software systems: An empirical study. Journal of Systems and Software **193**, 111458 (2022). https://doi.org/https://doi.org/10.1016/j.jss.2022.111458, https://www.sciencedirect.com/science/article/pii/S0164121222001480
52. Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. In: 2018 IEEE/ACM 40th Int. Conf. on Soft. Eng. (ICSE) (2018)
53. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D.: Detecting bad smells in source code using change history information. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 268–278 (2013)
54. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D.: Do they really smell bad? a study on developers' perception of bad code smells. In: 2014 IEEE Int. Conf. on Soft. Maintenance and Evolution (2014)

55. Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., Zaidman, A.: A textual-based technique for smell detection. In: 2016 IEEE 24th international conference on program comprehension (ICPC) (2016)
56. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research (2011)
57. Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N.: The jinx on the nasa software defect data sets. In: International Conference on Evaluation and Assessment in Software Engineering (EASE) (2016)
58. PMD: Pmd source code analyser, https://pmd.github.io/
59. Pornprasit, C., Tantithamthavorn, C., Jiarpakdee, J., Fu, M., Thongtanunam, P.: Pyexplainer: Explaining the predictions of just-in-time defect models. In: International Conference on Automated Software Engineering (ASE) (2021)
60. Raju, V.N.G., Lakshmi, K.P., Jain, V.M., Kalidindi, A., Padma, V.: Study the influence of normalization/transformation process on the accuracy of supervised classification. In: 2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT) (2020)
61. Ribeiro, M.T., Singh, S., Guestrin, C.: ”why should i trust you?”: Explaining the predictions of any classifier. In: International Conference on Knowledge Discovery and Data Mining (KDD) (2016)
62. Riel, A.: Object Oriented Design Heuristics. Addison-Wesley Professional (1996)
63. Santana, A., Cruz, D., Figueiredo, E.: An exploratory study on the identification and evaluation of bad smell agglomerations. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing (2021)
64. Santos, G., Figueiredo, E., Veloso, A., Viggiato, M., Ziviani, N.: Understanding machine learning software defect predictions. In: Automated Software Engineering Journal (ASEJ) (2020)
65. Santos, G.: gesteves91/artifact-fase-santos-23: FASE Artifact Evaluation 2023 (Jan 2023), https://doi.org/10.5281/zenodo.7502546
66. Sayyad S., J., Menzies, T.: The PROMISE Repository of Software Engineering Databases. (2005), http://promise.site.uottawa.ca/SERepository
67. Schumacher, J., Zazworka, N., Shull, F., Seaman, C.B., Shaw, M.A.: Building empirical support for automated code smell detection. In: International Symposium on Empirical Software Engineering and Measurement (ESEM) (2010)
68. Sjøberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dybå, T.: Quantifying the effect of code smells on maintenance effort. In: IEEE Trans. on Soft. Eng. (2013)
69. Stroulia, E., Kapoor, R.: Metrics of refactoring-based development: An experience report. 7th International Conference on Object Oriented Information Systems2001 (2001)
70. Tantithamthavorn, C., Hassan, A.E.: An experience report on defect modelling in practice: Pitfalls and challenges. In: International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (2018)
71. Tantithamthavorn, C., Hassan, A.E., Matsumoto, K.: The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. In: Transactions on Software Engineering (TSE) (2019)
72. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Ihara, A., Matsumoto, K.: The impact of mislabelling on the performance and interpretation of defect prediction models. In: International Conference on Software Engineering (ICSE) (2015)

73. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An empirical comparison of model validation techniques for defect prediction models (2017)
74. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: The impact of automated parameter optimization on defect prediction models. In: Transactions on Software Engineering (TSE) (2019)
75. Tóth, Z., Gyimesi, P., Ferenc, R.: A public bug database of github projects and its application in bug prediction. In: Computational Science and Its Applications (ICCSA) (2016)
76. Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J.: On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering (EMSE) (2009)
77. Turhan, B., Tosun, A., Bener, A.: Empirical evaluation of mixed-project defect prediction models. In: Proceedings of the 37th Conference on Software Engineering and Advanced Applications (SEAA) (2011)
78. Vale, G., Hunsen, C., Figueiredo, E., Apel, S.: Challenges of resolving merge conflicts: A mining and survey study. In: Transactions on Software Engineering (TSE) (2021)
79. Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: International Conference of Software Engineering (ICSE) (2016)
80. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer (2012)
81. Xuan, X., Lo, D., Xia, X., Tian, Y.: Evaluating defect prediction approaches using a massive set of metrics: An empirical study. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC) (2015)
82. Yamashita, A., Moonen, L.: Do developers care about code smells? an exploratory survey. In: 2013 20th Working Conf. on Reverse Eng. (WCRE) (2013)
83. Yamashita, A., Counsell, S.: Code smells as system-level indicators of maintainability: An empirical study. In: Journal of Systems and Software (2013)
84. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: International Workshop on Predictor Models in Software Engineering (PROMISE) (2007)