Evaluating a Continuous Feedback Strategy to Enhance Machine Learning Code Smell Detection

Daniel Cruz[a], Amanda Santana[a], Eduardo Figueiredo[a]

[a]*Computer Science Department, Av. Antônio Carlos, 6627, Pampulha, Belo Horizonte, 31270-901, Minas Gerais, Brazil*

Abstract

Code smells are symptoms of bad design choices implemented on the source code. Several code smell detection tools and strategies have been proposed over the years, including the use of machine learning algorithms. However, we lack empirical evidence on how expert feedback could improve machine learning based detection of code smells. This paper aims to propose and evaluate a conceptual strategy to improve machine-learning detection of code smells by means of continuous feedback. To evaluate the strategy, we follow an exploratory evaluation design to compare results of the smell detection before and after feedback provided by a service - acting as a software expert. We focus on four code smells - God Class, Long Method, Feature Envy, and Refused Bequest - detected in 20 Java systems. As results, we observed that continuous feedback improves the performance of code smell detection. For the detection of the class-level code smells, God Class and Refused Bequest, we achieved an average improvement in terms of F1 of 0.13 and 0.58, respectively, after 50 iterations of feedback. For the method-level code smells, Long Method and Feature Envy, the improvements of F1 were 0.66 and 0.72, respectively. Our promising results are a stepping stone towards the development of new strategies and tools relying on continuous feedback for machine learning detection of code smells.

Keywords: code smell, machine learning, feedback model update

Email addresses: `danielvsc@dcc.ufmg.br` (Daniel Cruz), `amandads@dcc.ufmg.br` (Amanda Santana), `figueiredo@dcc.ufmg.br` (Eduardo Figueiredo)

1. Introduction

Code smells are symptoms of bad design choices implemented on source code [1]. They negatively affect software quality attributes [2], such as software comprehension [3], code stability [4], and robustness [5, 6]. Furthermore, they are one of the key indicators of technical debts, specifically design debts [7], resulting in extra costs in the software development life-cycle, such as rework [8]. Even though a code smell should not always be removed [9], it is important to be aware of their existence to be able to manage the technical debt [7].

To detect code smells, several techniques and tools have been proposed [10, 11, 12, 13, 14]. In fact, these techniques present different types of detection strategies, such as software metrics, textual analysis, and AST analysis [15]. For the software metrics-based detection strategies, thresholds must be defined to identify code smells. However, defining proper thresholds is a major drawback due to their innate complexity, being addressed by several studies that propose methods for threshold derivation [16, 17, 18, 19].

Several studies relying on machine learning algorithms to detect code smells have been proposed in the literature [11, 20, 21, 22, 23, 24, 25]. For instance, Fontana et al. [26] provided an extensive comparison among different machine learning algorithms for detecting code smells, using the same data for training and evaluating their performance in the same way. In their study, almost all algorithms achieved good results for the code smell detection problem. However, a replication study conducted by Di Nucci et al. [27] found diverging results. Their study [27] indicates that the detection need to be improved for several algorithms and code smells.

In a previous work [28], we built a dataset of code smells for twenty Java systems from the Qualita Corpus [29], by using five code smell detection tools. This dataset presents realistic characteristics, such as the expected imbalanced distribution of code smells [30, 31]. We also performed a statistical evaluation of a representative sample of this dataset ground truth to verify if the strategies of the tools comply with the human perception of code smells. In this dataset, we identified and analyzed four code smells: *God Class*, *Long Method*, *Feature Envy*, and *Refused Bequest*. Based on this dataset, we also performed a comparative study of seven machine learning algorithms [28]: *Naive Bayes*, *Logistic Regression*, *Multilayer Perceptron*, *Decision Trees*, *K-Nearest Neighbors*, *Random Forest*, and *Gradient Boosting Machine*. We found a better performance for tree-based algorithms, such as

Random Forest.

Despite the advances in machine learning algorithms, the quality of data is an important matter to obtain high-performance machine learning systems. In a supervised learning scenario, obtaining the labels for the training samples is one of the main concerns, and it can be very expensive. As the use of machine learning to resolve problems has become a trend, we can find the most diverse domains in which models are being developed. In this context, it is common to find actors with domain knowledge, that we named *experts*, to support the creation of high-quality data. Their knowledge can help improve the performance of the models by providing insights and understanding about the data being used. They can also provide the knowledge to define if some model's prediction is right, acting as a labeler or evaluator.

By relying on a service to simulate experts, this paper first proposes a conceptual strategy to improve the models' accuracy using expert feedback. This feedback would be collected, for instance, after the detection being provided by a tool that uses a trained machine learning model. The feedback provided by the expert service can be used to update the initial model. We hypothesize that the model can keep evolving with a qualified feedback from the expert to improve its accuracy on the next detection. We then conduct an exploratory evaluation of the proposed conceptual strategy as follows.

We first evaluate the viability of the feedback as a source of improvement in the code smell detection, and searched for the feedback strategy parameters to assess an optimal configuration for its application. We then evaluate the strategy by simulating 50 iterations of feedback and model retraining. We then compare results of the smell detection before and after feedback. We focus on four code smells, God Class, Long Method, Feature Envy, and Refused Bequest, detected in our previous dataset of twenty Java systems [28]. Our goal in this study is to quantitatively evaluate the performance improvement of the machine learning models after several iterations of expert feedback.

Our results indicated that machine learning models achieved an improvement on the detection of all code smells after expert feedback (Section 4). For instance, for the *Feature Envy* detection, the mean improvement was about 0.72 in F1, after all feedback cycles. We also found a consistent pattern of improvement for all code smells. That is, the detection performance presents a small variation between a few cycles, but in the long run, they all tend to improve when continuous feedback is provided.

Thus, the contributions of this paper are as follows.

- A conceptual strategy to continuously update a machine learning model aiming to improve the detection performance of code smells by means of continuous feedback;
- A dataset of instances of four code smells, God Class, Refused Bequest, Long Method, and Feature Envy, detected by 5 software tools in 20 open-source Java systems;
- An empirical evaluation of the strategy to measure how much the code smell detection can be improved once a single-time feedback is provided; and
- An empirical evaluation of the strategy after 50 iterations of continuous expert feedback is provided. We simulate the expert feedback by a service.

The rest of this paper is organized as follows. Section 2 details the proposed conceptual strategy that updates the models based on expert feedback. Section 3 describes the study design to evaluate our feedback strategy. Section 4 presents our results and discusses the implications for practitioners. Section 5 elaborates on the main threats to the validity of our work. Section 6 describes relevant literature studies and highlights how our work differs and complements them. Finally, Section 7 concludes our work, presenting directions for future works.

2. The Conceptual Feedback Strategy

Software development is in constant evolution as should be the tools to identify or to refactor quality issues, such as code smells. Thus, assessing the quality of the system is not a one-time task since the quality criteria can also be evolved. That is, the software quality can degrade over time, for instance, when new features are implemented or tools are applied to refactor quality issues. This brings us to the scenario in which the quality assurance tools should also be able to evolve. In this case, the machine learning models trained should be integrated into a tool, being able to perform the detection, after extracting the software metrics from the source code and making the inference. This became even more necessary and feasible in industry with support of products and services, such as Amazon Ground Truth[1] and Google

[1]https://aws.amazon.com/sagemaker/groundtruth/

Data Labeling Service[2]. That is, in the perspective of the machine learning operations (MLOps), an area which combines existing practices and processes of DevOps [32] with the new requirements of machine learning, we need to keep the model updated and useful for its user.

Several studies evaluated machine learning performance for code smell detection [26, 27, 28], but none of them evolve the models based on expert feedback. To fill this gap, we propose in this section a conceptual strategy to evolve the machine learning models. We also discuss several issues, such as the complexity of dealing with small amount of feedback in relation to the training size. We select a pre-trained model based on the Random Forest classifier to accomplish the code smell detection task. Our aim is to evolve this model for detecting code smells by means of expert feedback, aligned to the idea of continuous training the supervised machine learning system discussed above. Section 2.1 describes the main components of our conceptual tool and Section 2.2 presents the proposed feedback strategy, in which users and experts interact with the tool components.

2.1. Feedback-based Detection Tool

This section defines the components of a prototype tool to support the detection strategy, considering the actors, entities, and some other important concepts for understanding. First, the actors present in this strategy are: *User* and *Domain Expert*. The *User* is someone in the context of the software development that wants to detect code smells. For instance, a user could be a software engineer who intends to analyze the quality of their newly created code. The *Domain Expert* is someone who is an expert in some domain or field. In our case, we define it as the *Software Engineer Expert*; i.e., a senior professional with high experience in software development or more specifically oriented to the area of software quality evaluation. While the first actor, *User*, is only executing the tool to obtain the detection outputs, the *Software Engineer Expert* participates in the process by providing feedback to improve the quality of the detection for all users.

We then developed a prototype tool, named *FeedSniffer*, with essential modules to act as a machine learning system that detects code smells. Figure 1 depicts these modules. They are briefly described as follows.

- **Metrics Extraction**. The first module of *FeedSniffer* is responsible

[2]https://cloud.google.com/ai-platform/data-labeling/docs

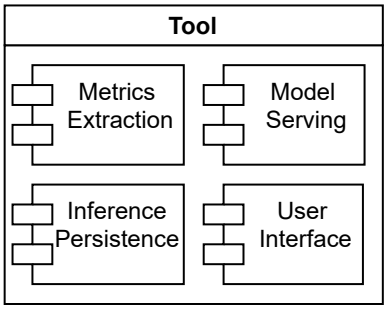

Figure 1: FeedSniffer Modules

to calculate the software metrics for the source code. Several tools have been proposed [33] and are available to extract metrics. The software metrics extracted are the same used for training the models.

- **Model Serving**. The tool is also responsible to load the trained machine learning model into memory and use it to perform the detection. That is, it uses the model to classify the target instance, labeling it as smelly or not.

- **User Interface**. An interface that helps the user to perform the detections. It could be a plugin, a standalone application, or even just a command-line interface. The current implementation provides a command-line user interface. For instance, Figure 2 presents a prototype screen that indicates how a human expert could provide feedback on an instance of Long Method detected by Decor. *FeedSniffer* also outputs a CSV file listing all elements (classes and methods) of the project and their respective labels (i.e., 1 for smelly and 0 for non-smelly).

- **Inference Persistence**. Besides providing the results in the interface, this module of *FeedSniffer* is responsible to store the results in the *Detection Storage*. In case the user changes the label of an element (e.g., turn 1 into 0), the updated list can also be stored.

2.2. The Learning Strategy from Expert Feedback

Figure 3 depicts the cyclic conceptual strategy proposed to improve the quality of the code smell detection. In a nutshell, the quality of the detection is improved by keeping the model updated; i.e., by means of the knowledge

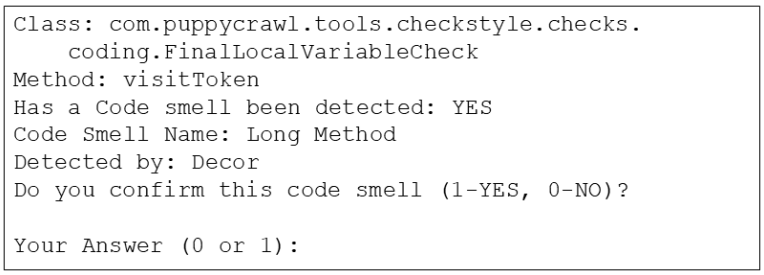

Figure 2: FeedSniffer Command Line Interface for Expert Feedback

of *Software Engineer Experts*. The process is divided in three main routines. Each routine can be identified in Figure 3 by the numbers in the circles. The first routine consists of the execution of a detection tool by the *User* to perform the code smell detection. The detection results can be stored into the *Detection Storage* for a later analysis. The second routine deals with the feedback loop. First, the *Software Engineer Expert* has access to the *Detection Storage*. In fact, the *User* and the *Software Engineer Expert* can be the same person since a developer is also supposed to have knowledge about the software design. Therefore, through a GUI or command-line program, they can interact with the classification of an element that interest them (see Figure 2). For instance, if one instance was detected as smelly, but the metrics that contributed more to this classification do not seem to make sense to the expert, they could seek to analyze the instance more deeply in the code. After analyzing the instance, the expert can decide to provide feedback: it the instance is indeed smelly or not. Experts can select as many detections as they want to analyze to provide feedback. This selection can be facilitated by explainable ML models, such as the SHAP results [34], and by the model's confidence. In our exploratory evaluation, we create a service to act as an expert by accessing the ground truth of our dataset (see Sections 3.3 and 3.4).

After providing feedback for a set of instances, the third routine is triggered. The automated pipeline updates the model currently used by the tool. As the number of instances used for training is, in general, much larger than the set of feedback, it is necessary to provide some augmentation for the data, by prioritizing them and making them impact the retraining of the model. The model retraining, in this case, is done by merging the augmented data with the initial data. Finally, after retraining the model, we can finish

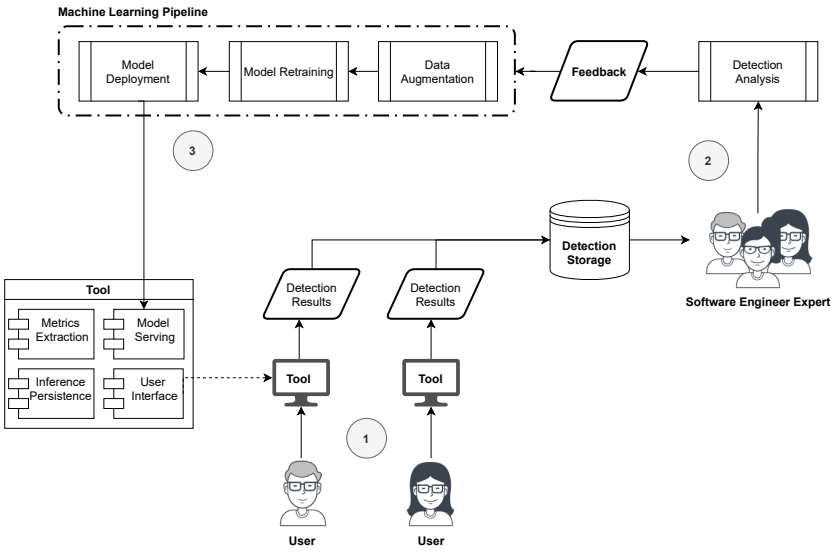

Figure 3: A Strategy to Detect Code Smells based on Continuous Feedback

the process by deploying the model.

As we have made some choices, two important aspects in the third routine of the process are: how it is executed and the other alternatives. The first aspect is related to how we retrain the entire model. That is, we keep the initial training data available from the dataset and, after merging it with the feedback data, we retrain the model entirely, but keeping the found hyperparameters. We also evaluated other alternatives, such as the online and batch training strategies, in which the model is trained considering the existent parameters when a new observation or a set of new observations are available. However, we prefer the offline training as our dataset is not very large and it does not take too long to train the model from the scratch. Therefore, we think it is a simpler approach when dealing with a tree-based model, such as the Random Forest classifier. Finally, we have also experimented with some methods that use online bagging for retraining ensembles [35], but we have not found improvements.

Another decision is related to the Data Augmentation step (see Figure 3). Several techniques can be used to perform this step. In fact, this is one of the focuses for some machine learning areas, such as computer vision [36]. In our scenario, with numeric features, we evaluated some oversampling techniques, such as SMOTE [37], and Adaptative ones [38] for the training

in the first phase of this study. However, as our goal is to provide more importance to the feedback data in the strategy, we decided to perform a simple oversampling by duplicating the data. Other alternatives could include evaluating a meta-model to deal with the data prioritizing. We did not delve into this possibility, to keep the focus of our research. Although it may be an interesting venue for future work, the current strategy is well defined in terms of components and steps. Further work could extend and evaluate other techniques, by comparing them with our results found when using the duplication oversampling.

3. The Exploratory Study Design

This section provides the study design to evaluate the proposed strategy. Section 3.1 defines the research questions and metrics used for evaluating the detection performance. To answer the research questions, we followed five steps – Dataset Collection, Ground Truth Creation, Data Preparation, Single Training Evaluation, and Continuous Training Evaluation – described in Sections 3.2 to 3.6.

3.1. Research Questions and Evaluation Metric

The first aspect to be considered is how the model retraining with the experts' feedback affects its performance. Therefore, before going into the evaluation of the entire process, we aim to measure how much the detection can be improved when the retraining occurs only once. We then define the following research question (RQ1).

RQ1: How much the model's performance can be improved by one-time feedback?

Moreover, we also need to evaluate the model performance after several cycles. That is, let's suppose that after some cycles of feedback the model could start to miss a lot of detections, that were being identified correctly before. Thus, to evaluate how the strategy improves the detection over several feedback cycles, we define the following research question (RQ2).

RQ2: How much the model's performance can be improved by continuous feedback?

Evaluation Metric. To correctly assess the performance of the models and obtain a valid comparison, it is important to select suitable metrics, such as

AUC-ROC [39] and Matthews Correlation Coefficient (MCC) [40]. In fact, code smell datasets are highly imbalanced (Section 3.3). For instance, for the Long Method smell, we have less than one percent of smelly instances. Thus, biased models (e.g., always predicting the instance as non-smelly) could be considered good models by some direct metrics, such as accuracy, which would not be reasonable. Since the code smell datasets are highly imbalanced, we evaluated feedback strategy mainly with *F-Measure* using the unitary weight (F1). However, we also made our raw data publicly available to allow for further analysis with different evaluation metrics. F1 is the harmonic mean of Precision and Recall and these evaluation metrics are defined as follows. Let's consider *TP* the *True Positives*, *FP* the *False Positives*, *TN* the *True Negatives*, and the *FN* the *False Negatives*.

Precision. It measures how much the predictions of smelly instances are correct. The precision is calculated as:

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

Recall. It measures how much of the existent smelly instances were detected by the model. The recall is calculated as:

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

F1. F1 is the harmonic mean of Precision and Recall, defined as follows.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

3.2. Dataset: Smells, Systems, and Metrics

Code Smells. In this work, we focused on four code smells: God Class, Refused Bequest, Long Method, and Feature Envy. God Class [1] consists of a class that accumulates several responsibilities. Refused Bequest [1] is also related to a class. It defines a class that inherits from another but does not even use the attributes and methods inherited. Long Method [1] is related to a method and consists of a method with a large and complex behavior, providing multiple functionalities. Finally, Feature Envy [1] is also related to a method and consists of a method that is more associated with another class than its own. We have selected these smells based on previous

studies that analyzed the perception of code smells by developers [41, 42]. For instance, Palomba et al. [41] investigated how developers perceive the severity of the code smell. They have also investigated how often they identify them. Considering a range from 1 to 5, they found that God Class and Long Method were classified with the highest severity, with a median value of 5 and 4, respectively. For the Feature Envy and Refused Bequest smells, the degree of severity achieved a median value of 4. Other studies [42, 43] reported similar findings, but they mostly found Refused Bequest as harmful as God Class and Long Method.

Software Systems and Metrics. In a previous work [28], we created a dataset with 20 open source Java systems that contain instances of four code smells. The systems were mainly extracted from the *Qualita Corpus* [29], with most of them being used in other machine learning studies [26, 27]. Table 1 presents the complete list of 20 selected systems. The first column presents the name of the system, while the second column presents a brief classification according to its purpose [28, 29]. The third and fourth columns depict the size of the systems in terms of number of classes and number of methods, respectively. For every class and method in the dataset, we extracted 17 metrics for the class level and 13 metrics for the method level. The class level metrics were extracted using the tool VizzMaintenance [44]. This tool calculates known metric suites [45, 46]. The method level metrics were extracted by the CK Metrics tool [47]. The tool computes known metrics, such as *Weighted Method Count* and more experimental ones, like *Quantity of Assignments*. We selected these tools and metrics because (i) both tools export measurement data in a way we could integrate to *FeedSniffer* and (ii) these metrics have been used in previous work about machine learning detection of code smells [28]. The full list of extracted metrics can be found in our online replication package [48].

3.3. Ground Truth Creation

To create the ground truth of code smells, we combined the result of five automatic detection tools: PMD[3], JDeodorant [49], JSpirit [14], an implementation[4] of DECOR [12], and Organic [50]. We rely on the agreement among tools in a majority voting ensemble. It consists of comparing the

[3]pmd.github.io

[4]ptidej.net/publications/Keyword/CODE-AND-DESIGN-SMELLS.php

Table 1: Selected Systems

Name	Description	Classes	Methods
Checkstyle	IDE	511	2,620
Commons Codec	Tool	145	1,481
Commons IO	Tool	282	2,638
Commons Lang	Tool	642	5,947
Commons Logging	Tool	74	546
Hadoop	Middleware	3,703	18,111
Hibernate	Database	6,745	43,828
HtmlUnit	Testing	882	7,485
JasperReports	Data Viewer	1,642	14,880
JFreeChart	Tool	1,037	11,528
JHotDraw	3D/Graphics	732	6,641
JMeter	Testing	1,023	8,689
Lucene	Tool	4,353	20,135
Quartz	Middleware	268	2,495
Spring Framework	Middleware	5,935	30,321
SquirrelSQL	Database	73	550
Struts	Middleware	2,139	12,656
Tapestry	Middleware	1,957	9,276
Tomcat	Middleware	1,794	14,017
Weka	Tool	1,663	18,110
	Total	**35,600**	**231,954**

results among the tools to determine the final outcome. These tools have some limitations, as each one can be used to detect only a small subset of code smells. Therefore, we have identified which tools could detect the same code smells. The goal was to have three different tools for every code smell. Thus, with this voting, an instance (class or method) is considered smelly if two or more tools detected the same smell.

As expected, the number of smell instances is small for all types of code smells when compared to non-smelly instances [30]. For *God Class*, about 4.77% of classes were detected as smelly (1,689 out of 35.6K classes). For *Long Method*, this proportion decreases even more, only 2,023 out of 232K methods were labeled on the ground truth as positive, i.e., 0.87% of all methods. For *Feature Envy*, considering the 232K methods, 3.46% instances were considered as smelly (8,016 instances). For *Refused Bequest*, 8.96% of classes were labeled as positive for this code smell (3,190 out of 35.6K classes).

We evaluate the agreement between the tools' results and the human perception, since we know about the subjective nature of code smells detection [2]. The manual evaluation was performed by 10 software engineering researchers; i.e., PhD candidates and junior developers in our research lab. Due to the large size of the dataset, a complete evaluation was not feasible.

Instead, we rely on statistically sampling. First, for every code smell, we determined a sample size to meet a confidence level of 90% and a maximum error of 10%. We then randomly extracted the necessary sample size from our ground truth and grouped the samples by system. Each researchers manually evaluated code smells of two systems and then we applied the Fleiss Kappa [51] to compute the agreement (a generalization of the Cohen Kappa [52] measure). With Fleiss Kappa, we were able to compute the reliability of the agreement between more than two raters. For every code smell, we computed this agreement, which is presented in Table 2. As can be seen from Table 2, for all smells we had the agreement higher than 0.2, indicating a significant agreement between our voting strategy and human perception.

Table 2: Evaluation Agreement

Code Smell	**Agreement**	**Interpretation**
Refused Bequest	0.65	Substantial
God Class	0.47	Moderate
Long Method	0.41	Moderate
Feature Envy	0.35	Fair

3.4. Data Preparation

We defined four groups of datasets (one for each analyzed code smell) instead of only one dataset. The features are the set of software metrics used to characterize each instance (class or method) and the target is a binary variable indicating if the instance is smelly or non-smelly. It is worth mentioning that not every system in the dataset contains all four code smells analyzed in this work. Table 3 depicts how many systems are affected by each code smell in our datasets. Each code smell is represented by one line and its name is presented in the first column, while the second column contains the number of systems with at least one instance of the respective code smell. We can observe in Table 3 that only the God Class smell was found in all systems, followed by Feature Envy with fifteen systems affected, Long Method with thirteen systems, and finally, Refused Bequest with eleven systems.

Furthermore, due to limited resources, we avoid an evaluation with a large group of actual experts to provide feedback for all experimental runs. As a result, we decide to create a service to act as an expert by accessing the ground truth of our dataset. Therefore, in the context of our experimental

Table 3: Number of Systems with Each Code Smell

Code Smell	Number of Systems
God Class	20
Feature Envy	15
Long Method	13
Refused Bequest	11

setup, we define Oracle as a service that represents the Software Engineer Expert (see Figure 3). The Oracle is responsible for identifying the wrong detections, by accessing the ground truth available for a specific instance. From these wrong detections, the Oracle is able to provide a fraction of correct detections as feedback (e.g., 10%, 20%).

3.5. Single Training Evaluation (RQ1)

To compare the effect of updating the model with feedback from the experts, we designed two moments for the experimentation: before and after any feedback is provided. One important choice for our strategy is which machine learning model to evaluate. Motivated by our previous study [28], in which we evaluated the performance of 7 different models, we have found that Random Forest (RF) and Gradient Boosting Machine using XGBoost Trees (GBM) were the best models to detect all four smells. Table 4 summarizes the F1 performance for both models.

Table 4: F1 Scores for Random Forest and GBM

Model	Code Smell	Mean F1	F1 STD
RF	God Class	*0.846*	0.010
	Long Method	*0.233*	0.052
	Feature Envy	*0.281*	0.058
	Refused Parent Bequest	*0.641*	0.024
GBM	God Class	0.839	0.020
	Long Method	0.211	0.092
	Feature Envy	0.146	0.119
	Refused Parent Bequest	0.593	0.102

The first column presents the machine learning model, while the second column shows the code smell under evaluation. The third and fourth columns

present, respectively, the mean performance and the std standard deviation (STD) of the F1 scores across the cross-validation. The items in italics represent the best performance. Complete results including Accuracy, Precision, and Recall can be found on [28].

As can be seen in Table 4, Random Forest obtained the best F1 mean scores across the 10-folds. However, we can observe that the performance for both smells at method-level (Long Method and Feature Envy) were poor (less than 0.3). This indicates the need of understanding how different techniques can improve the model classification, such as incorporating developer feedback. Our evaluation process for each Target System Dataset (TSD) is defined as follows.

1. We train a model using TSD's training data and the Random Forest algorithm, including its hyperparameters, found as a good choice in our previous work [28].
2. This trained model is used to detect the code smells on the test data, which consists of the instances of the target system.
3. The performance metric (F1) is calculated for the detection results.

In the second moment, we perform the following procedures to update the models with the feedback and evaluate their performance against the initial ones, obtained from the first moment. For each TSD:

1. We ask the Oracle for the feedback from the detection results obtained in the first moment. In this step, the amount of feedback requested varied. This amount is defined as the *Feedback Size (N)*.
2. We process the feedback data by duplicating them. In this step, the amount of oversampling applied varied and it is defined as the *Oversampling Ratio (K)*.
3. The initial training data is merged with the oversampled feedback data and is used to retrain the model, keeping the existing hyperparameters. We also remove from the initial test data the data provided as feedback, keeping the assessment fair.
4. The retrained model is used to detect the code smells on the test data.
5. The performance metrics are calculated again for the new detection results.

The *Feedback Size (N)* and *Oversampling Ratio (K)* provided by Oracle varied, as we intend to explore different values. The reason is because we

do not know upfront a common value, requiring empirical evaluation. For *Feedback Size (N)*, we tried seven possible configurations: 10%, 20%, 30%, 40%, 50%, 60%, and 70%. For the *Oversampling Ratio (K)*, we defined eleven configurations: 1%, 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45%, and 50%.

3.6. Continuous Training Evaluation (RQ2)

A key difference for this design is the control of the experimented variables. Based on the results of RQ1, Feedback Size (N) was kept constant with the value of 10% in the continuous feedback evaluation. The reason is that our goal here is not to explore the effects of large amounts of feedback at once, as we expect it to be provided more incrementally. First, we have conducted a pilot to evaluate the execution of seven cycles of the strategy. The pilot was enough to make clear the potential of the strategy and then we have expanded the number of cycles to 50. As it is not guaranteed nor expected that after several cycles the detection to be perfect (i.e., it would achieve the maximum value of a performance metric) these fifty iterations were enough to demonstrate the effects of the model updating in the code smell detection.

The other important factor to consider, which was also defined, is the Oversampling Ratio (K). In this case, we defined a fixed value of 1% for two reasons. First, this value is not aggressive, which is quite important in the strategy context, since the model is updated more than one time. For higher values, the model could lose its generalization power, as the data from the other systems would begin to be overwhelmed by the duplicated feedback data from the target system. Second, through the results of the pilot study, we found some quantitative indicators that most of the best results have used lower values for the oversampling configuration, in special, the value of 1%.

4. Results and Discussion

This section presents and discusses the results of this study. Section 4.1 reports the results regarding the single training evaluation, while Section 4.2 presents the results regarding the continuous training evaluation.

4.1. Single Time Feedback Results

Using the data collected before and after the feedback, we aim to understand for each system which configurations were superior to detect code

smells. That is, what are the best K and N configurations to update the model. In order to compare the models before and after the feedback, we focus on F1 as the main comparison metric. However, we made our raw data publicly available for further analysis with different evaluation metrics, such as Precision, Recall, AUC-ROC [39], and MCC [40] .

Figure 4 presents this comparison. In the horizontal axis (F1 before Feedback), we can observe the F1 score of the detection performed by the model before the feedback, and in the vertical axis (F1 after Feedback) the best F1 score achieved after model updating with single time feedback. Each instance in the chart is a system, and we can observe that the different marks are related to each code smell.

Note that the diagonal line in Figure 4 provides a visual way to identify if the feedback improved the detection or not. We can observe that the detection was not improved for only one system, and it was actually worsened. We also noticed that the farther from the line the marker is, the greater the difference in performance after the feedback. For instance, for one of the target systems, F1 was around 0.2 before feedback. However, after feedback with one of the provided configurations, it reached an F1 value of about 0.8. Nevertheless, to make sure they are significantly different, we have performed statistical tests to confirm it. Therefore, we state the following null hypotheses:

H_0: Updating the model with feedback has no effect on detection performance.

We perform the Shapiro-Wilk test for normality for each one of our experimental groups. We have selected this test because the normality tests are influenced by the sample size. As our samples are small (less than 30 units), this test will be less affected by Type I error (i.e., false positive). We found that for all smells except the God Class sample before feedback, the detection performances (i.e., measured by F1) follow a normal distribution with a significance level of 99%.

Since they follow a normal distribution, we applied the paired T-test for the samples of the code smells Long Method, Feature Envy, and Refused Bequest. For the samples of the God Class code smell, we applied the Wilcoxon signed-rank test, which is a non-parametric version of the paired T-test. Considering the following p-values obtained from the tests: 1.03×10^{-4} (God Class), 1.96×10^{-7} (Long Method), 6.06×10^{-7} (Feature Envy), 6.45×10^{-5} (Refused Bequest), we found that for all code smells, we could reject the null hypothesis. That is, updating the model with feedback affects detection

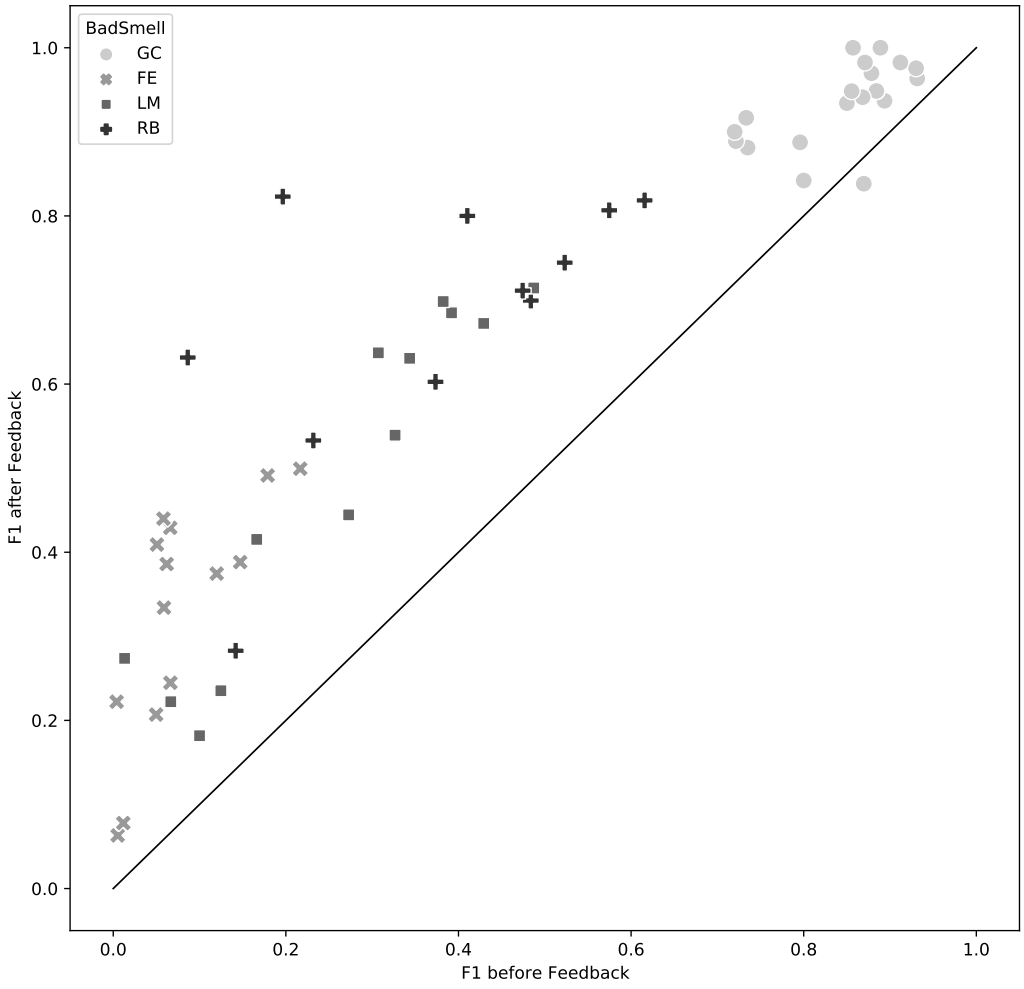

Figure 4: Performance Comparison

performance, with a significance level of 99%.

Considering the results depicted in Figure 4, in which we can see the improvement in the detection performance and the statistical significance provided by the hypothesis testing, we evaluated how much the model has been improved. We analyzed the performance differences by each code smell, separately. Figure 5 presents a boxplot with the distribution for the difference between F1 before feedback and F1 after feedback. The horizontal axis presents the segmentation by code smells, while the vertical axis presents the

distribution of the difference between F1 scores. That is, the F1 difference between the detection performance before and after using the feedback.

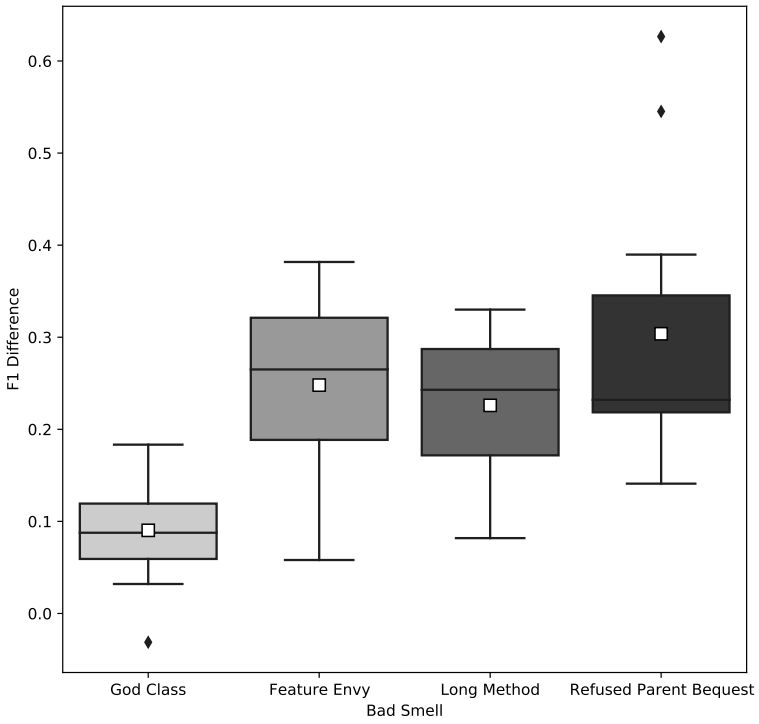

Figure 5: Performance Difference per Code Smell

We can observe that the performance improvement among the code smells varies, similar to the results in Figure 4. One of the clearest differences is the variation in performance between God Class and the other smells. One possible reason may be that the model had good performance from the beginning (before feedback) and there was not much space for improvement. It was also the only one in which the feedback got to worsen the detection performance, although in a little scale. Considering the other three smells, we notice that improvement was close among them, as we can see for the median lines of the boxplots, always close to 0.25. Still, by analyzing the means, indicated by squares inside each boxplot, it is possible to observe that Refused Bequest presented slightly better improvement results. While Feature Envy and Long Method achieved an average improvement of 0.25 and 0.23, respectively, Refused Bequest achieved an average improvement of

0.3. This result is reinforced by the size of the box above the median line and the outliers with even better results.

Notwithstanding, we can further explore the configurations and their results. We can observe independently from the code smell groups which values of Oversampling Ratio (K) and Feedback Size (N) provided the best results. Although we could not compare them directly because of their heterogeneous nature, we can obtain some insights looking for the performance results of the systems. For each system, we retrieved the configuration responsible for the best detection performance. We then calculated the frequency in which each configuration appeared. For some systems, the best performance was obtained for more than one configuration. In this case, we compute for all values used, not prioritizing any configuration. Therefore, the total number of configurations can exceed the sum of the number of systems in each group of code smells. Figure 6 presents this frequency regarding the available configurations. In the vertical axis, we can observe the number of systems in which the configuration provided the best detection performance. In the horizontal axis, we can observe the configuration. The label K represents Oversampling Ratio, and the label N represents Feedback Size.

As expected, the best results were achieved when the expert provides more feedback. Regarding oversampling, we found in most cases the best results provided by the configuration with the smallest Oversampling Ratio (1%). We can observe in Figure 6 that 24 out of the 58 targets achieved the best performance when the expert provides more feedback (i.e., N=70). Regarding oversampling, we found in most cases the best results provided by the configurations with the smallest oversampling ratio (i.e., K=1%). For some systems, the best performance was found with less feedback and then stabilized. In this way, we understand why the values 50% and 60% also appear more frequently in the top results. In fact, for 4 out of the 58 targets, the same F1 was achieved by three different feedback sizes: 50%, 60%, 70%. Regarding Oversampling Ratio, we found that except for the 1% found on the top configurations, and the 5% found in a few cases, the other ratios were found in one to four systems that achieved the same performance. Figure 6 also indicates the cases in which the high oversampling ratios were found paired with large feedback sizes (greater than 50%). The most extreme cases are related to two small systems with few instances that achieved the maximum F1 for twenty-one configurations, and twelve different configurations, respectively. These systems are responsible for almost all configurations presented in Figure 6 with a count below three.

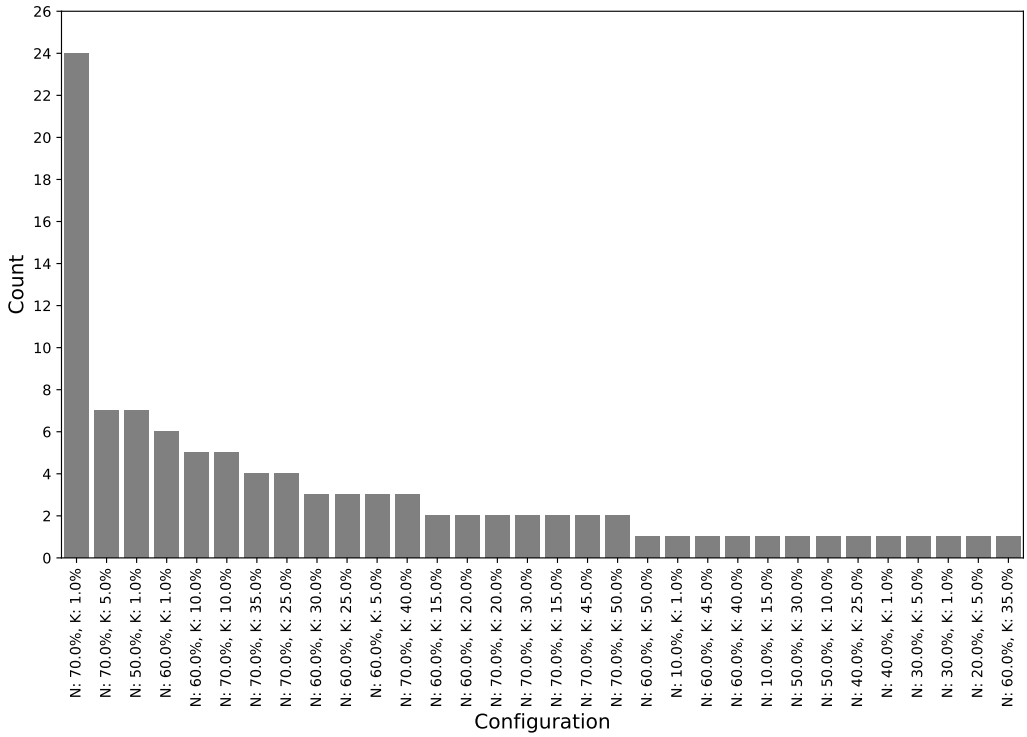

Figure 6: Best configurations

Note that for the feedback size, these results are not saying that detection is improved only after a great number of instances with the feedback. We can only observe that, in general, the more feedback is provided, the better the model behaves. For the oversampling rate configuration, we found that less oversampling is better. Finally, considering all these analyses, we can answer RQ1 as follows.

Answering RQ1. The performance improvement varies according to the code smell, with a wide improvement space, reaching an increase of up to 0.63 in F1 for some systems. God Class detection presented an average improvement of 0.1 in F1. Feature Envy, Long Method, and Refused Bequest achieved an average improvement of 0.26 in F1.

4.2. Continuous Feedback Results

This section presents the analysis performed in the data obtained aiming to answer the second research question (Section 3.6). That is, we evaluate

how continuously retraining the model with the expert feedback affects the model performance.

Figure 7 summarizes the model performance across the incremental feedback cycles for each code smell group. Each row in this figure is related to one code smell and it presents two charts divided into two columns. The charts in the left column present the evolution of the F1 metric in the vertical axis across the feedback cycles displayed in the horizontal axis. We can observe two series: the black line represents the median value and the blue line represents the mean value for all systems evaluated. The charts in the right column present the evolution of the F1 metric in the vertical axis across the feedback cycles displayed in the horizontal axis. However, as we performed ten repetitions when selecting the 10% feedback sample, we also present the 95% confidence interval for the scores, to avoid selection bias in the evaluation. Hence, the black lines also represent the mean value for F1, while the shading outlines the confidence interval area. It is noteworthy that we plotted the mean twice to analyze two different perspectives. For the charts in the left column, we aim to analyze the distance between the mean and the median and to understand if there is more variance of the results across the different systems evaluated. For the charts in the right column, we aim to analyze the impact of the random shuffling of the feedback sample through the mean and the confidence interval.

For the first smell, God Class, we can observe similar results to the obtained on the previous design. In general, each cycle provides a small increase in F1. At a short pace, the performance is increased, but as it reaches a great F1 (above 0.9), it seems to reach an asymptotic close to the maximum F1 possible. We understand this behavior as a small space for improvement. As we discussed before in Section 3.6, we do not expect the F1 score to reach the maximum value, even though it can happen for some systems. Looking at the God Class chart in the left column, we can observe only a small difference between the systems and, in general, they behave the same. From the chart in the right column, we observe that the selection of the feedback sample across the cycles does not affect the results, as the confidence interval is really small.

Regarding the detection of Long Method, we can observe a linear trend of performance increase. However, we first observe that both median and mean decrease in the first iterations with the feedback. This may be due to the random sample selection. After the first cycles, the performance starts to increase and stabilize when F1 reaches around 0.9. We also observe a

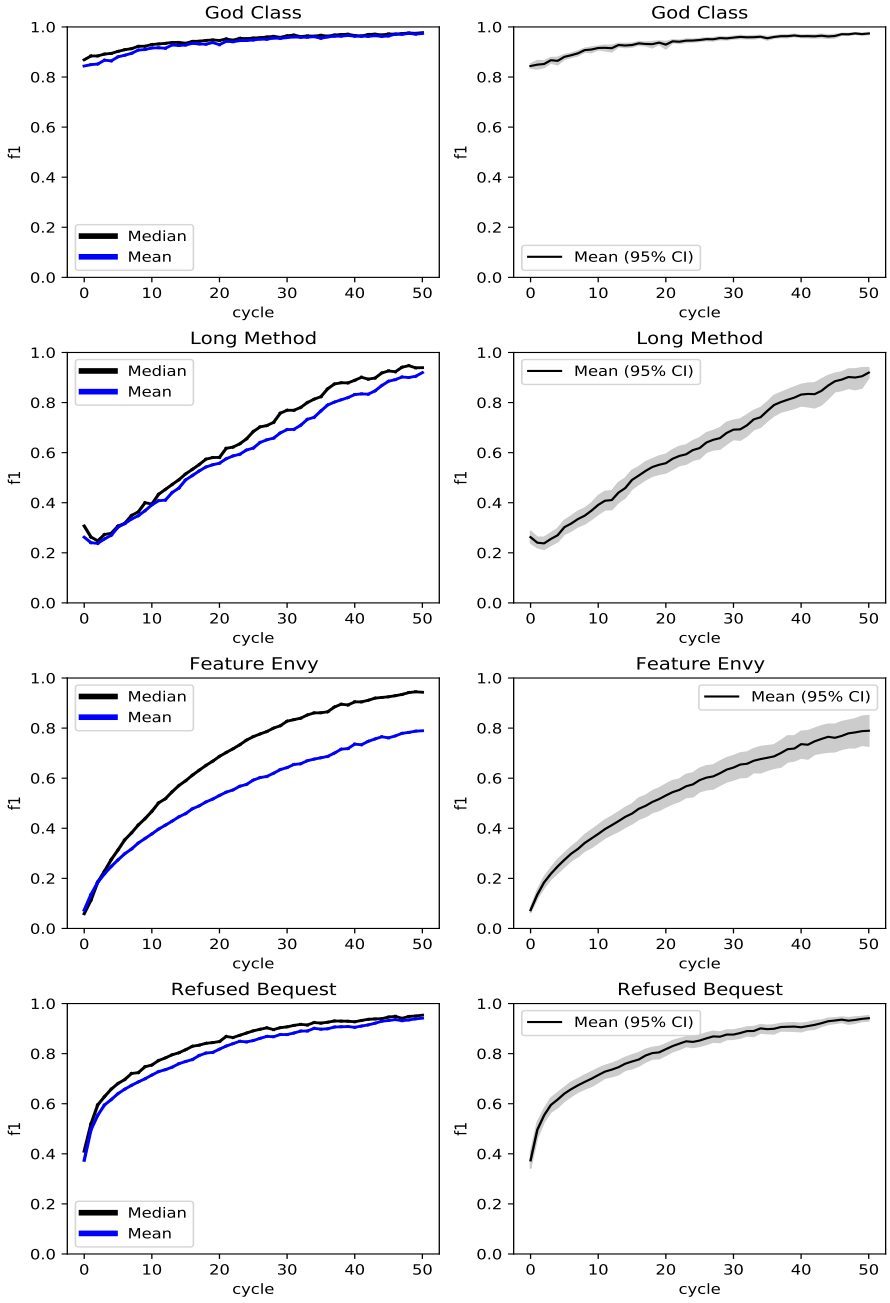

Figure 7: Continuous Feedback Results

small difference between the mean and the median, as most of the systems behave the same, increasing only a little bit less than some others. From the chart in the right column, we observe a larger band for the confidence interval. This means that, depending on the instances present in the feedback sample, the performance may vary between cycles. However, by looking at the broader perspective, we can observe a significant improvement between the fifty cycles. For instance, when looking for differences between the first cycle, without feedback, and after the last cycle, we found an improvement in F1 greater than 0.6. That is, F1 increased from an average of 0.26 to 0.92.

The Feature Envy detection presented an improvement behavior too, but not linear as the previous smell, Long Method. The first twenty cycles present a faster improvement, increasing from the average F1 of 0.08 to an average F1 of about 0.5. In the last thirty cycles, it kept increasing until reaching an average F1 of 0.79. However, observing the blue line of the left column chart, we can notice that the median presents a higher value. Investigating the results deeply, i.e., per system, we found that some systems did not perform well and had a very small increase, lowering the average value. However, in general the systems performed well as can be observed by the black line (median). For this code smell, we found two patterns of improvement. Most of the systems increased their performance since the first cycles. However, despite also improving, some systems follow a much slower pace until the end of the fifty cycles. Analyzing these systems, the main common attribute is the high number of methods. This suggests that F1 could further increase by executing more cycles. Looking at the chart in the right column, we also observe that the confidence interval is well contained and does not seem to affect the overall result. However, we can observe that the band becomes broader in the last cycles, although in general, the improvement is clear when looking over all cycles.

Finally, we have the Refused Bequest smell. For this one, we can observe a behavior similar to Feature Envy, in which a trend of increase in detection performance is presented since the first cycles. However, differently from the Feature Envy detection, the two lines, representing the median and the mean of F1, remain close throughout the cycles. All systems evaluated had their performance increased. Worth noticing that after only ten cycles, the mean F1 has almost doubled, from 0.37 to 0.71, ending with an average F1 of 0.94. The selection of the feedback sample also did not impact the results in general, as we see in the confidence interval band from the chart in the right column. Through these analyses, we can answer the RQ2 as follows.

As can be seen, there is indeed an improvement in the F1 measure when we provide feedback. However, retraining large models can be costly, since it requires powerful hardware resources to train and retrain more complex models for larger datasets; e.g., a server/database in which our Users and Experts can insert and update the instances. For the industry, online services providers, such as Amazon Web Service [5] and Microsoft Azure [6], can be used for the model retraining. However, we also further research work to investigate and minimize costs, for instance, associated with the impact of using feedback batches.

Answering RQ2. The performance improvement varies according to the code smell. God Class detection presented the smaller average improvement, an increase of 0.13 in F1, as the initial performance was already high. The Long Method detection presented an average improvement of 0.66 in F1. For the Feature Envy and Refused Bequest detections, we found average improvements of 0.72 and 0.58 in F1, respectively.

5. Threats to Validity

Despite the careful design of our empirical study, some limitations may affect the validity of our results. In this section, we discuss some of those threats and our actions to mitigate them, organizing them by construct, internal, external, and conclusion validity [53].

Construct Validity. With respect to the number of instances used to evaluate the models after several cycles, it is important to note that the number of test instances does not drive the results. That is, as we remove the feedback samples from the test data to keep the evaluation fair, we also evaluated how many instances were removed and if the results could be impacted by a small number of instances left to be evaluated. First, we did not keep the cycles running after the number of feedback available had finished, avoiding increasing the performance for the last cycles. Then, we also evaluated how many instances from the initial set were removed from the initial amount. As we have tested the entire dataset through our systems cross-validation, we compare the initial size of the test data for each code smell, before and after the last cycles, for each system.

[5] `https://aws.amazon.com/pt/`
[6] `https://azure.microsoft.com/`

Internal Validity. Regarding the internal validity of the continuous training evaluation study, as we set the value of the feedback size, we also replicate the process, changing the 10% randomly selected for feedback from each cycle. We aim to mitigate the possibility of obtained results being influenced by the selection of the instances belonging to the feedback. For instance, if in the first cycle the majority of the feedback is related to false negatives, this could influence our model differently from if the majority of instances in the first cycle were false positives. Beyond the fact that we can estimate the error of our model, through these replications, we can obtain the confidence interval for our measurements. Our focus is not on obtaining a fixed F1 value, but a range, providing more confidence to our results, even if more systems were added to the model. In total, ten repetitions for each system/group were performed. Each repetition preserves the random seed for all code smells and systems on its execution.

External Validity. First, concerning the external validity of our studies, the systems evaluated may not represent all systems available, and they are all written in Java. That is, our results depend directly on how the dataset was constructed. For instance, we analyze systems in the Qualita Corpus, which may present different levels of maturity, sizes, and domains of proprietary systems. Moreover, we are aware that the learning performance for systems from different languages and domains may lead to different results. Our goal was to propose and evaluate our feedback strategy, serving as an step stone to studies that explore different datasets. We highlight that the strategy itself and the experimental studies designed to evaluate it are not dependent on the programming language and could be performed in different datasets. To manually create a dataset for a large number of systems was not feasible due to our time constraints, since experts should inspect all system's classes and methods. To address this limitation, we opted to create our ground truth by using five different detection tools, that uses different identification approaches. To mitigate the bias of false positives, we have used a voting strategy, in which at least two tools have to agree that the instance is smelly. Consequently, we do not rely on the specific output of a tool, but on their agreement. We further cross-validated a sample of code smells detected with our voting strategy with ten developers, and found an fair-substantial agreement.

Conclusion Validity. Regarding the conclusion validity of the single train-

ing evaluation study, the results found could be based on the wrong choice of the statistical tests. To avoid it, we performed previous analyses regarding the normality of the samples and their dependency relation. Thus, we applied two tests separately to mitigate this threat. Another threat, for both studies, is the performance metric that drives our conclusion. As we have discussed in Section 3.1, the main metric selected to evaluate the performance is the F1 score. The main reason is that we do not want to prioritize neither precision nor recall. We also are dealing with heavily imbalanced data. Hence, we are focusing on the metrics related to the class that matters, the positive one that represents the presence of a smell.

6. Related Work

Several detection techniques and tools have been proposed in the literature. In fact, the adoption of machine learning techniques to detect code smells has became a trend [20, 21, 22, 23, 24, 26, 27, 54, 55]. For instance, Khomh et al. [11] proposed a Bayesian approach which initially converts existing detection rules to a probabilistic model to perform the predictions. Khomh et al. [22] later extended their previous work [11] by the introduction of Bayesian Belief Networks, improving the accuracy of the detection. Maiga et al. [23, 24] proposed an SVM-based approach that uses the feedback information provided by practitioners. Amorim et al. [20] presented an experience report on the effectiveness of Decision Trees for detecting code smells. They choose these classifiers due to their interpretability [20]. Thus, most of the proposed works focus only on one classifier. They were also trained in a dataset composed of few systems and, consequently, the results may be positive towards their approach due to overfitting.

Other studies [21, 26, 27, 28] evaluated and compared the performance of different machine learning algorithms on distinct sets of systems. For instance, Fontana et al. [21, 26] performed a larger comparison of classifiers [26]. The notorious impact of their work was the great performance reported. Even naive algorithms were able of achieving good results using a small training dataset. Di Nucci et al. [27] replicated the study and verified that the reported performance could be biased by the dataset and some procedures, such as unrealistic balanced data, in which one-third of the instances were smelly. Our previous work [28] contributed with further empirical evidences on the use of machine learning algorithms for code smells detection by using a larger dataset of curated systems.

Table 5 presents a comparison summary between similar previous work [26, 27] and ours with respect to the used dataset and ML algorithms. The columns are related to each work. The rows present characteristics of the studies, such as which code smells were studied. In our work, we aim to contribute with empirical evidences on the use of machine learning algorithms for code smells detection, by overcoming the known limitations [27]. The following considerations can also be made. (i) The dataset used in this paper is very close to the real world, being extremely imbalanced, with a very low proportion of smelly instances. For instance, for the Long Method smell, less than 1% of the methods are smelly. We observe in the last column of Table 5 that besides the lower number of systems, the number of instances in our dataset is much bigger than on other studies. (ii) Our features were analyzed, and only two strong correlations have been found and discarded. As shown in Table 5, on the *Features* row, we use less metrics. However, the feature selection of a replication study [27] detected that most of the 143 features were high correlated and were discarded accordingly. (iii) We found that it is harder to obtain good performance for smells at method level.

Few studies address a feedback process to improve the detection of code smells as we do here. Hozano et al. [57] proposed a platform to consider the developer feedback, to create personalized rules for a rule-based detection technique. However, we did not found in their work how the rules evolve. Their evaluation was also performed only on two small systems and using only the accuracy metric. It may lead to an incorrect conclusion, without taking into consideration the data distribution. Liu et al. [58] proposed an automatic way to update the thresholds used to perform code smell detection. This technique is based on feedback provided by a software engineer. Their strategy focuses on making it possible to interact with the threshold definition by explicitly directing the detection regarding one metric, in their case, precision. However, different from our work, they focused on finding metrics thresholds to provide better detection. Meanwhile, our goal was to propose and evaluate a high abstraction strategy, aiming at providing a trained ML model with explicit feedback from a expert service.

7. Conclusion

Code smells are important indicators of quality improvement opportunities in the source code of a system. Although many techniques and tools have been proposed, we lack a continuous improvement alternative for the

machine learning based detection of code smells. In this paper, we proposed a conceptual strategy to improve the code smell detection through a continuous feedback approach, in which a machine learning model is retrained over the iterations with a expert that provides feedback about the detection results. We also evaluated the strategy with an exploratory study in which we experimented different parameterizations and simulated the use of the strategy with fifty cycles of feedback in a dataset of 20 software systems. We found that by using the proposed strategy, code smell detection can be improved incrementally for all code smells. For the detection of *God Class*, a code smell with a detection performance initially good, we achieved an average improvement of 0.13 in terms of F1. For the other code smell related to a class, *Refused Bequest*, after all iterations of the strategy, we achieved an average improvement of 0.58 in terms of F1. For the method level code smells, *Long Method* and *Feature Envy*, we achieved an average improvement of 0.66 and 0.72 in terms of F1, respectively.

As future work, we foresee some opportunities to further improve the current strategy or its evaluation. For instance, our study used 20 systems from the Qualita Corpus, and replication studies could evaluate different datasets, such as the MLCQ [59]. They could also investigate factors influence the most on the effectiveness of code smell detection, such as system size, number of smells detected, maturity and CI/CD. Another possibility is to combine the ground truths of several datasets, generalizing the model, to evaluate different programming languages. Concerning our strategy, a possibility for future work is to generalize our feedback strategy to incorporate different machine learning models, such as deep learning and LLMs [60], and to evaluate the performance when they receive multiple feedback sources. We would like to understand how noises in the provided feedback impact the models. That is, to what extent the models deal with this wrong feedback. As consequence, we can further enhance the strategy by introducing a routine that checks for noise and decides what to do with the feedback, *e.g.,* not updating the model, adjusting the oversampling ratio, asking for a second expert to provide feedback, etc. We can also evaluate the prioritizing of which instances to ask for feedback. As we observed in our results, the random selection of the group of 10% of instances to be given as feedback did not impact the detection's performance. However, there are other techniques from related fields (e.g., Active Learning) that could lead us to a way of selecting a set of instances that could make the performance of the detection improve faster (i.e., with fewer cycles). Finally, it is worth mentioning as future work an

experimental evaluation with humans; i.e., actual software experts [61]. For this evaluation, our prototype tool could also be improved, for instance, with a friendly user interface.

Acknowledgements. This research was partially supported by Brazilian funding agencies: CAPES, CNPq, and FAPEMIG.

References

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

[2] A. Yamashita, S. Counsell, Code smells as system-level indicators of maintainability: An empirical study, J. of Systems and Software 86 (10) (2013) 2639–2653.

[3] M. Abbes, F. Khomh, Y. Gueheneuc, G. Antoniol, 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.

[4] A. Santana, E. Figueiredo, J. Pereira, Unraveling the impact of code smell agglomerations on code stability, in: Proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2024.

[5] F. Khomh, M. Di Penta, Y. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change-and fault-proneness, Empirical Software Engineering 17 (3) (2012) 243–275.

[6] G. Santos, A. Santana, G. Vale, E. Figueiredo, Yet another model! a study on model's similarities for defect and code smells, in: Proceedings of the 26th International Conference on Fundamental Approaches to Software Engineering (FASE), 2023.

[7] G. Suryanarayana, G. Samarthyam, T. Sharma, Refactoring for software design smells: managing technical debt, Morgan Kaufmann, 2014.

[8] T. Sedano, P. Ralph, C. Péraire, Software development waste, in: Proceedings of the Int'l Conf. on Software Engineering, 2017, pp. 130–140.

[9] F. Fontana, V. Ferme, S. Spinelli, Investigating the impact of code smells debt on quality code evaluation, in: Proceedings of the International Workshop on Managing Technical Debt, 2012, pp. 15–22.

[10] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Ten years of jdeodorant: Lessons learned from the hunt for smells, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 4–14.

[11] F. Khomh, S. Vaucher, Y. Guéhéneuc, H. Sahraoui, A bayesian approach for the detection of code and design smells, in: Int'l Conf. on Quality Software, 2009, pp. 305–314.

[12] N. Moha, Y. Gueheneuc, L. Duchien, A. Le Meur, Decor: A method for the specification and detection of code and design smells, Transactions on Software Engineering 36 (1) (2009) 20–36.

[13] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in: Int'l Conf. on Automated Software Engineering (ASE), 2013, pp. 268–278.

[14] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, W. Oizumi, Jspirit: a flexible tool for the analysis of code smells, in: Int'l Conf. of the Chilean Computer Science Society (SCCC), 2015, pp. 1–6.

[15] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the Int'l Conf. on Evaluation and Assessment in Software Engineering (EASE), 2016.

[16] F. Fontana, V. Ferme, M. Zanoni, A. Yamashita, Automatic metric thresholds derivation for code smell detection, in: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics, IEEE, 2015, pp. 44–53.

[17] S. Herbold, J. Grabowski, S. Waack, Calculation and optimization of thresholds for sets of software metrics, Empirical Software Engineering 16 (6) (2011) 812–841.

[18] D. Roy, S. Fakhoury, V. Arnaoudova, Reassessing automatic evaluation metrics for code summarization tasks, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, p. 1105–1116.

[19] G. Vale, E. Fernandes, E. Figueiredo, On the proposal and evaluation of a benchmark-based threshold derivation method, Software Quality Journal (SQJ) (2019).

[20] L. Amorim, E. Costa, N. Antunes, B. Fonseca, M. Ribeiro, Experience report: Evaluating the effectiveness of decision trees for detecting code smells, in: Int. Symposium on Software Reliability Engineering (ISSRE), 2015, pp. 261–269.

[21] F. Fontana, M. Zanoni, A. Marino, M. V. Mäntylä, Code smell detection: Towards a machine learning-based approach (icsm), in: Int'l Conf. on Software Maintenance, 2013, pp. 396–399.

[22] F. Khomh, S. Vaucher, Y. Guéhéneuc, H. Sahraoui, Bdtex: A gqm-based bayesian approach for the detection of antipatterns, Journal of Systems and Software 84 (4) (2011) 559–572.

[23] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, E. Aimeur, Smurf: A svm-based incremental anti-pattern detection approach, in: Working Conference on Reverse Engineering (WCRE), 2012, pp. 466–475.

[24] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, G. Antoniol, E. Aïmeur, Support vector machines for anti-pattern detection, in: Proceedings of Int'l Conf. on Automated Software Engineering (ASE), 2012, pp. 278–281.

[25] H. Nunes, A. Santana, E. Figueiredo, H. Costa, Tuning code smell prediction models: A replication study, in: Proceedings of the 32nd International Conference on Program Comprehension (ICPC), 2024.

[26] F. A. Fontana, M. V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empirical Software Engineering 21 (3) (2016) 1143–1191.

[27] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, A. De Lucia, Detecting code smells using machine learning techniques: are we there yet?, in: Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 612–621.

[28] D. Cruz, A. Santana, E. Figueiredo, Detecting bad smells with machine learning algorithms: an empirical study, in: Proceedings of the 3rd International Conference on Technical Debt, 2020, pp. 31–40.

[29] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, The qualitas corpus: A curated collection of java code for empirical studies, in: Asia Pacific Software Engineering Conference (APSEC), 2010, pp. 336–345.

[30] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering 23 (3) (2018) 1188–1221.

[31] J. Carneiro, J. A. Pereira, A. Damasceno, J. B. Ribas, E. Figueiredo, Improvmlcq: A feature-enriched dataset for advancing code smell detection, in: Proceedings of the Ibero-American Conference on Software Engineering (CIbSE), 2025.

[32] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano, Devops, Ieee Software 33 (3) (2016) 94–100.

[33] R. Lincke, J. Lundberg, W. Löwe, Comparing software metrics tools, in: Proceedings of the 2008 international symposium on Software testing and analysis, 2008, pp. 131–142.

[34] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system, in: Int'l Conf. on knowledge discovery and data mining (KDD), ACM, 2016, pp. 785–794.

[35] A. Saffari, C. Leistner, J. Santner, M. Godec, H. Bischof, On-line random forests, in: 2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops, 2009, pp. 1393–1400. doi:10.1109/ICCVW.2009.5457447.

[36] C. Shorten, T. M. Khoshgoftaar, A survey on image data augmentation for deep learning, Journal of Big Data 6 (1) (2019) 1–48.

[37] A. Fernandez, S. Garcia, F. Herrera, N. V. Chawla, Smote for learning from imbalanced data: Progress and challenges, marking the 15-year anniversary, Journal of Artificial Intelligence Research (2018).

[38] Ş. Ertekin, Adaptive oversampling for imbalanced data classification, in: Information Sciences and Systems 2013, Springer, 2013, pp. 261–269.

[39] A. M. Carrington, D. G. Manuel, P. W. Fieguth, T. Ramsay, V. Osmani, B. Wernly, C. Bennett, S. Hawken, O. Magwood, Y. Sheikh, M. McInnes, A. Holzinger, Deep roc analysis and auc as balanced average accuracy, for improved classifier selection, audit and explanation, IEEE Transactions on Pattern Analysis and Machine Intelligence (2023) 329–341.

[40] D. Chicco, M. J. Warrens, G. Jurman, The matthews correlation coefficient (mcc) is more informative than cohen's kappa and brier score in binary classification assessment, IEEE Access (2021) 78368–78381.

[41] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: International Conference on Software Maintenance and Evolution, 2014, pp. 101–110.

[42] D. Taibi, A. Janes, V. Lenarduzzi, How developers perceive smells in source code: A replicated study, Information and Software Technology 92 (2017) 223–235.

[43] A. Yamashita, L. Moonen, Do developers care about code smells? an exploratory survey, in: 2013 20th working conference on reverse engineering (WCRE), 2013.

[44] H. Barkmann, R. Lincke, W. Löwe, Quantitative evaluation of software quality metrics in open-source projects, in: Int'l Conf. on Advanced Information Networking and Applications Workshops (AINA), 2009, pp. 1067–1072.

[45] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, Transactions on Software Engineering 20 (1994) 476–493.

[46] M. Lanza, R. Marinescu, Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems, Springer Science & Business Media, 2007.

[47] M. Aniche, Java code metrics calculator (CK), available in github.com/mauricioaniche/ck/ (2015).

[48] D. Cruz, A. Santana, E. Figueiredo, Evaluating a continuous feedback strategy to enhance machine learning code smell detection - online replication package (2025).
URL `https://zenodo.org/record/8122453`

[49] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Jdeodorant: identification and application of extract class refactorings, in: Int'l Conf. on Software Engineering (ICSE), 2011, pp. 1037–1039.

[50] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira, C. Lucena, On the identification of design problems in stinky code: experiences and tool support, Journal of the Brazilian Computer Society 24 (1) (2018).

[51] J. L. Fleiss, Measuring nominal scale agreement among many raters., Psychological bulletin 76 (5) (1971) 378.

[52] J. Cohen, A coefficient of agreement for nominal scales, Educational and psychological measurement 20 (1) (1960) 37–46.

[53] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[54] A. Santana, D. Cruz, E. Figueiredo, An exploratory study on the identification and evaluation of bad smell agglomerations, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, 2021, p. 1289–1297.

[55] A. Santana, E. Figueiredo, J. Pereira, A. Garcia, An exploratory evaluation of code smell agglomerations, Software Quality Journal (SQJ) (2024).

[56] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: Int'l Conf. on Software Maintenance (ICSM), 2004, pp. 350–359.

[57] M. Hozano, H. Ferreira, I. Silva, B. Fonseca, E. Costa, Using developers' feedback to improve code smell detection, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 1661–1663.

[58] H. Liu, Q. Liu, Z. Niu, Y. Liu, Dynamic and automatic feedback-based threshold adaptation for code smell detection, IEEE Transactions on Software Engineering 42 (6) (2015) 544–558.

[59] L. Madeyski, T. Lewowski, Mlcq: Industry-relevant code smell data set, in: Evaluation and Assessment in Software Engineering, 2020, pp. 342–347.

[60] H. Nunes, E. Figueiredo, L. Soares, S. Nadi, F. Ferreira, G. Santos, Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects, in: Proceedings of the 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025.

[61] S. O. Barraood, H. Mohd, F. Baharom, A. Almogahed, Verifying agile black-box test case quality measurements: Expert review, IEEE Access 11 (2023) 106987–107003.

Table 5: Comparison of Selected Related Work

	Fontana [26]	**Di Nucci [27]**	**Ours**
Systems	74		20
Sample Size	1,680*	3,360**	267,000
Code Smells	Data Class, God Class, Feature Envy, Long Method		God Class, Feature Envy, Long Method, Refused Bequest
Detection Tools (Ground Truth)	iPlasma, PMD, Fluid Tool, Antipattern Scanner, Marinescu[56]		JDeodorant, JSpirit, PMD, DECOR, Organic
Ground Truth Validation	Manual (Entire sample)		Manual (Statistical sample)
Algorithms (no variations)	Decision Trees Rule Based Random Forest Naive Bayes SVM/SMO		Decision Trees Random Forest Naive Bayes Logistic Regression KNN Multilayer Perceptron Gradient Boosting Machine
Features	143		30
Feature Selection	No	Yes	Yes
Optmization	Grid Search		Random Search

** One dataset for each code smell, each one with 420 instances*
** The replication study claims to have duplicated the original dataset. This number was calculated based on this information and is not present in the original work.*