An Exploratory Evaluation of Continuous Feedback to Enhance Machine Learning Code Smell Detection

Daniel Cruz , Amanda Santana , Eduardo Figueiredo

[1]Computer Science Department (DCC)
Federal University of Minas Gerais (UFMG)

{danielvsc, amandads, figueiredo}@dcc.ufmg.br

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.*

1. Introduction

Code smells are symptoms of bad design choices implemented on source code [16]. They negatively affect software quality attributes [38], such as software comprehension [2] and robustness [22]. To detect code smells, several techniques and tools have been proposed [12, 20, 28, 29, 30, 32, 36]. In fact, these techniques present different types of detection strategies, such as software metrics, textual analysis, and AST analysis [11]. For the software metrics-based detection strategies, thresholds must be defined to identify code smells. Defining proper thresholds is a major drawback due to their innate complexity, being addressed by several studies that propose methods for threshold derivation [14, 17].

Several studies rely on machine learning algorithms to detect code smells have been proposed in the literature [3, 13, 20, 21, 26, 27]. For instance, Fontana et al. [15] provided an extensive comparison among different machine learning algorithms for detecting code smells, using the same data and evaluating them 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. [10] found diverging results. Their study [10] indicates that the detection need to be improved for several algorithms and code smells. In a previous work [9], we built a dataset of code smells for twenty Java systems by using five code smell detection tools. 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 [9]: *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 also can 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 (Section 2). 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 (Section 3). We first evaluate the viability of the feedback as a source of improvement in the code smell detection and searched for parameters of the strategy to assess an optimal configuration for its application. We then evaluate the strategy by simulating 50 iterations of feedback and model training. 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 twenty Java systems. 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 (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. Thus, the contributions of this paper is twofold: (1) a conceptual strategy to continuously update a machine learning model aiming to improve the detection performance of code smells by means of continuous feedback and (2) an exploratory evaluation of this strategy, through a service that allowed us to simulate 50 iterations of expert feedback.

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. Given this scenario, 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.

Figure 1 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 of *Software Engineer Experts*. The process is divided in three main routines. Each routine can be identified in Figure 1 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 Experts* have 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. Experts can select as many detections as they want to analyze and provide feedback. This selection can be facilitated by explainable ML models, such as the SHAP results [7], and by the model's confidence. 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. 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).

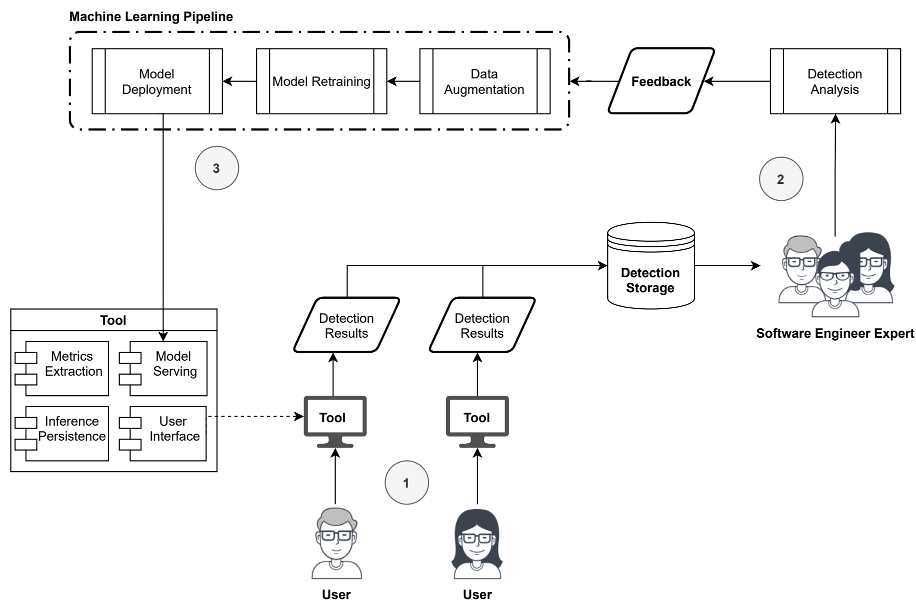

Figure 1. A Strategy to Detect Code Smells based on Continuous Feedback

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 the process by deploying the model.

We developed a prototype tool, named *FeedSniffer*, to support the proposed strategy with four essential modules described as follows. **Metrics Extraction**: The first module of *FeedSniffer* is responsible to calculate the software metrics for the source code. Since several tools are available to extract metrics [24], we integrated *FeedSniffer* with

two existing tools: CK Metrics [4] and VizzMaintenance [5]. The software metrics extracted are 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 pre-trained model to classify the target instance, labeling it as smelly or not. **User Interface**: An interface that helps the user to perform the detections, but in our case it has a command-line interface. That is, *FeedSniffer* 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**: 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.

3. The Exploratory Study Design

Section 3.1 defines the research questions and the metrics used for evaluation. To answer the research questions, we followed five steps 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, RQ1 aims to evaluate how much the detection can be improved when the retraining occurs only once. We the evaluate with RQ2 the model performance after several cycles.

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

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

Evaluation Metric. Since the code smell datasets are highly imbalanced, we evaluated feedback strategy with *F-Measure* using the unitary weight (F1). F1 is the harmonic mean of Precision and Recall and a robust measurement of machine learning performance.

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

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 [16] consists of a class that accumulates several responsibilities. Refused Bequest [16] 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 is related to a method and consists of a method with a large and complex behavior, providing multiple functionalities. Finally, Feature Envy 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 [33, 34]. For instance, Palomba et al. [33] 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 [34, 39] reported similar findings, but they mostly found Refused Bequest as harmful as God Class and Long Method.

Software Systems and Metrics. 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* [35], with most of them being used in other machine learning studies [15, 10]. 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 [9, 35]. 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 [5]. This tool calculates known metric suites [6, 8, 18, 23]. The method level metrics were extracted by the CK Metrics tool [4]. 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 [9]. The full list of extracted metrics can be found in our online replication package [1].

3.3. Ground Truth Creation

To create the ground truth of code smells, we combined the result of five automatic detection tools: PMD[1], JDeodorant [12], JSpirit [36], an implementation[2] of DECOR [30], and Organic [31]. We rely on the agreement among tools in a majority voting ensemble. It consists of comparing the 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. The result of this voting ensemble is also in our online replication package [1].

We evaluate the agreement between the tools' results and the human perception, since we know about the subjective nature of code smells detection [38]. 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 staticsally sampling. First, for every code smell, we determined a sample size to meet a confidence level of 90% and a maximum error of 10%. Then, we randomly extracted the necessary sample size from our ground truth and grouped the samples by system. Finally, each researchers manually evaluated code smells of two systems. The agreement between tools and a manual evaluation was high enough for our purposes (above 80% for all systems).

3.4. Data Preparation

We defined four groups of datasets (one for each 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. The God Class smell was found in all 20 systems, Feature Envy in 15 systems,

[1] pmd.github.io
[2] 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**

Long Method in 13 systems, and Refused Bequest in 11 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 setup, we define Oracle as a service that represents the Software Engineer Expert. 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. The 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 previous work [9].
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. The first one, Feedback Size (N), was kept constant with the value of 10%. 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 duplicate 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

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 if any of the configurations were superior to the initial result. If so, what is the best K and N configuration to update the model. Thus, we extract the best results among the different configurations, and compare them with the initial model performance. Based on our results, we observed 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, the base F1 was around 0.2 before feedback. However, after the feedback with one of the provided configurations, it reached an F1 of about 0.8. Nevertheless, to

make sure they are significantly different, we have performed statistical tests to confirm it. Then, 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 performance, with a significance level of 99%.

As expected, we observed that the best results were achieved 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%). Due to space constraints, detailed results about the single training evaluation is available in our replication package [1].

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

Figure 2 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 chart in the left column presents 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 chart in the right column presents 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 the 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.

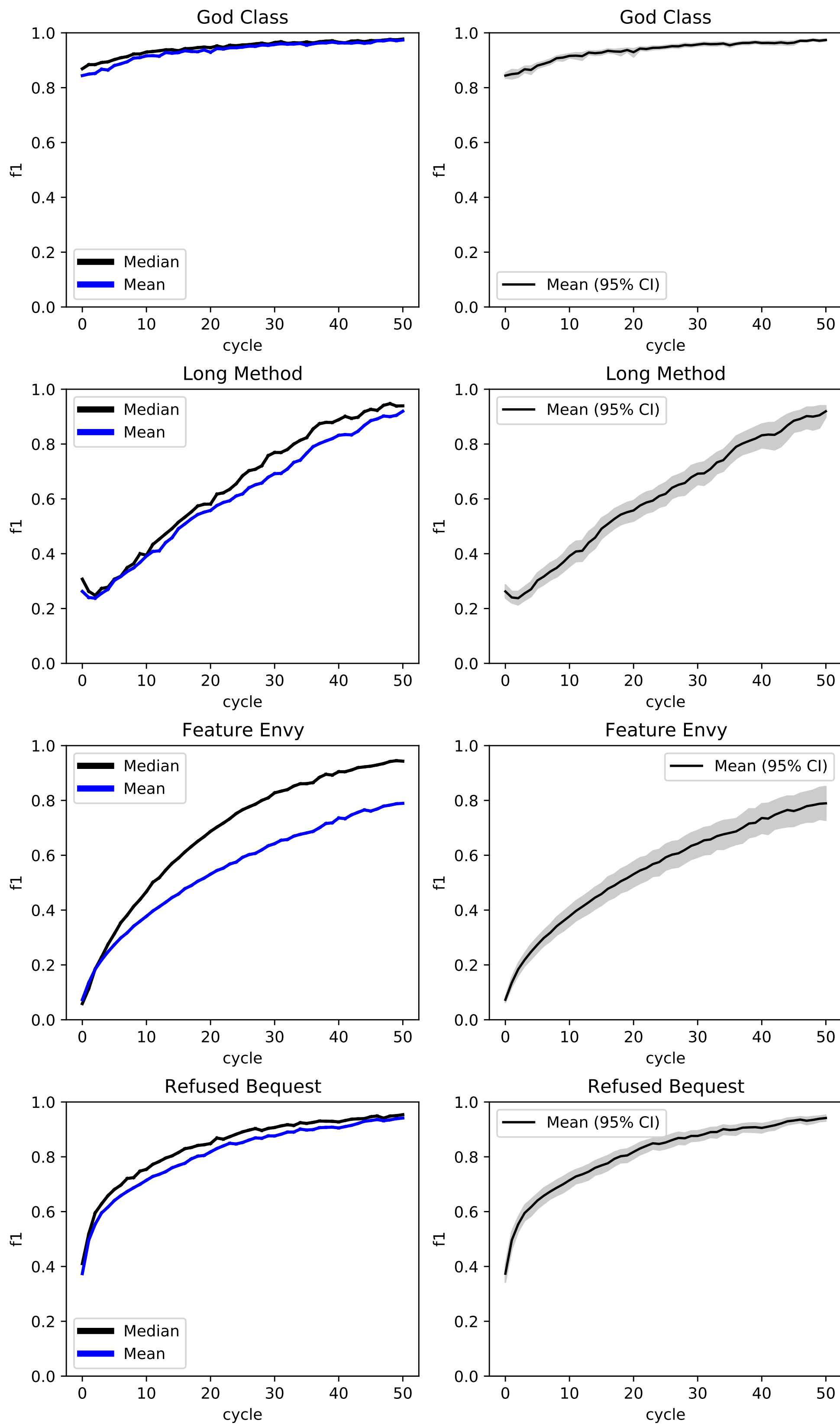

Figure 2. Continuous Feedback Results

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 asymptote close to the maximum F1 possible. We can understand this behavior as there is 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 on 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 can 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 can also observe a 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 can 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 can 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 av-

erage F1 of 0.94. The selection of the feedback sample also did not impact the results in general, as we can see in the confidence interval band from the chart in the right column. Through these analyses, we can answer the RQ2 as follows.

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 [37].

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.

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. However, 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.

Conclusion Validity. Regarding the conclusion validity of the single training 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. Recently, the adoption of machine learning techniques to detect code smells has became a trend [21, 26, 27, 3, 13, 15, 10]. For instance, Khomh et al. [20] proposed a Bayesian approach which initially converts existing detection rules to a probabilistic model to perform the predictions. Khomh et al. [21] later extended their previous work [20] by the introduction of Bayesian Belief Networks, improving the accuracy of the detection. Maiga et al. [26, 27] proposed an SVM-based approach that uses the feedback information provided by practitioners. Amorim et al. [3] presented an experience report on the effectiveness of Decision Trees for detecting code smells. They choose these classifiers due to their interpretability [3]. 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 [13, 15, 10, 9] evaluated and compared the performance of different machine learning algorithms on distinct sets of systems. For instance, Fontana et al. [13, 15] performed a larger comparison of classifiers [15]. 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. [10] 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. Cruz et al. [9] contributed with further empirical evidences on the use of machine learning algorithms for code smells detection by using a larger dataset of curated systems.

Few studies address a feedback process to improve the detection of code smells as we do here. Hozano et al. [19] 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. [25] 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, a good topic to be investigated in future work would be to add noise to the feedback acquisition. That is, provide some wrong feedbacks to understand how the model deals with this noisy data. Another topic for future work is the possibility of prioritizing the 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). It is also important to expand the dataset to cover other code smells and more systems. Finally, it is worth mentioning as future work an experimental evaluation with humans; i.e., actual software experts. 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] An exploratory evaluation of continuous feedback to enhance machine learning code smell detection - online replication package, 2023. URL `https://zenodo.org/record/8122453`.

[2] M. Abbes, F. Khomh, Y. Gueheneuc, and 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.

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

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

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

[6] J. M. Bieman and BK. Kang. Cohesion and reuse in an object-oriented system. *Software Engineering Notes (SEN)*, 20:259–262, 1995.

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

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Transactions on Software Engineering*, 20:476–493, 1994.

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

[10] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and 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)*, pages 612–621, 2018.

[11] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and 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.

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

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

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

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

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

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

[18] M. Hitz and B. Montazeri. *Measuring coupling and cohesion in object-oriented systems*. Citeseer, 1995.

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

[20] F. Khomh, S. Vaucher, YG. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Int'l Conf. on Quality Software*, pages 305–314, 2009.

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

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

[23] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[24] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 131–142, 2008.

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

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

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

[28] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume)*, pages 77–80, 2005.

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

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

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

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

[33] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *International Conference on Software Maintenance and Evolution*, pages 101–110, 2014.

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

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

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

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

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

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