

Repositório ISCTE-IUL

Deposited in *Repositório ISCTE-IUL*:

2022-06-03

Deposited version:

Accepted Version

Peer-review status of attached file:

Peer-reviewed

Citation for published item:

Reis, J., Brito e Abreu, F. & Figueiredo Carneiro, G. (2022). Crowdsmeiling: A preliminary study on using collective knowledge in code smells detection. *Empirical Software Engineering*. 27 (3)

Further information on publisher's website:

[10.1007/s10664-021-10110-5](https://doi.org/10.1007/s10664-021-10110-5)

Publisher's copyright statement:

This is the peer reviewed version of the following article: Reis, J., Brito e Abreu, F. & Figueiredo Carneiro, G. (2022). Crowdsmeiling: A preliminary study on using collective knowledge in code smells detection. *Empirical Software Engineering*. 27 (3), which has been published in final form at <https://dx.doi.org/10.1007/s10664-021-10110-5>. This article may be used for non-commercial purposes in accordance with the Publisher's Terms and Conditions for self-archiving.

Use policy

Creative Commons CC BY 4.0

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in the Repository
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Crowdsmelling: A preliminary study on using collective knowledge in code smells detection

José Pereira dos Reis · Fernando Brito e Abreu · Glauco de Figueiredo Carneiro

Received: date / Accepted: December 20, 2021

Abstract Code smells are seen as a major source of technical debt and, as such, should be detected and removed. However, researchers argue that the subjectiveness of the code smells detection process is a major hindrance to mitigating the problem of smells-infected code.

This paper presents the results of a validation experiment for the Crowdsmelling approach proposed earlier. The latter is based on supervised machine learning techniques, where the wisdom of the crowd (of software developers) is used to collectively calibrate code smells detection algorithms, thereby lessening the subjectivity issue.

In the context of three consecutive years of a Software Engineering course, a total “crowd” of around a hundred teams, with an average of three members each, classified the presence of 3 code smells (Long Method, God Class, and Feature Envy) in Java source code. These classifications were the basis of the oracles used for training six machine learning algorithms. Over one hundred models were generated and evaluated to determine which machine learning algorithms had the best performance in detecting each of the aforementioned code smells.

Good performances were obtained for God Class detection (ROC=0.896 for Naive Bayes) and Long Method detection (ROC=0.870 for AdaBoostM1), but much lower for Feature Envy (ROC=0.570 for Random Forrest).

The results suggest that Crowdsmelling is a feasible approach for the detection of code smells. Further validation experiments based on dynamic learning are required to comprehensive coverage of code smells to increase external validity.

José Pereira dos Reis
ISTAR-Iscte, Instituto Universitário de Lisboa, Lisboa, Portugal
E-mail: jvprs@iscte-iul.pt

Fernando Brito e Abreu
ISTAR-Iscte, Instituto Universitário de Lisboa, Lisboa, Portugal
E-mail: fba@iscte-iul.pt

Glauco de Figueiredo Carneiro
Universidade Salvador (UNIFACS), Salvador, Bahia, Brazil
E-mail: glauco.carneiro@unifacs.br

Keywords Crowdsourcing · code smells · code smells detection · software quality · software maintenance · collective knowledge · machine learning algorithms

1 Introduction

Maintenance tasks are incremental modifications to a software system that aim to add or adjust some functionality or to correct some design flaws and fix some bugs. It has been found that feature addition, modification, bug fixing, and design improvement can cost as much as 80% of total software development cost (Travassos et al., 1999). In addition, it is shown that software maintainers spend around 60% of their time in understanding code (Zitzler et al., 2003). Therefore, as much as almost half ($80\% \times 60\% = 48\%$) of total development cost may be spent on understanding code. This high cost can be reduced by the availability of tools to increase code understandability, adaptability, and extensibility (Mansoor et al., 2017).

In software development and maintenance, especially in complex systems, the existence of code smells jeopardizes the quality of the software and hinders several operations such as code reuse. Code smells are not bugs, since they do not prevent a program from functioning, but rather symptoms of software maintainability problems (Yamashita and Moonen, 2013). They often correspond to the violation of fundamental design principles and may slow down software evolution (e.g. due to code misunderstanding) or increase the risk of bugs or failures in the future. Code smells can then compromise software quality in the long term by inducing technical debt (Bavota and Russo, 2016).

Many techniques and tools have been proposed in the literature for detecting code smells (Pereira dos Reis et al., 2021), but that detection faces a few challenges. The first is that code smells lack a formal definition (Wang et al., 2015). Therefore, their detection is highly subjective (e.g. dependent on the developer's experience). Second, due to the dramatic growth in the size and complexity of software systems in the last four decades (Humphrey, 2009), it is not feasible to detect code smells thoroughly without tools.

Several approaches and tools for detecting code smells have been proposed. Kessentini et al. (2014) classified those approaches into 7 categories: metric-based approaches, search-based approaches, symptom-based approaches, visualization based approaches, probabilistic approaches, cooperative based approaches, and manual approaches. The most popular code smells detection approach is metric-based. The latter is based on the application of detection rules that compare the values of relevant metrics extracted from the source code with empirically identified thresholds. However, these techniques present some problems, such as subjective interpretation, a low agreement between detectors (Fontana et al., 2012), and threshold dependability.

To overcome the aforementioned limitations of code smell detection, researchers recently applied supervised machine learning techniques that can learn from previous datasets without needing any threshold definition. The main impediment for applying those techniques is the scarcity of publicly available oracles, i.e. tagged datasets for training detection algorithms. To mitigate this hindrance, we have proposed *Crowdsourcing* (Reis et al., 2017), a collaborative crowdsourcing approach, based in machine learning, where the wisdom of the crowd (of software

developers) is used to collectively calibrate code smells detection algorithms. The applications based in collective intelligence, where the contribution of several users allows attaining benefits of scale and/or other types of competitive advantage, are gaining increasing importance in Software Engineering (Stol and Fitzgerald, 2014) and other areas (Bigham et al., 2014; Bentzien et al., 2013). The most notable examples of crowdsourcing in Software Engineering are *crowdtesting* (Sharma and Padmanaban, 2014) and *code snippets recommendation* (Proksch et al., 2014).

In this paper we present the first results of applying *Crowdsmelling* in practice. The paper is organized as follows: next section introduces the related work; then, section 3 describes the experiment; results and corresponding analyses and the answers to the research questions are presented in section 4; discussing the results and threats to the validity are presented in section 5; and the concluding remarks, as well as scope for future research, are presented in section 6.

2 Related Work

The related work is organised in two subsections and chronologically within each one.

2.1 Crowd-based approaches

Palomba et al. (2015) presented LANDFILL, a Web-based platform for sharing code smell datasets, and a set of APIs for programmatically accessing LANDFILL's contents. This platform was created due to the lack of publicly available oracles (sets of annotated code smells). The web-based platform has a dataset of 243 instances of five types of code smells (Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy) identified from 20 open source software projects and a systematic procedure for validating code smell datasets. LANDFILL allows anyone to create, share, and improve code smell datasets.

Oliveira et al. (2016) performed a controlled experiment involving 28 novice developers, aimed at assessing the effectiveness of collaborative practices in the identification of code smells. The authors used Pair Programming (PP) and Coding Dojo Randori (CDR), which are two increasingly adopted practices for improving the effectiveness of developers with limited or no knowledge in Software Engineering tasks, including code review tasks, and compared this two practices (PP and CDR) with solo programming in order to better distinguish their impact on the effective identification of code smells. The results suggest that collaborative practices contribute to the effectiveness on the identification of a wide range of code smells. for nearly all types of inter-class smells, the average of smells identified by novice pairs or groups outperformed at least in 40% of the corresponding average of smells identified by individuals and collaborative practices tend to increase the rate of success in identifying more complex smells. In the same year Oliveira (2016) performed a research based on a set of controlled experiments conducted with more than 58 novice and professional developers, with the aim of knowing how to improve the efficiency in the collaborative identification of code smells, and reached the same conclusions as the first study.

Oliveira et al. (2017) in this paper reports an industrial case study aimed at observing how 13 developers individually and collaboratively performed smell identification in five software projects from two software development organizations. The results are in line with previous studies by these authors, where they suggest that collaboration contributes to improving effectiveness on the identification of a wide range of code smells.

de Mello et al. (2017) presents and discusses a set of context factors that may influence the effectiveness of smell identification tasks. The authors presented an initial set of practical suggestions for composing more effective teams to the identification of code smells. These suggestions are, i) be sure all team professionals are aware of the code smell concepts applied in the review, ii) be sure all team professionals are aware of the relevance of identifying code smells, iii) take preference to use collaboration in the reviews, iv) include professionals that had worked in the module and professionals without such experience, v) include professionals with different professional roles.

Tahir et al. (2018) presented a study where they investigate how developers discuss code smells and anti-patterns over Stack Overflow to understand better their perceptions and understanding of these two concepts. In this paper, both quantitative and qualitative techniques were applied to analyze discussions containing terms associated with code smells and anti-patterns. The authors reached conclusions like: i) developers widely use Stack Overflow to ask for general assessments of code smells or anti-patterns, instead of asking for particular refactoring solutions, ii) developers very often ask their peers ‘to smell their code’ (i.e., ask whether their own code ‘smells’ or not), and thus, utilize Stack Overflow as an informal, crowd-based code smell/anti-pattern detector, iii) developers often discuss the downsides of implementing specific design patterns, and ‘flag’ them as potential anti-patterns to be avoided. Conversely, the authors also found discussions on why some anti-patterns previously considered harmful should not be flagged as anti-patterns, iv) C#, JavaScript and Java were the languages with most questions on code smells and anti-patterns, constituting 59% of the total number of questions on these topics, v) Blob, Duplicated Code and Data Class are the most frequently discussed smells in Stack Overflow, vi) when authors analyzed temporal trends in posts on code smells and anti-patterns in Stack Overflow, show that there has been a steady increase in the numbers of questions asked by developers over time.

Oliveira et al. (2020) have carefully designed and conducted a controlled experiment with 34 developers. The authors exploited a particular scenario that reflects various organizations: novices and professionals inspecting systems they are unfamiliar with. They expect to minimize some critical threats to validity of previous work. Additionally, they interviewed 5 project leaders aimed to understand the potential adoption of the collaborative smell identification in practice. Statistical testing suggests 27% more precision and 36% more recall through the collaborative smell identification for both novices and professionals. The interviews performed by the authors showed that leaders would strongly adopt the collaborative smell identification. However, some organization and tool constraints may limit such adoption.

Baltes and Treude (2020) presented a study with similarities and differences between code clones in general and code clones on Stack Overflow and point to open questions that need to be addressed to be able to make data-informed deci-

sions about how to properly handle clones on this important platform. The results of his first preliminary investigation indicated that clones in Stack Overflow are common, diverse, similar to clones in regular software projects, affect the maintainability of posts and can lead to licensing issues. The authors further point to specific challenges, including incentives for users to clone successful answers and difficulties with bulk edits on the platform.

2.2 Multiple ML models based approaches

Regarding the use of the machine learning approach in the detection of code smells, most studies only use one algorithm, being the most usual algorithm the decision trees. We will present below the most relevant studies that use multiple machine learning algorithms.

Some of the most relevant studies in the area of machine learning were performed by Fontana et al. (2013, 2015). In the first work Fontana et al. (2013), they outlined some common problems of code smell detectors and described the approach they were following based on machine learning technology. In this study the authors focused on 4 code smells (Data Class, Large Class, Feature Envy, Long Method), considered 76 systems for analysis and validation and experimented 6 different machine learning algorithms. The results with a use of 10-fold cross-validation to assess the performance of predictive models shown that J48, Random Forest, JRip and SMO have accuracy values greater than 90% for the 4 code smells, and on average they have the best performances. In the second work Fontana et al. (2015), they performed the largest experiment of applying machine learning algorithms. They experimented 16 different machine-learning algorithms on four code smells (Data Class, Large Class, Feature Envy, Long Method) and 74 software systems, with 1986 manually validated code smell samples. They found that all algorithms achieved high performances in the cross-validation data set, yet the highest performances were obtained by J48 and Random Forest, while the worst performance were achieved by support vector machines. The authors concluded that the application of machine learning to the detection of these code smells can provide high accuracy (>96 %), and only a hundred training examples are needed to reach at least 95 % accuracy. The authors interpret the results as an indication that “using machine learning algorithms for code smell detection is an appropriate approach”.

Di Nucci et al. (2018) replicated the Fontana et al. (2015) study with a different dataset configuration. The dataset contains instances of more than one type of smell, with a reduced proportion of smelly components and with a smoothed boundary between the metric distribution of smelly and non-smelly components, and therefore more realistic. The results revealed that with this configuration the machine learning techniques reveal critical limitations in the state of the art which deserve further research. They concluded that, when testing code smell prediction models on the revised dataset, they noticed: i) accuracy of all the models is still noticeably high when compared to the results of the reference study (on average, 76% vs 96%), ii) that performances are up to 90% less accurate in terms of F-Measure than those reported in the Fontana et al. study. Thus, the problem of detecting code smells through the adoption of machine learning techniques may

still be worthy of further attention, e.g., in devising proper machine learning-based code smell detectors and datasets for software practitioners.

To the best of our knowledge, namely obtained while performing a systematic literature review on code smells detection techniques Pereira dos Reis et al. (2021), there is no study that uses a collective knowledge-based approach to detect code smells automatically, i.e. based on machine learning, with a dataset increment over 3 years. The use of groups of people in code smells detection is typically used in manual detection approaches and in the construction of oracles (a tagged dataset for training detection algorithms). A distinctive feature of our approach is the use of crowds. While in related work a group of 3 to 5 people is typically used to build an oracle, we used hundreds, thus embodying a much larger large diversity of opinions. However, manual techniques are human-centric, tedious, time-consuming, and error-prone. These techniques require a great human effort, therefore not effective for detecting code smells in large systems.

3 Experiment Planning

3.1 Research Questions

The concept of Crowdsampling – use of collective intelligence in the detection of code smells – aims to mitigate the aforesaid problems of subjectivity and lack of calibration data required to obtain accurate detection model parameters, by using machine learning techniques. We have formulated the following research questions to assess the feasibility of Crowdsampling:

- **RQ1:** What is the performance of machine learning techniques when trained with data from the crowd and therefore more realistic?
- **RQ2:** What is the best machine learning model to detect each one of the three code smells?
- **RQ3:** Is it possible to use Collective Knowledge for code smells detection?

The goal of these RQs is to understand if our Crowdsampling approach is feasible. For this, it is fundamental to understand the performance of machine learning techniques (RQ1), which will make our approach feasible. However, it is always important to know, in addition to performance, if there is any tendency towards algorithms (RQ2). If it is found that there is a tendency for one algorithm to overlap with the others, in the future we can simplify our research, focusing on fewer algorithms. This aspect will also propose the simplification of an application that automates this approach. Finally, based on this data, we intend to determine the feasibility of this approach in detecting code smells (RQ3).

3.2 Participants

Our approach consists of several teams using a tool, as an advisor, to detect code smells and then confirming the validity of the detection manually. In addition to the code smells detected by the advisor tool, teams can always add other code smells manually. In the end, code identification, code metrics and classification (presence or absence of code smells) are saved by creating an oracle for each code

smell. This oracle will allow training machine learning algorithms for code smells detection. These oracles have been increased for 3 years, with data collected each year.

The repetition of this process for 3 years, allowed every year to increase the oracle with data from new teams, thus increasing the variability of existing classifications. This variability of opinions in the code smells classification is very important, because it will allow collecting data from teams with different opinions on the definition of code smells, enriching the oracle.

Our subjects were the finalists (3rd year) of a B.Sc. degree in computer science at the ISCTE-IUL University, attending a compulsory Software Engineering course. They had similar backgrounds as they have been trained across the same set of courses along their academic path. However, there are differences between the students, as the skills and experience in code development are different. The knowledge about code smells was acquired in the Software Engineering curricular unit.

Table 1: Teams whose code smells detection was included in the oracles

Year	Number of teams	Total number of elements
2018	8	31
2019	51	152
2020	44	179

The teams had a variable size depending on the year (see Table 1) and the number of participants were increasing each year. In 2018, 19 teams were formed, essentially with 4 elements each, for a total of 73 elements, but in the end, only the data from 8 teams, for a total of 31 elements were used for the oracle. In subsection 3.3 we explain why the data from 11 teams were not used. In 2019 we had 51 teams, mainly made up of 3 members, with a total of 152 members. In 2020 we had 44 teams, mainly made up of 6 members, with a total of 179 members. These teams were requested to complete a code smells detection assignment.

3.3 Data

Participants were invited to perform the detection of 3 code smells (God Class, Feature Envy, Long Method) in a code extract (e.g. of their choice). They used JDeodorant¹ as an auxiliary tool in the detection. The use of tools to help detect code smells in the process of creating oracles is usual. For example, in the Fontana et al. (Fontana et al., 2015) study, 5 advisors were used, depending on the code smell that was intended to be detected. We chose to use JDeodorant because it detects refactoring opportunities for the 3 code smells we used, because it is one of the best known and used tools, as we can see in the paper by Tsantalis et al. (2018). To account for individual judgement in the oracle, teams could either decide to accept (true positives) or not (false positives) the tool suggestions or add additional manual detections (false negatives).

¹ <https://users.encs.concordia.ca/nikolaos/jdeodorant/>

In 2018 each team chose the Java project where they wanted to do code smells detection from a list of 8 open-source projects. The latter have already been used in other studies, namely in those using machine learning approaches mentioned in the related work section (Fontana et al., 2013, 2015; Di Nucci et al., 2018). However, in the end, only 3 projects/versions were considered: jasml-0.10², jgrapht-0.8.1³ and jfreechart-1.0.13⁴. We discarded the collected data from the other projects, chosen by 11 teams (42 participants), since those teams used diversified versions and, therefore, the collected metrics were not consistent from version to version, and that would be a validity threat. In the next 2 years we just used Jasml-0.10 to avoid the aforementioned issue.

The results of each team’s detection are saved in a file with the following fields for the code smells Feature Envy and Long Method: Team number, project, package, class, method, 82 metrics of code, is code smell. In the case of the code smell God Class, as the scope is to the class, does not have the method field and 61 code metrics are saved. At the end we have 3 files, one for each smell code.

The data obtained each year serve to reinforce the calibration datasets of the machine learning algorithms, with the objective of improving their detection performance over time. This way we will have several datasets, so we can evaluate which one gives the best results for each code smell.

Table 2: Datasets (Oracles) and their composition

Dataset	Code smell	N ^o Cases	True	% True	False	% False
2018	Feature Envy	10	3	30%	7	70%
2019	Feature Envy	197	110	56%	87	44%
2019+2018	Feature Envy	207	113	55%	94	45%
2020	Feature Envy	123	79	64%	44	36%
2020+2019	Feature Envy	320	189	59%	131	41%
2020+2019+2018	Feature Envy	330	192	58%	138	42%
2018	God class	22	8	36%	14	64%
2019	God class	129	74	57%	55	43%
2019+2018	God class	151	82	54%	69	46%
2020	God class	136	84	62%	52	38%
2020+2019	God class	265	158	60%	107	40%
2020+2019+2018	God class	287	166	58%	121	42%
2018	Long Method	59	24	41%	35	59%
2019	Long Method	414	180	43%	234	57%
2019+2018	Long Method	473	204	43%	269	57%
2020	Long Method	853	350	41%	503	59%
2020+2019	Long Method	1267	530	42%	737	58%
2020+2019+2018	Long Method	1326	554	42%	772	58%

In Table 2 we present the composition of the datasets, indicating the following elements, i) name of the dataset, ii) code smell to which the dataset refers, iii) number of cases, iv) number of true instances, v) percentage of true instances, vi) number of false instances, vii) percentage of false instances. Each dataset is identified by the year or the years that constitute it, for example, 2019 is the dataset of

² <http://jasml.sourceforge.net/>

³ <https://jgrapht.org/>

⁴ <https://www.jfree.org/>

the year 2019 and 2019+2020 is the dataset resulting from the aggregation of the datasets of the years 2019 and 2020. Unlike several authors, such as Fontana et al. (2015), we do not normalize our datasets in size in order to balance the number of positive and negative instances. Even with the risk of getting worse results, we used the datasets with all the cases classified by the teams. Thus, we believe that we are reproducing the reality of the teams' thinking about code smells. The size of the datasets varies widely depending on the type of the code smell. Since the datasets of code smell Feature Envy are too small, i.e., for a code smell in the scope of the method, they do not have a large enough variance of cases, it was not possible to obtain good results. Even so, we intend to use all the datasets, as they represent the obtained reality and serve as a basis for a future amplification and evolution of the crowd's study in code smells detection.

The 18 datasets are available on GitHub⁵.

3.4 Code Smells

In this study we considered three different types of code smells defined by Fowler et al. (1999):

- God Class. This smell characterises classes having a large size, poor cohesion, and several dependencies with other data classes of the system. Class that has many responsibilities and therefore contains many variables and methods. The same Single Responsibility Principle (SRP) also applies in this case;
- Feature Envy. When a method is more interested in members of other classes than its own, is a clear sign that it is in the wrong class;
- Long Method. Methods implementing more than one functionality are affected by this smell. Very large method/function and, therefore, difficult to understand, extend and modify. It is very likely that this method has too many responsibilities, hurting one of the principles of a good Object Oriented design (SRP: Single Responsibility Principle);

The choice of these 3 code smells is due to the fact that, according to the Systematic Literature Review we conducted, they are the three most detected code smells Pereira dos Reis et al. (2021). Therefore, it is easier for teams to obtain documentation and understand these 3 code smells for better detection.

3.5 Code Metrics

In this study, we used the same metrics that were used in the study of Fontana et al. (2015), since the metrics are publicly available.

The metrics extracted from the software which constitute the independent variables in the machine learning algorithms, are at class, method, package and project level. For God Class, we used a set of 61 metrics, and for the other two code smells, Feature Envy and Long Method, we used a set of 82 metrics, plus 21 metrics than God Class, since these codes smells are at the method level. The main metrics are described in the table Appendix A..

⁵ <https://github.com/dataset-cs-surveys/Crowdsmelling>

3.6 Machine Learning Techniques Experimented

The application used in this experiment to train and evaluate machine learning algorithms was Weka (open source software from Waikato University) (Hall et al., 2009), and the following algorithms available in Weka were implemented:

- J48 (Quinlan, 2014) is an implementation of the C4.5 decision tree, and its three types of pruning techniques: pruned, unpruned and reduced error pruning;
- Random Forest (Breiman, 2001) consists of a large number of individual decision trees, a forest of random trees, that operate as an ensemble;
- AdaBoostM1 (Freund and Schapire, 1996) Boosting works by repeatedly running a given weak learning algorithm on various distributions over the training data, and then combining the classifiers produced by the weak learner into a single composite classifier. Weka uses the Adaboost M1 method;
- SMO (Platt, 1999) is a Sequential Minimal Optimization algorithm widely used for training support vector machines. We use the Polynomial kernel;
- Multilayer Perceptron (Rumelhart et al., 1986) is a classifier that uses back-propagation to learn a multi-layer perceptron to classify instances;
- Naïve Bayes (John and Langley, 1995) is a probabilistic model based on the Bayes theorem.

Experiments were performed to evaluate the performance values of the machine learning algorithms used with their default parameters for each type of code smell. Also, no feature selection technique was used.

3.7 Model Evaluation

To assess the capabilities of the machine learning model, we adopted 10-Fold Cross Validation (Stone, 1974). This methodology randomly partitions the data into 10 folds of equal size, applying a stratified sampling (e.g., each fold has the same proportion of code smell instances). A single fold is used as test set, while the remaining ones are used as training set. The process was repeated 10 times, using each time a different fold as test set. The result of the process described above consisted of a confusion matrix for each code smell type and for each model (Pecorelli et al., 2019).

Several evaluation metrics can be used to assess model quality in terms of false positives/negatives (FP/FN), and true classifications (TP/TN). However, commonly used measures, such as **Accuracy**, **Precision**, **Recall** and **F-Measure**, do not perform very well in case of an imbalanced dataset or they require the use of a minimum probability threshold to provide a definitive answer for predictions. For these reasons, we used the **ROC**⁶, which is a threshold invariant measurement. Nevertheless, for general convenience, we kept present in results tables all the evaluation metrics (Caldeira et al., 2020).

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \quad (1)$$

⁶ Receiver operating characteristic (**ROC**) is a curve that plots the true positive rates against the false positive rates for all possible thresholds between 0 and 1.

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F - measure = 2 * \frac{Recall * Precision}{Recall + Precision} \quad (4)$$

ROC gives us a 2-D curve, which passes through (0, 0) and (1, 1). The best possible model would have the curve close to $y = 1$, with an area under the curve (**AUC**) close to 1.0. **AUC** always yields an area of 0.5 under random-guessing. This enables comparing a given model against random prediction, without worrying about arbitrary thresholds, or the proportion of subjects on each class to predict (Rahman and Devanbu, 2013).

3.8 Process

In this subsection, we describe the three stages that constitute the process adopted in this exploratory study.

3.8.1 Stage 1: Developer - Code smell classification

All Java developers use the Eclipse IDE with the JDeodorant plug-in installed. In the first year - the year 2018 - each team was free to choose, from a list of Java projects, the one they wanted to use to detect code smells. So, in the first year, the teams chose the Java projects jasml-0.10, jgrapht-0.8.1, and jfreechart-1.0.13. In the following two years, the teams just used jasml-0.10.

Figure 1 shows the code smells classification process by the programmer, where we can see that after importing the Java project, the participants were invited to perform the detection of 3 Code smell (Long Method, God Class, Feature Envy). In 2018, detections of code smells were performed in the 3 Java projects as follows: a) Long Method, 5 teams detected this smell in jasml-0.10, 2 teams detected it in jfreechart-1.0.13, and all detections performed in jgrapht-0.8. 1 were not used for the reasons described in subsection 3.3; b) God Class, 4 teams detected this smell in jasml-0.10, 2 teams detected it in jfreechart-1.0.13, and 1 team detected it in jgrapht-0.8.1; c) Feature Envy, only the detections made by 4 teams in jasml-0.10 were used, all other detections were discarded for the reasons already mentioned. In the following 2 years, all teams detected all 3 code smells in jasml-0.10.

In this detection the participants could use JDeodorant as an auxiliary tool in the detection of smells, i.e., they first used JDeodorant as an advisor, and then manually validated the result of the detection of the tool, saying whether or not they agreed with the code smells detected. JDeodorant detects refactoring opportunities (refactoring is a controlled technique for improving the design of an existing code base (Fowler et al., 1999)), consequently, when JDeodorant detects a refactoring opportunity it is detecting a code smell candidate. The use of JDeodorant also had the advantage that participants could export the code smells identified by this tool to a text file, where they later registered their agreement or not with this identification, i.e., they performed the final classification.

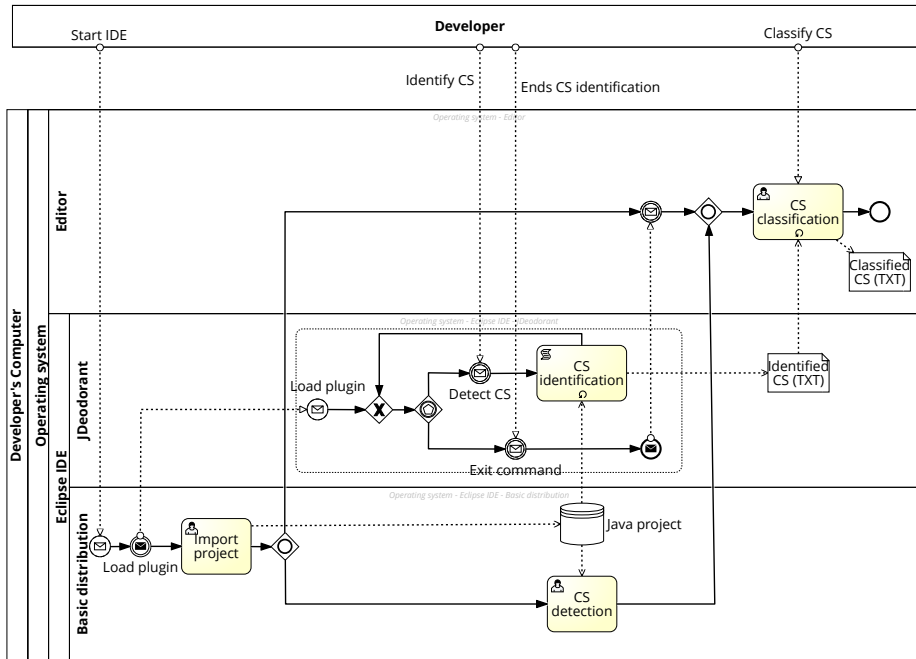


Fig. 1: Process of code smells classification by the developer

Regardless of the use of JDeodorant, all participants could identify the code smells directly in the Java project code (using the code metrics) and record their occurrence or not in a text file. In this case, the participants wrote in the text file the name of the class or method, and if there existed or not a code smell. The percentage of teams that performed code smells detection without the help of the JDeodorant advisor was 7%. Although, the work of detecting code smells without the use of the advisor is higher we found that, on average, the teams that did not use the advisor detected 30% more Long Methods and 20% more God Class. Regarding Feature Envy, the detection was on average 16% less than the teams that used JDeodorant.

With the use of JDeodorant, as advisor, in detecting smells, there is a risk that teams will only classify code smells resulting from advisor detection, in our case, code smells candidates detected by JDeodorant. To mitigate this risk the teams were asked to classify all classes and methods in a project package, thus extending the classification to cases not detected by JDeodorant. Another factor that minimizes this risk is the fact that JDeodorant identifies refactoring opportunities in code that is clearly not code smell, but the code can still be improved. This fact causes in JDeodorant's detection result a larger scope of cases and consequently a larger disagreement between the teams' classification and JDeodorant's identification. In the detection of the Long Method, the degree of disagreement with JDeodorant in the year 2018 was 66% (highest disagreement), and in the year 2019, it was 49% (lowest disagreement), being in total for the three years 54%. For God Class, the disagreement with JDeodorant for the three years was

47% and varied from 68% in the year 2018 to 40% in 2020. In Feature Envy the disagreement ranged from 70% in 2018 to 34% in 2020, being in the three years 45%.

Regarding the code classified by the teams, methods, and classes of the applications, we found that the majority was classified by more than one team. In the first year, 2018, due to the diversity of Java projects used there was a greater dispersion of the code classified, with most classes (75%) and methods (76%) classified by only one team. Consequently, the most used class was classified by four teams, and the most used method in the classification was classified by six teams. The next two years saw a reversal, with most classes and methods being classified by more than one team. Regarding classes, 60% in the year 2019 and 75% in the year 2020 were classified by more than one team, with the most used class being classified by 43 teams. Regarding methods, 85% and 60% were classified by more than one team in 2020 and 2019, respectively, with the most-used method being classified by 44 teams.

The time given to the teams to classify the three code smells was three weeks, and no indication was given on how they should work as a team, that is, how they should divide the code smells classification among the various team members. Hence, based on the data obtained from the experiment, we are not able to identify specifically which members performed a specific code analysis. However, we were able to identify which code smells were analyzed. For example, according to data available in GitHub⁷, it is possible to identify that in the 2020 Long Method dataset, the `private void consumeDigits()` method was classified by 37 teams, by applying a filter to the method field. We have made available on GitHub the file `code-classification-statistics.csv` with a set of statistics about the percentages of teams that classified the methods and classes. We also found that the teams divided the classification of the three code smells among their members, for example, when the team had six members, they created groups of two members, and each group classified one code smell in the code. In this way, the teams increased the reliability of the classification, since the code was classified by 2 team members.

As a result of this stage, all teams produced three files - one for each code smell - with the classification of a set of methods and classes of the Java project, i.e., with the record of the existence or not of code smells in those classes or methods. This stage was performed over 3 years, 2018, 2019 and 2020.

3.8.2 Stage 2: Researcher - Evaluation of machine learning models

After collecting data in three years, we proceeded to the second phase, which aimed to produce the datasets for the 3 code smells and evaluation of the different machine learning techniques. In figure 2 is represented the whole process of this second stage.

The first task to be performed by the researcher is the creation of the datasets described in section 3.3.

The creation of the datasets is done by joining all the text files with the classifications of a code smell, produced by the teams of each year, in a single Excel file. Then, to this excel file are added the code metrics for the methods or classes (see section 3.5), depending on the scope of the code smell to which the dataset

⁷ <https://github.com/dataset-cs-surveys/Crowdsmelling>

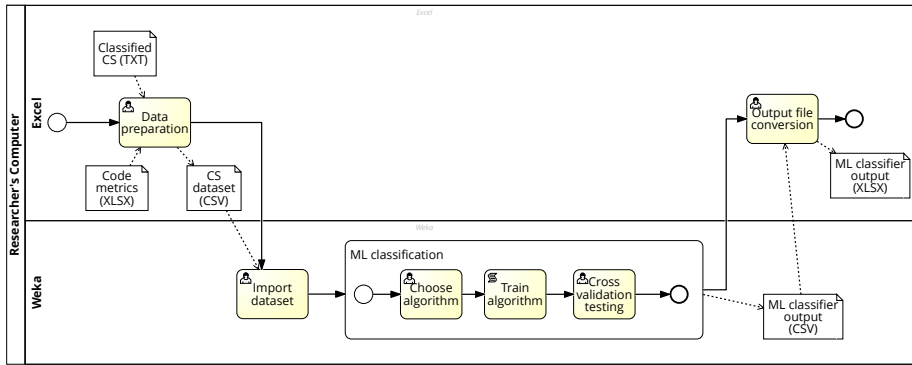


Fig. 2: Process of creation of the datasets and evaluation of the machine learning techniques by the researcher

belongs. Thus, in a first step, datasets are created - usually called oracles - with the data for each year, for each of the three code smells, for a total of 6 datasets. These datasets have been given the name of the year to which they belong, i.e., 2018, 2019 and 2020. In a second step, we proceed to aggregate the dataset of the year with those of previous years to make the dataset larger, increasing the number of instances. In the end, we created six datasets for each code smell, with a total of 18 datasets (see table 2).

After creating the datasets, we proceed to the creation and evaluation of the machine learning models using Weka (open source software from Waikato University) (Hall et al., 2009). To import datasets into Weka, we convert the datasets files, from excel XLSX to CSV. At Weka, we trained the six algorithms described in section 3.6, with each of the 18 datasets, and evaluated the model produced using the 10-Fold Cross Validation methodology. In the end, 36 machine learning models were created for each code smell, with a total of 108 models for the three code smells. Finally, all the metrics (Accuracy, Precision, Recall, F-Measure, and ROC) resulting from the evaluation of each model were saved in the "ML classifier output" file (see section 3.7).

3.8.3 Stage 3: Researcher - Model variance test

To check if there were significant differences between the classifications presented by the different models, we proceeded to the analysis of variance through a one-way analysis of variance (ANOVA) (see figure 3).

To test the variance between the machine learning models we use the ROC value. Thus, the first step was to produce a data file, for each code smell, with the identification of the machine learning models and the respective ROC. This file was created aggregating the results of the evaluations of all models produced by Weka, by code smell.

To analyze if there were differences between the classifications of the machine learning models for each code smell, we performed an analysis of variance using a one-way analysis of variance (ANOVA) test in the IBM SPSS Statistics 27 software.

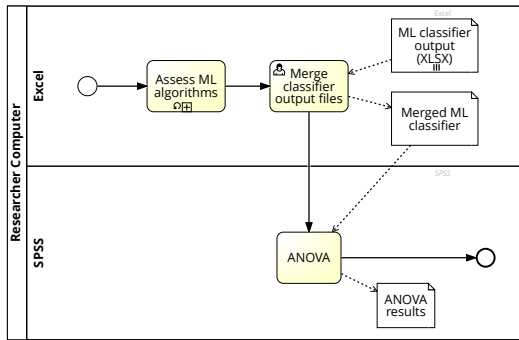


Fig. 3: Process of testing the variance between machine learning models

4 Results

In this section, we present the experiment results with respect to our research questions.

4.1 RQ1. What is the performance of machine learning techniques when trained with data from the crowd and therefore more realistic?

In this **RQ** we will evaluate the performance of the 36 models for each code smell in a total of 108 models. These models resulted from the training of the 6 machine learning algorithms (J48, Random Forest, AdaBoostM1, SMO, Multilayer Perceptron, Naïve Bayes), described in section 3.6, by the datasets presented in table 2. These algorithms were trained with the various datasets resulting from the crowd, and as explained in 3.3 we want these datasets to be as real as possible, to represent as faithfully as possible what the detection teams think about the code smells.

From the various existing metrics for evaluating machine learning models, we have chosen to use the ROC as the primary metric, but we also use accuracy, precision, recall, and f-measure. For testing, we used the 10-Fold Cross Validation, for the reasons presented in 3.7.

4.1.1 Performance of machine learning techniques for the code smell Long Method

Starting by analyzing the machine learning techniques for the Long Method data, described in Table 3, we observed that the best results were obtained by the Random Forrest and AdaBoostM1 algorithms. The best result with a ROC of 0.870 was obtained by AdaBoostM1 when trained by the dataset 2020, followed by the Random Forrest with ROC of 0.869 for the same dataset. For the dataset 2018, the best result was also that of AdaBoostM1. However, the most uniform algorithm was Random Forrest, with the best results in 4 of the 6 datasets (2020+2019+2018, 2020+2019, 2019+2018, 2019) and for the dataset 2020, the difference for AdaBoostM1 is insignificant (0.001). The Multilayer Perceptron and J48 algorithms, were two other algorithms to present ROC results above 0.800. Especially the

Multilayer Perceptron algorithm which for the datasets of the year 2020 presented an ROC between 0.868 and 0.822.

Table 3: Long Method: ROC Area results for the machine learning algorithms trained by the 3 years datasets

Algorithm	year dataset	2020			2019		2018
		2020+2019+2018	2020+2019	2020	2019+2018	2019	2018
J48		0.792	0.801	0.832	0.677	0.678	0.617
Random Forest		0.828	0.828	0.869	0.684	0.679	0.671
AdaBoostM1		0.807	0.818	0.870	0.665	0.673	0.707
SMO		0.753	0.753	0.803	0.634	0.649	0.524
MultilayerPerceptron		0.822	0.822	0.868	0.683	0.667	0.604
NaiveBayes		0.736	0.742	0.783	0.584	0.614	0.471

The worst results were obtained by the NaiveBayes algorithm with ROC between 0.783 and 0.471. The second worst algorithm was SMO, with ROC results between 0.803 and 0.524.

In table 3, we can still observe that the best results were obtained when the algorithms were trained with the datasets for the year 2020, with ROC of 0.870 for the dataset 2020 and ROC of 0.828 for the datasets 2020+2019+2018 and 2020+2019. In opposition is the year 2019, with the worst results, ROC of 0.684 and 0.679 for the datasets 2019+2018, 2019, respectively.

4.1.2 Performance of machine learning techniques for the code smell God Class

Table 4 shows the results of the machine learning techniques for the God Class data. The best result was obtained by the NaiveBayes algorithm, when trained by the dataset 2020, with the ROC value of 0.896. The algorithms that obtained the best performances were NaiveBayes and MultilayerPerceptron, with the best result in 3 of the datasets each one. NaiveBayes obtained the best results for the datasets 2020, 2020+2019, 2019, with ROC values of 0.896, 0.859 and 0.804, respectively. Also with the best result in 3 datasets (2020+2019+2018, 2019+2018, 2018) the MultilayerPerceptron algorithm presented ROC values between 0.768 and 0.885. The Random Forest and AdaBoostM1 algorithms presented their best ROC values of 0.893 and 0.876, respectively, for the dataset 2020.

The worst results were presented by J48 and SMO, with their best ROC values for the dataset 2020 of 0.759 and 0.857, respectively.

Regarding the datasets that presented the best results were those of the year 2020, with the dataset only with data of the year 2020 being the best (dataset 2020) with ROC values between 0.896 and 0.791. The dataset with the worst results was 2018, with the ROC between 0.491 and 0.0768.

4.1.3 Performance of machine learning techniques for the code smell Feature Envy

The ROC Results for the machine learning algorithms trained by the 3-year datasets for the code smell Feature Envy are presented in table 5. Feature Envy

Table 4: God Class: ROC Area results for the machine learning algorithms trained by the 3 years datasets

		2020			2019		2018
year							
dataset							
Algorithm		2020+2019+2018	2020+2019	2020	2019+2018	2019	2018
J48		0.763	0.759	0.791	0.693	0.725	0.692
Random Forest		0.853	0.850	0.893	0.781	0.802	0.491
AdaBoostM1		0.854	0.857	0.876	0.771	0.793	0.571
SMO		0.815	0.800	0.857	0.716	0.751	0.741
MultilayerPerceptron		0.880	0.853	0.885	0.805	0.797	0.768
NaiveBayes		0.731	0.859	0.896	0.669	0.804	0.651

detection results are low, with the Random Forest algorithm having the best ROC value of 0.570 when trained by dataset 2019. As already explained in point 3.3, the datasets for Feature Envy are very small, considering the variance of cases, however we are convinced that when we have bigger datasets the results will be better. The machine learning algorithms showed better results when trained with the datasets of the year 2019, with ROC values between 0.570 and 0.508.

Table 5: Feature Envy: ROC Area results for the machine learning algorithms trained by the 3 years datasets

		2020			2019		2018
year							
dataset							
Algorithm		2020+2019+2018	2020+2019	2020	2019+2018	2019	2018
J48		0.518	0.484	0.467	0.552	0.563	0
Random Forest		0.539	0.494	0.486	0.542	0.570	0
AdaBoostM1		0.498	0.437	0.468	0.554	0.548	0
SMO		0.520	0.491	0.500	0.551	0.508	0
MultilayerPerceptron		0.533	0.498	0.536	0.548	0.544	0
NaiveBayes		0.524	0.519	0.482	0.548	0.547	0

4.1.4 The one-way analysis of variance (ANOVA)

To determine if there were significant differences between the performance of machine learning techniques when trained with data from the crowd and therefore more realistic, a One-way ANOVA was conducted to compare effect of machine learning techniques on the ROC. Before performing the ANOVA, we checked all the assumptions for its application, namely, the inexistence of outliers, the normality of the distribution (Shapiro-Wilk test), and the homogeneity of variances (Levene's test). All assumptions were fulfilled and the following results were obtained:

i) For the code smell Long Method, an analysis of variance showed that the effect of the performance of machine learning techniques on ROC value was not significant, $F(5,30)=1.096$, $p=.383$.

ii) For the code smell God Class, an analysis of variance showed that the effect of the performance of machine learning techniques on ROC value was not significant, $F(5,30)=.655$, $p=.660$.

ii) For the code smell Feature Envy, an analysis of variance showed that the effect of the performance of machine learning techniques on ROC value was not significant, $F(5,24)=.585$, $p=.712$.

The results of the variance tests show there was no statistically significant difference between the performance of the six machine learning models, when trained with data from the crowd and therefore more realistic.

4.1.5 Summary of RQ1 results

For the code smell Long Method the best result with a ROC of 0.870 was obtained by AdaBoostM1 when trained by the dataset 2020, followed by the Random Forest with a ROC of 0.869 for the same dataset. For the code smell God Class the best result was obtained by the NaiveBayes algorithm, when trained by the dataset 2020, with the ROC value of 0.896. Feature Envy detection results are low, with the Random Forest algorithm having the best ROC value of 0.570 when trained by dataset 2019.

The results of the variance tests (performed through One-way ANOVA) show there was no statistically significant difference between the performance of the six machine learning models when trained with data from the crowd and therefore more realistic.

4.2 RQ2. What is the best machine learning model to detect each one of the three code smells?

In this RQ we want to know which is the best model to detect each of the code smells. To do so, we analyzed the various metrics that evaluate the performance of code smells prediction models in detecting each of the 3 code smells. Of course, the best model will vary with the metric we choose to analyze the model performance (accuracy, precision, recall, f-measure, ROC), but for the reasons described in 3.7 we will use as the main metric the ROC.

Tables 6, 7, and 8 present the performance of the prediction models for the 3 code smells, where the best values for each of the evaluation metrics are marked.

4.2.1 Best machine learning model for the code smell Long Method

For the code smell Long Method, the model that best performs its detection is AdaBoostM1, presenting the best values for all evaluation metrics. As we can see in the table 6, AdaBoostM1 obtained a ROC value of 0.870, a accuracy of 81.36%, a precision of 82.90%, a recall of 81.40%, and F-measure of 81.50%. However, two more models present an almost equal ROC, Random Forest and Multilayer Perceptron, with ROC values of 0.869 and 0.868, respectively.

Except for NaiveBayes, all the other five models have values higher than 0.803 for ROC and values higher than 80.00% for f-measure, precision, and recall in the detection of code smell Long Method.

Table 6: Long Method: Performance of the code smell prediction models

Dataset	Classifier	Accuracy	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
2018	J48	61.02%	61.00%	37.20%	63.40%	61.00%	61.30%	0.617
2018	Random Forest	61.02%	61.00%	45.10%	60.00%	61.00%	60.00%	0.671
2018	AdaBoostM1	67.80%	67.80%	36.50%	67.30%	67.80%	67.40%	0.707
2018	SMO	55.93%	55.90%	51.20%	54.30%	55.90%	54.50%	0.524
2018	MultilayerPerceptron	57.63%	57.60%	46.10%	57.40%	57.60%	57.50%	0.604
2018	NaiveBayes	61.02%	61.00%	47.70%	59.40%	61.00%	58.70%	0.471
2019	J48	64.73%	64.70%	34.10%	66.10%	64.70%	64.90%	0.678
2019	Random Forest	66.18%	66.20%	34.50%	66.40%	66.20%	66.30%	0.679
2019	AdaBoostM1	66.67%	66.70%	29.60%	70.70%	66.70%	66.30%	0.673
2019	SMO	65.46%	65.50%	35.70%	65.50%	65.50%	65.50%	0.649
2019	MultilayerPerceptron	63.29%	63.30%	38.60%	63.10%	63.30%	63.10%	0.667
2019	NaiveBayes	61.11%	61.10%	40.80%	60.90%	61.10%	61.00%	0.614
2019+2018	J48	65.75%	65.80%	33.20%	67.00%	65.80%	65.90%	0.677
2019+2018	Random Forest	65.33%	65.30%	35.30%	65.60%	65.30%	65.40%	0.684
2019+2018	AdaBoostM1	66.60%	66.60%	29.10%	71.40%	66.60%	66.20%	0.665
2019+2018	SMO	63.85%	63.80%	37.10%	64.00%	63.80%	63.90%	0.634
2019+2018	MultilayerPerceptron	63.00%	63.00%	39.40%	62.70%	63.00%	62.80%	0.683
2019+2018	NaiveBayes	58.35%	58.40%	43.70%	58.20%	58.40%	58.30%	0.584
2020	J48	79.95%	80.00%	20.30%	80.30%	80.00%	80.00%	0.832
2020	Random Forest	80.66%	80.70%	20.70%	80.60%	80.70%	80.70%	0.869
2020	AdaBoostM1	81.36%	81.40%	16.70%	82.90%	81.40%	81.50%	0.870
2020	SMO	80.77%	80.80%	20.20%	80.90%	80.80%	80.80%	0.803
2020	MultilayerPerceptron	80.07%	80.10%	21.50%	80.00%	80.10%	80.00%	0.868
2020	NaiveBayes	73.39%	73.40%	33.00%	73.70%	73.40%	72.30%	0.783
2020+2019	J48	76.32%	76.30%	22.10%	77.80%	76.30%	76.50%	0.801
2020+2019	Random Forest	77.19%	77.20%	22.60%	77.70%	77.20%	77.30%	0.828
2020+2019	AdaBoostM1	76.80%	76.80%	20.30%	79.40%	76.80%	76.90%	0.818
2020+2019	SMO	75.53%	75.50%	25.00%	75.80%	75.50%	75.60%	0.753
2020+2019	MultilayerPerceptron	75.85%	75.80%	24.60%	76.10%	75.80%	75.90%	0.822
2020+2019	NaiveBayes	68.43%	68.40%	35.70%	68.00%	68.40%	67.90%	0.742
2020+2019+2018	J48	76.40%	76.40%	22.70%	77.40%	76.40%	76.50%	0.792
2020+2019+2018	Random Forest	76.77%	76.80%	22.70%	77.50%	76.80%	76.90%	0.828
2020+2019+2018	AdaBoostM1	76.40%	76.40%	20.50%	79.30%	76.40%	76.50%	0.807
2020+2019+2018	SMO	75.19%	75.20%	24.60%	75.80%	75.20%	75.30%	0.753
2020+2019+2018	MultilayerPerceptron	76.92%	76.90%	22.50%	77.70%	76.90%	77.10%	0.822
2020+2019+2018	NaiveBayes	68.18%	68.20%	35.70%	67.80%	68.20%	67.70%	0.736

4.2.2 Best machine learning model for the code smell God Class

Table 7 presents the results of God Class detection using the 10-Fold Cross-Validation technique and where the best values are marked. As we can see in table 7, the model that presents the best value for the ROC is Naive Bayes with a value of 0.896. For the remaining four evaluation metrics, the Random Forest model presents the same values as the Naive Bayes. Thus, the Naive Bayes and Random Forest models present an accuracy value of 88.97%, a precision value of 89.70%, a recall value of 89.00%, and an f-measure value of 88.70%.

When we evaluate the models by the ROC value, we verify that, except for the J48 model, all the other five models have values higher than 0.857. For the remaining evaluation metrics all six models have: a) accuracy values higher or equal to 87.50%, b) precision values higher or equal to 87.80%, c) recall values higher or equal to 87.50%, and d) f-measure values higher or equal to 87.20%.

When we compare the results of the code smell God Class detection with those of the Long Method, we verify that the results of God class are better.

4.2.3 Best machine learning model for the code smell Feature Envy

Regarding the code smell Feature Envy, we present in table 8 the results of the evaluation of the different models. For the dataset 2018 of Feature Envy it was not possible to obtain precision, and consequently f-measure, since all the instances classified as TRUE were poorly classified, i.e., all the instances were classified as

Table 7: God Class: Performance of the code smell prediction models

Dataset	Classifier	Accuracy	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
2018	J48	81.82%	81.80%	26.50%	82.00%	81.80%	81.10%	0.692
2018	Random Forest	63.64%	63.60%	47.60%	61.90%	63.60%	62.30%	0.491
2018	AdaBoostM1	68.18%	68.20%	39.60%	67.40%	68.20%	67.70%	0.571
2018	SMO	77.27%	77.30%	29.10%	76.90%	77.30%	76.90%	0.741
2018	MultilayerPerceptron	72.73%	72.70%	31.70%	72.70%	72.70%	72.70%	0.768
2018	NaiveBayes	68.18%	68.20%	45.00%	66.70%	68.20%	66.10%	0.651
2019	J48	72.87%	72.90%	29.50%	72.70%	72.90%	72.70%	0.725
2019	Random Forest	73.64%	73.60%	28.50%	73.50%	73.60%	73.50%	0.802
2019	AdaBoostM1	72.87%	72.90%	29.50%	72.70%	72.90%	72.70%	0.793
2019	SMO	76.74%	76.70%	26.60%	76.90%	76.70%	76.30%	0.751
2019	MultilayerPerceptron	75.97%	76.00%	27.70%	76.10%	76.00%	75.50%	0.797
2019	NaiveBayes	76.74%	76.70%	26.60%	76.90%	76.70%	76.30%	0.804
2019+2018	J48	70.86%	70.90%	30.00%	70.80%	70.90%	70.80%	0.693
2019+2018	Random Forest	67.55%	67.50%	32.40%	67.80%	67.50%	67.60%	0.781
2019+2018	AdaBoostM1	69.54%	69.50%	30.90%	69.50%	69.50%	69.50%	0.771
2019+2018	SMO	72.19%	72.20%	28.90%	72.10%	72.20%	72.00%	0.716
2019+2018	MultilayerPerceptron	71.52%	71.50%	29.00%	71.50%	71.50%	71.50%	0.805
2019+2018	NaiveBayes	74.83%	74.80%	26.50%	74.90%	74.80%	74.60%	0.669
2020	J48	87.50%	87.50%	17.30%	87.80%	87.50%	87.20%	0.791
2020	Random Forest	88.97%	89.00%	16.40%	89.70%	89.00%	88.70%	0.893
2020	AdaBoostM1	88.24%	88.20%	16.80%	88.70%	88.20%	87.90%	0.876
2020	SMO	88.24%	88.20%	16.80%	88.70%	88.20%	87.90%	0.857
2020	MultilayerPerceptron	88.24%	88.20%	16.80%	88.70%	88.20%	87.90%	0.885
2020	NaiveBayes	88.97%	89.00%	16.40%	89.70%	89.00%	88.70%	0.896
2020+2019	J48	82.64%	82.60%	21.70%	82.90%	82.60%	82.30%	0.759
2020+2019	Random Forest	83.02%	83.00%	21.50%	83.40%	83.00%	82.60%	0.850
2020+2019	AdaBoostM1	82.64%	82.60%	21.70%	82.90%	82.60%	82.30%	0.857
2020+2019	SMO	82.26%	82.30%	22.30%	82.60%	82.30%	81.90%	0.800
2020+2019	MultilayerPerceptron	82.26%	82.30%	22.30%	82.60%	82.30%	81.90%	0.853
2020+2019	NaiveBayes	83.02%	83.00%	21.50%	83.40%	83.00%	82.60%	0.859
2020+2019+2018	J48	81.88%	81.90%	21.70%	82.30%	81.90%	81.50%	0.763
2020+2019+2018	Random Forest	81.53%	81.50%	22.00%	81.90%	81.50%	81.20%	0.853
2020+2019+2018	AdaBoostM1	80.84%	80.80%	22.70%	81.20%	80.80%	80.50%	0.854
2020+2019+2018	SMO	83.28%	83.30%	20.30%	83.80%	83.30%	82.90%	0.815
2020+2019+2018	MultilayerPerceptron	82.23%	82.20%	20.10%	82.20%	82.20%	82.10%	0.880
2020+2019+2018	NaiveBayes	81.88%	81.90%	21.30%	82.10%	81.90%	81.60%	0.731

FALSE. For the dataset 2020, we also did not obtain precision and f-measure, because all the instances classified as FALSE were badly classified, i. e., all the models created from this dataset to classify the future envy, classified all the instances of the dataset 2020 as TRUE. For this reason we will not consider in the response to the RQ the models resulting from the training by these two datasets.

When we evaluate the models by the ROC metric, we find that the best model is the Random Forrest with a ROC of 0.570. However, if we compare the various evaluation metrics we find that all the other evaluation metrics have better values than the ROC metric. The best performance in the detection of Feature Envy is obtained by the Naive Bayes model for precision with a value of 61.40%. The Random Forrest model also obtains the best accuracy with 59.69% and recall with a value of 59.70%.

When we compare the results of the models for the detection of the three smells, we verify that the worst results are obtained by the Feature Envy detection models.

4.2.4 Summary of RQ2 results

For the code smell Long Method, the model that best performs its detection is AdaBoostM1, presenting the best values for all evaluation metrics. For the God Class, the model that presents the best value for the ROC is Naive Bayes with a value of 0.896. For Feature Envy, when we evaluate the models by the ROC metric, we find that the best model is Random Forrest. However, for this code smell, the best performance is obtained by the Naive Bayes model, for the precision

Table 8: Feature Envy: Performance of the code smell prediction models

Dataset	Classifier	Accuracy	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
2018	J48	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	Random Forest	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	AdaBoostM1	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	SMO	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	MultilayerPerceptron	70.00%	70.00%	70.00%	-	70.00%	-	0.000
2018	NaiveBayes	30.00%	30.00%	87.10%	35.00%	30.00%	32.30%	0.000
2019	J48	56.85%	56.90%	46.10%	56.20%	56.90%	56.20%	0.563
2019	Random Forest	58.38%	58.40%	44.70%	57.80%	58.40%	57.70%	0.570
2019	AdaBoostM1	54.82%	54.80%	51.40%	52.90%	54.80%	51.40%	0.548
2019	SMO	52.79%	52.80%	51.30%	51.50%	52.80%	51.40%	0.508
2019	MultilayerPerceptron	51.78%	51.80%	52.30%	50.40%	51.80%	50.40%	0.544
2019	NaiveBayes	52.28%	52.30%	45.40%	54.30%	52.30%	52.00%	0.547
2019+2018	J48	57.97%	58.00%	42.80%	57.90%	58.00%	58.00%	0.552
2019+2018	Random Forest	57.49%	57.50%	43.80%	57.30%	57.50%	57.30%	0.542
2019+2018	AdaBoostM1	53.62%	53.60%	48.80%	52.90%	53.60%	52.90%	0.554
2019+2018	SMO	55.56%	55.60%	45.40%	55.50%	55.60%	55.50%	0.551
2019+2018	MultilayerPerceptron	53.62%	53.60%	47.50%	53.50%	53.60%	53.50%	0.548
2019+2018	NaiveBayes	51.69%	51.70%	47.30%	52.60%	51.70%	51.70%	0.548
2020	J48	64.23%	64.20%	64.20%	-	64.20%	-	0.467
2020	Random Forest	64.23%	64.20%	64.20%	-	64.20%	-	0.486
2020	AdaBoostM1	64.23%	64.20%	64.20%	-	64.20%	-	0.468
2020	SMO	64.23%	64.20%	64.20%	-	64.20%	-	0.500
2020	MultilayerPerceptron	64.23%	64.20%	64.20%	-	64.20%	-	0.536
2020	NaiveBayes	51.22%	51.20%	38.20%	61.40%	51.20%	50.90%	0.482
2020+2019	J48	59.38%	59.40%	56.70%	57.00%	59.40%	48.40%	0.529
2020+2019	Random Forest	59.69%	59.70%	56.10%	58.00%	59.70%	49.40%	0.548
2020+2019	AdaBoostM1	58.75%	58.80%	59.30%	34.80%	58.80%	43.70%	0.519
2020+2019	SMO	59.06%	59.10%	56.50%	55.70%	59.10%	49.10%	0.513
2020+2019	MultilayerPerceptron	57.50%	57.50%	56.40%	52.80%	57.50%	49.80%	0.545
2020+2019	NaiveBayes	52.81%	52.80%	40.70%	58.70%	52.80%	51.90%	0.532
2020+2019+2018	J48	57.58%	57.60%	57.80%	50.50%	57.60%	44.50%	0.518
2020+2019+2018	Random Forest	58.48%	58.50%	55.90%	56.20%	58.50%	47.70%	0.539
2020+2019+2018	AdaBoostM1	57.88%	57.90%	58.40%	33.80%	57.90%	42.70%	0.498
2020+2019+2018	SMO	58.79%	58.80%	54.70%	56.90%	58.80%	49.70%	0.520
2020+2019+2018	MultilayerPerceptron	54.85%	54.80%	58.50%	46.80%	54.80%	45.50%	0.533
2020+2019+2018	NaiveBayes	51.82%	51.80%	43.20%	56.10%	51.80%	51.10%	0.524

metric, with a value of 61.40%. When we compare the results of the models for the detection of the three smells, we verify that the worst results are obtained by the Feature Envy detection models and the best results by God Class.

4.3 RQ3. Is it possible to use Collective Knowledge for code smells detection?

Several studies present code smells detection results through machine learning techniques with accuracy, precision, recall, and f-measure, very close to 100%. However, these studies use very treated datasets to obtain good results, which makes the datasets unrealistic. A proof of this is the replication of one of the most important studies on code smells detection using machine learning techniques by Di Nucci et al. (2018), where more realistic datasets were used in this replication. The results of this replication show that the accuracy value, on average, decreased from 96% to 76%, but the f-measure presented results 90% lower than in the reference work. When we compare our results with the Di Nucci et al. (2018) study, we find that the results are similar in some metrics, and better in others.

As reported in the answers to RQ1 and RQ2, we obtained values for some machine learning models close to 90%, which can be considered very good. The fact that the most recent datasets are the ones that usually present the best results, mainly the year 2020, leaves us with expectations of being able to improve the results further. This improvement is mainly due to the improvement of the

methodological process, which has been progressively refined each year. Thus, the answer to this RQ is yes, it is possible to use Collective Knowledge for code smells detection.

4.3.1 Summary of RQ3 results

The answer to this RQ is yes, it is possible to use Collective Knowledge for code smells detection. The Crowdsourcing approach obtained values for some ML models close to 90%, which can be considered very good. Overall our results are similar to those of Di Nucci et al. (2018)'s study and are even better in some metrics. The fact that the most recent datasets are the ones that usually show the best results, mainly the year 2020, leaves us with expectations that we can improve the results even more.

5 Discussion

5.1 Research Questions (RQ)

In this section, we present the discussion of the results considering the three RQs. Regarding the comparison of our results with existing works, we will compare with Di Nucci et al. (2018) study, since it is the one that presents more similarities with ours, also using more realistic datasets.

For the long Method, the model with the best prediction is AdaBoostM1, trained on the 2020 dataset, with a ROC of 0.870 (see Table 3), but also F-Measure and Accuracy show values higher than 80%, namely 81.50%, and 81.36%, respectively (see table 6). Random Forest, with a ROC of 0.869, shows a value almost equal to AdaBoostM1. These two models show good results, in line with the results presented in Di Nucci et al. (2018)'s study. The MultilayerPerceptron and J48 models also show good results with the best ROC of 0.868 and 0.832, respectively, for the 2020 dataset. Namely, MultilayerPerceptron is the second-best model for three datasets (2019+2018, 2020+2019, 2020+2019+2018). In opposition is NaiveBayes and SMO which show the worst results for all datasets, for example, for the 2020 dataset where they have their best values, the ROC is 0.783 for NaiveBayes, and 0.803 for SMO.

Also, regarding the code smell Long Method, the models trained with the most recent dataset, the year 2020, have the best values, with a ROC greater than or equal to 0.803 for five models (J48, Random Forest, AdaBoostM1, SMO, MultilayerPerceptron) out of the six we used. Only the NaiveBayes model has a ROC lower than 0.800, more precisely 0.783, but still higher than all models trained with datasets from previous years. The fact that the models trained on the most recent dataset show the best results is important because it means that there has been an evolution in the production of the datasets over the three years by this approach.

For the code smell God Class, prediction values very close to 90% were obtained, as such, we consider these to be good values compared to similar studies. The model that presented the best ROC value was NaiveBayes with 0.896, followed by Random Forest with 0.893 (see table 4) for the 2020 dataset. These two models also had the best values for the other metrics, with both having equal values for

F-Measure 88.70% and Accuracy of 88.97% (see table 7) for the 2020 dataset. For the MultilayerPerceptron model, good results were also obtained, with a ROC of 0.885, for the 2020 dataset and a ROC of 0.880 for the 2020+2019+2018 dataset. Hence, this model presented the 3rd and 4th best values. The AdaBoostM1 and the SMO models obtained their best values with the 2020 dataset, with a ROC value of 0.876 and 0.857, respectively. The worst values were presented by the J48 model, with its best ROC value of 0.791, thus being the only model that failed to exceed the ROC value of 0.800. For the code smell God Class it happened the same as for the code smell Long Method, all models presented their best ROC values when trained with the most recent year datasets, the year 2020.

For the code smell Feature Envy, it was not possible to obtain the values for all the evaluation metrics for the reasons already explained in the results section (see subsection 4.2.3). The models for this code smell showed low results, being the worst results of the three code smells. Thus, the best ROC value was 0.570 for Random Forrest, but far from the values obtained for God Class and Long Method, 0.896 and 0.870, respectively. The NaiveBayes model showed the best result of all the evaluation metrics with a value of 61.40% for Precision. Random Forrest again presented the best value for Recall 59.70% and Accuracy 59.69%. For F-Measure, the best value of 58.00% is obtained with the J48 model. Regarding the datasets that show better results for Feature Envy, the Accuracy, Precision, and Recall metrics were the 2020+2019, for the F-measure and ROC metrics were the 2019+2018 and 2019, respectively. Hence, no dataset concentrates most of the best values for the various metrics.

In the following subsections we provide a summary for each of the RQs.

5.1.1 RQ1. What is the performance of machine learning techniques when trained with data from the crowd and therefore more realistic?

The best result was obtained for the code smell God Class with a ROC value of 0.896, however, the Long Method with a ROC of 0.870 is very close. The worst result was obtained for Feature Envy with a ROC of 0.570. The difference in ROC value between the best and worst code smell is 0.326. This considerable difference is due to the constitution of the Feature Envy datasets. When we analyze the composition of the datasets, we see that the diversity of cases (classified methods) is much smaller compared to that of the Long Method datasets, which is also a code smell in the method scope, and consequently uses the same code metrics. The solution to this problem is to continue to grow the dataset by classifying methods that are not already part of it. However, this problem alerts us to the classification of more complex code smells, as such, with fewer occurrences in the code, and where programmers tend to follow more of the advisors' detection results.

When we compare our ROC values with those obtained by Di Nucci et al. (2018), we find that for the code smells God Class and Long Method, we obtain similar values in the range of 0.89 and 0.87, respectively. Regarding the code smell Feature Envy, for the reasons already presented, our value of 0.57 is considerably lower than the one presented by Di Nucci et al. (2018), which is 0.89.

5.1.2 RQ2. What is the best machine learning model to detect each one of the three code smells?

Having the ROC as the reference metric, for the Long Method the best models were AdaBoostM1 and Random Forrest, for the God Class it was Naive Bayes and Random Forrest that presented the best values, and for the Feature Envy, it was Random Forrest and Naive Bayes models. Thus, we can conclude that in regards to which is the best ML model for the detection of the three code smells, we do not have a model that guarantees the best detection value in the three smells, however, Random Forrest stands out.

When we compare the results of the models for the detection of the three smells, we verify that the best results are obtained by the God Class detection models, and the worst results are obtained by the Feature Envy detection models.

In the Di Nucci et al. (2018) study, the best performances (for all code smells) were obtained by the Random Forrest and J48 models. These two models have in common that they are based on decision trees. When we compare them to our models we can conclude, i) Random Forrest was also the model with which we obtained the best results when considering all smells, ii) regarding J48, it was not one of our best models, because only for the F-Measure in the Feature Envy code smell it presented the best value, iii) Naive Bayes, which was one of our best models, did not present significant results in the Di Nucci et al. (2018) study.

5.1.3 RQ3. Is it possible to use Collective Knowledge for code smells detection?

This study is the first to use the Crowdsampling approach for code smell detection, as such, there are always methodological aspects that could have been improved. As a result of this study, we have also already found more aspects that can be improved in the future which we present in this discussion.

When we have dozens of participants, it is not possible to have total control over the actions of each participant. In our case, this was reflected in the non-use, in the first year, of data from 11 teams, out of a total of 42 participants. To have better control over the participants' actions, we removed the possibility for them to choose the Java project on which to detect the code smells, and all the teams started using the same project. This decision resulted in a lack of diversity of cases when a code smell is more complex and consequently has fewer existences in the code (in our case, Feature Envy). Another consequence was that the participants started to follow the advisor's suggestions more since the code smell is more complex.

In this experiment, we performed many processes manually, such as the data aggregation process, which proved to be time and effort consuming, and therefore impractical to implement in a company's reality. However, our goal was to perform the first experiment to verify the potential of this approach.

This first study presents promising results, therefore, we are convinced of the applicability of this approach in detecting code smells. One of the reasons for this is that this approach allowed us to gather in the datasets a wide variety of opinions regarding the classification of code smells since it has the contribution of more than 350 participants. Another reason is that the datasets have borderline smells (where it is not clear whether it is a smell or not)

which make it harder to detect. Finally, the fact that the datasets are not balanced also contributes to the datasets being considered more realistic.

We have organized the datasets by year so that we can compare the results of each year and thus understand the progress in implementing this approach. The results obtained show progress over these years, with the best values being obtained in 2020. Thus, we are led to conclude that we are on the right track and that we can improve these results much more.

From a methodological point of view, more validation experiments are needed to cover more code smells, build more broad datasets, and to increase external validity. To address many of the issues presented, we are developing a tool based on micro-tasks to automate the whole process of the Crowdselling approach, from the extraction of metrics from the Java project code to the validation of code smells by the developers. With these tools, our goal is to move to the next level, which is to study the implementation of this approach in an industrial environment.

5.2 Implications and limitations of the Crowdselling Approach

The Crowdselling approach has several advantages for developers and researchers because it is a dynamic approach that does not require the definition of rules for the detection of each code smell and its thresholds. This approach through the input given by developers produces datasets more and more adapted to the developers' reality, which implies the production of better ML models and consequently a better detection of code smells. These dynamics presented by the approach has two main advantages: i) although we have used only three code smells in this study, it is not limited to these code smells, and can be generalized to other code smells; ii) it makes the detection accuracy improve as the feedback from the developers grows (by improving the learning datasets) and leads the ML models to converge to maximum accuracy.

The learning dynamics presented in the previous paragraph, are also the main limitation of the approach, because it is dependent on feedback from developers, and it is not possible to predict exactly how much convergence in learning can be achieved, i.e., what is the maximum detection accuracy.

To better demonstrate our approach, we will exemplify two scenarios: i) a company where there is a set of development rules, namely, code smells, that is known and respected by all developers. In this scenario all developers are aligned with the code smells rules, thus contributing to a clear definition of what a code smell is in the datasets. In this scenario, we have faster convergence and will achieve higher detection accuracy; ii) A successful open-source project, where many developers contribute. In this scenario, it will be more complicated for all developers to respect the rules since there will be less alignment among developers, and therefore more divergence on what is a code smell. Our approach will always translate in the datasets what the developers understand to be a code smell, but the convergence will take longer and the detection accuracy will be lower.

In both scenarios, this approach learns from the context in which it is used by learning the code smells detection rules used, thus always translating the developers' reality. The more precise the detection rules, the higher the detection accuracy.

5.3 Threats to validity

In our study, we made assumptions that may threaten the validity of our results. In this section, we discuss possible sources of threats, and how we mitigated them.

5.3.1 Conclusion Validity

Threats in this category impact the relation between treatment and outcome.

The first threat is in the evaluation methodology, so we adopted the 10-fold cross-validation, which is one of the most used in machine learning, and to directly compare our results with those achieved in the other study's.

As for the evaluation metrics adopted to interpret the performance of the experimented models, we have adopted the most common machine learning metrics, which have been used in other studies with some similarity.

To test if there was a statistically significant difference between the performance of the six machine learning models, we used the one-way analysis of variance (ANOVA).

5.3.2 Construct Validity

As for potential issues related to the relationship between theory and observation, we may have been subject of problems in the adopted methodology. To avoid bias in process, we elaborated a script in which we detailed all the steps that the teams had to carry out to detect code smells, so that there would be uniformity in the process. However, we cannot guarantee the correct use of this script by all the teams.

Code metrics are extremely important because they play the role of independent variables in the machine learning algorithms. To avoid bias in metrics extraction we used the same metrics as in Fontana et al. (2015), since they are publicly available. As for the experimented prediction models, we exploited the implementation provided by the Weka framework (Hall et al., 2009), which is widely considered as a reliable tool. To avoid bias in the parameterization of the Weka algorithms, we used the default values for the parameters.

5.3.3 Internal Validity

This threat is related to the correctness of the experiments' outcome. Since the definition of code smells is subjective, it may cause different interpretations and, as such, the manual evaluation is not entirely reliable. To mitigate this problem, an advisor is used in the experiment to serve as a basis for identifying code smells, although the final decision is always made by the team, which is composed of several developers, and all had the same training.

To avoid participants only classifying code smells detected by the JDeodorant advisor (although it was optional to use it), it was indicated that they would have to classify at least one package, however, not all teams did so. This requires teams to manually classify, based directly on code metrics, a set of false positive and negative code smells as detected by JDeodorant.

The participants in this study were students attending a compulsory Software Engineering course. In the scope of this course, an optional assignment was done

where this experiment was carried out. To have rigor in the accomplishment of this work, since it was optional, the works were evaluated by the teachers and a grade was assigned according to the quality demonstrated. The fact that only students were used can be a threat, however, these are finalists who in three months will be working in companies. On the other hand, the use of students has advantages and disadvantages as we can see in the paper by Feldt et al. (2018).

The maturity, experience, and knowledge of team members about code smells is a variable that we cannot control. As such, there may be variations in the accuracy and precision of the detection of code smells. To minimize the possible bias, the decisions are not individual, but taken by the team. The time given to do this work was 3 weeks, which may have been a reason for bias, but considering that it was a team effort we thought it was sufficient.

Because code smells is only detected in three Java projects, there may be some bias as to the number and type of code smells existing in these Java projects. We chose these projects because they are open-source, are widely used in code smells detection, and are not toy examples due to their considerable dimension.

5.3.4 External Validity

Finally, the External validity is concerned with whether we can generalize the results outside the scope of our study.

With respect to the generalizability, we used the three most common code smells in this type of studies. Regarding the code metrics, we used a high number, 61 metrics for God Class and 82 metrics for Feature Envy and Long Method, thus ensuring a wide scope.

In terms of programming languages, we only used Java projects, but Java is by far the most used language in code smells detection studies, accounting for 77.1% of the cases Pereira dos Reis et al. (2021).

The fact that this study has a very manual component does not make it easy to reproduce, however, all the necessary indications are in the study, and a set of materials is available on GitHub.

6 Conclusion and future work

In (Reis et al., 2017) we have proposed the concept of Crowdsmelling – use of collective intelligence in the detection of code smells – to mitigate the aforesaid problems of subjectivity and lack of calibration data required to obtain accurate detection model parameters. In this paper we reported first results of a study investigating the approach Crowdsmelling, a collaborative crowdsourcing approach, based in machine learning, where the wisdom of the crowd (of software developers) will be used to collectively calibrate code smells detection algorithms.

For 3 years we collected code smells detection data by several teams manually, although they could use JDeodorant as an advisor if they wanted. Combining the data from each year with the previous ones, we created several oracles for each of the three code smells (Long Method, God Class, Feature Envy). The latter were used to train a set of machine learning algorithms, creating the detection models for each of the three code smells, in a total of 108 models. Finally, to evaluate the models we tested them using the 10-Fold Cross-Validation methodology, and

analyzed the metrics Accuracy, Precision, Recall, and F-Measure, with special emphasis on ROC, because the datasets were not treated, for example, balanced. This way we created the most realistic datasets possible. To check if there were significant differences between the classifications presented by the different models, we proceeded to the analysis of variance through a one-way analysis of variance (ANOVA).

Regarding RQ1, we conclude that the best results for the code smell Long Method were obtained by the Random Forrest and AdaBoostM1 algorithms. The best result with a ROC of 0.870 was obtained by AdaBoostM1 when trained by the dataset 2020, followed by the Random Forrest with ROC of 0.869 for the same dataset. For the code smell God Class, the best result was obtained by the NaiveBayes algorithm, when trained by the dataset 2020, with the ROC value of 0.896. For Feature Envy the results are low, with the Random Forest algorithm having the best ROC value of 0.570 when trained by dataset 2019. The results of the variance tests (ANOVA) show there was no statistically significant difference between the performance of the six machine learning models, when trained with data from the crowd and therefore more realistic.

As for RQ2, the best machine learning model for Long Method detection is AdaBoostM1, presenting the best values for all evaluation metrics, a ROC value of 0.870, a accuracy of 81.36%, a precision of 82.90%, a recall of 81.40%, and F-measure of 81.50%. For the God Class, the model that presents the best value for the ROC is Naive Bayes with a value of 0.896. the Naive Bayes and Random Forest models present an accuracy value of 88.97%, a precision value of 89.70%, a recall value of 89.00%, and an f-measure value of 88.70%. For the Feature Envy, best model is the Random Forrest with a ROC of 0.570. However, the best performance in the detection of Feature Envy is obtained by the Naive Bayes model for precision with a value of 61.40%.

Regarding RQ3, it is possible to use the Crowdsmeeling – use of collective intelligence in the detection of code smells – as a good approach for the detection of code smells, because we obtained values for some machine learning models close to 90%, which can be considered very good, for realistic datasets, which reflect the detection performed by developers. The fact that the most recent datasets, the year 2020, are the ones that usually presented the best results, leaves us with great motivation to continue developing this detection approach because we think that we can even better the results.

We are currently developing a plugin for the Eclipse IDE, which extracts the code metrics, detects the code smells, identifies the code smells in the code, receives the programmer’s opinion regarding the detection of the code smell (i.e., if the programmer agrees or not with the code smell and stores all this information in a database). This plugin is expected to simplify the use of the Crowdsmeeling approach, making it simple for programmers to use when developing their Java projects.

Acknowledgements This work was partially funded by the Portuguese Foundation for Science and Technology, under ISTAR’s projects UIDB/04466/2020 and UIDP/04466/2020, and by Anima Institute (Edital N^o 43/2021).

References

- Baltes, S. and Treude, C. (2020). Code duplication on stack overflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER '20*, page 13–16, New York, NY, USA. Association for Computing Machinery.
- Bavota, G. and Russo, B. (2016). A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 315–326.
- Bentzien, J., Muegge, I., Hamner, B., and Thompson, D. C. (2013). Crowd computing: Using competitive dynamics to develop and refine highly predictive models. *Drug Discovery Today*, 18(9-10):472–478.
- Bigham, J. P., Bernstein, M. S., and Adar, E. (2014). Human-Computer Interaction and Collective Intelligence. In Malone, T. W. and Bernstein, M. S., editors, *The Collective Intelligence Handbook*. CMU.
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.
- Caldeira, J., Brito e Abreu, F., Cardoso, J., and dos Reis, J. P. (2020). Unveiling process insights from refactoring practices.
- de Mello, R., Oliveira, R., Sousa, L., and Garcia, A. (2017). Towards effective teams for the identification of code smells. In *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 62–65.
- Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., and De Lucia, A. (2018). Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621.
- Feldt, R., Zimmermann, T., Bergersen, G. R., Falessi, D., Jedlitschka, A., Juristo, N., Münch, J., Oivo, M., Runeson, P., Shepperd, M., Sjøberg, D. I., and Turhan, B. (2018). Four commentaries on the use of students and professionals in empirical software engineering experiments. *Empirical Software Engineering*, 23(6):3801–3820.
- Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2).
- Fontana, F. A., Mäntylä, M. V., Zanoni, M., and Marino, A. (2015). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*.
- Fontana, F. A., Zanoni, M., Marino, A., and Mäntylä, M. V. (2013). Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*, pages 396–399.
- Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc.
- Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on International Conference on Machine Learning, ICML'96*, page 148–156, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

- Humphrey, W. (2009). The Future of Software Engineering: I. *The Watts New? Collection: Columns by the SEI's Watts Humphrey*.
- John, G. and Langley, P. (1995). Estimating continuous distributions in bayesian classifiers. In *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345. Morgan Kaufmann.
- Kessentini, W., Kessentini, M., Sahraoui, H., Bechikh, S., and Ouni, A. (2014). A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering*, 40(9):841–861.
- Mansoor, U., Kessentini, M., Maxim, B. R., and Deb, K. (2017). Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, 25(2):529–552.
- Oliveira, R. (2016). When more heads are better than one? understanding and improving collaborative identification of code smells. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 879–882.
- Oliveira, R., de Mello, R., Fernandes, E., Garcia, A., and Lucena, C. (2020). Collaborative or individual identification of code smells? on the effectiveness of novice and professional developers. *Information and Software Technology*, 120:106242.
- Oliveira, R., Estácio, B., Garcia, A., Marczak, S., Prikladnicki, R., Kalinowski, M., and Lucena, C. (2016). Identifying code smells with collaborative practices: A controlled experiment. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 61–70.
- Oliveira, R., Sousa, L., de Mello, R., Valentim, N., Lopes, A., Conte, T., Garcia, A., Oliveira, E., and Lucena, C. (2017). Collaborative identification of code smells: A multi-case study. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 33–42.
- Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2015). Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 482–485.
- Pecorelli, F., Di Nucci, D., De Roover, C., and De Lucia, A. (2019). On the role of data balancing for machine learning-based code smell detection. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, page 19–24, New York, NY, USA. Association for Computing Machinery.
- Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G., and Anslow, C. (2021). Code smells detection and visualization: A systematic literature review. *Archives of Computational Methods in Engineering*.
- Platt, J. C. (1999). Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*, page 185–208. MIT Press.
- Proksch, S., Amann, S., and Mezini, M. (2014). Towards standardized evaluation of developer-assistance tools. In *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering - RSSE 2014*, pages 14–18, New York, New York, USA. ACM Press.
- Quinlan, J. R. (2014). *C4.5: Programs for Machine Learning*. Elsevier.

- Rahman, F. and Devanbu, P. (2013). How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, page 432–441. IEEE Press.
- Reis, J. P., Brito e Abreu, F., and de F. Carneiro, G. (2017). Code smells detection 2.0: Crowdsourcing and visualization. In *2017 12th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–4.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA.
- Sharma, M. and Padmanaban, R. (2014). *Leveraging the wisdom of the crowd in software testing*. CRC Press.
- Stol, K.-J. and Fitzgerald, B. (2014). Researching crowdsourcing software development: Perspectives and concerns. In *Proceedings of the 1st International Workshop on CrowdSourcing in Software Engineering, CSI-SE 2014*, page 7–10, New York, NY, USA. Association for Computing Machinery.
- Stone, M. (1974). Cross-validators choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 36(2):111–147.
- Tahir, A., Yamashita, A., Licorish, S., Dietrich, J., and Counsell, S. (2018). Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, EASE'18*, page 68–78, New York, NY, USA. Association for Computing Machinery.
- Travassos, G., Shull, F., Fredericks, M., and Basili, V. R. (1999). Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th conference on object oriented programming, systems, languages, and applications*, pages 47–56, New York, NY, USA. ACM Press.
- Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2018). Ten years of jdeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14.
- Wang, C., Hirasawa, S., Takizawa, H., and Kobayashi, H. (2015). Identification and Elimination of Platform-Specific Code Smells in High Performance Computing Applications. *International Journal of Networking and Computing*, 5(1):180–199.
- Yamashita, A. and Moonen, L. (2013). To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Inf. Softw. Technol.*, 55(12):2223–2242.
- Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2003). Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transaction on Evolutionary Computation*, 7(2):117–132.

Appendices

Appendix A. Code metrics

Metric	Acronym
Lines of Code	LOC
Lines of Code Without Accessor or Mutator Methods	LOCNAMM
Number of Packages	NOPK
Number of Classes	NOCS
Number of Methods	NOM
Number of Not Accessor or Mutator Methods	NOMNAMM
Number of Attributes	NOA
Cyclomatic Complexity	CYCLO
Weighted Methods Count	WMC
Weighted Methods Count of Not Accessor or Mutator Methods	WMCNAMM
Average Methods Weight	AMW
Average Methods Weight of Not Accessor or Mutator Methods	AMWNAMM
Maximum Nesting Level	MAXNESTING
Weight of Class	WOC
Called Local Not Accessor or Mutator Methods	CLNAMM
Number of Parameters	NOP
Number of Accessed Variables	NOAV
Access to Local Data	ATLD
Number of Local Variable	NOLV
Tight Class Cohesion	TCC
Lack of Cohesion in Methods	LCOM
Fanout	FANOUT
Access to Foreign Data	ATFD
Foreign Data Providers	FDP
Response for A Class	RFC
Coupling Between Objects Classes	CBO
Called Foreign Not Accessor or Mutator Methods	CFNAMM
Coupling Intensity	CINT
Coupling Dispersion	CDISP
Maximum Message Chain Length	MAMCL
Number of Message Chain Statements	NMCS
Mean Message Chain Length	MEMCL
Changing Classes	CC
Changing Methods	CM
Number of Accessor Methods	NOAM
Number of Public Attributes	NOPA
Locality of Attribute Accesses	LAA
Depth of Inheritance Tree	DIT
Number of Interfaces	NOI
Number of Children	NOC
Number of Methods Overridden	NMO
Number of Inherited Methods	NIM
Number of Implemented Interfaces	NOII
Number of Default Attributes	NODA
Number of Private Attributes	NOPVA
Number of Protected Attributes	NOPRA
Number of Final Attributes	NOFA
Number of Final and Static Attributes	NOFSA
Number of Final and Non - Static Attributes	NOFNOSA
Number of Not Final and Non - Static Attributes	NOFNOSA
Number of Static Attributes	NOSA
Number of Non - Final and Static Attributes	NOFNOSA
Number of Abstract Methods	NOABM
Number of Constructor Methods	NOCM
Number of Non - Constructor Methods	NONCM
Number of Final Methods	NOFM
Number of Final and Non - Static Methods	NOFNOSM
Number of Final and Static Methods	NOFSM
Number of Non - Final and Non - Abstract Methods	NOFNABM
Number of Final and Non - Static Methods	NOFNOSM
Number of Non - Final and Static Methods	NOFNOSM
Number of Default Methods	NODM
Number of Private Methods	NOPM
Number of Protected Methods	NOPRM
Number of Public Methods	NOPLM
Number of Non - Accessors Methods	NONAM
Number of Static Methods	NOSM