Preprint of an article: Lech Madeyski and Tomasz Lewowski (2023), "Detecting code smells using industry-relevant data", *Information and Software Technology*, Volume 155, Pages 107112, DOI: 10.1016/j.infsof.2022.107112
Preprint: https://madeyski.e-informatyka.pl/download/MadeyskiLewowski23ISTpreprint.pdf

Detecting code smells using industry-relevant data

Lech Madeyski[a,b,1,*], Tomasz Lewowski[a,2,*]

[a] *Wroclaw University of Science and Technology, Department of Applied Informatics, Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland*
[b] *Blekinge Institute of Technology, Department of Software Engineering, Karlskrona, Sweden*

Abstract

Context Code smells are patterns in source code associated with an increased defect rate and a higher maintenance effort than usual, but without a clear definition. Code smells are often detected using rules hard-coded in detection tools. Such rules are often set arbitrarily or derived from data sets tagged by reviewers without the necessary industrial know-how. Conclusions from studying such data sets may be unreliable or even harmful, since algorithms may achieve higher values of performance metrics on them than on models tagged by experts, despite not being industrially useful.

Objective Our goal is to investigate the performance of various machine learning algorithms for automated code smell detection trained on code smell data set(MLCQ) derived from actively developed and industry-relevant projects and reviews performed by experienced software developers.

Method We assign the severity of the smell to the code sample according to a consensus between the severities assigned by the reviewers, use the Matthews Correlation Coefficient (MCC) as our main performance metric to account for the entire confusion matrix, and compare the median value to account for non-normal distributions of performance. We compare 6720 models built using eight machine learning techniques. The entire process is automated and reproducible.

Results Performance of compared techniques depends heavily on analyzed smell. The median value of our performance metric for the best algorithm was 0.81 for Long Method, 0.31 for Feature Envy, 0.51 for Blob, and 0.57 for Data Class.

Conclusions Random Forest and Flexible Discriminant Analysis performed the best overall, but in most cases the performance difference between them and the median algorithm was no more than 10% of the latter. The performance results were stable over multiple iterations. Although the F-score omits one quadrant of the confusion matrix (and thus may differ from MCC), in code smell detection, the actual differences are minimal.

Keywords: reproducible research, software engineering, machine learning, code smells

1. Introduction

The amount of software written each year increases dramatically, and even just the cost of correcting defects is measured in billions of dollars [1]. Research on the cost of changes suggests that the effort spent to introduce a modification increases substantially with the evolution of the project [2]. This, of course, poses a number of challenges for large projects.

Some studies suggest that both maintenance effort and number of defects are distributed according to the Pareto principle: a small number of entities consume most of the effort and have most of the defects [3, 4, 5, 6]. Detecting those entities (regardless of whether they are files, classes, modules, subsystems, or microservices) is likely to help software maintainers refactor the critical areas and avoid a number of problems.

However, the detection of such fragments prior to production deployment is a challenging task, as it requires a thorough review of the whole code base. While this may be a viable option for system maintainers (since they need to know the code anyway), it is hardly a scalable solution. It is also not a solution that can be adopted by any consultancy (except for a few high-risk industries, such as medical and military). While it is possible to detect these fragments retroactively by analyzing incoming defect reports, it is generally preferable to solve problems without involving users. This is true especially in non-open-source environments, where there may be only a few paying customers. In such a context, defects may attract higher management attention or even cause the company to lose contracts.

Therefore, it is reasonable to look for automated tools and heuristics that can help software developers detect modules that are likely to contain defects. To aid in this detection, the concept of code smell was coined by Fowler and Beck in the late 1990s [7]. A "code smell" is a code snippet (or several snippets) that "usually corresponds to a deeper problem in the system". Once such code smells

*Corresponding author
Email addresses: lech.madeyski@pwr.edu.pl (Lech Madeyski), tomasz.lewowski@pwr.edu.pl (Tomasz Lewowski)
[1]ORC ID: 0000-0003-3907-3357
[2]ORC ID: 0000-0003-4897-1263

are detected, increased focus must be placed on them, as they are likely to require extra effort during maintenance and implementation of new features. Sometimes, such a detection may lead to reworking the module in the form of a refactoring. A "code smell" is a heuristic concept, so it is not necessarily a problem itself—a developer may have planned such a design and may very well have had valid reasons to use such a pattern. However, if the "smelly" code spontaneously emerges, it is likely to cause problems. Even if it was planned, extra caution should be exercised when performing modifications that may affect it (e.g., additional layer of regression testing, additional documentation, etc.), as unplanned consequences are likely.

That said, "usually corresponds to a deeper problem in the system" does not say anything about how to detect such problems, so by itself it is not a useful criterion even for experienced software developers, not to mention automated detection. To address this problem and present some more concrete examples (not an exhaustive list), in the initial publication Fowler [7] presented a catalog of 22 named code smells ("Feature Envy", "Functional Decomposition", "Long Method" etc.) and a brief description, meant for human developers (in a fashion similar to how Gang of Four described the original set of design patterns in [8]). While software developers may find those descriptions useful and perhaps even prefer them vague, those descriptions are also not well suited for automated tools.

There have been many attempts to define code smells using various techniques and predictors–from classic object-oriented metrics [9, 10] through text-based detection [11] up to analyzing clusters of changes and co-changes [12, 13]. However, even though much effort has already been put into the domain, we believe that there is still room for an exploratory study like this one. There are four main differences between this study and similar studies in the area:

1. we use a separately published data set built from actively maintained software projects and reviewed by software developers with industrial experience [14],
2. our primary focus is on optimizing Matthews' Correlation Coefficient (MCC) metric, taking into account recommendations from recent studies by Shepperd et al. [15] and Yao and Shepperd [16],
3. we focus on median results and quartiles to account for non-normal distributions of performance metrics,
4. our study is focused on reproducibility [17, 18, 19], providing all source data, execution scripts, and parameters in a complete reproduction package (described in detail in Section 3.6).

The rest of the paper is structured as follows: Section 2 describes the approaches to code smell detection taken by other researchers, Section 3 describes details of our research setup, and Section 4 contains the results of our study. Then, in Section 5, we discuss the findings and possible shortcomings, and we conclude the paper in Section 6.

2. Related work

Code smell detection is an area that has been intensively researched. We found seven systematic reviews published in the last six years that cover the area of code smell detection. Some of the reviews, e.g., [20, 21, 22] focus solely on code smell detection using machine learning techniques, while others, like [23], also include more traditional methods, such as preset detection rules. Reviews by Caram et al. [24] and Santos et al. [25] cover a wider perspective on the general state of the art, the usefulness of code smells (mainly in the areas of code maintainability, change proneness, error proneness, and bug prediction), and their wider applicability.

A particularly large-scale survey (spanning papers from 1990 to 2017) is presented in [26] and discusses the overall state of the art in code smell research, including who the researchers are, what the trends are, what the research goals are, and what the future goals are for the domain.

In general, methods used for code smell detection include a plethora of solutions: static algorithms (e.g., rules like in DETEX [10]), general-purpose machine learning solutions (such as decision trees and random forests [9, 27, 28], SVMs [9, 27], Bayesian techniques [29, 9], neural networks [30] or rules [27]) and some methods developed specifically for code smells, such as analysis of co-change patterns [12], code history analysis [31], text processing [11] and deep learning methods [32, 33, 34, 35, 36], which were not yet thoroughly researched in regard to application in code smell detection. Some researchers do not create full machine learning models but instead decide to only tune parameters of pre-created models [37, 38]. Often, this tuning uses sophisticated statistical [39] or evolutionary [40] techniques.

A recent comparison between machine learning techniques and rules (represented by the DÉCOR tool) [41] suggests that rules may achieve better performance than machine learning algorithms, but are still not good enough to be practically useful. Other research, e.g., [36] shows machine learning algorithms performing better than predefined rules. Overall, the performance of the methods used by more than one researcher varies greatly between different studies (we show specific examples in Section 5.3). In [19], we concluded that one of the reasons for this variation may be the lack of a shared data set that would enclose the understanding of the "code smell" concept that comes from experienced software developers or, more generally, the lack of a standardized way to create such data sets, as it is obvious that a single data set would not be sufficient.

From a practical standpoint, it is also important to understand that not all smells are equal; developers prioritize addressing them not only by their severity, but also by the importance of the module in which they occur [42]. Some smells are not perceived as problems, but as suboptimal (but acceptable) design choices [43]. There are also papers suggesting that refactoring does not decrease the severity of code smells [44].

There are several drawbacks that we found in most of the existing body of research: first, the most popular data sets, including the one provided by Fontana et al. [9], were tagged by students. As we argue in [14], existence and severity of code smells is part of technical knowledge that is not taught in universities, but rather obtained during industrial practice. Since students, by definition, lack this experience, their tagging of code smells has to be superficial. While the overall agreement between developers as to what constitutes a code smell does not seem to be high [45], developers are the domain experts and ones who, in practice, get to decide whether a given code snippet is or is not smelly, simply by virtue of being the people who work with code. Although the understanding of code smells exhibited by students may be similar to the understanding exhibited by some of the developers, we are not aware of any study that demonstrates this to a satisfying degree. Second, as demonstrated in [19], most studies do not provide a complete reproduction package, and a substantial part of the studies do not provide a description that is detailed enough for reproduction.

Perception of code smells severity is an area of active research. Fontana et al. [46] used four-point severity scale from 0 (no-smell), through 1 (non-severe smell), 2 (smell) to 3 (severe smell). The scale is used to prioritize code smell and acquire more information from the data set reviewers – however, the number of severities is arbitrary. Pecorelli et al. [47] used a five-point severity scale during the data collection phase to assess the severity of code smell (called *criticality* in their paper), but later decided to use a three-point scale for modeling purposes. Taibi et al. [48] conducted a medium-scale survey on perceived code smell harmfulness. They also used a five-point scale, but the paper does not report any modeling. Results of [48] suggest that there may be an important difference between the theoretical sentiment toward code smells (even by senior developers) and practical consequences of this sentiment (e.g., in the form of refactoring recommendations). A further study by Sae-Lim et al. [42] selected task relevance and module importance as two important factors (apart from the severity of the smell) that guide the refactoring recommendations of developers.

Performance metrics that researchers most often use include precision, recall, accuracy, and F-score [20]. However, accuracy behaves badly in the presence of imbalanced classes, while precision, recall, and F-score only include three quadrants of the confusion matrix. Yao and Shepperd [16] reported that this may have a substantial impact on the interpretation of the results. Their research is related to defect prediction, but the same argument can also be made for code smells. Thus, we present the most common metrics together with the MCC metric, while at the same time reporting the full confusion matrices in the reproduction package described in Section 3.6.

This is our first study on creating general purpose code smell models; previously we focused on creating a large-scale data set [14], systematic reviews of the literature on applying machine learning techniques to code smell detection [20] and the reproducibility of code smell research [19], and, on a smaller scale, investigating the relationship between new Java language features and the existence of code smells [49].

In this study, we use a curated data set that was built by software engineers with industrial expertise [14] and provide a full reproduction package that contains all operations performed on the data set to obtain the results presented. Of course, since the process of machine learning relies on randomness (e.g., to generate folds for cross-validation), the results that other researchers obtain during reproduction may be slightly different. Nevertheless, the results we obtained were remarkably stable for virtually every algorithm and data set, so there shall be no significant differences.

We used the MLCQ data set reported by us in data paper [14] based on a set of industry-relevant projects selected using a technique presented in [50]. The same data set was also used in [51] and [36]. The main difference between this study and [51] lies primarily in the data preprocessing and details of the machine learning procedure — the authors of [51] used a decision tree and random forest algorithm, while we analyzed a much wider range of algorithms. We also use the hyperparameter optimization method for training, since, wherever applicable, we optimize for MCC (hyperparameter optimization was not reported by the authors of [51]). The third difference lies in the set of predictors used. While both our paper and [51] use metrics provided by PMD[3], we added an additional set of metrics, calculated by internal CODEBEAT tools[4]. These metrics are briefly described in Tables A.7 and A.8. On the other hand, the authors of [51] also used results from a commercial tool, Understand by SciTools. On the other hand, the authors of [36] used a different approach and, in addition to using code metrics, they also decided to use three source code embeddings (code2vec, code2seq and CuBERT). Their method to resolve the severity of code smell is the same as the method we used for DS1 (the severity resolution in this paper is described in Section 3.3). The authors of [36] also decided to focus on Blob (called God Class in their paper) and Long Method.

3. Method

In this section, we describe the details of our research setup and process, including the characteristics of the data set used, the set of predictors, algorithms, and performance metrics used.

The main goal of this paper is to investigate which algorithms achieve the best performance metrics in the

[3] `https://pmd.github.io/`

[4] `https://codebeat.co/` is an automated online code review service developed by the code quest company that assisted in creating the MLCQ data set that, in turn, we needed to help them develop code smell prediction models.

problem of code smell detection (**RQ1**), and to determine whether the difference in performance between various algorithms is substantial. However, we are aware that there may be predictors that we did not include, and that matter; to estimate this effect, we decided to compare our work with other publications that use same data set (but possibly different predictors). We reflect upon this problem in **RQ2**.

Lastly, we also briefly compare our results with the results that other researchers achieved on different data sets, using various techniques. We did not replicate all other studies, but instead we simply included the performance reported by the authors of each method. This activity is the subject of **RQ3** and is meant to present the scale of differences that can be expected when changing data sets.

The research questions we will attempt to answer are the following:

RQ1 Which machine learning algorithms are best used for the detection of code smells?

RQ2 How do our results compare to the results that other researchers were able to achieve on the MLCQ data set?

RQ3 How do our results compare with the results that other researchers were able to achieve (irrespective of the data set)?

3.1. Data set

We use the data set that was originally published in [14]. The data set contains over 14000 reviews of around 5000 Java code snippets gathered from 26 software developers. Reviews are split into four code smells: Blob, Data Class (DC), Feature Envy (FE), and Long Method (LM). Each review assigns exactly one severity of a single smell to a single code snippet. There are four available severity levels: `none`, `minor`, `major`, and `critical`. The exact number of reviews with each severity for each smell is presented in Table 1.

Table 1: Number of reviews per severity per smell

Name	#total	#critical	#major	#minor	#none
Blob	4019	127	312	535	3045
DC	4021	146	401	510	2964
LM	3362	78	274	454	2556
FE	3337	24	142	288	2883

The data set has an embedded cross-check, i.e., every snippet that was selected to exhibit some features of a given smell (i.e., the first reviewer assigned a minor, major, or critical severity to the sample) is cross-checked by another developer. If the reviewers disagree, a third review is provided. The reviewers responsible for performing the cross-check did not see the severity assigned by the first ones, but were aware that they are performing a cross-check.

There were 26 reviewers from at least eight companies (some of the reviewers declined to answer this question), with a maximum of eight developers known to be from a single company. These eight developers contributed around 55% of the total number of reviews. All the developers had industrial experience. 19% of the total number of reviews were contributed by developers with less than two years of experience, 14% by developers with two to five years of experience, and the remaining 67% by developers with more than five years of experience.

The published version of the data set presented in [14] does not contain predictors. This was a deliberate decision, and its goal was to avoid locking researchers into predefined predictors and, possibly, into the errors that the authors have made when calculating those.

Compared to other code smell data sets, there are several important differences: in this one, the reviews were made by experienced software developers, random sampling was used for obtaining candidate code snippets (ones that were later reviewed), and an automated approach was used for project acquisition. These properties make it possible to replicate a code smell data set that should have similar properties.

3.2. Predictors

Predictors are obviously required to perform classification. We decided to use two sets of software metrics: one provided by our industrial partner (i.e., the code quest company that owns the CODEBEAT platform) and the other provided by PMD. Occasionally, some metrics may have been duplicated between those two sets, which might have caused issues for some algorithms, for example for Naive Bayes. PMD is an open-source tool that can analyze source code and calculate metrics (using a special module – metrics framework). Its default set of metrics is limited, but it allows for the creation of custom user-defined metrics. It is also a tool used by other researchers to generate code metrics [52, 51, 53]. CODEBEAT metrics have not yet been used in research, because they are metrics generated by our industrial partner, who calculates them in several languages and was interested in knowing how useful they are for code smell detection, since they were looking at expanding the capabilities of their product.

The initial set of default predictors provided by CODEBEAT and PMD was extended to include the predictors that were most commonly used by other researchers. The analysis of the use of predictors in the literature was done as part of a joint research project with the code quest company. The results were partially published in [20]. The detailed analysis of the usage of predictors is not part of [20], but is based on the data included in the reproduction package for [20], thus is fully reproducible.

Since there are more than 50 metrics available in total, we do not describe all of them in detail here. Instead, we include a brief description in tables A.7 and A.8 and complete documentation in the reproduction package in Section 3.6. Some metrics were only relevant for classes, while

others were only relevant for methods. In total, 49 metrics were used in class smell models and 14 metrics were used for method smell models. We generally did not remove any predictors from the source data set, but instead filtered them out during data preprocessing in our R scripts. A predictor was removed from the data set if its variability was low (below 2% of the samples was different from the mode of the data set), since constant features are of little relevance to machine learning algorithms[54].

We explicitly decided to focus only on the source code metrics and ignore process metrics due to how data set was built: developers were presented with a code sample and then they had to decide whether the sample exhibits a smell or not. Although they did have a link to the entire repository and could look up file history, in practice this option was not used during review (some reviewers used it to look up projects from particularly interesting code snippets, but this was usually done post-review). This means that developers were able to detect the smell from the structure of the code snippet alone, without knowing its history. Therefore, the model should be able to detect the structure as well. This does not mean that process metrics cannot be used as predictors as is the case in software defect prediction [55]; in fact, it is possible that some features that lead to smelly code stem from interactions that can be detected using process metrics [56]. However, our assumption is that it should not be necessary to use process metrics to reach the same conclusions, since the developer does not use them.

In software engineering, in general, obtaining predictors with low correlation is challenging and requires a complex feature engineering process (e.g., all "size" metrics are usually correlated with each other). We decided not to perform such a process, at risk of impacting performance of models that are more sensitive to correlated variables (e.g., Naïve Bayes). This decision should not affect the results of algorithms with built-in resilience to correlation between predictors, such as Random Forest. After performing a post hoc correlation analysis (after the models were already evaluated), we discovered that multiple predictors have a high correlation (occasionally even more than 0.9), which impacts some, but not all, of the models. In particular, the Naïve Bayes classifier is heavily affected, which may be the reason why it performed poorly.

3.3. Severities

Each review from the data set published in [14] contains an assessment of a severity assigned by a reviewer (software developer) to a given code sample for a given code smell – each reviewer assigns a single severity to a sample/smell pair, but there may be multiple reviews of the same sample. There are four possible severities, in ascending order of severity: none, minor, major, and critical. Developers were not presented with any specific guidelines for assigning severities and were asked to select according to their expertise.

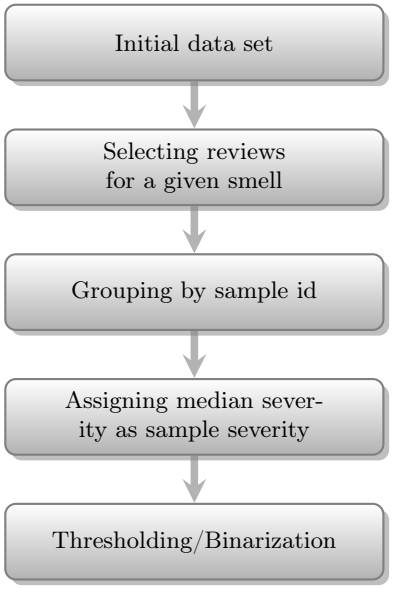

Figure 1: Data set preparation process

For some samples, the original data set has more than one review. For all samples in which any of the reviews points to a severity other than `none` a cross-check was employed. In case of disagreement, a third review was requested. A direct usage of the data set would mean assigning several severities to some of the samples. To avoid this, we assign to each sample the median severity of all its reviews, disregarding who the reviewer was. In the rare case where the number of reviews was even and did not result in a clean median, the higher severity value was used.

After that operation, we have decided to perform the training as a binary classification task. To do that, for each of the code smells, we have created two data sets based on two different cutoff points:

DS1 C1: `none`, C2: `minor` + `major` + `critical`,
DS2 C1: `none` + `minor`, C2: `major` + `critical`.

A diagram showing this process is presented in Figure 1. Samples with severity at the threshold or below are assigned a negative class in the final binary data set ("no smell"), samples above are assigned a positive class ("smell present"). The process was executed two times for each code smell – once with the threshold at `severity=none` (DS1 – positive class contains all samples with median severity equal to minor, major, or critical), second with the threshold at `severity=minor` (DS2 – positive class contains all samples with median severity equal to major or critical).

We also considered using DS3 with only `critical` samples marked as *smelly*, but there were too few samples marked as `critical` to provide a solid training data set. Even in the case of DS2 the data set is severely imbalanced, and in the case of one smell – Feature Envy – it was not possible to build models performing better than random,

thus we do not include analysis for DS2 for Feature Envy in this paper.

Other data sets, e.g., one used in [9], are often balanced to some extent and include a similar number of smells of each kind. This is often done by using advisors (automated scripts that suggest "potential candidates") and/or arbitrary selection of samples for review and training process. Which of the methods should be used to provide data sets that can be used in practical applications may be a subject of further research and we will not discuss it in this paper.

3.4. Algorithms

To preprocess data and build machine learning models, we use the R package `mlr` [57]. Pre-processing steps and models were initially developed by the authors as part of the joint research and development project with the code quest company, but evolved further to the form published on GitHub[5]. We used the following algorithms: Conditional Inference Trees (CTree) [58], Flexible Discriminant Analysis (FDA) [59] Mixture Discriminant Analysis (MDA) [60], K-Nearest Neighbor (KNN) [61], Support Vector Machine (SVM) [62, 63], Naïve Bayes [63], and Random Forests [64]. To rectify the imbalance in the data set, we oversampled the minority class by a factor of 10 using the bagging technique. Originally, we also tried using SMOTE, but in preliminary tests (not reported in this paper), there was no substantial performance improvement.

Obviously, we were unable to verify every possible set of parameters for these algorithms, but a wide range of them were tested using the hyperparameter tuning feature of `mlr`. We present a summary in Section 4.

Whenever the tuning of hyperparameters was applicable, we optimized for the highest value of MCC.

3.5. Performance metrics

As we found in [20], most studies use accuracy, precision, recall, and F-score as their performance metrics. To provide some comparison possibility, we do report those metrics, but at the same time we also provide MCC (recommended by Shepperd et al. [15] and Yao and Shepperd[16]), as well as balanced accuracy.

The final results of our performance metrics are based on average values from 120 runs of a five-fold cross-validation procedure for each machine learning algorithm. Sampling during fold creation is stratified to ensure that each fold receives data from each class. We also provide standard deviation and quartiles in the box plots.

The value of MCC also serves as a basic benchmark, since it indicates how well a classifier performs compared to random guessing. In response to **RQ3**, we briefly present results published in the scientific literature for other data sets. These results are incomparable, since the data set acquisition process is very different. Problems with comparison and reproducibility of code smell data set creation are discussed in detail in [19] – here we only note that very high performance on one data set does not mean high performance for another one. The differences in performance, as noted in [24], can be attributed to the differences in the data sets, and discussing the details of the data set creation procedures is outside the scope of this study.

3.6. Reproduction package

Reproducibility of research is important [17, 19, 18]. To streamline the uptake of the research results of this paper and to guarantee maximum reproducibility of the study, a complete reproduction package is provided. The package can be downloaded from Zenodo `https://doi.org/10.5281/zenodo.7319860`. It contains the following artifacts:

1. a complete data set (including values of all predictors for all samples),
2. brief description of all available predictors,
3. a set of R scripts used to preprocess data, as well as create and evaluate models,
4. a set of R scripts used to generate diagrams and data tables,
5. an instruction on how to use the scripts,
6. a complete set of results, including full confusion matrices and built R models,
7. description of the environment where experiments were conducted,
8. tables with detailed results,
9. scripts used for statistical analysis and their results,
10. a data sheet that describes the contents in detail.

The source code used in the study was also published on GitHub[6] with the tag `1.0.0-paper`. We encourage researchers to verify our results and submit pull requests in case of any issues.

4. Results

In this section, we briefly study the results obtained for each of the smells and methods. Due to limited space, we only present boxplot diagrams with the results here; exact values can be found in the data tables present in a separate data appendix. Statistical analysis of differences between distributions is performed using a global Kruskal-Wallis test with a follow-up pairwise Nemenyi test. Statistical analysis is performed only for MCC values. The nonparametric effect size was estimated using the probability of superiority (Vargha and Delaney's A) $\hat{p}$ – its value ranges from 0 to 1, with 0.5 indicating stochastic equality, and 1 indicating that the first group dominates the second [65].

[5] https://github.com/thecodebeat/model-build-scripts

[6] `https://github.com/tlewowski/code-smell-modelling`

4.1. Feature Envy

Feature Envy in this research is treated as a method-level smell, because the data set for it was gathered on a per-method basis. In the subject literature, Feature Envy is sometimes also treated as a class-level smell. Conceptually, Feature Envy is related to methods that use operations from external classes more often than their own operations. Feature Envy detection performance results for DS1 are shown in Figure 2, while detailed data tables are available in the data appendix included in the reproduction package. Analysis was not performed on DS2 due to a heavy imbalance (20 samples of *smelly* code and 2394 samples of *non-smelly* code).

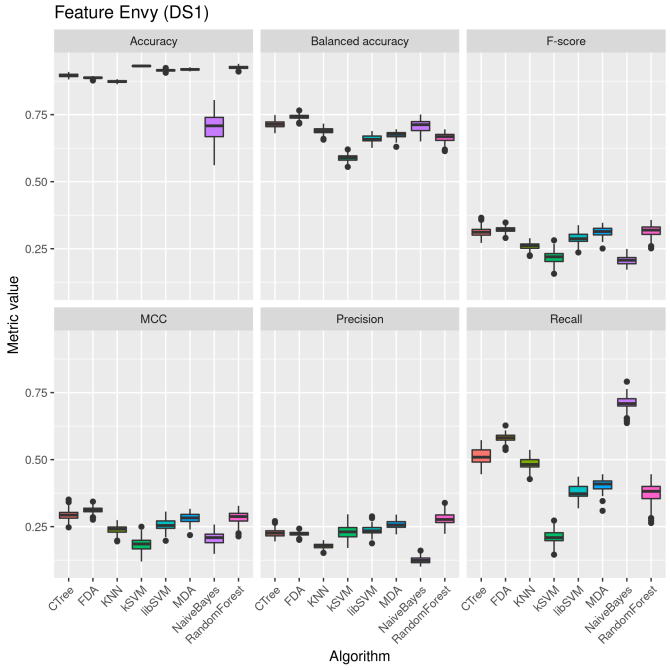

Figure 2: Feature Envy (DS1)

The performance metrics achieved for Feature Envy are by far the lowest among the code smells analyzed. They are also fairly uniform, with a large overlap between most of the techniques (in terms of MCC) and little variance for each technique.

The best performance for DS1 was achieved by the FDA classifier with a median MCC value of 0.31. In terms of other performance metrics, the F-score values follow the same pattern as MCC. Precision is uniformly low for all methods, with the highest median value of 0.26, but the median value of recall in case of Naive Bayes is 0.71, which is 0.13 (over 22%) more than the second best algorithm in terms of recall – FDA. In fact, Naive Bayes classifier focuses on recall so much that it visibly affects not only precision (lowest median of all algorithms – 0.12) but also accuracy (median for Naive Bayes – 0.71, median for the second-worst algorithm – KNN – 0.87).

In the case of Feature Envy detection, MCC was significantly different for different machine learning algorithms ($H(7) = 781.06$, p-value $< 2.2 \times 10^{-16}$). The Nemenyi post hoc test with the chi-square approximation yields a p-value $< 2.2 \times 10^{-16}$ for all comparisons with FDA, except MDA (where the p-value was 8.9×10^{-7}) and Random Forest (where the p-value was 2.2×10^{-5}). The value of $\hat{p}$ for all pairwise comparisons with FDA is at least 0.847, which (according to Kitchenham et al. [65]) is considered a large effect size. This confirms that in the case of Feature Envy detection, for DS1, the FDA algorithm achieves the best results in terms of MCC.

The number of positive samples for this code smell is very low, which may be the reason why the performance metrics are so low (compared to other code smells). This can be interpreted as a problem with data gathering, but since the data were gathered using the same flow for all code smells, this may also be an indicator that Feature Envy is either not a common problem in modern software projects (at least in the eyes of the software engineers involved in code smell reviews) or that it substantially differs from other analyzed smells. We were unable to conduct follow-up interviews with the reviewers to decide whether this is the case, so we decided that it remains a problem to be solved in the future.

4.2. Long Method

Long Method is a smell at the method level. Contrary to its name, it is associated not only with methods that are too long (span over too many lines), but also with ones that perform too many actions and have too many responsibilities or are too complex – these properties usually correlate with size. Long Method detection performance results for DS1 are shown in Figure 3 and for DS2 in Figure 4, while detailed data tables are available in the data appendix included in the reproduction package.

Performance metrics achieved for Long Method are the highest among the code smells analyzed. They are also uniform both between algorithms – there is less than 10% difference between the best and worst algorithm for each data set (in terms of median MCC) – and also for multiple iterations of a single algorithm. There is also a large overlap between algorithms when it comes to results.

Generally, it seems easier to separate classes in DS2 than in DS1. The best performance was achieved by the Random Forest classifier, with a median MCC value greater than 0.80 for DS2 and greater than 0.75 for DS1.

In terms of other performance metrics, the values of the F-score follow the same pattern as MCC. An interesting case is DS1 – MCC values of both Naive Bayes and Random Forest algorithms are very similar (highest of the algorithms analyzed), but there is a huge difference in terms of precision and recall – Naive Bayes algorithm has by far the highest precision (with minimum above maximum for any other algorithm) and lowest recall (with maximum overlapping only with minimum for KNN), while Random Forest has median precision slightly above others (save Naive Bayes) and median recall in the

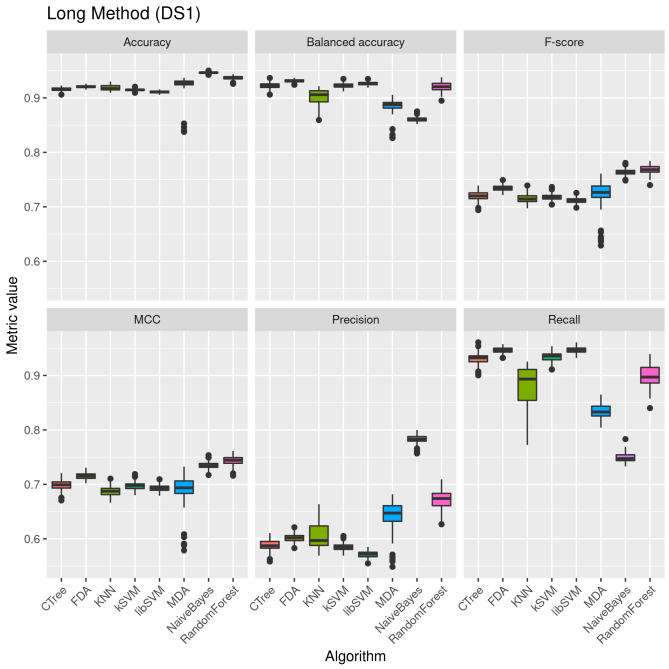

Figure 3: Long Method (DS1)

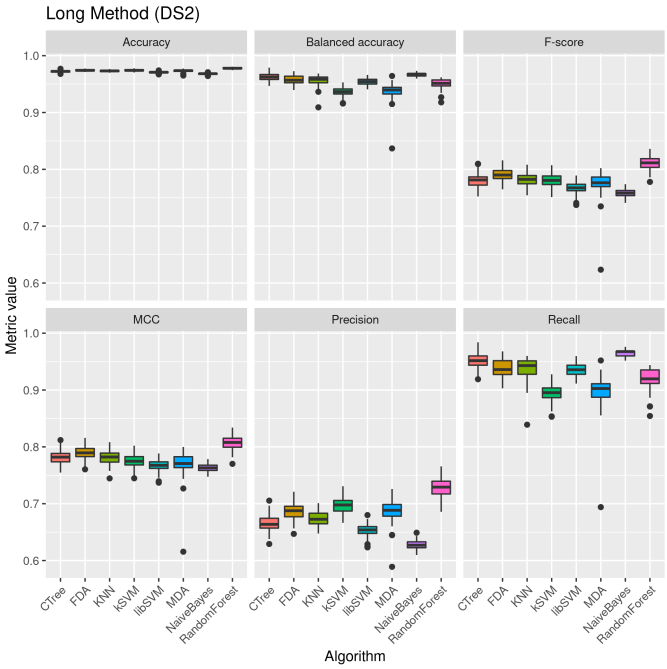

Figure 4: Long Method (DS2)

middle of the pack, with a lot of overlap with other methods in both cases. In the case of DS2, such an effect cannot be observed.

In the case of Long Method detection, MCC was significantly different for different machine learning algorithms, i.e., ($H(7) = 724.12$, p-value $< 2.2 \times 10^{-16}$) for DS1, while ($H(7) = 531.62$, p-value $< 2.2 \times 10^{-16}$) for DS2. For DS1, the Nemenyi post hoc test with the chi-square approximation produces a p-value $< 2.2 \times 10^{-16}$ for all comparisons with Random Forest, apart from FDA (where the p-value was 2.5×10^{7}) and Naive Bayes (where the p-value was 0.7695). The nonparametric effect size $\hat{p}$ is at least 0.799 in case of comparisions of Random Forest with all other algorithms, which (according to [65]) is considered a large effect size. For DS2, the situation is similar. The Nemenyi post hoc test with the chi-square approximation yields a p-value $< 2.2 \times 10^{-16}$ for all comparisons with Random Forest, apart from FDA (where the p-value was 0.00012), CTree (where the p-value was $1.3e-14$), and KNN (where the p-value was 4.4×10^{-16}). The nonparametric effect size $\hat{p}$ is at least 0.866 in case of comparisions of Random Forest with all other algorithms, which (according to [65]) is considered a large effect size. This confirms that in the case of Long Method detection, both for DS1 and DS2, the Random Forest algorithm achieves the best results in terms of MCC.

MDA seems to be the least stable algorithm, with a number of low-precision models affecting its overall range of results.

4.3. Blob

Blob is a class-level smell. Its meaning is similar to the meaning of **Long Method** – it represents long, complex, and contrived fragments of code that are hard to understand for developers. There are several smells closely related to Blob (God Class, Large Class, Brain Class), which share most of the properties with Blob, perhaps with slightly different focus. They are all included in MLCQ under the "Blob" label. Blob and its sibling smells are the most researched code smells according to a recent review [20]. Blob detection performance results are shown for DS1 in Figure 5 and DS2 in Figure 6, while detailed data tables are presented in the data appendix included in the reproduction package.

Generally, it seems marginally easier to separate classes in DS1 than in DS2. The best performance for DS1 was achieved by the Random Forest Classifier with a median MCC value of 0.51. In the case of DS2, the best performance was achieved by the FDA and MDA classifiers, with a median MCC value of 0.46.

Blob is a smell with fairly uniform performance with regard to MCC, F-score, and precision, but substantially varying when it comes to recall, with high variability for each method and big overlap between methods, especially in the case of DS2. The FDA classifier achieved a particularly high recall, in the case of DS1 reaching 0.88. While there are several outliers below this value, it is worth noting that the upper three quartiles cover only values between 0.88 and 0.90, which is a remarkably stable result. The algorithm with the second recall value, CTree, had a median recall of 0.84 with the upper three quartiles between 0.84 and 0.87. All other medians are below 0.80.

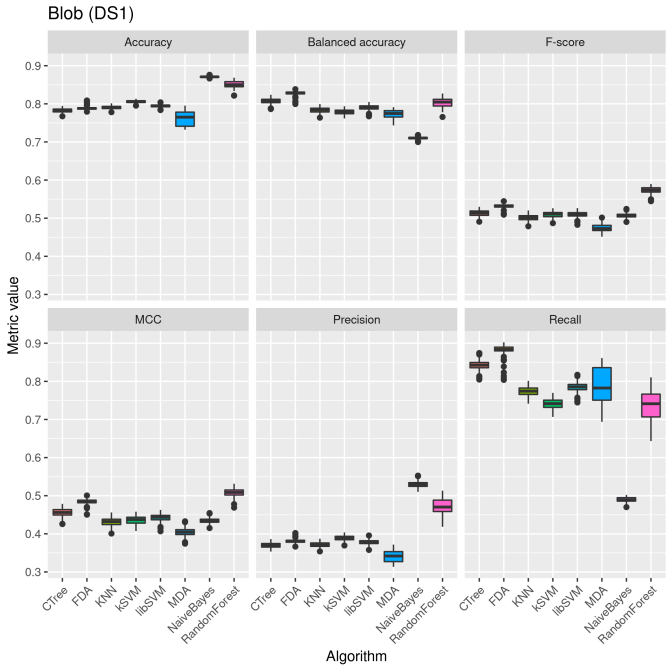

Figure 5: Blob (DS1)

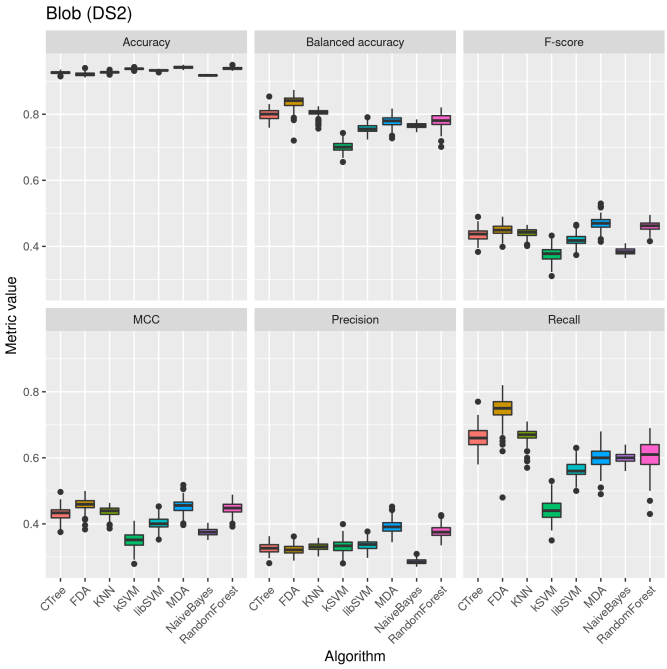

Figure 6: Blob (DS2)

For the same data set, the highest precision was achieved in case of Naive Bayes - a median of 0.53 with the next algorithm – Random Forest – reaching a median value of 0.47.

In case of Blob detection, MCC was significantly different for different machine learning algorithms, i.e., ($H(7) = 810.43$, p-value $< 2.2 \times 10^{-16}$) for DS1, while ($H(7) = 715.17$, p-value $< 2.2 \times 10^{-16}$) for DS2. For DS1, the Nemenyi post hoc test with the chi-square approximation produces a p-value $< 2.2 \times 10^{-16}$ for all comparisons with Random Forest, apart from FDA (where the p-value was 0.1769) and CTree (where the p-value was 4.0×10^{-10}). The nonparametric effect size $\hat{p}$ is at least 0.965 in case of comparisions of Random Forest with all other algorithms, which (according to [65]) is considered a large effect size. This confirms that in the case of Blob detection, for DS1, the Random Forest algorithm achieves the best results in terms of MCC. For DS2, the Nemenyi post hoc test with the chi-square approximation yields a p-value $< 2.2 \times 10^{-16}$ for all comparisons with FDA, except MDA (where the p-value was 0.99874), Random Forest (where the p-value was 0.53945), and KNN (where the p-value was 6.1×10^{-4}). The nonparametric effect size $\hat{p}$ is at least 0.798 in case of comparisions of Random Forest with CTree, KNN, kSVM, libSVM, Naïve Bayes, which (according to [65]) is considered a large effect size, whilst in comparisions of FDA with Random Forest $\hat{p} = 0.667$ and MDA $\hat{p} = 0.556$, which are considered medium-to-large and small effect size, respectively. This confirms that in the case of Blob detection, for DS2, the FDA algorithm achieves the best results in terms of MCC, while MDA is the second.

4.4. Data Class

Data Class is a particularly interesting smell, since, despite including it in the original list of code smells, it is not universally considered harmful. On the contrary, Data Classes are a typical pattern when it comes to data transfer or persistence even in object-oriented programming. Indeed, Data Class is only viewed as a problem in the object-oriented paradigm when acting as a domain object. We will not discuss here whether using Data Class is the right design choice, and instead we will focus on the reviews provided by experts. Please note that even if using a Data Class is justified, it is still marked as a Data Class in the data set.

Detection of Data Class using the metrics that we had access to is by no means straightforward. The performance results of the Data Class detection are shown for DS1 in Figure 7 and for DS2 in Figure 8, while detailed data tables are presented in the data appendix included in the reproduction package.

The differences between various methods' performance results in Data Class detection are the biggest in the study.

Generally, it seems easier to separate classes in DS1 than in DS2. The best performance for both DS1 and DS2 was achieved by the Random Forest classifier, with a median MCC value of 0.57 for DS1 and 0.53 for DS2.

The performance results for Data Class are generally consistent for multiple iterations of the same algorithm (except for recall for KNN for both data sets, Random Forest for DS1 and FDA for DS2). However, they do vary substantially between algorithms. Naive Bayes has the worst recall results in the case of DS1 (median of 0.38, compared to next algorithm – 0.74 for kSVM). This is par-

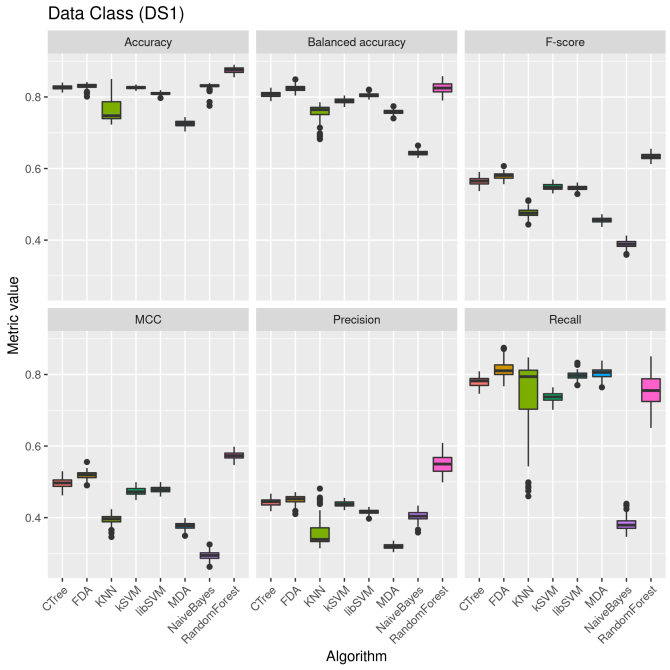

Figure 7: Data Class (DS1)

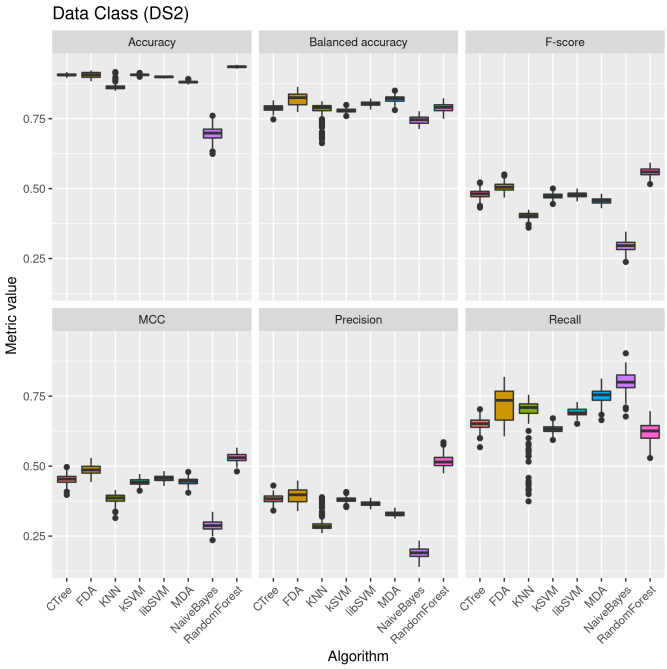

Figure 8: Data Class (DS2)

ticularly surprising considering that for DS2 median recall for Naive Bayes is the highest of all methods – 0.80 (on the other hand, precision is very low – 0.19 – making the results in terms of both F-score and MCC worst out of the compared algorithms). The highest precision is achieved by Random Forest for both DS1 and DS2 – 0.55 and 0.51, respectively.

In the case of Data Class detection, MCC was significantly different for different machine learning algorithms, i.e., ($H(7) = 909.77$, p-value $< 2.2 \times 10^{-16}$) for DS1, while ($H(7) = 828.85$, p-value $< 2.2 \times 10^{-16}$) for DS2. For DS1, the Nemenyi post hoc test with the chi-square approximation produces a p-value $< 2.2 \times 10^{-16}$ for all comparisons with Random Forest, apart from FDA (where the p-value was 0.0536) and CTree (where the p-value was 1.3×10^{-8}). The nonparametric effect size $\hat{p}$ is greater than 0.999 in the case of comparisons of Random Forest with all other algorithms, which (according to [65]) is considered a large effect size. For DS2, the Nemenyi post hoc test with the chi-square approximation produces a p-value $< 2.2 \times 10^{-16}$ for all comparisons with Random Forest, apart from FDA (where the p-value was 0.047), and libSVM (where the p-value was 4.4×10^{-16}). The nonparametric effect size $\hat{p}$ is at least 0.960 in case of comparisions of Random Forest with all other algorithms, which (according to [65]) is considered a large effect size. This confirms that in the case of Data Class detection, both for DS1 and DS2, the Random Forest algorithm achieves the best results in terms of MCC.

5. Discussion

In two of the three compared cases (Blob and Data Class), a better performance was achieved with respect to the median MCC in the case of DS1 (Figure 5, Figure 7) than in the case of DS2 (Figure 6, Figure 8). In the case of the third, Long Method, better median MCC was achieved for DS2 (Figure 4) than for DS1 (Figure 3). This does not give a conclusive answer regarding whether developers perceive "code smelliness" as a binary parameter or a continuum. In fact, this may even vary between smells and developers, e.g., a claim in [51] is made that for Blob and Data Class the main variation is between minor/none and critical/major severities, while for Long Method and Feature Envy no such grouping was identified. This subject requires further research involving software developers. This will be relevant for both researchers and practitioners who aim to create data sets of code smells, as it would give them guidelines on whether to expect developers to rank "smelliness" of the code on a scale or as a binary attribute (or whether different strategies should be employed for different smells).

5.1. RQ1: which machine learning algorithms are best used for detection of code smells?

To decide which algorithm is the best for detecting code smells, we have to take into account the performance for all smells and all data sets. We decided to create a ranking of methods and then recommend the top ones. In this study, we rank on the basis of the median MCC achieved by the algorithm. The complete ranking is presented in Table A.4. For each code smell and data set, we rank the

methods for their performance (median MCC) – for each data set, a method gets assigned a number of points equal to the number of methods that performed worse than it on the given data set. Finally, we sum the points for each method, performing a de facto Borda count voting.

A winner in this setup is the Random Forest algorithm with a total score of 45, closely followed by the FDA algorithm with a total score of 43. In every analyzed case, one of those algorithms performed best, and both were in the top three for every analyzed case. Of course, if the desired characteristics differ from what we assumed (e.g., the goal is to reduce false positives, even at the expense of false negatives), the best choice may be different. However, if using an F-score specifically, the top three results are the same (although with slightly different scores), as shown in Table A.5. A Friedman test performed on the MCC result yields a p-value below 10^{-4} which suggests that those results come from more than one distribution. However, a post hoc Nemenyi test between all pairs yields only six pairs for which the p-value is below 0.05: FDA-KNN, FDA-kSVM, FDA-Naive Bayes, Random Forest-KNN, Random Forest-kSVM, Random Forest-Naive Bayes. The nonparametric effect size $\hat{p}$ is at least 0.918 in case of comparisons of Random Forest with all algorithms except FDA, for which the value is 0.633. All other comparisons with FDA also have an effect size of at least 0.918. This can be interpreted as a large dominance of Random Forest and FDA over all other algorithms and a medium dominance of Random Forest over FDA.

We observed that in the case of our research, using F-score instead of MCC would not affect the qualitative output of the study – while the exact values would differ, those two metrics follow a similar distribution for all analyzed cases. Detailed differences are shown in Table A.6 – the largest difference in ranking is two places in three cases. The overall ranking would change only in spots 4-7, where differences between methods are minimal anyway. Although this result does not invalidate the general recommendation of Shepperd et al. and Yao and Shepperd [15, 16] to use performance metrics that account for all four quadrants of the confusion matrix, we agree with, it is a suggestion that, in the particular area of code smell detection, previous research that relied on F-score is likely to achieve comparable results in terms of MCC.

In most cases, the best machine learning techniques are able to achieve MCC values that correspond to moderate to strong correlation with the actual existence of a code smell. While, in general, the choice of an algorithm has to be done on a case-by-case basis, Random Forest and FDA algorithms have consistently achieved top results in terms of median MCC and median F-score and can be recommended as a starting point for further research.

Although Random Forest and FDA achieved the best overall results, in many cases other methods have achieved comparable results, often overlapping. Such uniformity cannot be attributed to algorithm-specific shortcomings, as a wide range of types of algorithms was analyzed–including tree-based, analytic, instance-based, SVM and statistical. It may be that different samples are classified correctly by different types of classifiers, which would mean that the results may be further improved by using classifier ensembles, but according to preliminary results (this research is still ongoing), using ensembles is unlikely to introduce a substantial difference in classification performance (the more so that Random Forrest is already a kind of ensemble model). This observation led us to believe that it is unlikely that the use of other machine learning algorithms alone would yield substantially different results. The consequence of this conclusion is that, to improve the performance of the classifiers, new attributes should be added to the samples. Those attributes should be new code metrics – while other areas of research on software engineering (e.g., research on defect prediction) also use process metrics, we claim that these are also unlikely to strongly improve the performance of code smell detection. The reason for that is the process used for data set creation – while code smell severities are assigned by domain experts in [14], those experts are not affiliated with reviewed projects in any way. In fact, in most cases they see the reviewed code for the first time – thus, they were not able to use any process metrics for their assessment. Determining which specific code metrics would improve performance the most is subject to further research. It is possible that less typical features (related to properties such as code styling, variable/function naming, and indentation) may positively affect models' results. Some work in this area has been done in [36], where neural embeddings are used as predictors with promising results, but this research is still in its early stage.

5.2. RQ2: how do our results compare to results that other researchers were able to achieve on the MLCQ data set?

We were not able to compare our results with other code smell detection tools, because existing tools (such as JDeodorant [66, 67] or JSpIRIT [68]) are focused on improving developers' performance, not on large-scale studies. As such, those tools work as IDE plugins. Since MLCQ is composed from over 500 projects, it is not feasible to manually import to an IDE and analyze all of them.

There are two papers that we know of that attempt to solve the same problem – detect code smells – using the same MLCQ data set and machine learning techniques [51] and [36]. Results of [36] are directly comparable to ours on DS1, while results of [51] are not, since the authors of [51] focused on the effect of the severity cut-off threshold on precision and accuracy, while we focused on performance, particularly MCC, of various machine learning techniques on fixed, two data splits into binary classes.

The main reason for incomparability is different data preprocessing: we decided to first assign a higher median severity to each sample and then split into DS1 and DS2, while the authors of [51] decided to use a different severity

aggregation technique, based on a weighted mean of all severities.

Since it is important to establish a common baseline, we decided to replicate the aggregation technique presented in [51] to verify that our results match. It is important since we use a slightly different set of predictors, thus it is possible that we have no access to some relevant information. This is particularly important since one of the conclusions of **RQ1** is that additional predictors are needed.

Replication was done only by training ten Random Forest models for each data point. The replication was run twice – once when hyperparameters were optimized for the best value of MCC, and the second time when they were optimized for best value of precision. We present only results optimized for MCC but results of the run optimized for precision are not much different. The authors of [51] decided to analyze only Blob and Data Class, thus only those are included in the replication.

The reproduction was executed using the same workflow as our original research with three exceptions:

1. sample aggregation was done using a sum instead of median,
2. thresholding into positive/negative class was done based on the sum and not on the median,
3. in one of the runs (200 models) we optimize hyperparameters for precision, not for MCC.

We trained a total of 400 models (200 for Data Class and 200 for Blob). The results are shown in Figures 9 and 10, while detailed performance data are presented in the data appendix included in the reproduction package.

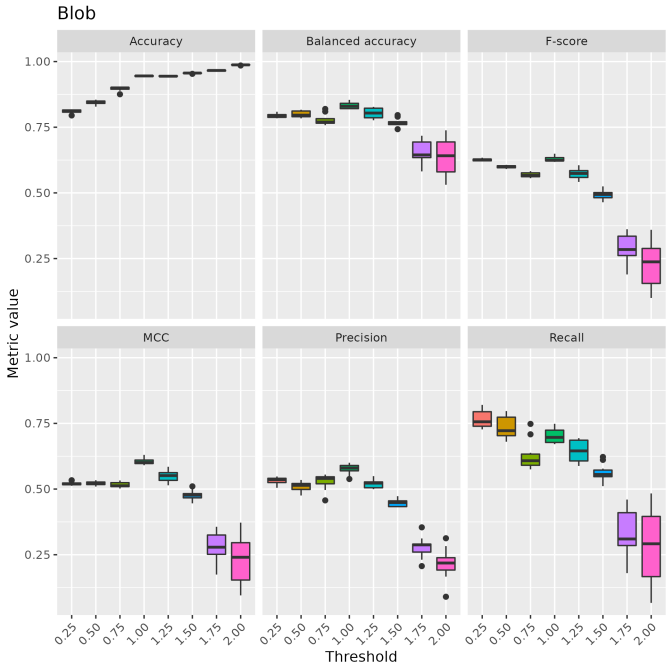

Figure 9: Blob, Random Forest (reproduction)

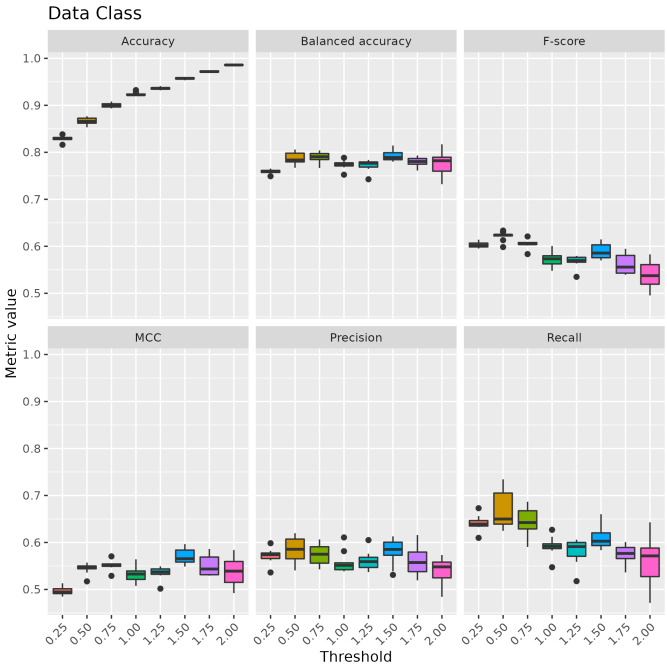

Figure 10: Data Class, Random Forest (reproduction)

Overall, we have not been able to reproduce the results shown in [51]. With a threshold above 2.0, the learning process was getting unstable due to the small number of positive samples (17 for Blob at 2.25, 6 for Blob at 2.50, 29 for Data Class at 2.25, 18 for Data Class at 2.50), thus this area is not shown on Figure 9 and Figure 10.

In the area analyzed, our results, compared to the original results shown in [51], are worse in terms of precision (accuracy is not a valid metric for imbalanced data sets, so we will not discuss it). Assuming that the results of [51] are valid, the most likely explanation is that the Understand [69] metrics are much more informative with regard to code smell detection than the CODEBEAT metrics (both Understand and CODEBEAT are commercial offerings). A list of more than 100 metrics provided by Understand, together with their definitions, is available online[7]. On the other hand, the list of metrics provided by CODEBEAT is much smaller – there are a total of 28 metrics, which are briefly discussed in Table A.7.

The research described in [36] was also concerned only with Long Method and Blob (called God Class there). We present their results and compare them to the median Random Forest result on DS1 (column Δ_{RF}) in Table 2 (only F-scores are included).

As presented in Table 2, results achieved by our Random Forest classifier achieved higher results in terms of F score compared to the approaches used in [36].

[7]See `https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have-`

Method	Smell	F-score	Δ_{RF}
ML_code2vec	Long Method	0.24	-0.53
H_metrics	Long Method	0.48	-0.29
ML_code2seq	Long Method	0.55	-0.22
ML_metrics_votes	Long Method	0.63	-0.14
ML_metrics	Long Method	0.67	-0.10
ML_CuBERT	Long Method	0.75	-0.02
ML_code2vec	Blob	0.26	-0.31
H_metrics	Blob	0.41	-0.16
ML_code2seq	Blob	0.49	-0.08
ML_metrics_votes	Blob	0.51	-0.06
ML_metrics	Blob	0.52	-0.05
ML_CuBERT	Blob	0.53	-0.04

Table 2: Results achieved in [36] and their comparison to median results of Random Forest from the current study

5.3. RQ3: how do our results compare to the results that other researchers were able to achieve (regardless of the data set)?

Our study, like other studies discussed in Section 5.2, uses the MLCQ data set as the source of truth. However, there are many other data sets used in the scientific literature, often used only in a single study. The problems with this approach are discussed in Section 3.5 and in more detail in [19].

Here, we only discuss papers that report detection of at least one of the following four code smells: Blob (or God Class/Large Class/Brain Class), Data Class, Feature Envy and Long Method (or Brain Method), and use Java projects for that. Although those results are not directly comparable, we include them here to present the reader with the range of results that may be expected in this sort of research. We also want to show that selecting a data set is a critical aspect for code smell detection, and the domain will face challenges until the data acquisition procedures are not agreed upon.

In Table 3, we gathered some of the results published by other researchers in recent years. We extracted the value of the F1 metric (MCC was frequently unavailable, and we decided it is preferable to provide a comparable metric to all results), and only included results from papers that reported it. If the values presented in the table were universally considered valid, the detection problem would already be solved in 2015, when the best value of F1 for each smell exceeds 0.95. Presented results are only example results from the last few years. The point of presenting those is not to make a complete review, but to give the reader a perspective on how big the differences can be. Interestingly, the performance results do not appear to correlate with the publication time, since the oldest paper in Table 3 ([9]) is also the one that presents the highest (or second-highest for some smells) performance results.

However, later research on the same smells using the same algorithms and similar sets of predictors (predictors are not included in Table 3, but the algorithms predominantly used code metrics as their input data) show that those results are not stable across data sets. This was pointed out in [24] and was also an inspiration for creating MLCQ [14].

5.4. Study implications

Machine learning algorithm performance—results of our research clearly show that Random Forest is the algorithm that performs best overall. This is not surprising, as this algorithm is commonly considered one of the best and is widely used to estimate the ceiling. However, interestingly, Flexible Discriminant Analysis (FDA) performs not much worse. To our knowledge [20], this is the first paper on code smell detection that includes the FDA algorithm, and, considering the results, we believe that this may be a promising path to follow.

MCC vs F-score—despite the fact that the F-score omits one quadrant of the confusion matrix and, theoretically speaking, the results may differ substantially between F-score and MCC, in our research we have shown that in the area of code smell detection, the actual differences are minimal and using any of those metrics will lead to similar conclusions.

Performance boundaries—while the research community struggles to achieve the best possible performance metrics, it is not obvious what those maximum values are. Considering that the agreement between developers on what constitutes code smell is not very high [45], additional research is needed to estimate the achievable target performance.

Missing predictors—our results suggest that the main factor that can improve the detection results are additional code metrics. These may be metrics from the Understand tool or new ones (e.g., ones that include meaning of variable names), but changes in machine learning algorithms alone are unlikely to substantially affect the results.

Open source tools—both this paper and [51] use predictors calculated by commercial tools (CODEBEAT in our case, Understand in case of [51]). This poses a threat to the reproducibility of these studies, since other

Reference	Year	Smell	Algorithm	F1	Dataset source
[9]	2015	Data Class	Random Forest	0.989	[9]
[9]	2015	Data Class	Naïve Bayes	0.980	[9]
[9]	2015	Data Class	SVM	0.71	[9]
[9]	2015	Data Class	J48	0.987	[9]
[70]	2019	Data Class	GA	0.65-0.89	[70]
[70]	2019	Data Class	PEA	0.72-0.90	[70]
[70]	2019	Data Class	MOGP	0.65-0.89	[70]
[70]	2019	Data Class	SP-J48	0.70-0.90	[70]
[9]	2015	Blob	Random Forest	0.980	[9]
[9]	2015	Blob	Naïve Bayes	0.981	[9]
[9]	2015	Blob	SVM	0.959	[9]
[9]	2015	Blob	J48	0.980	[9]
[41]	2019	Blob	Naïve Bayes	0.41	[71]
[41]	2019	Blob	Rules (DÉCOR)	0.16	[71]
[33]	2019	Blob	Deep Learning	0.223	[33]
[70]	2019	Blob	GA	0.90-1.00	[70]
[70]	2019	Blob	PEA	0.99-1.00	[70]
[70]	2019	Blob	MOGP	0.94-1.00	[70]
[70]	2019	Blob	SP-J48	1.00	[70]
[72]	2020	Blob	GBM	0.843	[33]
[72]	2020	Blob	Random Forest	0.847	[33]
[72]	2020	Blob	KNN	0.725	[33]
[72]	2020	Blob	Naïve Bayes	0.547	[33]
[9]	2015	Feature Envy	Random Forest	0.973	[9]
[9]	2015	Feature Envy	Naïve Bayes	0.936	[9]
[9]	2015	Feature Envy	SVM	0.941	[9]
[9]	2015	Feature Envy	J48	0.970	[9]
[70]	2019	Feature Envy	GA	0.45-0.80	[70]
[70]	2019	Feature Envy	PEA	0.91-1.00	[70]
[70]	2019	Feature Envy	MOGP	0.49-0.89	[70]
[70]	2019	Feature Envy	SP-J48	1.00	[70]
[33]	2019	Feature Envy	Deep Learning	0.519	[33]
[72]	2020	Feature Envy	GBM	0.146	[33]
[72]	2020	Feature Envy	Random Forest	0.300	[33]
[72]	2020	Feature Envy	KNN	0.044	[33]
[72]	2020	Feature Envy	Naïve Bayes	0.005	[33]
[9]	2015	Long Method	Random Forest	0.996	[9]
[9]	2015	Long Method	Naïve Bayes	0.984	[9]
[9]	2015	Long Method	SVM	0.976	[9]
[9]	2015	Long Method	J48	0.993	[9]
[41]	2019	Long Method	Naïve Bayes	0.23	[71]
[41]	2019	Long Method	Rules (DÉCOR)	0.44	[71]
[33]	2019	Long Method	Deep Learning	0.555	[33]
[72]	2020	Long Method	GBM	0.242	[33]
[72]	2020	Long Method	Random Forest	0.225	[33]
[72]	2020	Long Method	KNN	0.084	[33]
[72]	2020	Long Method	Naïve Bayes	0.196	[33]

Table 3: Results achieved by other researchers

researchers may have trouble acquiring those tools. To alleviate this risk, we provide descriptions of those metrics. Although we think this is acceptable as a step in an exploratory study, we also believe that metrics that prove to be useful should be ported to open-source tools, such as PMD[8] or JavaMetrics[9]. PMD out-of-the-box is able to calculate over a dozen metrics; JavaMetrics offers several dozen metrics. As such, they could be extended during each study (separately by each researcher), which could decrease reproducibility of the results. Understand itself is capable of calculating more than 100 metrics, and the results of this study suggest that some of them deliver value. Researchers should identify which ones and port them to open source tools so that they could be used by a wider community.

Data sets—as shown in Section 5.3, the results for the same algorithms and similar sets of predictors can vary greatly. This variability is attributed to the data selection process, which is not yet standardized in the research community. Lack of standarization may be caused by vague definitions of code smells and lack of agreement between experts on what constitutes a smell. Although earlier literature suggests that code smells are a useful concept [73, 74], perhaps it would be more efficient to infer them from defect data sets (as structural defect predictors), rather than to rely on developers' understanding.

5.5. Threats to validity

To analyze threats to the validity of our study, we follow the classification of threats given by Wohlin [75] and discuss the internal and external validity of the study.

5.5.1. Internal validity

The whole analysis presented in this study was automated. It is always possible that an important defect exists somewhere in the relevant source code. Nowadays, this is even more likely, since not only our code is relevant to the results but also code of all libraries that are used in the study. Of course, we cannot guarantee their correctness. However, we used well-known and tested machine learning libraries in a very mature environment. We believe that this minimizes the likelihood of a library error that is relevant for the results, but since this possibility cannot be rejected, in the reproduction package, we provide a full list of dependencies and their versions, including both a list of R packages and a list of operating system packages that were installed when performing calculations.

We also reviewed our code several times, but the possibility of it containing defects cannot be excluded. To aid researchers who wish to reproduce this study, we published the whole source code together with the results. Each artifact used in the study can be downloaded from an online appendix described in Section 3.6.

A likely problem with some of the algorithms – particularly Naïve Bayes – is their assumption about independent predictors. In our study, this criterion was not fulfilled (as proven by high correlation yield by a post hoc analysis of predictors), so solutions that do not have embedded resilience to this sort of problems may be improved by feature engineering.

5.5.2. External validity

The models described in this study were built from reviews created by a group of engineers with more or less similar professional backgrounds, using a single set of open source Java projects. We do not claim that these techniques are transferable outside the realm of open-source Java projects. Indeed, we do not even claim that the smells themselves are transferable. Further research is needed to decide on that. This study has simply set the bounds on performance of machine learning models that use product metrics as predictors for samples obtained by a wide group of engineers, as there are studies showing low agreement on what constitutes a code smell [76, 45].

6. Conclusions

Our study shows that Random Forest is the best algorithm overall to detect code smells in Java source code – performance measure that it was able to achieve was the best in five our of seven experiments. In all cases, the difference was statistically significant, with large effect size in four of those cases. Flexible Discriminant Analysis was the second algorithm which performed very well and consistently achieved performance in top three algorithms.

Overall, the degree to which we are able to detect a code smell varies substantially depending on the smell itself – we are able to detect Long Method with high confidence, Blob and Data Class with medium confidence and Feature Envy with low confidence. For all cases, the algorithms performed better than random guessing.

It should be noted that using F-score instead of MCC would not affect the conclusions, as it appeared that these two metrics follow a similar distribution for all the cases analyzed. This means that although MCC is generally a recommended performance measure (see [15, 16]), this does not necessarily invalidate conclusions based on F-score, widely used, especially but not only in older publications.

We were able to achieve an F-score higher than that presented in [36], that used the same data set and severity preprocessing, although [36] used very advanced algorithms, including the CuBERT model. On the other hand, we were not able to replicate the results from [51], especially with higher severity thresholds, perhaps due to the lower number of metrics used.

We have shown that results achieved on different data sets cannot be reasonably compared – results achieved by researchers when they use different data sets can range

[8] `https://pmd.github.io/`
[9] `https://github.com/LechMadeyski/JavaMetrics`

from 0.005 to 0.936, even for the same algorithm (Naïve Bayes for Feature Envy). Until a widely accepted benchmark data set is established, the performance of the methods should only be compared when applied to the same data set(s).

With baseline results established in this study, there are two main directions for further research: first, improving code smell detection results. It seems that the most promising way to achieve that is to increase the number of software metrics available to the model, for example, with metrics calculated by the Understand [69] tool. Second, a benchmark should be developed that would allow practical evaluation of code smell models, since the goal is not to reduce the number of code smells *per se*, but to decrease the number of defects in software and/or decrease overall maintenance effort, evaluation of models should take this into account. In particular, we would like to evaluate whether detection models trained on MLCQ can positively affect the performance of defect prediction models or maintenance effort prediction models.

CRediT authorship contribution statement

Lech Madeyski: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing – original draft, Writing - review & editing, Supervision, Funding acquisition. **Tomasz Lewowski**: Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing - review & editing, Visualization.

Declaration of competing interest

This research was carried out in collaboration with code quest sp. z o.o. The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Reproduction package (including data and code) is discussed in detail in Section 3.6.

Acknowledgement

This work has been partially carried out as part of the research and development project POIR.01.01.01-00-0792/16 supported by the National Centre for Research and Development (NCBiR), and the research internship of Lech Madeyski at BTH. The authors thank Tomasz Korzeniowski and Marek Skrajnowski from code quest sp. z o.o. for all of the support, comments, and feedback from the real-world software engineering environment.

References

[1] H. Krasner, The cost of poor software quality in the US: A 2020 report, 2021. URL: https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf.

[2] T. Bakota, P. Hegedűs, G. Ladányi, P. Körtvélyesi, R. Ferenc, T. Gyimóthy, A cost model based on software maintainability, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 316–325. doi:10.1109/ICSM.2012.6405288.

[3] A. Endres, An analysis of errors and their causes in system programs, IEEE Transactions on Software Engineering 1 (1975) 140–149. doi:10.1109/TSE.1975.6312834.

[4] C. Ebert, M. Bundschuh, R. Dumke, A. Schmietendorf, Defect Detection and Quality Improvement, Best Practices in Software Measurement: How to use metrics to improve project and process performance, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 133–156. doi:10.1007/3-540-26734-4_9.

[5] Z. Li, N. Madhavji, S. Murtaza, M. Gittens, A. Miranskyy, D. Godwin, E. Cialini, Characteristics of multiple-component defects and architectural hotspots: A large system case study, Empirical Software Engineering 16 (2011) 667–702. doi:10.1007/s10664-011-9155-y.

[6] T. D. Oyetoyan, D. S. Cruzes, R. Conradi, A study of cyclic dependencies on defect profile of software components, Journal of Systems and Software 86 (2013) 3162–3182. doi:10.1016/j.jss.2013.07.039.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, MA, USA, 1999.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[9] F. A. Fontana, M. V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empirical Software Engineering 21 (2016) 1143–1191. doi:10.1007/s10664-015-9378-4.

[10] N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (2010) 20–36. doi:10.1109/TSE.2009.50.

[11] F. Palomba, Textual Analysis for Code Smell Detection, in: Proceedings - International Conference on Software Engineering, volume 2, 2015, pp. 769–771. doi:10.1109/ICSE.2015.244.

[12] S.-J. Lee, L. Lo, Y.-C. Chen, S.-M. Shen, Co-changing code volume prediction through association rule mining and linear regression model, Expert Systems with Applications 45 (2016) 185–194. doi:10.1016/j.eswa.2015.09.023.

[13] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings, 2013, pp. 268–278. doi:10.1109/ASE.2013.6693086.

[14] L. Madeyski, T. Lewowski, MLCQ: Industry-Relevant Code Smell Data Set, in: Proceedings of the Evaluation and Assessment in Software Engineering, EASE '20, Proceedings of the Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA, 2020, p. 342–347. doi:10.1145/3383219.3383264.

[15] M. Shepperd, D. Bowes, T. Hall, Researcher Bias: The Use of Machine Learning in Software Defect Prediction, IEEE Transactions in Software Engineering 40 (2014) 603–616. doi:10.1109/TSE.2014.2322358.

[16] J. Yao, M. Shepperd, The impact of using biased performance metrics on software defect prediction research, Information and Software Technology 139 (2021) 106664. doi:10.1016/j.infsof.2021.106664.

[17] L. Madeyski, B. Kitchenham, Would wider adoption of reproducible research be beneficial for empirical software engineering

research?, Journal of Intelligent & Fuzzy Systems 32 (2017) 1509–1521. doi:10.3233/JIFS-169146.

[18] B. Kitchenham, L. Madeyski, P. Brereton, Meta-analysis for Families of Experiments in Software Engineering: A Systematic Review and Reproducibility and Validity Assessment, Empirical Software Engineering 25 (2020) 353–401. doi:10.1007/s10664-019-09747-0.

[19] T. Lewowski, L. Madeyski, How far are we from reproducible research on code smell detection? a systematic literature review, Information and Software Technology 144 (2022) 106783. doi:10.1016/j.infsof.2021.106783.

[20] T. Lewowski, L. Madeyski, Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review, Developments in Information & Knowledge Management for Business Applications : Volume 3, Springer International Publishing, Cham, 2022, pp. 285–319. doi:10.1007/978-3-030-77916-0_12.

[21] A. Al-Shaaby, H. Aljamaan, M. Alshayeb, Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review, Arabian Journal for Science and Engineering 45 (2020) 2341–2369. doi:10.1007/s13369-019-04311-w.

[22] M. I. Azeem, F. Palomba, L. Shi, Q. Wang, Machine learning techniques for code smell detection: A systematic literature review and meta-analysis, Information and Software Technology 108 (2019) 115 – 138. doi:10.1016/j.infsof.2018.12.009.

[23] G. Rasool, Z. Arshad, A review of code smell mining techniques, Journal of Software: Evolution and Process 27 (2015) 867–895. doi:10.1002/smr.1737.

[24] F. Caram, B. R. de Oliveira Rodrigues, A. Campanelli, F. Silva Parreiras, Machine learning techniques for code smells detection: A systematic mapping study, International Journal of Software Engineering and Knowledge Engineering 29 (2019) 285–316. doi:10.1142/S021819401950013X.

[25] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, M. G. de Mendonça, A systematic review on the code smell effect, Journal of Systems and Software 144 (2018) 450 – 477. doi:10.1016/j.jss.2018.07.035.

[26] E. V. d. P. Sobrinho, A. De Lucia, M. d. A. Maia, A systematic literature review on bad smells–5 w's: Which, when, what, who, where, IEEE Transactions on Software Engineering 47 (2021) 17–66. doi:10.1109/TSE.2018.2880977.

[27] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, A. De Lucia, Detecting code smells using machine learning techniques: Are we there yet?, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 612–621. doi:10.1109/SANER.2018.8330266.

[28] M. Mhawish, Predicting code smells and analysis of predictions: Using machine learning techniques and software metrics, Journal of Computer Science and Technology Vol. 35 (2020) 1428–1445. doi:10.1007/s11390-020-0323-7.

[29] F. Khomh, S. Vaucher, Y.-G. Guéehéeneuc, H. Sahraoui, A Bayesian Approach for the Detection of Code and Design Smells, in: Proceedings - International Conference on Quality Software, 2009, pp. 305–314. doi:10.1109/QSIC.2009.47.

[30] J. Yu, C. Mao, X. Ye, A novel tree-based neural network for android code smells detection, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), 2021, pp. 738–748. doi:10.1109/QRS54544.2021.00083.

[31] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, IEEE Transactions on Software Engineering 41 (2015) 462–489. doi:10.1109/TSE.2014.2372760.

[32] H. Liu, Z. Xu, Y. Zou, Deep learning based feature envy detection, in: ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 385–396. doi:10.1145/3238147.3238166.

[33] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, L. Zhang, Deep learning based code smell detection, IEEE Transactions on Software Engineering (2019). doi:10.1109/TSE.2019.2936376.

[34] M. Hadj-Kacem, N. Bouassida, Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder, in: Proceedings of the International Joint Conference on Neural Networks, volume 2019-July, 2019. doi:10.1109/IJCNN.2019.8851854.

[35] T. Lin, X. Fu, F. Chen, L. Li, A novel approach for code smells detection based on deep leaning, in: B. Chen, X. Huang (Eds.), Applied Cryptography in Computer and Communications, Applied Cryptography in Computer and Communications, Springer International Publishing, Cham, 2021, pp. 171–174.

[36] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, G. Sladić, Automatic detection of long method and god class code smells through neural source code embeddings, Expert Systems with Applications 204 (2022) 117607. doi:https://doi.org/10.1016/j.eswa.2022.117607.

[37] H. Liu, Q. Liu, Z. Niu, Y. Liu, Dynamic and automatic feedback-based threshold adaptation for code smell detection, IEEE Transactions on Software Engineering 42 (2016) 544–558. doi:10.1109/TSE.2015.2503740.

[38] Y. Guo, C. Seaman, N. Zazworka, F. Shull, Domain-specific tailoring of code smells: An empirical study, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, Association for Computing Machinery, New York, NY, USA, 2010, p. 167–170. doi:10.1145/1810295.1810321.

[39] F. Arcelli Fontana, V. Ferme, M. Zanoni, A. Yamashita, Automatic Metric Thresholds Derivation for Code Smell Detection, in: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics, 2015, pp. 44–53. doi:10.1109/WETSoM.2015.14.

[40] S. Boutaib, M. Elarbi, S. Bechikh, F. Palomba, L. B. Said, A possibilistic evolutionary approach to handle the uncertainty of software metrics thresholds in code smells detection, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), 2021, pp. 574–585. doi:10.1109/QRS54544.2021.00068.

[41] F. Pecorelli, F. Palomba, D. Di Nucci, A. De Lucia, Comparing heuristic and machine learning approaches for metric-based code smell detection, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 93–104. doi:10.1109/ICPC.2019.00023.

[42] N. Sae-Lim, S. Hayashi, M. Saeki, How do developers select and prioritize code smells? a preliminary study, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 484–488. doi:10.1109/ICSME.2017.66.

[43] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 101–110. doi:10.1109/ICSME.2014.32.

[44] T. Saika, E. Choi, N. Yoshida, S. Haruna, K. Inoue, Do developers focus on severe code smells?, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 4, 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 1–3. doi:10.1109/SANER.2016.117.

[45] M. Hozano, N. Antunes, B. Fonseca, E. Costa, Evaluating the Accuracy of Machine Learning Algorithms on Detecting Code Smells for Different Developers, in: Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2: ICEIS,, INSTICC, Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2: ICEIS, SciTePress, 2017, pp. 474–482. doi:10.5220/0006338804740482.

[46] F. Arcelli Fontana, M. Zanoni, Code smell severity classification using machine learning techniques, Knowledge-Based Systems 128 (2017) 43 – 58. URL: https://doi.org/10.1016/j.knosys.2017.04.014. doi:10.1016/j.knosys.2017.04.014.

[47] F. Pecorelli, F. Palomba, F. Khomh, A. De Lucia, Developer-driven code smell prioritization, Association for Computing Machinery, New York, NY, USA, 2020. URL: https://doi.org/10.1145/3379597.3387457. doi:10.1145/3379597.3387457.

[48] D. Taibi, A. Janes, V. Lenarduzzi, How developers perceive smells in source code: A replicated study, Information and Software Technology 92 (2017) 223–235. URL: https://www.sciencedirect.com/science/article/pii/S0950584916304128. doi:https://doi.org/10.1016/j.infsof.2017.08.008.

[49] H. Grodzicka, A. Ziobrowski, Z. Łakomiak, M. Kawa, L. Madeyski, Code Smell Prediction Employing Machine Learning Meets Emerging Java Language Constructs, in: A. Poniszewska-Marańda, N. Kryvinska, S. Jarząbek, L. Madeyski (Eds.), Data-Centric Business and Applications: Towards Software Development (Volume 4), volume 40 of book series Lecture Notes on Data Engineering and Communications Technologies, Springer International Publishing, Cham, 2020, pp. 137–167. doi:10.1007/978-3-030-34706-2_8.

[50] T. Lewowski, L. Madeyski, Creating Evolving Project Data Sets in Software Engineering, in: S. Jarzabek, A. Poniszewska-Marańda, L. Madeyski (Eds.), Integrating Research and Practice in Software Engineering, volume 851 of *Studies in Computational Intelligence*, Springer, Cham, 2020, pp. 1–14. doi:10.1007/978-3-030-26574-8_1.

[51] C. Soomlek, J. van Rijn, M. Bonsangue, Automatic Human-Like Detection of Code Smells, Discovery Science, Springer International Publishing, Cham, 2021, pp. 19–28. doi:10.1007/978-3-030-88942-5_2.

[52] M. Gradišnik, T. Beranič, S. Karakatič, G. Mausaš, Adapting god class thresholds for software defect prediction: A case study, in: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2019, pp. 1537–1542. doi:10.23919/MIPRO.2019.8757009.

[53] B. Soltanifar, S. Akbarinasaji, B. Caglayan, A. B. Bener, A. Filiz, B. M. Kramer, Software analytics in practice: A defect prediction model using code smells, in: Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS '16, Proceedings of the 20th International Database Engineering & Applications Symposium, Association for Computing Machinery, New York, NY, USA, 2016, p. 148–155. doi:10.1145/2938503.2938553.

[54] G. Chandrashekar, F. Sahin, A survey on feature selection methods, Computers & Electrical Engineering 40 (2014) 16–28. doi:10.1016/j.compeleceng.2013.11.024, 40th-year commemorative issue.

[55] L. Madeyski, M. Jureczko, Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study, Software Quality Journal 23 (2015) 393–422. URL: https://doi.org/10.1007/s11219-014-9241-7. doi:10.1007/s11219-014-9241-7.

[56] S. Fu, B. Shen, Code bad smell detection through evolutionary data mining, in: 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2015, pp. 1–9. doi:10.1109/ESEM.2015.7321194.

[57] B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, Z. M. Jones, mlr: Machine learning in r, Journal of Machine Learning Research 17 (2016) 1–5. URL: https://jmlr.org/papers/v17/15-066.html.

[58] T. Hothorn, A. Zeileis, partykit: A modular toolkit for recursive partytioning in r, Journal of Machine Learning Research 16 (2015) 3905–3909. URL: http://jmlr.org/papers/v16/hothorn15a.html.

[59] S. Milborrow. Derived from mda:mars by T. Hastie and R. Tibshirani., earth: Multivariate Adaptive Regression Splines, 2011. URL: http://CRAN.R-project.org/package=earth, r package.

[60] F. Leisch, K. Hornik, B. D. Ripley, B. Narasimhan, mda: Mixture and Flexible Discriminant Analysis, 2020. URL: https://CRAN.R-project.org/package=mda, r package version 0.5-2.

[61] K. Hechenbichler, K. Schliep, Weighted k-nearest-neighbor techniques and ordinal classification, 2004. doi:10.5285/ubm/epub.1769.

[62] A. Karatzoglou, A. Smola, K. Hornik, A. Zeileis, kernlab – an S4 package for kernel methods in R, Journal of Statistical Software 11 (2004) 1–20. doi:10.18637/jss.v011.i09.

[63] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, F. Leisch, e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien, 2021. URL: https://CRAN.R-project.org/package=e1071, r package version 1.7-9.

[64] A. Liaw, M. Wiener, Classification and regression by randomforest, R News 2 (2002) 18–22. URL: https://CRAN.R-project.org/doc/Rnews/.

[65] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, A. Pohthong, Robust Statistical Methods for Empirical Software Engineering, Empirical Software Engineering 22 (2017) 579–630. URL: https://doi.org/10.1007/s10664-016-9437-5. doi:10.1007/s10664-016-9437-5.

[66] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of feature envy bad smells, in: 2007 IEEE International Conference on Software Maintenance, 2007, pp. 519–520. doi:10.1109/ICSM.2007.4362679.

[67] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Jdeodorant: identification and application of extract class refactorings, in: 2011 33rd International Conference on Software Engineering (ICSE), 2011, pp. 1037–1039. doi:10.1145/1985793.1985989.

[68] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, W. Oizumi, Jspirit: a flexible tool for the analysis of code smells, in: 2015 34th International Conference of the Chilean Computer Science Society (SCCC), 2015, pp. 1–6. doi:10.1109/SCCC.2015.7416572.

[69] SciTools, Understand by SciTools, https://www.scitools.com/, 2022. Accessed: 2022-04-13.

[70] A. Kaur, S. Jain, S. Goel, SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells, Neural Computing and Applications (2019). doi:10.1007/s00521-019-04175-z.

[71] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, 2018, pp. 482–482. doi:10.1145/3180155.3182532.

[72] D. Cruz, A. Santana, E. Figueiredo, Detecting bad smells with machine learning algorithms: An empirical study, in: Proceedings of the 3rd International Conference on Technical Debt, TechDebt '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 31–40. doi:10.1145/3387906.3388618.

[73] P. Piotrowski, L. Madeyski, Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review, Data-Centric Business and Applications: Towards Software Development (Volume 4), Springer International Publishing, Cham, 2020, pp. 77–99. doi:10.1007/978-3-030-34706-2_5.

[74] D. Bán, R. Ferenc, Recognizing antipatterns and analyzing their effects on software maintainability, in: Computational Science and Its Applications – ICCSA 2014, Computational Science and Its Applications – ICCSA 2014, Cham, 2014, pp. 337–352. doi:10.1007/978-3-319-09156-3_25.

[75] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wessln, Experimentation in Software Engineering, Springer Publishing Company, Incorporated, 2012.

[76] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 101–110. doi:10.1109/ICSME.2014.32.

Appendix A.

This appendix includes ranking tables Tables A.4 to A.6 useful for selecting the best overall machine learning algorithm for code smell detection. Detailed results, data tables, source code, and data sheet are available in an external appendix. The appendixcan be downloaded from Zenodo at `https://doi.org/10.5281/zenodo.7319860`

Smell	DS	CTree	FDA	KNN	kSVM	libSVM	MDA	Naïve Bayes	Random Forest
Blob	DS1	5	6	1	3	4	0	2	7
Blob	DS2	3	7	4	0	2	6	1	5
Data Class	DS1	5	6	2	3	4	1	0	7
Data Class	DS2	4	6	1	2	5	3	0	7
Long Method	DS1	4	5	0	3	1	2	6	7
Long Method	DS2	4	6	5	3	1	2	0	7
Feature Envy	DS1	6	7	2	0	3	4	1	5
Total	-	31	43	15	14	20	18	10	45

Table A.4: Ranking of machine algorithms for each of code smell data sets (based on median MCC) - higher is better

Smell	DS	CTree	FDA	KNN	kSVM	libSVM	MDA	Naïve Bayes	Random Forest
Blob	DS1	5	6	1	4	3	0	2	7
Blob	DS2	3	5	4	0	2	7	1	6
Data Class	DS1	5	6	2	4	3	1	0	7
Data Class	DS2	5	6	1	3	4	2	0	7
Long Method	DS1	3	5	1	2	0	4	6	7
Long Method	DS2	4	6	5	3	1	2	0	7
Feature Envy	DS1	4	7	2	1	3	5	0	6
Total	-	29	41	16	17	16	21	9	47

Table A.5: Ranking of machine algorithms for each of code smell data sets (based on median F-score) - higher is better

Smell	DS	CTree	FDA	KNN	kSVM	libSVM	MDA	Naïve Bayes	Random Forest	**Swaps**
Blob	DS1	0	0	0	-1	1	0	0	0	1
Blob	DS2	0	2	0	0	0	-1	0	-1	2
Data Class	DS1	0	0	0	-1	1	0	0	0	1
Data Class	DS2	-1	0	0	-1	1	1	0	0	2
Long Method	DS1	1	0	-1	1	1	-2	0	0	3
Long Method	DS2	0	0	0	0	0	0	0	0	0
Feature Envy	DS1	2	0	0	-1	0	-1	1	-1	3
Total	-	2	2	-1	-3	4	-3	1	-2	-

Table A.6: Differences in ranking of machine algorithms for each of code smell data sets between ranking based on MCC and ranking based on F-score

Name	Description
ABC_SIZE	Assignment Branch Condition size
ARITY	Number of method arguments
BLOCK_NESTING	Maximum block nesting
CYCLO	Cyclomatic complexity
LINES_OF_CODE	Number of lines of code, excluding comments and whitespaces
FUNCTIONS	Total number of methods in a class
INSTANCE_VARIABLES	Number of fields in a class
METHODS	Number of non-static methods in a class
CHILDREN	Number of classes that extend/implement class/interface
DEPTH	Total inheritance depth
USES	Number of references to the class
USED_BY	Number of other classes that reference the class
TREE_IMPURITY	Density of inter-class connections
INSTABILITY	$\frac{USED}{USED+USED\_BY}$
ATTRIBUTES	Number of non-static fields
ACCESSORS	Number of getters
MUTATORS	Number of setters
NOT_ACCESSORS_OR_MUTATORS	$METHODS - ACCESSORS - MUTATORS$
METHODS_WEIGHTED	Sum of CYCLO values for all non-static methods
NOT_ACCESSORS_OR_MUTATORS_WEIGHTED	Sum of CYCLO values for non-static methods that are neither getters nor setters
AVERAGE_METHODS_COMPLEXITY	Average CYCLO value for non-static methods
AVERAGE_NOT_ACCESSOR_OR_MUTATOR_METHOD_COMPLEXITY	Average CYCLO value for non-static methods that are neither getters nor setters

Table A.7: Description of metrics provided by CODEBEAT. Metrics are calculated from a single codebase.

Name	Description
AMW	Average method weight
ATFD	Access to foreign data
CBO	Coupling between objects
CYCLO	Cyclomatic complexity
DIT	Depth of inheritance tree
FANOUT	Number of called classes
LCOM5	Lack of cohesion in methods
LOC	Lines of code
LOCNAMM	Lines of code without accessors or mutators
NCSS	Non-commenting source statements
NLV	Number of local variables
NMO	Number of methods overridden
NOA	Number of attributes
NOAM	Number of accessort methods
NOC	Number of children
NOCM	Number of constructor methods
NOFA	Number of final attributes
NOM	Number of methods
NOMNAMM	Number of nont accessor or mutator methods
NONFNSA	Number of non-final and non-static attributes
NONFNSM	Number of final and non-static methods
NOP	Number of parameters
NOPA	Number of public attributes
NOPM	Number of private methods
NOPRA	Number of protected attributes
NOPVA	Number of private attributes
NOS	Number of statements
NPATH	n-path complexity
TCC	Tight class cohesion
WMC	Weighted methods count
WMCNAMM	Weighted methods count of non accessor or mutator methods
MOC	Weight of class

Table A.8: Description of metrics provided by PMD