Knowledge-Based Systems 128 (2017) 43–58

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Code smell severity classification using machine learning techniques

Francesca Arcelli Fontana, Marco Zanoni*

Università degli Studi di Milano-Bicocca, Milan, Italy

ARTICLE INFO

Article history:
Received 8 August 2016
Revised 6 April 2017
Accepted 24 April 2017
Available online 25 April 2017

Keywords:
Code smells detection
Machine learning
Code smell severity
Ordinal classification
Refactoring prioritization

ABSTRACT

Several code smells detection tools have been developed providing different results, because smells can be subjectively interpreted and hence detected in different ways. Machine learning techniques have been used for different topics in software engineering, e.g., design pattern detection, code smell detection, bug prediction, recommending systems. In this paper, we focus our attention on the classification of code smell severity through the use of machine learning techniques in different experiments. The severity of code smells is an important factor to take into consideration when reporting code smell detection results, since it allows the prioritization of refactoring efforts. In fact, code smells with high severity can be particularly large and complex, and create larger issues to the maintainability of software a system. In our experiments, we apply several machine learning models, spanning from multinomial classification to regression, plus a method to apply binary classifiers for ordinal classification. In fact, we model code smell severity as an ordinal variable. We take the baseline models from previous work, where we applied binary classification models for code smell detection with good results. We report and compare the performance of the models according to their accuracy and four different performance measures used for the evaluation of ordinal classification techniques. From our results, while the accuracy of the classification of severity is not high as in the binary classification of absence or presence of code smells, the ranking correlation of the actual and predicted severity for the best models reaches 0.88–0.96, measured through Spearman's ρ.

1. Introduction

Different tools for code smells [1] detection have been developed, which exploit different techniques, usually metric-based techniques [2]. The results provided by the tools are usually different, as we observed in a previous paper [3], where we outline how different detectors for the same smell do not meaningfully agree in their answers. We found the best agreement in the results for the detection of three code smells: God Class, Large Class and Long Parameter List. Moreover, we outlined another problem regarding the reliability of the results of the detectors, in particular related to the recall values. We found also a high number of false positive results: smells that are not real smells to be removed, and that can be considered in some cases as domain or design dependent smells [4].

Starting from these observations, on the low agreement on the results provided by different code smells detectors, and on the high number of false positive smells detected by a tool, we started working on code smells detection by exploiting machine learning techniques [5,6]. We followed this direction because detection approaches based on machine learning techniques have not been largely explored before and we had a previous large experience in exploiting machine learning techniques also for design pattern detection [7]. Most approaches for code smell detection are based on fixed rules exploiting metric thresholds. A tool exploiting a different technique is JDeodorant [8], which is able to detect four smells through the opportunities of applying refactoring steps to remove them.

Another aspect of code smell detection that did not receive large attention in the literature is the fact that different code smell instances can have different size, intensity, or *severity*, and therefore they shall be managed in different ways, since their effect on software quality is different. In this paper, we briefly describe the main steps of our machine learning-based approach, and then we focus our attention on the possibility to classify not only the presence or absence of code smells, but also their severity. Each code smell has specific characteristics that make it detrimental for software quality. We define severity as an estimation of the amount of these characteristics in a code smell instance. For example, a God Class has high severity if it is very large and complex, and/or centralizes a large part of the intelligence of the system.

An approach based on machine learning has the advantage of being able to exploit any code characteristic a developer consid-

* Corresponding author.
E-mail addresses: arcelli@disco.unimib.it (F. Arcelli Fontana), marco.zanoni@disco.unimib.it (M. Zanoni).

http://dx.doi.org/10.1016/j.knosys.2017.04.014
0950-7051/

ers important to define the severity of a code smell, as far as this characteristic can be estimated through one or more software metrics. Moreover, the developer does not need to formalize its definition of severity, since it will be synthesized by the learning models. This makes the method highly flexible and independent from explicit classification rules and metric thresholds. Therefore, the learning by example approach made possible through this method allows for different scenarios, e.g., where a common and highly agreed benchmark dataset is shared by many experts, creating a common ground for both learning and evaluation of code smell severity classification models, or where organizations can target their severity definition through the use of example instances, without the need to explicitly formalize severity classification rules.

Because we define severity using an ordinal variable with 4 values, we approach our experiment as an ordinal classification task. We propose an experiment where we apply 11 learning models, in multiple setups (a total of 177), for the classification of the severity of four code smells (Data Class, God Class, Feature Envy, Long Method). We selected a first group of models from a set that proved to provide good performances in the binary classification (presence/absence) of code smells [6]. We combine them with an ordinal classification method [9] to exploit the ordinal characteristics of severity. We test regression models and Fuzzy Rule Learners, and compare them with the best classifiers. This set of models is experimented on a dataset composed of 420 data points (classes or methods) for each code smell, sampled from 76 open source Java projects.

Our aim in this paper is to assess the performances of code smells severity classification, and select the best models to carry out this task. Good performances in the ranking of code smell detection results are important to provide developers reliable information, allowing them to prioritize refactoring and re-engineering efforts, e.g., recommending them to fix only the smells with highest severity. We explore the performances of learning models for this task through four research questions, defined in Section 4.

The paper is organized through the following sections: in Section 2, we introduce some related work; in Section 3, we briefly describe our code smell detection approach based on machine learning techniques; in Section 4, we explain code smells severity and its classification, and define research questions for this work; in Section 5, we describe the experimental method we followed, and describe the learning models and performance evaluation measures we applied; in Section 6, we describe our results, answering research questions; in Section 7, we outline the threats to the validity of our work; finally, in Section 8, we provide some concluding remarks and future investigations.

2. Related work

Many tools for code smells detection have been proposed, both commercial tools and research prototypes. Tools exploit different techniques to detect code smells: some techniques are metrics-based [10,11], while others use a dedicated specification language [12], use program analysis to identify refactoring opportunities [8,13], are based on the analysis of software repositories [14] or use machine learning techniques [6]. In this section, we consider only tools or approaches which exploit machine learning techniques and the approaches that introduce some kind of code smell Intensity or Severity index.

2.1. Machine learning approaches for code smells detection

In the literature we found, other than our approach, only few works exploiting machine learning techniques for code smell detection.

Maiga et al. [15] introduce SVMDetect, an approach to detect anti-patterns, based on support vector machines. The subjects of their study are Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife antipatterns, on three open-source programs: ArgoUML, Azureus, and Xerces. Khomh et al. [16] present BDTEX (Bayesian Detection Expert), a Goal Question Metric approach to build Bayesian Belief Networks from the definitions of antipatterns; they validate BDTEX with Blob, Functional Decomposition, and Spaghetti Code antipatterns on two open-source programs.

Maiga et al. [17] introduce SMURF, an approach to detect antipatterns, based on a machine learning technique using support vector machines and taking into account practitioners' feedback. Yang et al. [18] study the judgment of individual users by applying machine learning algorithms on code clones.

Maneerat and Muenchaisri [19] collect datasets from the literature regarding the evaluation of 7 bad-smells, and apply 7 machine learning algorithms for bad-smells prediction, using 27 design model metrics extracted by a tool as independent variables. The author make no explicit reference to the applied datasets.

As we can see, the principal differences of these works respect to our approach is that most of the above papers are principally focused on the detection of antipatterns and not of code smells as defined by Fowler [1]. The experiments they performed considered only 2 or 3 systems and they usually experiment only one machine learning algorithm. In one case [19], the number of systems in not known, while 7 smells have been investigated. In our approach, we focus our attention on 4 code smells, we consider 76 systems for the analysis and our validation and we experiment 11 different machine learning algorithms in different setups, actually leading to the evaluation of 177 learning models.

2.2. Severity of code smells

With respect to the severity classification, a similar idea has been applied for software defects and its industrial adaption is widespread. Several authors worked in predicting defect severity with machine learning techniques [20–22]. For what concerns code smells severity, we found the following approaches.

Vidal et al. [23] developed a tool that suggests a ranking of code smells, based on a combination of three criteria: past component modifications (its stability, if many changes occurred over the history of the application), important modifiability scenarios for the system, and relevance of the kind of smell. The relevance is a subjective value that a developer can assign to each kind of smell to indicate how harmful he considers it. This value might vary from developer to developer, or might also be system-specific. They propose a semi-automated approach, called smart identification of refactoring opportunities (SpIRIT) that prioritizes the code smells of a system according to their criticality. The approach has been evaluated in two case-studies, and they assert that the obtained results are useful to developers.

Vidal et al. [24] extend the previous work by proposing criteria for prioritizing groups of code anomalies as indicators of architectural problems in evolving systems. They analyzed several versions of 2 applications. The prioritization is based on two scoring criteria that have the goal of ranking in higher positions the agglomerations that likely indicate certain types of architectural problems.

Lanza and Marinescu [10] defined different metric-based rules to detect code smells, or identify code problems called disharmonies. The disharmonies are defined based on a combination of different metrics that have to exceed a predetermined threshold. They propose a naive approach to prioritize the disharmonies, in which the classes (or methods) that present a high number of disharmonies are considered more critical.

Marinescu [25] proposes a Flaw Impact Score to measure the criticality of code smells and rank them. The author proposed to

measure the impact of code smells considering three factors: *(i)* the negative influence of a kind of code smell on a good design (according to low coupling, high cohesion, moderate complexity and proper encapsulation) using a three value nominal scale (low, medium, high); *(ii)* granularity, the kind of entity that the smell affects (method or a class); and *(iii)* severity, based on the most critical symptoms of the smell measured by one or more metrics.

Arcelli Fontana et al. [11] propose a code smell Intensity Index computed considering the values of all the metrics used in the detection strategy of the smell, giving the same weight to all the metrics involved. Hence, to compute the Intensity index, they take into account all the features used as hints of the presence of a smell. This index has been also exploited to evaluate a smell-aware combined bug-prediction model [26].

The Stench Blossom [27] code smell detection tool provides severity as a visual effect on a petal corresponding to a code smell: increased severity corresponds to increased petal size. It uses the count of the number of detection rule violations to determine the length of the petal; each rule violation is given equal weight. This is different from our Intensity index because we compute a single measure aggregating the value of the detection rule violations compared with a reference distribution.

Mkaouer et al. [28], with the aim to suggest refactoring solutions, propose a code smell Severity, that is a level assigned to a code smell by a developer. This assessment can change over time. They also define the code smell Importance of a class that contains a code smell, related to the number and size of the features that the class supports. Also this property can vary over time as classes are added/deleted/split.

In another direction, Zhao and Hayes [29] propose a hierarchical approach to identify and prioritize refactorings, that are ranked based on predicted improvement to the maintainability of the software. They analyzed two systems, but they do not define an Intensity Index for code smells.

With respect to previous work, our approach is new, since we apply machine learning techniques to predict code smell severity. According to our knowledge, this direction has not been explored in the literature yet. As we already introduced, this machine learning approach allows the definition of severity through examples, allowing different organizations or maintainers to customize the definition according to their needs.

3. Machine learning-based code smell detection

In this paper, we extend the code smell detection approach based on machine learning that we defined in a previous work [6]. In the following, we summarize the approach we followed to apply and evaluate machine learning models for the detection of code smells, our training data collection process and the obtained dataset.

3.1. Data modeling

We model code smell detection as a supervised learning task. We exploit a large set of software metrics as independent variables, and the presence of a code smell as the dependent variable. We recall that in the previous work the dependent variable is binary, representing absence or presence of a code smell, while in this paper it represents the severity of a code smell, described in the next section. The subjects of the classification are classes or methods, depending on the kind of code smell to be identified (code smells are defined at different levels of granularity). Therefore, our datasets are composed of one feature vector for each class or method, where each vector contains the software metrics associated to the respective subject, plus the class variable.

Software metrics are defined at different levels of granularity, too. The metrics we consider can be associated to packages, types (classes or interfaces) and methods. Since these different granularity levels form a containment relation (package contains type contains method), each feature vector contains the metrics associated to the respective class or method, plus the metrics associated to all its containers. This choice allows the representation of an element into its context, e.g., a method with a large number of lines of code may be evaluated differently if contained in a much larger class or in a class only slightly larger.

3.2. Data collection

For our analysis, we considered the Qualitas Corpus of systems collected by Tempero et al. [30]. The corpus, and in particular the version we use, 20120401r, is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. We selected 76 systems of different size and computed a large set of object-oriented metrics, to be used as independent variables. To be able to correctly compute the metric values, most of the 76 systems were completed (e.g., adding missing libraries) in order to make them compile. We were not able to do this operation for all the systems of the Qualitas Corpus. Moreover, we decided to remove systems of small size. The metrics we selected are reported in Appendix A in Tables A.9 and A.10.

As our dependent variables, we choose code smells with high frequency [31], that may have the greatest negative impact [32] on the software quality, and which can be recognized by some available detection tools [3]. As outlined above, code smells may be defined at the level of class or method. At class level, we decided to detect God Class [10] and Data Class [1], while at method level, we detect Long Method [1] and Feature Envy [1]. In our definition, on which our learning oracle definition is based, we include Large Class [1] into the definition of God Class, and Brain Method and God Method [10] into the definition of Long Method. Since we merged definition of different smells, and for clarity, we report in the following the definition we followed for the smells:

- *God Class*: The God Class code smell refers to classes that tend to centralize the intelligence of the system. A God Class tends to be complex, to have too much code, to use large amounts of data from other classes and to implement several different functionalities.
- *Data Class*: The Data Class code smell refers to classes that store data without providing complex functionality, and having other classes strongly relying on them. A Data Class reveals many attributes, it is not complex, and exposes data through accessor methods.
- *Long Method*: The Long Method code smell refers to methods that tend to centralize the functionality of a class. A Long Method tends to have too much code, to be complex, to be difficult to understand and to use large amounts of data from other classes.
- *Feature Envy*: The Feature Envy code smell refers to methods that use much more data from other classes than from their own class. A Feature Envy tends to use many attributes of other classes, considering also attributes accessed through accessor methods.

Once we identify the systems to be used for the code smell evaluation, and the code smells to be evaluated, we need to build a dataset to be used for training and validation of machine learning models. We know from previous experience [6] that code smell density is usually low (5–8% of classes, 1–4% of methods, considering the same code smells experimented in this paper), so random sampling the classes and methods of the 76 systems is not a viable solution. For this reason, we helped our sampling through

Table 1
Advisors.

Code smell	Reference tool/Detection rules
God Class	iPlasma (God Class, Brain Class) [33], PMD[1]
Data Class	iPlasma, Fluid Tool [34], Anti-Pattern Scanner [32]
Long Method	iPlasma (Brain Method), PMD, Marinescu detection rule [35]
Feature Envy	iPlasma, Fluid Tool

Advisors. For each code smell, we identified in the literature a set of available detection tools and rules that are able to detect it; we call them Advisors. We selected as many heterogeneous Advisors as possible, favoring the detection rules implemented by tools, because they should be more reliable and have a larger user base. The Advisors selected for each code smell are reported in Table 1.

We apply the selected Advisors on the 76 systems, recording the results for each class and method. The detection output has been used as a hint to select a class (or method) to be manually evaluated, producing a label for each class (or method). In particular, the number of Advisors reporting a class/method as affected by a smell has been used as a nominal class label to perform a stratified sampling (without repetition) over the dataset.

Following the output of Advisors, the reported code smell candidates are manually evaluated. The labelling process is supported also by graphical code representations, like dependencies, call, or hierarchy graphs. This information is used to support the decision of which label to assign. The values of software metrics are not available during evaluation, nor the number of Advisors suggesting an instance, to avoid biases. Three MsC students performed the labelling process, after being trained both theorically and practically for the task. The three raters performed the labelling independently, and in case their evaluation were not all equal for a single instance, they reached an agreement through discussion.

Finally, the datasets have been normalized, setting their size to 420 instances and their balance between 33% and 66% positive (affected by code smells) instances.

4. Code smells severity classification

During the manual evaluation of code smells, a severity value is assigned to each evaluated instance. Our code smell severity classification can take one among four possible values:

0 *No smell*: the class (or method) is not affected by the smell;
1 *Non-severe smell*: the class (or method) is only partially affected by the smell;
2 *Smell*: the smell characteristics are all present in the class (or method);
3 *Severe smell*: the smell is present, and has particularly high values of size, complexity or coupling.

The code smell severity classification can have two possible benefits. First, ranking of classes (or methods) by the severity of their code smells can aid software developers in prioritizing their work on the most severe code smells. Often, developers have to work under time constraints and they might not have time to fix all the code smells automatically detected. Second, the usage of the code smell severity classification for the labelling provides more information than a more traditional binary classification, as for machine learning purposes the code smell severity classification will be interpreted as an ordinal variable.

In previous work [6], we collapsed this information back to a binary classification by grouping together the values, i.e., {0} → *INCORRECT* (code smell absence), {1, 2, 3} → *CORRECT* (code smell presence). Using this setting, we obtained very good detection performances (96–99% Accuracy, 97–99% F-Measure) with different learning models.

In this paper, we experiment with machine learning techniques to classify the severity of code smells, exploiting all the information available.

To this end, we have to reconfigure our learning approach to address the new goal. In fact, our previous approach dealt with binary classification (presence or absence of a code smell), while in this case the prediction targets an ordinal value (the code smell severity). For this reason, we perform an ordinal classification (or regression) [36] experiment. This task can be addressed with different strategies:

- *Multinomial classification*: this is achieved by considering the ordinal variable as a nominal variable, where the different values are independent; this approach is straightforward, and can be applied to most learning models, since the available implementations allow both binary and multinomial classification;
- *Ordinal meta-learning*: some methods [9,37] have been defined to exploit binary classifiers in an ordinal classification task; this approach seeks to enhance the previous one, by exploiting the ordering information contained in the variable;
- *Regression*: since ordinal values can be represented as numbers, it is possible to apply regression models to fit the ordinal variable; this approach is sensitive to the numerical representation of the ordinal value, and may suffer from a wrong representation choice.

While the high-level goal of our experiment is to assess to what extent machine learning models can classify code smell severity, we identify a set of more specific research questions we want to address:

RQ1 How do the best learning models identified in our previous work perform for this task?
RQ2 Does the application of a technique for ordinal meta-learning enhance the detection performances?
RQ3 What performance can be achieved by regression models?
RQ4 What is the best learning model experimented for severity classification?

5. Experiment definition

In this section, we define the process we applied to prepare the data for our analysis, and to select, perform and evaluate supervised learning models.

5.1. Preprocessing

Each of the four datasets is composed of 420 instances (classes or methods), associated to a feature vector consisting of 63 values for God Class and Data Class and of 84 values for Long Method and Feature Envy. An additional feature contains the severity value of the smell.

Each dataset is then processed by a feature selection procedure, common to all learning models:

1. Features are normalized in the range 0–1 using their minimum and maximum values;
2. Features with variance lower than 0.01 are excluded from the feature set;
3. Linear correlation is computed across all pairs of features, and features with correlation higher than 0.8 are selected as candidates to be dropped; among the candidates, only features that are correlated (above the threshold) with the lowest number of other features are dropped; this behaviour is embedded in the "Correlation Filter" node available in KNIME.

[1] https://pmd.github.io/.

5.2. Machine learning models selection

In this experiment, we apply a first set of models that had the best performances in our previous work [6] about binary classification of code smells:

- J48 is an implementation of the C4.5 decision tree. This algorithm produces human understandable rules for the classification of new instances. The Weka implementation offers three different approaches to compute the decision tree, based on the types of pruning techniques: pruned, unpruned and reduced error pruning. In the experimentation, all the available pruning techniques were used, and reported as separated classifiers.
- JRip is an implementation of a propositional rule learner. It is able to produce simple propositional logic rules for the classification, which are understandable to humans and can be simply translated into logic programming.
- Random Forest is a classifier that builds many classification trees as a forest of random decision trees, each one using a specific subset of the input features. Each tree gives a classification (vote) and the algorithm chooses the classification having most votes among all the trees in the forest.
- Naïve Bayes is the simplest probabilistic classifier based on applying Bayes' theorem. It makes strong assumptions on the input: the features are considered conditionally independent among each other.
- SMO is an implementation of John Platt's sequential minimal optimization algorithm to train a support vector classifier. In the experimentation, RBF (Radial Basis Function) kernel and Polynomial kernel were used in combination with this classifier.
- LibSVM is an integrated software for support vector classification. It is available for Weka by using an external adapter. Two support vector machine algorithms are experimented (C-SVC, ν-SVC), in combination with four different kernels (Linear, Polynomial, RBF, Sigmoid).

The choice of the algorithms was made by selecting different algorithms representing different approaches. All the models have been also applied in combination with the AdaBoost boosting method. Models implementations are the ones from Weka [38]. These models are applied in this work using the best parameter settings for each dataset found in our previous work, on the same dataset, but for binary classification purposes. This is the initial set of models experimented to answer RQ1.

We extend the first set of models with other classification models:

- Fuzzy Rule Learner: this model learns classification rules using a method defined by Gabriel and Berthold [39,40]. We applied the implementation of the model available in KNIME[2]. We selected this model because it uses techniques very different from the ones we already experimented.
- Decision Tree Learner: this is an alternative (to Weka J48) decision tree that combines C4.5 with techniques defined by Shafer et al. [41], available in KNIME. We added this model because decision trees already proved to have good performances, and we wanted to understand if an alternative implementation of similar concepts could reach similar results.

We will apply this extended set of models to answer RQ4.

All the classification models have been experimented also in combination with the ordinal classification method defined by Frank and Hall [9]. RQ2 explicitly targets the effectiveness of this method on our datasets and the set of models applied in RQ1. RQ4 will be answered also considering models enhanced by this method. This method allows applying any binary classifier able to provide estimation probabilities for each predicted class and exploit it for ordinal classification. This is achieved by building synthetic binary class variables derived from the ordinal variable. For example, considering our 4-level severity variable, and mapping them to integer values from 1 to 4, three binary class variables will be defined using this criterion:

1. $\{1\} \rightarrow$ *false*, $\{2, 3, 4\} \rightarrow$ *true*;
2. $\{1, 2\} \rightarrow$ *false*, $\{3, 4\} \rightarrow$ *true*;
3. $\{1, 2, 3\} \rightarrow$ *false*, $\{4\} \rightarrow$ *true*.

These three variables are independently used for model training, and the probabilities assigned to each variable value are combined to decide the original value having the highest probability according to the model. In our experiment, we used the Weka implementation of this method when applying Weka models, and a custom-built implementation when applying models implemented in KNIME.

Finally, we experiment different regression models, all in the implementations provided by KNIME, to answer RQ3:

- Linear Regression;
- Polynomial Regression, with maximum degree = 3;
- Random Forest Regression: applies Random Forests for regression.

To make the application of these models possible, severity has to be represented as a number, and we mapped the four severity values to integers from 1 to 4.

5.3. Models performance evaluation

The performance evaluation experiment has been carried out using repeated 10-fold cross validation, using 10 repetitions. Each repetition of cross validation uses a different dataset randomization, and performs stratified sampling of the dataset for the creation of folds. Therefore, we have $10^*10 = 100$ different evaluations of each model on different subsets of the same dataset.

Evaluation of learning performances in ordinal classification is different than in binary or multinomial classification. In ordinal classification, accuracy is important but also the ordering of results is important. According to the literature [36,37], different performance measures can be applied specifically to this context:

- Kendall's τ_b and Spearman's ρ rank-order correlation coefficients are used as measures of how good prediction results are ordered w.r.t. the original data;
- mean absolute deviation (MAD, also called mean absolute error) and mean square error (MSE): these measures are based on the association to each severity value to a number (we used integers from 1 to 4, like for regression models above), and quantify the prediction error as the difference between the actual value and the predicted one; MAD is the mean of the absolute value of errors, MSE is the mean of their square.

The first two measures are particularly suited to the evaluation of the ranking produced by ordinal classification models, but lack some readability about the learning error, which is the focus of the last two measures. Therefore, the two pairs of measures are complementary. In our experiment, we apply all these measures, and consider also accuracy, applying it as in multinomial classification.

6. Results

In this section, we report the outcomes of our experiments, organized to address the research questions we defined.

[2] http://www.knime.org.

Table 2
Dataset composition.

Code smell	Severity			
	1	2	3	4
God Class	154	29	110	127
Data Class	151	32	113	124
Long Method	280	11	95	34
Feature Envy	280	23	95	22

Table 3
Feature selection results.

Code smell	Original features	Variance filter	Correlation filter
God Class	63	36	27
Data Class	63	34	26
Long Method	84	59	37
Feature Envy	84	56	34

6.1. Dataset composition and preprocessing

As it was already stated, we performed our experiments on four datasets, one for each considered code smell. Each dataset is composed of 420 manually evaluated classes or methods. In Table 2, we report the composition of our datasets, with the number of instances assigned to each severity level. The less frequent severity level in the datasets is 2, and the two class-based smells have a different balance than the two method-based smells w.r.t. severity levels 1 and 4. This makes positive smell instances more prevalent in the first two datasets than in the other two.

We applied two feature selection steps on the dataset, as explained in Section 5: a low variance filter and a high linear correlation filter. As we can see from Table 3, both steps significantly reduced the number of features, reaching less than a half of the original features in all datasets.

6.2. Binary classification models applied to multinomial class

Our first experiment addresses the use of the best models found in our previous experiment regarding binary classification [6]. Table 4 reports the performance measures of the best 5 models of each dataset for multinomial classification. The best performance values for each dataset are reported in bold font (we will use this convention in next similar tables). Models are sorted by descending values of Spearman's ρ, which is used also to select the top 5 models; the full performance report tables can be found in Appendix B. Performance measures are averaged over the 10 repetitions, and are reported with the respective standard deviations (σ). Since we applied models both with and without boosting, we prefix boosted models with "B-".

The best results report 0.75–0.76 accuracy for class-based smells and 0.92–0.93 for method-based smells, with low deviations, $\leq$0.01 in most cases. The top 5 models are always represented by some variant of decision trees or random forests, plus the boosted version of the JRip rule extraction algorithm. The best classifiers in this experiment have extremely similar performance values, meaning that no one is clearly better than the other. This result is aligned with our previous experience in binary classification.

6.2.0.1. Answer to RQ1. In RQ1, we ask: *How do the best learning models identified in our previous work perform for this task?* We applied the same models used in our previous work [6], in a multinomial classification setting. Accuracy values are lower on this class variable: on the binary class variable the classifiers reached 0.96–0.99 Accuracy, while for the classification of severity values span from 0.75 (for God Class and Data Class) to 0.93 (Long Method and Feature Envy). A similarity between the results obtained through binary and multinomial classification is in the models that performed best for each dataset. In fact, decision trees and Random Forests always obtain the best performances. In summary, the answer to this question is that the models are not as accurate as in binary classification, but are able to obtain good accuracy at least on method-based smells. Anyway, considering performance measures more appropriate for ordinal classification, we can see that the expected error is proportionally smaller. For example, MAD for B-JRip on God Class is 0.3, and accuracy is 0.75. This means that while 1/4 predictions are wrong, most errors have an absolute value of 1 (considering 20 predictions, with five errors, four errors must have absolute value 1 and one 2 to obtain these scores). This is reflected also in the value of ρ, reporting 0.85.

6.3. Ordinal classification through binary classifiers

Since multinomial classifiers are not built to understand the ordering of values in an ordinal variable, we applied an existing method, defined by Frank and Hall [9], to apply binary classifiers to an ordinal variable.

We executed the same classifiers applied in the previous section, combined with the ordinal classification method. This can easily achieved in Weka, since it provides a meta-classifier for this. The classifiers have been executed in the same setup, and evaluated through the same procedure, i.e., 10 repetitions of 10-fold cross validation. Throughout all the experiments preformed in this paper, the random seeds have been fixed for each repetition, so every cross-validation fold is exactly equal to the others, allowing us to properly compare the results of each fold in each repetition one another.

To compare the performances of ordinal and multinomial classifiers, we applied the Wilcoxon signed rank test, with the alternative hypothesis set to "greater", i.e., we are interested in knowing if the ordinal method generated an improvement in the classifier's performances. More formally, we have:

- H_0: the distribution of performances of the two compared models are equal, i.e., the median of their difference is 0;
- H_1: the distribution of performances of the ordinal-enhanced classifier is higher than the non-ordinal one, i.e., the median of their difference is $>$0.

We pair the tests results with Cliff's δ effect size measure, to assess the amount of improvement. The performance measure we choose to use to compare the results is Spearman's ρ. Therefore, all tests are applied to 100 (10 repetitions $*$ 10 folds) values of ρ on each side (ordinal vs multinomial) for each model and dataset.

Table 5 shows the results of the tests we performed (p-values and effect size magnitude), also reporting the mean ρ value of the ordinal version of the algorithm (the respective multinomial classification value can be found in Tables B.11–B.14). As we can see, the ordinal method is more effective on the God Class and Data Class smells than on the other two. Considering a significance level $\alpha = 0.05$, we can see that only the first two models of the list achieved an improvement by applying the ordinal method, while for the other models the test is not significant on all datasets. As for effect sizes, we can see that the better performing models received a lower improvement than the others. In fact, large effect sizes are found only on the lower performing models (e.g., LibSVM on different dataset, JRip on Data Class). The only model with good performances in the multinomial classification experiment achieving a medium effect size is B-J48-Pruned on Data Class.

6.3.0.2. Answer to RQ2. In RQ2, we ask: *Does the application of a technique for ordinal meta-learning enhance the detection perfor-*

Table 4
Existing model performances on multinomial classification - Top 5.

DS	Model	ρ	(σ)	τ_b	(σ)	MAD	(σ)	MSE	(σ)	Acc.	(σ)
GC	B-JRip	**0.85**	0.01	0.61	0.01	**0.30**	0.02	**0.43**	0.04	0.75	0.01
	B-J48-Unpruned	**0.85**	0.01	0.61	0.01	0.31	0.02	0.46	0.03	0.75	0.02
	J48-Pruned	**0.85**	0.01	**0.62**	0.01	0.31	0.02	0.46	0.05	**0.76**	0.01
	B-RandomForest	**0.85**	0.01	0.61	0.01	0.31	0.01	**0.43**	0.03	0.74	0.01
	B-J48-Pruned	0.84	0.01	0.61	0.01	0.32	0.01	0.48	0.02	0.75	0.01
DC	B-RandomForest	**0.86**	0.01	**0.62**	0.01	**0.29**	0.01	**0.39**	0.03	0.76	0.01
	J48-Pruned	**0.86**	0.01	**0.62**	0.01	0.30	0.02	0.43	0.03	0.76	0.01
	B-JRip	0.85	0.01	**0.62**	0.01	**0.29**	0.02	0.41	0.04	0.76	0.01
	RandomForest	0.85	0.01	0.61	0.01	**0.29**	0.02	0.43	0.02	**0.77**	0.01
	J48-Unpruned	0.84	0.01	0.61	0.01	0.33	0.02	0.47	0.04	0.73	0.02
LM	B-RandomForest	**0.96**	0.01	0.47	0.0	**0.10**	0.01	**0.12**	0.02	**0.92**	0.01
	RandomForest	**0.96**	0.01	0.47	0.0	**0.10**	0.01	0.13	0.01	0.91	0.0
	B-J48-Pruned	**0.96**	0.0	**0.48**	0.0	0.11	0.01	0.14	0.01	0.91	0.01
	B-J48-Unpruned	**0.96**	0.01	0.47	0.0	0.11	0.01	0.14	0.02	0.90	0.01
	B-JRip	0.95	0.01	0.47	0.0	0.11	0.01	0.14	0.02	0.91	0.01
FE	J48-Pruned	**0.93**	0.01	**0.48**	0.0	**0.09**	0.01	**0.14**	0.01	**0.93**	0.0
	J48-Unpruned	**0.93**	0.01	**0.48**	0.01	**0.09**	0.01	**0.14**	0.01	**0.93**	0.0
	RandomForest	**0.93**	0.01	0.46	0.01	0.11	0.01	0.15	0.02	0.91	0.01
	B-JRip	**0.93**	0.01	0.47	0.01	0.11	0.01	0.15	0.01	0.91	0.01
	B-J48-Unpruned	0.92	0.01	0.46	0.01	0.11	0.01	0.16	0.02	0.92	0.0

DS: Dataset; GC: God Class; DC: Data Class; LM: Long Method; FE: Feature Envy.

Table 5
Ordinal vs. multinomial ρ comparison.

Model	God Class				Data Class				Long Method				Feature Envy			
		p	δ	ρ		p	δ	ρ		p	δ	ρ		p	δ	ρ
B-J48-Pruned	•	0.0	s.	0.86	•	0.0	m.	0.87	○	0.02	n.	0.96	•	0.01	s.	0.94
B-J48-ReducedErrorPruning	•	0.0	s.	0.85	•	0.0	s.	0.86	○	0.02	n.	0.96		0.11	n.	0.92
B-J48-Unpruned		0.29	n.	0.85	•	0.0	s.	0.87	○	0.01	n.	0.96	•	0.01	n.	0.94
B-JRip	•	0.0	n.	0.87	•	0.0	n.	0.87		0.97	n.	0.93		0.9	n.	0.92
B-LibSVM-C-Linear	•	0.0	l.	0.74	•	0.0	s.	0.76		0.09	n.	0.92	○	0.03	s.	0.84
B-LibSVM-C-RBF	•	0.0	l.	0.16	•	0.0	l.	0.21		1.0	n.	0.0		0.8	n.	0.0
B-LibSVM-C-Sigmoid		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0
B-LibSVM-n-Linear		1.0	m.	0.36	•	0.0	l.	0.8	•	0.0	l.	0.63		1.0	n.	0.0
B-LibSVM-n-RBF	•	0.0	l.	0.16	•	0.0	s.	0.1		1.0	n.	0.0		1.0	n.	0.0
B-LibSVM-n-Sigmoid		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0
B-NaiveBayes		0.94	n.	0.68		1.0	s.	0.68		0.39	n.	0.88	•	0.0	l.	0.85
B-RandomForest	•	0.0	s.	0.87	•	0.0	s.	0.88		0.4	n.	0.96	•	0.0	s.	0.94
B-SMO-RBF	•	0.0	m.	0.15	•	0.0	s.	0.84		1.0	n.	0.0	•	0.0	s.	0.83
J48-Pruned		1.0	s.	0.83		1.0	s.	0.84	○	0.04	n.	0.96		0.48	n.	0.93
J48-ReducedErrorPruning		0.15	n.	0.83		0.17	n.	0.83	•	0.01	n.	0.95	•	0.0	n.	0.93
J48-Unpruned		1.0	s.	0.82		1.0	s.	0.82	○	0.03	n.	0.96		0.99	n.	0.93
JRip	•	0.0	s.	0.8	•	0.0	l.	0.82		0.64	n.	0.86		0.35	n.	0.89
LibSVM-C-Linear	•	0.0	s.	0.73	•	0.0	s.	0.76		0.07	n.	0.9	•	0.0	l.	0.86
LibSVM-C-RBF	•	0.0	l.	0.15	•	0.0	l.	0.21		1.0	n.	0.0		1.0	n.	0.0
LibSVM-C-Sigmoid		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0
LibSVM-n-Linear		0.17	n.	0.59	•	0.0	l.	0.8	•	0.0	l.	0.95		1.0	n.	0.0
LibSVM-n-RBF	•	0.0	l.	0.15	•	0.0	s.	0.1		1.0	n.	0.0		1.0	n.	0.0
LibSVM-n-Sigmoid		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0		1.0	n.	0.0
NaiveBayes		1.0	s.	0.65		1.0	m.	0.65	•	0.0	n.	0.83		1.0	s.	0.71
RandomForest	•	0.0	n.	0.85	•	0.0	s.	0.87		0.24	n.	0.96		0.96	n.	0.92
SMO-RBF	•	0.0	m.	0.15	•	0.0	s.	0.83		1.0	n.	0.0	•	0.0	m.	0.89

Legend: • = $p \leq 0.01$; ○ = $p \leq 0.05$; n. = Negligible, s. = Small, m. = Medium, l. = Large

mances? The technique we applied enhanced the performances of the binary classification models in many, but not all, cases. There is no explicit pattern determining which models or datasets benefit more from the application of this technique, except for two observations: *1)* boosted J48 classifiers are enhanced by this technique, while their non-boosted counterparts do not; *2)* class-based smells are enhanced more through the ordinal method than method-based smells.

6.4. Regression models applied to ordinal classification

An ordinal classification task can be approached also through regression models. To be able to apply regression, the ordinal variable has to be mapped to a numeric variable. In our experiment, we mapped the four severity levels to integer values in the range 1–4. Whenever a regression model makes an estimation, it is mapped to the nearest of the 4 admitted values. This allows us to evaluate the performances of each regression model using the same procedure applied to all the other classification models. In fact, also in this experiment we applied 10 repetitions of 10-fold cross validation, using the same folds evaluated for all other models experimented in this paper.

Table 6 reports the performance obtained by the experimented regression models. The results suggest that RandomForests are superior to the other two models on all datasets. In fact, all performance indicators (except τ_b) are better for RandomForests than in the other two models in each dataset. Kendall's τ_b does not follow this pattern only for Long Method, which is the dataset where models performed better on average (including the classification models experimented in the previous sections).

Table 6
Regression models performances.

DS	Model	ρ	(σ)	τ_b	(σ)	MAD	(σ)	MSE	(σ)	Acc.	(σ)
GC	Regression-RandomForest	**0.88**	0.00	**0.65**	0.00	**0.29**	0.01	**0.34**	0.01	**0.73**	0.01
	Regression-Poly	0.80	0.01	0.61	0.01	0.45	0.01	0.58	0.03	0.61	0.01
	Regression-Linear	0.76	0.01	0.57	0.01	0.55	0.01	0.68	0.02	0.52	0.01
DC	Regression-RandomForest	**0.87**	0.00	**0.65**	0.00	**0.30**	0.01	**0.34**	0.01	**0.72**	0.01
	Regression-Poly	0.79	0.02	0.61	0.01	0.45	0.02	0.61	0.07	0.61	0.01
	Regression-Linear	0.75	0.01	0.57	0.00	0.56	0.01	0.70	0.01	0.50	0.01
LM	Regression-RandomForest	**0.95**	0.00	0.48	0.00	**0.14**	0.01	**0.14**	0.01	**0.86**	0.01
	Regression-Linear	0.89	0.01	**0.50**	0.00	0.22	0.01	0.23	0.01	0.78	0.01
	Regression-Poly	0.86	0.02	0.47	0.01	0.25	0.01	0.35	0.03	0.79	0.01
FE	Regression-RandomForest	**0.92**	0.01	**0.49**	0.00	**0.16**	0.01	**0.18**	0.01	**0.85**	0.01
	Regression-Poly	0.87	0.01	0.47	0.01	0.21	0.01	0.27	0.02	0.82	0.01
	Regression-Linear	0.87	0.01	0.48	0.00	0.24	0.01	0.25	0.01	0.77	0.01

DS: Dataset; GC: God Class; DC: Data Class; LM: Long Method; FE: Feature Envy.

6.4.0.3. Answer to RQ3. In RQ3, we ask: *What performance can be achieved by regression models?* As we can see from the results, regression models can reach performance values very near to the ones already observed through the use of ordinal-enhanced classifiers. RandomForests-based regression is superior than the other two regression models w.r.t. all the considered performance measures in almost all cases. This result is in line with the good performances obtained so far through the use of RandomForests and decision tree classifiers, i.e., that tree-based models are particularly suitable for this task.

6.5. Benchmarking the best prediction models

In the previous sections, we assessed the performances achieved by the multinomial classification models, which we use as baselines. We tested that the ordinal classification method often helps classifiers improving their performances on our datasets. Finally, we assessed that regression models (Random Forests in particular) also achieve good performances, comparable to the ones of the best applied classifiers.

In this section, we compare the best models found during the previous experiments, to find, possibly, a model having top performance in all datasets. Since the number of models (considering also the combinations with boosting and ordinal method) we applied in this paper is large, we perform our comparison over the best performing models found in the previous sections. We select the top 5 models found in multinomial classification, the top 5 ordinal-enhanced versions of the multinomial classification models (the top 5 among all the ordinal-enhanced models, data is not shown in this paper), and the best regression model we found. The list of the selected models is reported in the first column of Table 7.

In Section 5, we introduced another classifier in the pool of the experimented models, i.e., the Fuzzy Rule Learner. We separately experimented with this classifier, with different results according to different configurations. For this reason, we first tried to find the best configurations of this classifier for each dataset, to be added to the set of classifiers to be compared.

The Fuzzy Rule Learner [39,40] supports this set of options and accepted values[3]:

1. Shrink after commit: true, false;
2. Use class with max coverage: true, false;
3. Fuzzy Norm: Min/Max Norm, Product Norm, Lukasiewicz Norm, Yager[2.0] Norm, Yager[0.5] Norm;
4. Shrink Function: VolumeBorderBased, VolumeAnchorBased, VolumeRuleBased.

[3] Please refer to the cited resources and KNIME documentation for an explanation of the options..

We applied all the 60 combinations of the above options on the four datasets, plus their combination with the ordinal classification method. Each combination reports a suffix encoding the set of options, e.g., FuzzyRule-ff41 means "Shrink after commit" is false (f), "Use class with max coverage" is false (f), "Fuzzy Norm" is Yager[0.5] (4, counting from 0), "Shrink Function" is VolumeAnchorBased (1, counting from 0). All the model settings have been experimented through 10 repetitions of 10-fold cross-validation, on the same folds of all the other models.

Our goal is to select the top-5 performing FuzzyRule models for each dataset, to be compared with the other selected models. We applied the same method used by Weka in its Experimenter module to rank the models. The method can be summarized as:

- Apply a test comparing the performances of all the pairs of models on the performance values of each repetition/fold;
- After setting a significance level of the test, count the times a model wins (significant test, has higher performances), and loses (significant test, has lower performances); ties can happen, when the test is not significant, and are recorded separately;
- Compute a score "W-L" as the wins-losses difference for each model, to be used to rank models by descending value.

In our case, we applied Wilcoxon signed rank test with $\alpha = 0.01$ to the Spearman's ρ values of each of the 100 folds into each dataset.

After ranking the best classifiers, we selected the top 5 for each model; we report them in the second column of Table 7.

We then performed another model ranking task, using the same parameters applied for the selection of the best FuzzyRule models. In this task, we considered all the top-performing models we selected, i.e., multinomial, order-enhanced, regression, fuzzy rules, and the KNIME decision tree introduced in Section 5 (DTree). Please note that, for the models selected as the best classifiers for each dataset, we considered their performances on all datasets in this comparison, also on the datasets where they were not in the best-performing group. Table 8 shows the top 5 ranked classifiers of all datasets (the respective confusion matrices are reported in Appendix D). The only models listed for all datasets are OB-RandomForest and OB-J48-Pruned. The two class-based smells report some more common models, i.e., Regression-RandomForest and OB-JRip, not reported for the other two smells, which share the OB-J48-Unpruned model. It is worth noting that Regression-RandomForest is the top-performing model in both God Class and Data Class. As a final consideration on the results, we can see that the absolute performance scores are extremely similar in all the models reported for each dataset.

6.5.0.4. Answer to RQ4. in RQ4, we ask: *What is the best learning model experimented for severity classification?* No single model can

Table 7
Models selected for global comparison.

Multinomial, ordinal-enhanced, regression	Fuzzy Rule
B-J48-Pruned	
B-J48-Unpruned	
B-JRip	FuzzyRule-ff41
B-RandomForest	FuzzyRule-ft41
DTree	O-FuzzyRule-ff20
J48-Pruned	O-FuzzyRule-ff22
J48-Unpruned	O-FuzzyRule-ff40
O-DTree	O-FuzzyRule-ff41
O-J48-Pruned	O-FuzzyRule-ff42
O-J48-ReducedErrorPruning	O-FuzzyRule-ft20
O-RandomForest	O-FuzzyRule-ft40
OB-J48-Pruned	O-FuzzyRule-ft41
OB-J48-ReducedErrorPruning	O-FuzzyRule-ft42
OB-J48-Unpruned	O-FuzzyRule-tf22
OB-JRip	O-FuzzyRule-tf42
OB-RandomForest	O-FuzzyRule-tt22
RandomForest	
Regression-RandomForest	
model names can be prefixed with "O-", "B-" or "OB-"; O ="Ordinal", B = "Boosted"	

Table 8
Best prediction models - Top 5.

DS	Model	W-L	ρ	(σ)	τ_b	(σ)	MAD	(σ)	MSE	(σ)	Acc.	(σ)
GC	Regression-RandomForest	31	**0.88**	0.04	**0.66**	0.03	**0.29**	0.08	**0.34**	0.11	0.73	0.07
	OB-JRip	23	0.87	0.05	0.64	0.03	0.30	0.09	0.39	0.15	**0.74**	0.07
	OB-RandomForest	22	0.87	0.05	0.64	0.03	0.30	0.08	0.39	0.14	**0.74**	0.06
	OB-J48-Pruned	19	0.86	0.05	0.63	0.04	0.31	0.08	0.41	0.14	**0.74**	0.06
	O-DTree	17	0.86	0.06	0.64	0.04	0.32	0.08	0.44	0.16	**0.74**	0.06
DC	Regression-RandomForest	27	**0.88**	0.04	**0.66**	0.03	0.30	0.07	**0.34**	0.1	0.72	0.06
	OB-RandomForest	26	**0.88**	0.04	0.65	0.03	**0.27**	0.07	0.35	0.12	0.76	0.06
	OB-JRip	23	0.87	0.04	0.65	0.03	0.29	0.07	0.37	0.12	0.75	0.06
	O-RandomForest	21	0.87	0.05	0.64	0.04	0.28	0.08	0.39	0.15	**0.77**	0.06
	OB-J48-Pruned	21	0.87	0.04	0.64	0.04	0.29	0.07	0.37	0.12	0.75	0.06
LM	OB-J48-Pruned	22	**0.96**	0.03	**0.49**	0.01	**0.10**	0.05	**0.12**	0.07	0.91	0.04
	B-RandomForest	21	**0.96**	0.04	0.48	0.01	**0.10**	0.05	**0.12**	0.09	**0.92**	0.04
	OB-J48-Unpruned	21	**0.96**	0.03	**0.49**	0.01	**0.10**	0.05	**0.12**	0.07	0.91	0.04
	OB-RandomForest	20	**0.96**	0.03	**0.49**	0.02	**0.10**	0.05	**0.12**	0.09	0.91	0.04
	O-RandomForest	19	**0.96**	0.03	0.48	0.02	**0.10**	0.05	**0.12**	0.08	0.91	0.04
FE	OB-J48-Unpruned	22	**0.94**	0.04	**0.49**	0.02	0.10	0.04	**0.14**	0.08	0.91	0.04
	OB-J48-Pruned	19	**0.94**	0.04	**0.49**	0.02	0.10	0.05	**0.14**	0.09	0.91	0.04
	OB-RandomForest	18	**0.94**	0.04	0.48	0.02	0.10	0.04	**0.14**	0.09	0.92	0.03
	J48-Pruned	16	0.93	0.05	**0.49**	0.02	**0.09**	0.05	**0.14**	0.09	**0.93**	0.04
	O-J48-Pruned	16	0.93	0.04	**0.49**	0.02	**0.09**	0.05	**0.14**	0.09	**0.93**	0.03

DS: Dataset; GC: God Class; DC: Data Class; LM: Long Method; FE: Feature Envy.

be determined to be the best for this task. First, all the top 5 models in each dataset reach extremely similar performance values, meaning that while some small difference exist, they can all be applied reaching performances among the best. Second, some models seem to be more appropriate according to the type of smell, i.e., Regression-RandomForest for God Class and Data Class and OB-J48-Pruned/Unpruned for Long Method and Feature Envy. Notably, none of the FuzzyRule models we experimented appear in the top 5 ranked models, confirming our previous results, i.e., that decision tree-based models are best suited for this classification task, in both binary and ordinal classification. A specific advantage of Regression-RandomForests is that it does not need to be repeated different times with different goals or different input sampling, as it is needed with the ordinal method and boosting, thus making the learning phase much faster.

7. Threats to validity

This section discusses the threats to validity of our experiments. Threats to internal validity are related to the correctness of the experiments' outcome, while threats to external validity are related to the generalization of the obtained results.

7.1. Threats to internal validity

The manual evaluation of code smells is subject to a certain degree of error, and can be considered the main threat to the internal validity of the experiment. Several factors can be taken into account, such as the developers' experience in object-oriented programming, the knowledge and the ability to understand design issues and other factors. All these factors can change the ability of the evaluator of choosing the proper value for the class variable. This could obviously cause a distortion in the training set. We managed this threat by aggregating the evaluation of three raters [6].

Another threat to internal validity may lie in the data preprocessing phase. In this paper, we do not show the use of normalization or sophisticated feature selection approaches. During an initial data exploration phase, we applied different normalization approaches to the datasets, never obtaining significantly different results, especially in classifier's rankings. Adding the type of normalization applied as another dimension of the study would have extremely increased the size of results reporting, without adding information for the reader. As for feature selection, we opted for a lightweight feature selection phase (variance and linear correlation), which removed more than a half of the features. Other more sophisticated feature selection procedures can be applied. For ex-

ample, backward feature selection can be used to select feature according to a specific classifier's performances. This kind of optimization procedure is extremely demanding of computing power, and would have implied extremely long experiments.

7.2. Threats to external validity

The generalization of our results can be limited by different factors. A first factor is the representativeness of the dataset of each code smell. The datasets have been built by sampling classes and methods from 76 systems belonging to different domains. All systems are open source and written in Java. This means that our results may not be generalized to proprietary software, or non-Java software. The dataset composition is influenced also by the balance between code smell of different severity levels. The sampling procedure applied to build the datasets has been targeted, through the use of Advisors, to the extraction of code smells of different severity levels. The balance between absence and presence of code smells has been kept as much as possible in the 33%–66% range. The balance between the severity levels implying different amount of code smell presence (all except the first), anyway is not homogeneous. Our results may not be applied successfully in contexts where the balance among different severity levels is significantly different than ours.

As a final point, our selection of four code smell, due to the cost of building reliable datasets, implies that we cannot generalize our results to other code smells. Anyway, the selection of the code smells has been driven by different criteria, e.g., diffusion, impact on code quality, to make our results as useful as possible to researchers and practitioners.

8. Concluding remarks

In this paper, we described our approach for code smell detection based on machine learning-techniques, by outlining in particular how the severity of code smells can be classified through learning models. We approached the task as an ordinal classification (or regression) problem, since severity is defined on an ordinal scale composed of four values. We applied classification models in two setups, the first simulating a multinomial classification, and the second by applying a known method to use classification models for ordinal classification. Moreover, we applied regression models, by assigning integer values to the ordinal variable.

The datasets and a first set of classification models are taken from previous work [6], where we applied binary classification for code smell detection. In this paper, we use the same datasets, but with code smell severity information, to classify the severity of each code smell. This task is more complex than binary classification, because of the ordinal class variable, and the higher sparsity of the single severity levels, but is also more useful to the potential users of the extracted learning models. In fact, code smells do not always need to be removed, sometimes they bring only minor disadvantages to the software they are into. Software developers and maintainers should therefore concentrate only on the more important code smells removals, and the techniques experimented in this paper have this target.

The importance of a good ranking of code smells, more than a perfectly accurate severity classification, is also the reason of some choices made during the design of our experiments. For example, we applied different performance measures to the classifiers' outputs, but we chose to apply Spearman's ρ as the primary evaluation measure. This measure, in fact, targets the accuracy of the ordering of the provided results, more than the actual prediction values. For the users of these techniques, in fact, the ranking of code smells by their severity is potentially more important than their exact severity value.

In our experiments, the overall goal has been to understand the performance levels achievable on our datasets for code smell severity classification. We think that our experiments demonstrated the viability of code smell severity classification as a reliable support to decision-making when approaching the refactoring of a large software system, allowing the prioritization of the code smells to be inspected and fixed. Since different techniques can be applied in this context, we organized our research according to four more specific research questions:

RQ1 *How do the best learning models identified in our previous work perform for this task?* We applied the best models we found for binary classification on this datasets, which reached 96–99% accuracy values, and applied them as-is in a multinomial classification task. The result is that they can reach ∼75% and ∼93% accuracy on class and method smells respectively. While 75% is a large accuracy drop, if considering the 96% accuracy of binary classification, the respective ρ values are ∼85%, pointing to a rather good correlation, and MAD ∼0.3 means that most errors differ for only one severity level from the correct result, making the overall ranking still reliable.

RQ2 *Does the application of a technique for ordinal meta-learning enhance the detection performances?* We applied the ordinal classification method by Frank and Hall [9] to the models used to answer RQ1. The result is that the method reached significantly better ρ values, but for different models in different datasets. In particular, the two class-based smells had more benefit from this method than the remaining two smells, which started from higher multinomial classification performances.

RQ3 *What performance can be achieved by regression models?* After applying Linear, Polynomial and RandomForests regression models, we found that RandomForests have the best overall performances, and that these performances are very similar to the best performances obtained through multinomial and ordinal classification by the other models in RQ1 and RQ2.

RQ4 *What is the best learning model experimented for severity classification?* We compared the performances of the top 5 learning models we experimented for each dataset, for both multinomial and ordinal classification, the best regression model, and many configurations of a Fuzzy Rule Learner model. The result is that RandomForests regression is the best model for God Class and Data Class, while RandomForests and decision trees, combined with boosting and the experimented ordinal classification method, have generally the best performances on all datasets.

We publicly share our datasets[4] to foster replication and extension by other research groups.

We would like to continue this work in several directions. First, we would like to extend our experiments to new datasets (including other smells and other manual validations), possibly generated by different research groups, to test the resilience of our approach to other severity classifications. In fact, code smell severity classification can be subjective, since the definition of code smells are informal, and different developers may consider different aspects of code smells when manually evaluating their instances. In this direction, a shared code smell dataset, taking into consideration severity, could be extremely valuable. Some initial proposals exist [14] going in this direction, but a real benchmark platform, as

[4] Datasets are available for reviewers at http://essere.disco.unimib.it/wiki/research/mlcsd ; archives are password-protected: password is "AF-MZ-mlcsd-severity-2016".

preliminary done, e.g., for design pattern detection [42], is still missing. When working on new datasets, it will also be important to optimize the parameters of the learning models, since the ones used in this work have been partially taken from our previous work on binary classification of code smells. Second, the learning approach used in this experiment could be extended including other pre-processing techniques helping the learning algorithms to reach higher performances. One relevant pre-processing phase that could be introduced is resampling: this would balance the input classes, possibly improving detection performances. Anyway, the datasets used in this work have originally been generated by trying to avoid extreme class imbalance. This has been necessary, since code smells are rare in software systems, and any attempt to extract detection models from smell data has to deal with this fact.

Another point of extension we are interested in is to provide tooling for the applied techniques, to build, integrate with, or support the building of, a code smell detection tool able to recommend developers the most meaningful issues to be approached. Since the approach uses standard software metrics as independent variables, this technique could be integrated, e.g., in our code smell detection tool JCodeOdor [11], or any other code smell detection tool based on software metrics.

Appendix A. Selected software metrics

Table A.9 reports the metrics we selected, classified by the aspect of software quality (Size, Complexity, Cohesion, Coupling, Encapsulation, Inheritance) they address most, while in Table A.10 we report another set of metrics we considered, which are based on the count of the number of element in classes or packages declared with some language modifier keyword (e.g., public, private, abstract, final). For a complete reference and definition of the metrics, please refer to this web page: http://essere.disco.unimib.it/wiki/research/mlcsd.

Table A.9
Selected metric names.

Quality dimension	Metric label	Metric name	Granularity
Size	LOC	Lines of Code	Project, Package, Class, Method
	LOCNAMM	Lines of Code Without Accessor or Mutator Methods	Class
	NOPK	Number of Packages	Project
	NOCS	Number of Classes	Project, Package
	NOM	Number of Methods	Project, Package, Class
	NOMNAMM	Number of Not Accessor or Mutator Methods	Project, Package, Class
	NOA	Number of Attributes	Class
Complexity	CYCLO	Cyclomatic Complexity	Method
	WMC	Weighted Methods Count	Class
	WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods	Class
	AMW	Average Methods Weight	Class
	AMWNAMM	Average Methods Weight of Not Accessor or Mutator Methods	Class
	MAXNESTING	Maximum Nesting Level	Method
	CLNAMM	Called Local Not Accessor or Mutator Methods	Method
	NOP	Number of Parameters	Method
	NOAV	Number of Accessed Variables	Method
	ATLD	Access to Local Data	Method
	NOLV	Number of Local Variable	Method
	FANOUT	–	Class, Method
	FANIN	–	Class
	ATFD	Access to Foreign Data	Method
	FDP	Foreign Data Providers	Method
	RFC	Response for a Class	Class
Coupling	CBO	Coupling Between Objects Classes	Class
	CFNAMM	Called Foreign Not Accessor or Mutator Methods	Class, Method
	CINT	Coupling Intensity	Method
	MaMCL	Maximum Message Chain Length	Method
	MeMCL	Mean Message Chain Length	Method
	NMCS	Number of Message Chain Statements	Method
	CC	Changing Classes	Method
	CM	Changing Methods	Method
	NOAM	Number of Accessor Methods	Class
Encapsulation	NOPA (NOAP)	Number of Public Attribute	Class
	LAA	Locality of Attribute Accesses	Method
	DIT	Depth of Inheritance Tree	Class
	NOI	Number of Interfaces	Project, Package
Inheritance	NOC	Number of Children	Class
	NMO	Number of Methods Overridden	Class
	NIM	Number of Inherited Methods	Class
	NOII	Number of Implemented Interfaces	Class

Table A.10
Custom modifier-based metrics names.

Metric label	Metric name
NODA	Number of Default Attributes
NOPVA	Number of Private Attributes
NOPRA	Number of Protected Attributes
NOFA	Number of Final Attributes
NOFSA	Number of Final and Static Attributes
NOFNSA	Number of Final and non - Static Attributes
NONFNSA	Number of not Final and non - Static Attributes
NOSA	Number of Static Attributes
NONFSA	Number of non - Final and Static Attributes
NOABM	Number of Abstract Methods
NOCM	Number of Constructor Methods
NONCM	Number of non - Constructor Methods
NOFM	Number of Final Methods
NOFNSM	Number of Final and non - Static Methods
NOFSM	Number of Final and Static Methods
NONFNABM	Number of non - Final and non - Abstract Methods
NONFNSM	Number of non - Final and non - Static Methods
NONFSM	Number of non - Final and Static Methods
NODM	Number of Default Methods
NOPM	Number of Private Methods
NOPRM	Number of Protected Methods
NOPLM	Number of Public Methods
NONAM	Number of non - Accessors Methods
NOSM	Number of Static Methods

Appendix B. Experiments detailed results

This section contains extended results tables, cited in Section 6.

Table B.11
Existing model performances on multinomial classification - God Class.

Model	ρ	(σ)	τ_b	(σ)	MAD	(σ)	MSE	(σ)	Acc.	(σ)
B-JRip	0.85	0.01	0.61	0.01	0.30	0.02	0.43	0.04	0.75	0.01
B-J48-Unpruned	0.85	0.01	0.61	0.01	0.31	0.02	0.46	0.03	0.75	0.02
J48-Pruned	0.85	0.01	0.62	0.01	0.31	0.02	0.46	0.05	0.76	0.01
B-RandomForest	0.85	0.01	0.61	0.01	0.31	0.01	0.43	0.03	0.74	0.01
B-J48-Pruned	0.84	0.01	0.61	0.01	0.32	0.01	0.48	0.02	0.75	0.01
J48-Unpruned	0.84	0.02	0.61	0.01	0.33	0.02	0.49	0.06	0.75	0.02
B-J48-ReducedErrorPruning	0.83	0.02	0.60	0.01	0.35	0.02	0.55	0.06	0.74	0.01
RandomForest	0.83	0.01	0.59	0.01	0.34	0.02	0.53	0.05	0.75	0.01
J48-ReducedErrorPruning	0.82	0.02	0.59	0.01	0.36	0.02	0.59	0.05	0.74	0.01
JRip	0.76	0.03	0.55	0.02	0.43	0.04	0.74	0.10	0.69	0.02
B-NaiveBayes	0.69	0.01	0.50	0.01	0.67	0.01	1.16	0.04	0.55	0.01
NaiveBayes	0.69	0.01	0.50	0.01	0.67	0.01	1.16	0.04	0.55	0.01
LibSVM-C-Linear	0.67	0.01	0.49	0.01	0.59	0.02	1.05	0.04	0.60	0.02
B-LibSVM-n-Linear	0.65	0.04	0.47	0.03	0.64	0.07	1.09	0.14	0.55	0.04
B-LibSVM-C-Linear	0.64	0.03	0.47	0.02	0.62	0.04	1.11	0.11	0.58	0.02
LibSVM-n-Linear	0.55	0.05	0.40	0.04	0.88	0.06	1.54	0.14	0.41	0.03
B-SMO-RBF	0.08	0.02	0.02	0.01	1.46	0.01	3.75	0.03	0.38	0.00
SMO-RBF	0.08	0.02	0.02	0.01	1.46	0.01	3.75	0.03	0.38	0.00
LibSVM-C-RBF	0.06	0.02	0.02	0.01	1.46	0.01	3.74	0.03	0.38	0.00
LibSVM-n-RBF	0.06	0.02	0.02	0.01	1.46	0.01	3.74	0.03	0.38	0.00
B-LibSVM-C-RBF	0.05	0.03	0.02	0.01	1.46	0.01	3.74	0.03	0.38	0.00
B-LibSVM-n-RBF	0.05	0.03	0.02	0.01	1.46	0.01	3.74	0.03	0.38	0.00
B-LibSVM-n-Sigmoid	0.01	0.00	0.00	0.00	1.23	0.05	2.09	0.25	0.15	0.04
LibSVM-n-Sigmoid	0.00	0.01	0.00	0.00	1.45	0.02	3.50	0.10	0.28	0.01
B-LibSVM-C-Sigmoid			0.00	0.00	1.50	0.00	3.84	0.00	0.37	0.00
LibSVM-C-Sigmoid			0.00	0.00	1.50	0.00	3.84	0.00	0.37	0.00

Table B.12
Existing model performances on multinomial classification - Data Class.

Model	ρ	(σ)	τ_b	(σ)	MAD	(σ)	MSE	(σ)	Acc.	(σ)
B-RandomForest	0.86	0.01	0.62	0.01	0.29	0.01	0.39	0.03	0.76	0.01
J48-Pruned	0.86	0.01	0.62	0.01	0.30	0.02	0.43	0.03	0.76	0.01
B-JRip	0.85	0.01	0.62	0.01	0.29	0.02	0.41	0.04	0.76	0.01
RandomForest	0.85	0.01	0.61	0.01	0.29	0.02	0.43	0.02	0.77	0.01
J48-Unpruned	0.84	0.01	0.61	0.01	0.33	0.02	0.47	0.04	0.73	0.02
B-J48-Unpruned	0.84	0.01	0.61	0.01	0.33	0.01	0.46	0.03	0.74	0.01
B-J48-Pruned	0.84	0.01	0.61	0.01	0.33	0.02	0.47	0.04	0.73	0.01
B-J48-ReducedErrorPruning	0.83	0.01	0.60	0.01	0.34	0.02	0.53	0.04	0.75	0.01
J48-ReducedErrorPruning	0.82	0.02	0.59	0.01	0.36	0.03	0.57	0.07	0.73	0.02
B-SMO-RBF	0.81	0.01	0.59	0.01	0.37	0.01	0.55	0.03	0.70	0.01
SMO-RBF	0.80	0.01	0.58	0.00	0.36	0.01	0.55	0.03	0.72	0.00
JRip	0.74	0.02	0.55	0.01	0.46	0.02	0.80	0.05	0.68	0.02
B-LibSVM-C-Linear	0.73	0.03	0.54	0.02	0.48	0.04	0.79	0.09	0.64	0.02
B-NaiveBayes	0.71	0.01	0.52	0.00	0.66	0.01	1.11	0.02	0.55	0.01
NaiveBayes	0.71	0.01	0.52	0.00	0.66	0.01	1.11	0.02	0.55	0.01
LibSVM-C-Linear	0.71	0.02	0.52	0.02	0.52	0.03	0.89	0.07	0.63	0.01
B-LibSVM-C-RBF	0.09	0.01	0.04	0.00	1.42	0.01	3.60	0.01	0.39	0.00
LibSVM-C-RBF	0.08	0.01	0.03	0.00	1.43	0.01	3.64	0.02	0.39	0.00
LibSVM-n-RBF	0.07	0.02	0.02	0.01	1.45	0.01	3.68	0.03	0.38	0.00
B-LibSVM-n-RBF	0.07	0.01	0.02	0.00	1.45	0.01	3.67	0.02	0.38	0.00
B-LibSVM-n-Sigmoid	0.00	0.01	0.00	0.00	1.46	0.03	3.57	0.23	0.33	0.03
B-LibSVM-C-Sigmoid	0.00	0.01	0.00	0.00	1.50	0.01	3.81	0.02	0.35	0.01
B-LibSVM-n-Linear			0.00	0.00	1.50	0.00	3.81	0.00	0.36	0.00
LibSVM-C-Sigmoid			0.00	0.00	1.50	0.00	3.81	0.00	0.36	0.00
LibSVM-n-Linear			0.00	0.00	1.50	0.00	3.81	0.00	0.36	0.00
LibSVM-n-Sigmoid			0.00	0.00	1.50	0.00	3.81	0.00	0.36	0.00

Table B.13
Existing model performances on multinomial classification - Long Method .

Model	ρ	(σ)	τ_b	(σ)	MAD	(σ)	MSE	(σ)	Acc.	(σ)
B-RandomForest	0.96	0.01	0.47	0.00	0.10	0.01	0.12	0.02	0.92	0.01
RandomForest	0.96	0.01	0.47	0.00	0.10	0.01	0.13	0.01	0.91	0.00
B-J48-Pruned	0.96	0.00	0.48	0.00	0.11	0.01	0.14	0.01	0.91	0.01
B-J48-Unpruned	0.96	0.01	0.47	0.00	0.11	0.01	0.14	0.02	0.90	0.01
B-JRip	0.95	0.01	0.47	0.00	0.11	0.01	0.14	0.02	0.91	0.01
B-J48-ReducedErrorPruning	0.95	0.01	0.47	0.01	0.11	0.02	0.16	0.03	0.91	0.01
J48-Pruned	0.95	0.01	0.47	0.00	0.14	0.01	0.18	0.03	0.88	0.01
J48-Unpruned	0.95	0.01	0.47	0.00	0.14	0.01	0.19	0.03	0.88	0.01
J48-ReducedErrorPruning	0.94	0.01	0.47	0.01	0.13	0.01	0.19	0.02	0.90	0.01
JRip	0.91	0.01	0.45	0.01	0.16	0.01	0.24	0.03	0.88	0.01
B-LibSVM-C-Linear	0.91	0.01	0.46	0.01	0.16	0.02	0.25	0.03	0.88	0.01
LibSVM-C-Linear	0.89	0.01	0.44	0.01	0.19	0.01	0.29	0.03	0.86	0.01
B-NaiveBayes	0.88	0.01	0.44	0.00	0.21	0.01	0.35	0.03	0.85	0.01
NaiveBayes	0.80	0.01	0.44	0.00	0.32	0.01	0.57	0.02	0.79	0.00
B-LibSVM-C-RBF			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
B-LibSVM-C-Sigmoid			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
B-LibSVM-n-Linear			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
B-LibSVM-n-RBF			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
B-LibSVM-n-Sigmoid			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
B-SMO-RBF			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
LibSVM-C-RBF			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
LibSVM-C-Sigmoid			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
LibSVM-n-Linear			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
LibSVM-n-RBF			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
LibSVM-n-Sigmoid			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00
SMO-RBF			0.00	0.00	0.72	0.00	1.66	0.00	0.67	0.00

Table B.14
Existing model performances on multinomial classification - Feature Envy.

Model	ρ	(σ)	τ_b	(σ)	MAD	(σ)	MSE	(σ)	Acc.	(σ)
J48-Pruned	0.93	0.01	0.48	0.00	0.09	0.01	0.14	0.01	0.93	0.00
J48-Unpruned	0.93	0.01	0.48	0.01	0.09	0.01	0.14	0.01	0.93	0.00
RandomForest	0.93	0.01	0.46	0.01	0.11	0.01	0.15	0.02	0.91	0.01
B-JRip	0.93	0.01	0.47	0.01	0.11	0.01	0.15	0.01	0.91	0.01
B-J48-Unpruned	0.92	0.01	0.46	0.01	0.11	0.01	0.16	0.02	0.92	0.00
J48-ReducedErrorPruning	0.92	0.01	0.47	0.00	0.11	0.01	0.16	0.01	0.92	0.00
B-J48-Pruned	0.92	0.01	0.46	0.01	0.11	0.01	0.17	0.01	0.91	0.00
B-RandomForest	0.92	0.01	0.45	0.00	0.11	0.01	0.16	0.02	0.91	0.00
B-J48-ReducedErrorPruning	0.91	0.01	0.47	0.01	0.11	0.01	0.17	0.02	0.91	0.01
JRip	0.89	0.01	0.45	0.01	0.15	0.02	0.23	0.03	0.88	0.01
SMO-RBF	0.84	0.01	0.42	0.01	0.20	0.01	0.32	0.02	0.86	0.01
B-LibSVM-C-Linear	0.81	0.02	0.43	0.01	0.22	0.01	0.33	0.02	0.83	0.01
B-SMO-RBF	0.79	0.01	0.39	0.01	0.25	0.01	0.42	0.02	0.83	0.01
B-NaiveBayes	0.76	0.02	0.42	0.01	0.31	0.02	0.48	0.03	0.77	0.01
NaiveBayes	0.75	0.01	0.42	0.01	0.34	0.01	0.55	0.02	0.76	0.01
LibSVM-C-Linear	0.66	0.03	0.40	0.01	0.37	0.03	0.50	0.04	0.69	0.03
B-LibSVM-n-Linear	0.29	0.07	0.07	0.05	0.60	0.05	1.27	0.12	0.69	0.02
B-LibSVM-C-RBF	0.03	0.06	0.00	0.00	0.66	0.00	1.43	0.01	0.67	0.00
B-LibSVM-C-Sigmoid			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00
B-LibSVM-n-RBF			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00
B-LibSVM-n-Sigmoid			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00
LibSVM-C-RBF			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00
LibSVM-C-Sigmoid			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00
LibSVM-n-Linear			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00
LibSVM-n-RBF			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00
LibSVM-n-Sigmoid			0.00	0.00	0.66	0.00	1.43	0.00	0.67	0.00

Appendix C. Feature vectors used for the training of models

Table C.15 reports all the features we used to train the learning models. We report only the features that have been kept after the feature selection phase, and for each code smell we report a • sign if the feature has been used, ○ otherwise. There are two kinds of features. The first kind are metrics, i.e., the ones reported in Tables A.9 and A.10. In addition to the metric name, in many cases the feature name contains a suffix, separated by a "_" sign, specifying the granularity of the metric reported in that feature. For example, `ATFD_method` and `ATFD_type` both refer to the ATFD metric, but the first is measured on methods, and the second is measured/aggregated on types. We use metrics gathered at different granularities, i.e., method, type, package, project, as explained in Section 3.1. The second kind of features is represented by additional properties of types or methods that we recorded, i.e., `isStatic`, `modifier`, `visibility`, all assigned to a granularity. The first is a boolean value reporting if a type or method is declared as `static`, the second reports which other modifier it has, e.g., `abstract`, `final`, and the third reports the visibility attribute of the type or method.

Table C.15
Features used for training the machine learning models.

Feature	God class	Data class	Long method	Feature envy
AMW_type	○	○	•	○
ATFD_method	○	○	•	•
ATFD_type	•	•	○	○
ATLD_method	○	○	○	•
CBO_type	○	○	•	○
CDISP_method	○	○	•	•
CLNAMM_method	○	○	•	•
DIT_type	•	•	•	•
FANOUT_method	○	○	•	•
FANOUT_type	•	•	•	•
LAA_method	○	○	•	•
LCOM5_type	•	•	•	•
LOC_method	○	○	•	•
LOC_type	•	•	○	○
MAXNESTING_method	○	○	•	•
MaMCL_method	○	○	•	•
NIM_type	•	•	•	•
NMO_type	•	•	○	○
NOAM_type	•	•	•	•
NOA_type	•	•	○	•
NOII_type	○	○	•	•
NOI_package	•	•	•	•
NOMNAMM_package	•	•	•	•
NOM_project	•	•	•	•
NOPK_project	•	•	•	•
NOP_method	○	○	•	•
RFC_type	•	•	•	•
TCC_type	•	•	•	•
WMC_type	○	○	•	○
WOC_type	•	•	•	•
NOFA	•	○	•	○
NONFNSA	•	•	•	•
NOCM	•	•	•	•
NONFNABM	○	○	○	•
NONFNSM	•	•	○	○
NONFSM	○	○	•	•
NODA	○	○	•	○
NOPVA	•	•	•	•
NOPM	•	•	○	○
NOPRA	•	•	○	○
NOPRM	○	○	•	•
NOSM	○	○	•	○
isStatic_method	○	○	•	•
isStatic_type	•	•	○	○
modifier_type	•	•	•	•
visibility_type	•	•	•	•

Appendix D. Confusion matrices for the best classifiers

In the following, we report the confusion matrices for the top 5 classifiers found in the experiment and reported in Table 8. Table D.16 represents the confusion matrix for God Class, Table D.17 for Data Class, Table D.18 for Long Method, and Table D.19 for Feature Envy. In each table, rows correspond to the actual class label, and columns to the prediction values.

Table D.16
CM for best classifiers for God Class.

(a) Regression-RandomForest

	1	2	3	4
1	1239	217	84	0
2	20	150	112	8
3	0	137	866	97
4	0	10	436	824

(b) OB-JRip

	1	2	3	4
1	1329	116	88	7
2	42	114	107	27
3	29	125	712	234
4	1	18	293	958

(c) OB-RandomForest

	1	2	3	4
1	1355	102	75	8
2	49	98	120	23
3	39	102	703	256
4	5	17	283	965

(d) OB-J48-Pruned

	1	2	3	4
1	1358	88	83	11
2	59	94	113	24
3	38	124	700	238
4	2	27	288	953

(e) O-DTree

	1	2	3	4
1	1271	144	120	5
2	49	122	102	17
3	30	173	758	139
4	10	40	273	947

Table D.17
CM for best classifiers for Data Class.

(a) Regression-RandomForest

	1	2	3	4
1	1201	230	79	0
2	20	164	133	3
3	0	152	878	100
4	0	10	441	789

(b) OB-RandomForest

	1	2	3	4
1	1368	61	72	9
2	43	110	143	24
3	21	102	775	232
4	1	6	287	946

(c) OB-JRip

	1	2	3	4
1	1358	75	64	13
2	33	130	139	18
3	23	146	748	213
4	0	27	289	924

(d) O-RandomForest

	1	2	3	4
1	1398	28	68	16
2	59	111	116	34
3	43	92	729	266
4	7	13	223	997

(e) OB-J48-Pruned

	1	2	3	4
1	1358	74	74	4
2	44	127	129	20
3	39	132	724	235
4	2	23	281	934

Table D.18
CM for best classifiers for Long Method.

(a) OB-J48-Pruned

	1	2	3	4
1	2758	21	21	0
2	7	15	85	3
3	12	41	810	87
4	0	0	116	224

(b) B-RandomForest

	1	2	3	4
1	2761	1	37	1
2	8	7	95	0
3	14	9	866	61
4	0	0	122	218

(c) OB-J48-Unpruned

	1	2	3	4
1	2757	21	21	1
2	5	21	81	3
3	11	39	820	80
4	1	0	125	214

(d) OB-RandomForest

	1	2	3	4
1	2760	11	26	3
2	6	7	96	1
3	14	15	844	77
4	0	0	112	228

(e) O-RandomForest

	1	2	3	4
1	2769	4	27	0
2	9	2	98	1
3	23	14	840	73
4	0	2	106	232

Table D.19
CM for best classifiers for Feature Envy.

(a) OB-J48-Unpruned

	1	2	3	4
1	2706	52	32	10
2	41	139	50	0
3	4	72	828	46
4	0	2	69	149

(b) OB-J48-Pruned

	1	2	3	4
1	2699	54	37	10
2	32	155	43	0
3	10	59	831	50
4	0	2	72	146

(c) OB-RandomForest

	1	2	3	4
1	2729	29	33	9
2	56	121	53	0
3	14	32	864	40
4	3	1	77	139

(d) J48-Pruned

	1	2	3	4
1	2686	45	59	10
2	51	128	51	0
3	3	21	876	50
4	0	0	20	200

(e) O-J48-Pruned

	1	2	3	4
1	2680	50	60	10
2	49	131	50	0
3	0	20	880	50
4	0	0	20	200

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1999. http://www.refactoring.com/.

[2] F.A. Fontana, E. Mariani, A. Morniroli, R. Sormani, A. Tonello, An experience report on using code smells detection tools, in: IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, RefTest 2011, IEEE Computer Society, Berlin, 2011, pp. 450–457, doi:10.1109/ICSTW.2011.12.

[3] F.A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: an experimental assessment, J. Object Technol. 11 (2) (2012) 1–38, doi:10.5381/jot.2012.11.2.a5. 5.

[4] F.A. Fontana, V. Ferme, M. Zanoni, Poster: filtering code smells detection results, in: Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Vol. 2, IEEE, Florence, Italy, 2015, pp. 803–804, doi:10.1109/ICSE.2015.256.

[5] F.A. Fontana, M. Zanoni, A. Marino, M.V. Mäntylä, Code smell detection: towards a machine learning-based approach, in: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013), IEEE Computer Society, Eindhoven, The Netherlands, 2013, pp. 396–399, doi:10.1109/ICSM.2013.56. ERA Track.

[6] F.A. Fontana, M.V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empir. Softw. Eng. (2015) 1–49, doi:10.1007/s10664-015-9378-4.

[7] M. Zanoni, F.A. Fontana, F. Stella, On applying machine learning techniques for design pattern detection, J. Syst. Softw. 103 (0) (2015) 102–117, doi:10.1016/j.jss.2015.01.037.

[8] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367, doi:10.1109/TSE.2009.1.

[9] E. Frank, M. Hall, A Simple Approach to Ordinal Classification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 145–156, doi:10.1007/3-540-44795-4_13.

[10] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer-Verlag, 2006.

[11] F.A. Fontana, V. Ferme, M. Zanoni, R. Roveda, Towards a prioritization of code debt: a code smell intensity index, in: Proceedings of the Seventh International Workshop on Managing Technical Debt (MTD 2015), IEEE, Bremen, Germany, 2015, pp. 16–24, doi:10.1109/MTD.2015.7332620. In conjunction with ICSME 2015.

[12] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F.L. Meur, DECOR: a method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng. 36 (1) (2010) 20–36, doi:10.1109/TSE.2009.50.

[13] N. Tsantalis, A. Chatzigeorgiou, Ranking refactoring suggestions based on historical volatility, in: Proceedings of the 15th European Conference on Software Maintenance and Reengineering, IEEE, 2011, pp. 25–34, doi:10.1109/CSMR.2011.7.

[14] F. Palomba, D.D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, A. De Lucia, Landfill: an open dataset of code smells with public evaluation, in: Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15), IEEE, Florence, Italy, 2015, pp. 482–485, doi:10.1109/MSR.2015.69.

[15] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, E. Aïimeur, Support vector machines for anti-pattern detection, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), ACM, Essen, Germany, 2012, pp. 278–281, doi:10.1145/2351676.2351723.

[16] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, Bdtex: a GQM-based Bayesian approach for the detection of antipatterns, J. Syst. Softw. 84 (4) (2011) 559–572, doi:10.1016/j.jss.2010.11.921. The 9th International Conference on Quality Software

[17] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, E. Aimeur, SMURF: a SVM-based incremental anti-pattern detection approach, in: Proceedings of the 19th Working Conference on Reverse Engineering (WCRE 2012), IEEE, Kingston, Ontario, Canada, 2012, pp. 466–475, doi:10.1109/WCRE.2012.56.

[18] J. Yang, K. Hotta, Y. Higo, H. Igaki, S. Kusumoto, Filtering clones for individual user based on machine learning analysis, in: Proceedings of the 6th International Workshop on Software Clones (IWSC 2012), IEEE Computer Society, Zurich, Switzerland, 2012, pp. 76–77, doi:10.1109/IWSC.2012.6227872.

[19] N. Maneerat, P. Muenchaisri, Bad-smell prediction from software design model using machine learning techniques, in: Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE 2011), IEEE, Nakhon Pathom, Thailand, 2011, pp. 331–336, doi:10.1109/JCSSE.2011.5930143.

[20] T. Menzies, A. Marcus, Automated severity assessment of software defect reports, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2008), IEEE, Beijing, China, 2008, pp. 346–355, doi:10.1109/ICSM.2008.4658083.

[21] A. Lamkanfi, S. Demeyer, E. Giger, B. Goethals, Predicting the severity of a reported bug, in: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), IEEE, Cape Town, South Africa, 2010, pp. 1–10, doi:10.1109/MSR.2010.5463284.

[22] Y. Tian, D. Lo, C. Sun, Information retrieval based nearest neighbor classification for fine-grained bug severity prediction, in: Proceedings of the 19th Working Conference on Reverse Engineering (WCRE 2012), IEEE, Kingston, ON, Canada, 2012, pp. 215–224, doi:10.1109/WCRE.2012.31.

[23] S.A. Vidal, C. Marcos, J.A. Díaz-Pace, An approach to prioritize code smells for refactoring, Autom. Softw. Eng. 23 (3) (2016a) 501–532, doi:10.1007/s10515-014-0175-x.

[24] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A.D. Pace, C. Marcos, On the criteria for prioritizing code anomalies to identify architectural problems, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC 2016), ACM, Pisa, Italy, 2016b, pp. 1812–1814, doi:10.1145/2851613.2851941.

[25] R. Marinescu, Assessing technical debt by identifying design flaws in software systems, IBM J. Res. Dev. 56 (5) (2012) 9:1–9:13, doi:10.1147/JRD.2012.2204512.

[26] F. Palomba, M. Zanoni, F.A. Fontana, A. De Lucia, R. Oliveto, Smells like teen spirit: improving bug prediction performance using the intensity of code smells, in: Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016), IEEE, Raleigh, North Carolina, USA, 2016, pp. 244–255, doi:10.1109/ICSME.2016.27.

[27] E. Murphy-Hill, N. Carolina, A.P. Black, An interactive ambient visualization for code smells, in: Proc. 5th international symposium on Software visualization (SOFTVIS '10), ACM, Salt Lake City, Utah, USA, 2010, pp. 5–14.

[28] M.W. Mkaouer, M. Kessentini, S. Bechikh, M. Cinnéide, A robust multi-objective approach for software refactoring under uncertainty, in: C.L. Goues, S. Yoo (Eds.), Search-Based Software Engineering, Vol. 8636 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 168–183, doi:10.1007/978-3-319-09940-8_12.

[29] L. Zhao, J. Hayes, Rank-based refactoring decision support: two studies, Innov. Syst. Softw. Eng. 7 (3) (2011) 171–189, doi:10.1007/s11334-011-0154-3.

[30] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, The qualitas corpus: a curated collection of java code for empirical studies, in: Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC 2010), IEEE Computer Society, 2010, pp. 336–345, doi:10.1109/APSEC.2010.46.

[31] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, J. Softw. Mainten. Evol. 23 (3) (2011) 179–202, doi:10.1002/smr.521.

[32] S. Olbrich, D. Cruzes, D.I.K. Sjoberg, Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2010), Timisoara, Romania, 2010, pp. 1–10, doi:10.1109/ICSM.2010.5609564.

[33] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, R. Wettel, iPlasma: an integrated platform for quality assessment of object-oriented design, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), IEEE, Budapest, Hungary, 2005, pp. 77–80. (Industrial & Tool Proceedings), Tool Demonstration Track.

[34] K. Nongpong, Integrating "Code Smells" Detection with Refactoring Tool Support, University of Wisconsin Milwaukee, 2012 Ph.D. thesis.

[35] R. Marinescu, Measurement and Quality in Objectoriented Design, Department of Computer Science, "Politehnica" University of Timisoara, 2002 Ph.d. thesis.

[36] S. Baccianella, A. Esuli, F. Sebastiani, Evaluation measures for ordinal regression, in: 2009 Ninth International Conference on Intelligent Systems Design and Applications, 2009, pp. 283–287, doi:10.1109/ISDA.2009.230.

[37] J. Cardoso, J.P.D. Costa, Learning to classify ordinal data: the data replication method, J. Mach. Learn. Res. 8 (2007) 1393–1429.

[38] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The weka data mining software: an update, SIGKDD Explor. Newsl. 11 (2009) 10–18, doi:10.1145/1656274.1656278.

[39] M.R. Berthold, Mixed fuzzy rule formation, Int. J. Approx. Reason. 32 (2) (2003) 67–84, doi:10.1016/S0888-613X(02)00077-4.

[40] T.R. Gabriel, M.R. Berthold, Influence of fuzzy norms and other heuristics on "mixed fuzzy rule formation", Int. J. Approximate Reasoning 35 (2) (2004) 195–202, doi:10.1016/j.ijar.2003.10.004.

[41] J.C. Shafer, R. Agrawal, M. Mehta, SPRINT: a scalable parallel classifier for data mining, in: Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96), Morgan Kaufmann Publishers Inc., Bombay, India, 1996, pp. 544–555.

[42] F.A. Fontana, A. Caracciolo, M. Zanoni, DPB: a benchmark for design pattern detection tools, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, IEEE Computer Society, Szeged, Hungary, 2012, pp. 235–244, doi:10.1109/CSMR.2012.32.