Evaluating the Accuracy of Machine Learning Algorithms on Detecting Code Smells for Different Developers

Mário Hozano[1], Nuno Antunes[2], Baldoino Fonseca[3] and Evandro Costa[3]
[1]*COPIN, Federal University of Campina Grande, Paraíba, Brazil*
[2]*CISUC, Department of Informatics Engineering, University of Coimbra, Portugal*
[3]*IC, Federal University of Alagoas, Maceió-Alagoas, Brazil*

Keywords: Code Smell, Machine Learning, Experiment.

Abstract: Code smells indicate poor implementation choices that may hinder the system maintenance. Their detection is important for the software quality improvement, but studies suggest that it should be tailored to the perception of each developer. Therefore, detection techniques must adapt their strategies to the developer's perception. Machine Learning (ML) algorithms is a promising way to customize the smell detection, but there is a lack of studies on their accuracy in detecting smells for different developers. This paper evaluates the use of ML-algorithms on detecting code smells for different developers, considering their individual perception about code smells. We experimentally compared the accuracy of 6 algorithms in detecting 4 code smell types for 40 different developers. For this, we used a detailed dataset containing instances of 4 code smell types manually validated by 40 developers. The results show that ML-algorithms achieved low accuracies for the developers that participated of our study, showing that are very sensitive to the smell type and the developer. These algorithms are not able to learn with limited training set, an important limitation when dealing with diverse perceptions about code smells.

1 INTRODUCTION

Code smells are poor implementation choices that often worsen software maintainability (Fowler, 1999). Studies (Khomh et al., 2009a; Fontana et al., 2013) have found that the presence of *code smells* may lead to design degradation (Oizumi et al., 2016), increasing effort for comprehension (Abbes et al., 2011), and contributing to introduction of faults (Khomh et al., 2011a). Therefore, these *smells* must be carefully detected and refactored in order to improve the software longevity (Rasool and Arshad, 2015).

Several studies indicate that Machine Learning algorithms (ML-algorithms) (Witten and Frank, 2005) are a promising way to automate part of the code smell detection process, without asking the developers to define their own strategies of code smell detection (Khomh et al., 2011b; Maiga et al., 2012; Amorim et al., 2015). In a nutshell, the MLalgorithms require a set of code examples classified as smell or non-smell, i.e. the *training set* or the *oracle*. From them, the learning algorithms generate smell detection models based on the software metrics, aiming at detecting smells.

In this context, studies evaluated the accuracy of *Bayesian Belief Networks (BBNs)* (Khomh et al., 2011b), *Support Vector Machine (SVM)* (Maiga et al., 2012), and *Decision Trees* (Amorim et al., 2015), in the detection of code smells in existing software projects. Moreover, Fontana *et al.* (Fontana et al., 2015) performed a broader study on comparing the efficiency of these learning algorithms on detecting code smells. The efficiency was evaluated in terms of the detection accuracy and the training effort (i. e., the number of examples) required to the algorithm to reach a high accuracy.

However, the abstract definitions of smell types in existing catalogs (e.g. (Fowler, 1999)) lead developers to have different perceptions about the occurrence of smells (Mäntylä and Lassenius, 2006; Schumacher et al., 2010; Santos et al., 2013). For instance, while a developer may consider that a method containing more than 100 lines of code as being a Long Method smell (Fowler, 1999), other developers may not necessarily agree. As a consequence, the developers have to customize such definitions to their specific context.

In this case, to use the ML-algorithms, it is ne-

Hozano, M., Antunes, N., Fonseca, B. and Costa, E.
Evaluating the Accuracy of Machine Learning Algorithms on Detecting Code Smells for Different Developers.
DOI: 10.5220/0006338804740482
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 2*, pages 474-482
ISBN: 978-989-758-248-6

cessary that developers provide examples that represent their understanding about smells. However, there is still no knowledge about the accuracy of ML-algorithms on detecting smells for developers that may perceive such anomalies differently. Indeed, the existing studies performed evaluations on datasets containing code examples annotated by a single person or a small number of developers that shared the same perception about the analyzed code smells.

In this paper we present a study aimed at evaluating the accuracy of ML-algorithms on **detecting code smells for different developers**, and that may have different perceptions about code smells. We analyzed the 6 widely used algorithms on detecting 4 types of code smells from three different perspectives: the overall accuracy, the accuracy for different developers, and the learning efficiency (the number of examples necessary to reach a specific accuracy). For this study we built a **dataset based on the input of 40 diverse developers** which classified 15 code snippets according to their individual perspective. This resulted in dataset containing 600 (non-)smells examples representing the individual perception of all 40 developers involved in our study.

The results of our study indicate that, in average, the analyzed ML-algorithms were not able to reach a high accuracy on detecting code smells for developers with different perceptions. The results also indicate that the **detection process is very sensitive not only the smell type but also the individual perception of each developer**. These two factors must be considered in the design of approaches that aim to accurately detect code smells.

2 RELATED WORK

The use of intelligent techniques has been widely investigated in order to deal with the different perceptions concerning code smells (Mäntylä and Lassenius, 2006; Schumacher et al., 2010; Santos et al., 2013). In this context, several machine learning algorithms have been adapted in order to enable an automatic detecting customization, based on a set of examples manually validated (Khomh et al., 2011b; Maiga et al., 2012; Amorim et al., 2015; Fontana et al., 2015). Although these studies report important results concerning the efficiency of techniques based on Machine Learning (ML) algorithms, they do not discuss about how such techniques deal on customizing the detection for different developers. After all, the studies did not evaluate the efficiency of these techniques when customized from different training sets validated individually by single developers.

Bayesian Belief Network (BBN) algorithm has been proposed to detect instances of *God Class* (Khomh et al., 2009b). The authors used 4 graduate students to validate, manually, a set of classes, reporting if each class contains a *God Class* instance or not. From such procedure was created a single oracle containing 15 consensual smell instances. After performing a 3-fold cross-validation procedure in order to calibrate the BBN, the authors assessed an accuracy (precision) of 0.68 on detecting *God Class* instances. In (Khomh et al., 2011b) the same authors extended their previous work, by applying the BBNs to detect three types of code smells. By following the same procedure of their previous work, the authors submitted 7 students to create a single oracle. After calibrating, the produced BBN reached an accuracy of almost 0.33.

The work presented in (Maiga et al., 2012) evaluated the efficiency of a technique based on a *Support Vector Machine* (SVM) to detect code smells. In order to evaluate the proposed technique, the authors consider the some oracles defined in a previous work (Moha et al., 2010). Although defined by several developers, such oracles are not indexed by their evaluators. After considering these oracles, the SVM approach reached, in average, accuracies up to 0.74.

In (Amorim et al., 2015) the authors proposed the use of *Decision Tree* algorithm to detect code smells. The authors used a single training set containing a huge number of examples and validated by different developers. The work used a third party dataset as an oracle for the training the modules. At end, the paper reported the algorithm was able to reach accuracies up to 0.78.

More recently, Fontana *et al.* (Fontana et al., 2015) presented a large study that compares and experiments different configurations of 6 ML-algorithms on detecting 4 smell types. For training, the authors considered a set of oracles composed of several examples of code smells manually validated by different developers. However, these oracles did not identify these developers. As results, the authors reported that all evaluated techniques present a high accuracy. The highest one was obtained by two algorithms based on *Decision Trees* (*J48* and *Random Forest* (Mitchell, 1997)). In addition, the authors also affirmed that were necessary a hundred training examples to the techniques reach an accuracy of, at least, 0.95.

3 STUDY DESIGN

The main goal of this study was to evaluate the ability of Machine Learning (ML) algorithms to detect code

smells for different developers, considering their individual perception about code smells. Although previous work (Fontana et al., 2015) studied the accuracy and efficiency of ML-algorithms, the different perspectives of the developers was not considered. In summary, the study tries to answer the following research questions:

- **RQ1**: *How accurate are the ML-algorithms in detecting smells?*
- **RQ2**: *How do the ML-approaches deal with different perceptions about code smells?*
- **RQ3**: *How efficient are the ML-algorithms on detecting smells?*

To answer these questions, an experimental approach was defined based on the usage of machine learning algorithms in code with and without smells. An overview of the approach is presented in Figure 1.

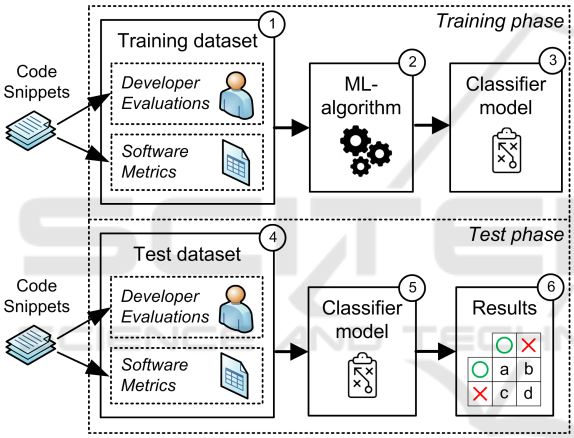


Figure 1: Training and testing the ML-algorithms.

The approach is based on two main parts: training phase and testing phase. The **training dataset (1)** is composed of a set of evaluations provided by developers, reporting the presence (or absence) of a smell in code snippets, and the software metrics extracted from those snippets. Next, the **ML-algorithm (2)** is trained to learn which metrics may determine the smell evaluations given as input. Based on this learning, the algorithm produces a **model (3)**, able to classify any snippet as a smell or non-smell.

In a test phase, the classifier model are tested to verify if it is able to classify novel instances as the developer does. For that, **test dataset (4)** is produced with evaluations and software metrics related to other code snippets. Next, the **classifier model (5)** tries to predict the developer's evaluations. Finally, classifications are compared with the developer's answers to obatin the **results (6)**.

3.1 Smell Types and ML-algorithms

In our study, we analyzed four types of code smells, described in Table 1. We chose these smell types for two main reasons. First, they affect different scopes of a program. While *God Class* and *Data Class* affect mainly classes, the *Long Method* and *Feature Envy* are related to methods. Second, these four smell types were also evaluated by previous studies on code smell detection (Khomh et al., 2011b; Maiga et al., 2012; Amorim et al., 2015; Fontana et al., 2015).

Table 1: Types of Code Smells investigated in this study.

Name	Description
God Class	Classes that tend to centralize the intelligence of the system
Data Class	Classes that have fields, getting and setting methods for the fields, and nothing else
Long Method	A method that is too long and tries to do too much
Feature Envy	Methods that use more attributes from other classes than from its own class, and use many attributes from few classes

We evaluated and compared 6 ML-algorithms, which reached a high accuracy in detecting code smells during the experiments presented in (Fontana et al., 2015). Initially, we applied these algorithms by following the same configuration adopted (Fontana et al., 2015) and then we tried other configurations in order to find one in which the analyzed algorithm could reach its highest efficiency. We present below a short description of the algorithms analyzed in our study:

- **J48**: an implementation of the C4.5 decision tree.
- **JRip**: an implementation of a propositional rule learner.
- **Random Forest**: a classifier that builds many classification trees as a forest of random decision trees.
- **Naive Bayes**: a simple probabilistic classifier based on applying Bayes' theorem.
- **SMO**: an implementation of John Platt's sequential minimal optimization algorithm to train a support vector classifier.
- **SVM**: an integrated software for support vector classification.

We used the Weka (Hall et al., 2009) implementation of these ML-algorithms. In order to provide a fair comparison, we created an environment in which all algorithms were executed by considering a same set of data in each execution.

3.2 Building the Oracle Datasets

We built a set of data containing examples of (non-)smells according to the developers' input. For this, we sent an invitation message to several contacts from academy and industry. We were able to recruit 40 developers with at least 3 years experience in software development and code smell detection, which had emphasis on software quality.

The selected developers evaluated a set of code snippets by looking for a specific smell type. The code snippets comprehend methods or classes suspects of containing the smell type under analysis. For example, to recognize the occurrence of *God Class*, the developers analyzed classes that could contain this smell type. The developer concluded each classification by answering: **`YES`**, if he agrees that the code snippet contained the specified smell type; and **`NO`**, otherwise. The code snippets used in our study were extracted from GanttProject[1] (v2.0.10), an open source Java project. We selected this project because they had been used in other studies related to code smells (Moha et al., 2010; Khomh et al., 2011b; Fontana et al., 2015). Moreover, such studies reported a variety of suspicious code smells in this project. In this context, the code snippets that were evaluated in our study counted with several of these reported instances.

In order to create the datasets that were considered in our study, we divided the `40` developers into `4` groups. Each group, composed of `10` participants, was responsible for analyzing one of the smell types described in Table 1. To this aim, each group analyzed a set of `15` code snippets, and each participant of the group classified each snippet indicating the occurrence or not of the smell type assigned to the group. Each set of `15` evaluations performed by a single developer detecting a given code smell formed an oracle that was used to train the ML-algorithms in our study. This way, the `600` classifications formed, in total, **`40` oracle datasets**.

To verify if the participants had different perceptions about smells, all the developers of a group analyzed the same set of `15` code snippets. From that, we could assess the agreement degree among the 10 developers that analyzed a single code smell type. Such agreement was calculated by using the *Fleiss*' Kappa, which is a measure to evaluate the concordance or agreement among multiple raters (Fleiss, 1971). This measure reports a number lower or equal to **`1`**. A *Kappa* value equal to *1*, means that the raters are in complete agreement. The categories presented in (Landis and Koch, 1977), propose that a low agreement is verified in values close or below 0, and that values above 0.6 represent substantial agreement.

[1] http://www.ganttproject.biz/

Table 2: Inter-rater agreement for each code smell type.

Code Smell	Agreement	Strength defined by (Landis and Koch, 1977)
God Class	0.308	Fair
Data Class	0.421	Moderate
Long Method	0.322	Fair
Feature Envy	0.222	Fair

After collecting all developers' evaluations and assessed the Kappa measure, we verified the inter-rater agreement for each smell type as described in Table 2. The developers that evaluated the *Data Class* smell, presented the higher, albeit still weak, agreement. They reached an *Kappa* value equals to 0.421, that is considered a *Moderate* agreement. All the remaining smells were evaluated by developers that presented a *Fair* agreement. The *Kappa* results for them varied from 0.222 (*Feature Envy*) to 0.308 (*God Class*). From such results, we confirmed that the developers that built the oracle datasets present different perceptions about the smells analyzed in our study. Such fact enable us to analyze the accuracy of the ML-algorithms on detecting code smells from these oracles aiming at answering our research questions.

3.3 Experimentation Phase

Using the datasets containing the developers' evaluations and the software metrics for each analyzed snippet, we performed two different experiments trying to answer the research questions, as follows:

1) Accuracy – Analyzes the accuracy of the ML-algorithms globally and by developer. The algorithms are used to produce a classifier model for each developer that evaluated a single smell type. In this scenario we used each one of the 40 oracle datasets (10 for each smell) as input through a 5-fold cross validation procedure. The creation of the experimented folds was guided to balance the number of code smell instances for each fold. As each participant classified manually each oracle used in our experiment, we could not control that such oracle was composed of a closest number of smell and non-smell instances. Moreover, in order to limit over-fitting problems, we repeated this procedure 10 times. In the end, we measured the mean of the accuracy reached by the produced classifier models on detecting the smell instances over the remaining fold.

2) Efficiency – Analyzes the accuracy of the ML-approaches according to the number of examples used as training. For each oracle dataset, we randomly ordered their evaluations and ran the ML-algorithms

with a varied number of these evaluations. Next, we executed the ML-algorithms 15 times, where the *n*-execution considers as learning the *n* first evaluations in the ordered dataset. The produced classifier models were tested on the entire dataset, containing the 15 examples. At end of each execution, we assessed the accuracy of the classifier model defined by the algorithms over the dataset. Each procedure with the 15 executions were repeated 10 times, in order to limit overfitting problems. Again, we measure the mean of accuracy reached by each algorithm on considering all executions.

4 RESULTS AND DISCUSSION

This section presents and discusses the main results of the study.

4.1 Overall Accuracy

Figure 2 presents the results concerning the mean values of accuracy obtained by the ML-algorithms on detecting the smell types *God Class*, *Long method*, *Data Class* and *Feature Envy*. We attach the exact *mean* value to the bar associated with each algorithm. The legend on top of the figure describes the colors used to identify each algorithm.

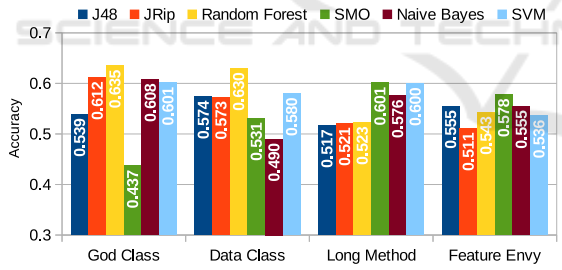


Figure 2: Mean of accuracy reached by the ML-algorithms on detecting smells.

As we can observe, the results of each algorithm in each smell type are quite diverse. In fact, the algorithms that present better results in a smell present in many cases much lower results for other smells. Furthermore, none of the ML-algorithms analyzed in our study was able to reach a mean above 0.635.

Random Forest presented the best results on detecting *God Class* and *Data Class*, but SMO performed better in *Long Method* and *Feature Envy* smells. These results suggest that the accuracy of the algorithms may be influenced by the scope of the type of code smell analyzed. While the *God Class* and *Data Class* are related to classes, the *Long Method* and *Feature Envy* are more related to methods.

Regarding the **God Class**, the SMO reached only ≈0.44, which is the lowest mean value of accuracy observed in all analysis. On the other hand, the *Random Forest* reached a mean 45% higher than SMO by reaching ≈0.64, which also correspond to highest observed mean value. JRip, Naive Bayes and SVM were still able to reach values greater than 0.6. Finally, J48 reached ≈0.54, which is an intermediate value between the mean obtained by the SMO and the other algorithms.

Similarly, Random Forest also reached the highest mean value of accuracy on detecting **Data Class**. Indeed, Random Forest was the only algorithm to reach a mean greater than 0.6. The algorithms J48, JRip, SMO and SVM reached values between 0.57 and 0.58. The lowest mean 0.49 was obtained by the Naive Bayes.

For **Long Method**, Random Forest was not able to reach the highest mean. Regarding this smell type, the highest mean values were obtained by the algorithms SMO and SVM, both reached ≈0.60. Next, the algorithm Naive Bayes reached ≈0.58. The remaining algorithms reached values around ≈0.52.

Regarding the **Feature Envy**, we observe that even though none of the algorithms was able to reach a mean above 0.6 on detecting this smell type, they did not reach a mean below 0.5. While the highest mean 0.578 was obtained by the SMO, the JRip reached the lowest mean 0.511.

The analysis of these results provides us with the answer to the research question **RQ1**. **The results indicate that, in average, the algorithms were not able to detect smells with high accuracy for developers with different perceptions.** In fact, we observe that the accuracy of each ML-algorithm depends on the type of smells being studied.

4.2 Accuracy by Developer

After analyzing the overall accuracy obtained by the ML-algorithms, we decided to perform a deeper investigation on the accuracy reached by each algorithm in order to better understanding how these algorithms deal with the individual perception of each developer.

Table 3 presents the results obtained. The first column describes the smell type in analysis. The second column identifies the developer that evaluated the code snippets related to each smell type, as described in Section 3.2. The following six columns report the accuracy reached by each ML-algorithm. Besides the accuracy values, we also report the *mean* of the accuracy values obtained by each algorithm on detecting the smell type in analysis. From the results described in Table 3, we can identify the accu-

racy obtained by each algorithm on detecting smells for each developer. In particular, we can recognize the algorithms that reached higher accuracy to a specific developer. In order to improve our discussion, we highlight (in gray) the cells containing the higher accuracies for each developer in the table. In addition, the lowest accuracy reached by the algorithms on detecting a given smell are marked in red.

God Class. The SVM reached an accuracy higher than the other algorithms on detecting *God Class* to the developers 1, 3 and 7-10. However, it also obtained the lowest accuracy, among all the *God Class* analysis, on detecting smells to the *Developer 2*. On the other hand, the *Random Forest* reached higher accuracy only to the developers 2 and 4-6, but it obtained the highest value of accuracy, among all the *God Class* analysis, on detecting smells to the *Developer 4*. The SMO, JRip, J48 and Naive Bayes did not reach an accuracy higher than the other algorithms for any developer.

Data Class. Regarding the *Data Class*, even the Naive Bayes that reached the lowest mean, it could reach an accuracy equal or higher than the other algorithms on detecting this smell type to the *Developers 18 and 20*. We also observe that the SVM reached the same accuracy of the Naive Bayes to the developers 18 and 20. In addition, the SVM reached higher accuracy than the other algorithms to the *Developer 16*. However, we note that the SVM reached the lowest accuracy, among all the *Data Class* analysis, on detecting smells to the developer 17.

The Random Forest, which obtained the highest mean, could reach an accuracy equal or higher than the other algorithms only to the developers *13, 14* and *17*. Note the JRip obtained the same accuracy of the Random Forest to the *Developer 14*. Moreover, the JRip reached a higher accuracy to the developers *15* and *19*. In the case of the *Developer 19*, the JRip obtained the highest accuracy, among all the Data Class analysis. Finally, the J48 obtained higher accuracy than the other algorithms to the developers 11-12.

Long Method. The SVM and SMO reached the highest means on detecting *Long Method* to the developers. While the SVM could reach higher accuracy than the other algorithms only to the *Developer 30*, the SMO reached higher accuracy to five developers: 21, 22, 26, 28 and 29. However, the SMO obtained the lowest accuracy among all the algorithms on detecting *Long Method* to the *Developer 27*.

The J48 and Random Forest reached higher accuracy than the other algorithms on detecting *Long Method* to the developers 25. In addition, the J48 obtained higher accuracy to the *Developer 23*. Finally, the Naive Bayes obtained higher accuracy to the *Developers 24 and 27*.

Table 3: Accuracy of the ML-algorithms on detecting smells for each developer.

	Dev	Accuracy					
		J48	JRip	RF	SMO	NB	SVM
GC	1	0.407	0.633	0.493	0.400	0.640	0.667
	2	0.547	0.647	0.707	0.587	0.493	0.360
	3	0.473	0.573	0.593	0.387	0.593	0.667
	4	0.640	0.667	0.760	0.433	0.747	0.667
	5	0.587	0.667	0.753	0.420	0.720	0.667
	6	0.533	0.580	0.747	0.467	0.500	0.387
	7	0.500	0.580	0.467	0.427	0.547	0.600
	8	0.567	0.540	0.607	0.440	0.647	0.667
	9	0.480	0.593	0.627	0.373	0.613	0.667
	10	0.660	0.640	0.600	0.440	0.580	0.667
	Mean	0.539	0.612	0.635	0.437	0.608	0.601
	Dev	J48	JRip	RF	SMO	NB	SVM
DC	11	0.693	0.673	0.513	0.453	0.380	0.600
	12	0.693	0.400	0.640	0.533	0.533	0.533
	13	0.420	0.480	0.613	0.413	0.433	0.600
	14	0.627	0.813	0.813	0.627	0.420	0.600
	15	0.627	0.780	0.733	0.640	0.400	0.600
	16	0.480	0.433	0.453	0.420	0.540	0.600
	17	0.473	0.520	0.640	0.500	0.453	0.333
	18	0.460	0.340	0.600	0.427	0.667	0.667
	19	0.660	0.840	0.753	0.633	0.407	0.600
	20	0.607	0.447	0.540	0.667	0.667	0.667
	Mean	0.574	0.573	0.630	0.531	0.490	0.580
	Dev	J48	JRip	RF	SMO	NB	SVM
LM	21	0.487	0.500	0.500	0.627	0.600	0.600
	22	0.560	0.600	0.600	0.780	0.700	0.600
	23	0.627	0.600	0.480	0.487	0.520	0.600
	24	0.407	0.433	0.560	0.667	0.700	0.667
	25	0.593	0.527	0.593	0.547	0.480	0.533
	26	0.427	0.480	0.520	0.547	0.487	0.533
	27	0.427	0.473	0.400	0.373	0.747	0.667
	28	0.480	0.467	0.467	0.827	0.540	0.533
	29	0.527	0.493	0.507	0.747	0.600	0.600
	30	0.633	0.640	0.600	0.413	0.387	0.667
	Mean	0.517	0.521	0.523	0.601	0.576	0.600
	Dev	J48	JRip	RF	SMO	NB	SVM
FE	31	0.593	0.520	0.540	0.567	0.467	0.580
	32	0.453	0.547	0.447	0.473	0.273	0.600
	33	0.693	0.593	0.607	0.620	0.467	0.587
	34	0.547	0.467	0.547	0.527	0.613	0.420
	35	0.613	0.533	0.513	0.540	0.533	0.600
	36	0.600	0.600	0.587	0.587	0.687	0.600
	37	0.533	0.413	0.580	0.740	0.760	0.453
	38	0.527	0.533	0.660	0.653	0.760	0.593
	39	0.453	0.360	0.407	0.420	0.453	0.327
	40	0.533	0.547	0.547	0.653	0.533	0.600
	Mean	0.555	0.511	0.543	0.578	0.555	0.536

GC-God Class, DC-Data Class, LM-Long Method, FE-Feature Envy

Feature Envy. Regarding the *Feature Envy*, the SMO reached higher accuracy than the other algorithms only to the *Developer 40*. Similarly to the SMO, the SVM also reached higher accuracy only to a specific developer (*32*).

We also note that the Naive Bayes could reach hig-

her accuracy than the other algorithms to five developers: 34 and 36-39. However, it also obtained the lowest accuracy among all the algorithms on detecting *Feature Envy* to the *Developer 32*. Finally, the J48 obtained a higher accuracy to four developers: 31, 33, 35 and 39.

These results help us to answer the **RQ2** presented on Section 3. Such results indicate that each algorithm reached a wide range of accuracy values on detecting a same smell type for different developers. In fact, the results show that **the accuracy obtained by each algorithm is very sensitive to the smell type and the developer**. To make matters worse, some algorithms that reached higher accuracy than the other algorithms on detecting a specific smell type to a developer, also obtained the lowest accuracy on detecting the same smell type to other developers.

4.3 Efficiency

In this section, we analyze the efficiency of the ML-algorithms, i.e., the number of examples required by each algorithm to reach its accuracy. Figure 3 presents the results of our evaluations that support the research question **RQ3**. The charts describe the learning curves that represent the efficiency reached by each algorithm on detecting the smell types *God Class*, *Data Class*, *Long Method* and *Feature Envy*, respectively. The *y-axis* represents the *mean* of the accuracy values obtained by each algorithm on detecting smells for different developers. The *x-axis* represents the number of examples used in the learning phase by the algorithm to reach such *mean*. For instance, in the case of the *God Class*, the algorithm SMO reached a mean accuracy of 0.450 by using only 7 examples. We ranged the number of examples from 1 to 15 because each developer analyzed 15 code snippets, as described in Section 3.2.

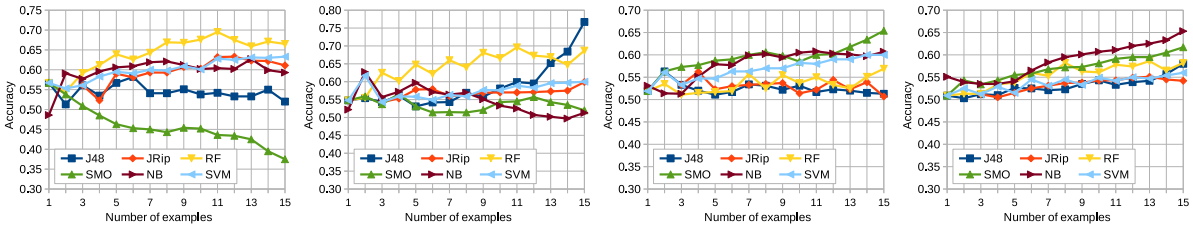
Concerning the **God Class**, all the analyzed algorithms reached a mean between 0.55 and 0.6 by using only one example. The only exception is the Naive Bayes (NB) that obtained a mean lower than 0.5. However, the efficiency of these algorithms suffered different variations as we increase the number of examples. We observe that the Random Forest (RF) was the only algorithm to reach mean values above 0.65. Indeed, none algorithm was able to overcome the mean obtained by the Random Forest when we considered more than three examples in the learning phase. On the other hand, the algorithm SMO presented the lowest mean by using more than three examples. In addition, we note that this algorithm tended to decrease its mean in almost all cases analyzed from 1 to 15 examples, reaching a mean lower than 0.4 by using 15 examples. Such result was a surprising for us because we expected that the algorithms could improve its accuracy as we increase the number of examples. The remaining algorithms reached an mean between 0.45 and 0.65.

Similarly, the *Random Forest* also reached the highest mean in the majority of the cases related to the **Data Class**. The only exceptions occurred when we considered 2 or more than 13 examples. In the first case, the SVM, SMO and Naive Bayes reached a mean equal or greater than the Random Forest. In the second case, the J48 overcame the Random Forest. In addition, we observe that the J48 presented a consistence increase in its accuracy after considering more than 12 examples. As a consequence, only the J48 could reach a mean above 0.7 among all the cases investigated in the four smell types analyzed in our study. Regarding the SVM and Naive Bayes, although they had reached the highest mean values by using only two examples, their mean values decreased on considering three examples. From this point, they could not reach a mean above 0.6 as we increase the number of examples.

For **Long Method**, the algorithm SMO reached mean values equal or greater than the other algorithms in almost all cases. The only exception occurred when we considered 10 examples, where the Naive Bayes overcame the SMO. We also note that the SVM presented a consistent increase in its accuracy from three examples, reaching a mean of 0.6. However, it was not able to overcome the SMO in any analyzed case. The algorithms J48 and JRip always reach values between 0.5 and 0.55 in the majority of the cases. Although the Random Forest had obtained the highest efficiency in the vast majority of the cases related to *God Class* and *Data Class*, it was not able to maintain its efficiency on detecting *Long Method*.

Regarding the *Feature Envy*, the algorithm Naive Bayes reached the highest mean in the most of the cases. The exceptions occurred when we considered from 2 to 5 examples, where the SMO reached mean values equal or greater than Naive Bayes. Even though the SMO has been overcome by the Naive Bayes in most of the analyzed cases, the SMO presented an increase in its accuracy as we considered more examples. Indeed, the SMO reached a mean greater than the remaining algorithms (J48, JRip, Random Forest and SVM) in the vast majority of the analyzed cases. The only exception occurred when considered 8 examples, where the SMO was overcome by the Random Forest and Naive Bayes.

The analysis of these results provides us with the answer to the research question **RQ3**. The results show that in most cases, **the algorithms are not**



(a) God Class (b) Data Class (c) Long Method (d) Feature Envy

Figure 3: Learning curves of the ML-algorithms on detecting smells.

accurate when trained with a small training set. In fact, the tendency in most cases is for increasing mean accuracy. The results also shows that the efficiency of the algorithm varies substantially depending on the type of code smells being evaluated.

5 THREATS TO VALIDITY

In this section we discuss the threats to validity by following the criteria defined in (Wohlin et al., 2000).

Construct Validity. The oracle datasets that supported our study were built with from a huge quantity of code snippets manually evaluated by developers. In this case, the developers evaluated each snippet by reporting the option "YES" or "NO", referring the presence or absence of a given code smell into the snippet. Providing only these two options may be a threat, since the developers could not inform the degree of confidence in their answers. However, we adopted such procedure aiming at ensuring that the developers were able to decide about the existence of a code smell and we could obtain a set of examples that enables to evaluate the efficiency of the ML-algorithms.

Internal Validity. The use of the Weka tool to implement the algorithms analyzed in our study enabled to experiment a variety of configurations, which affect the learning process. Thus, the configurations considered in our experiments may impact in the accuracy and efficiency of the algorithms. In order to mitigate this threat, we configured all algorithms according to the better settings defined in (Fontana et al., 2015). Indeed, that study performed a variety of experiments in order to find the best adjust for each algorithm.

External Validity. The code snippets evaluated by the developers were extracted from only 1 Java project, named GanttProject. Such project has been widely used in other works related to code smell (Khomh et al., 2011b; Moha et al., 2010; Maiga et al., 2012). However, although the implementation of this project presents classes and methods with different characteristics (i.e. size and complexity), our results might not hold to other projects. In the same way, even though we have performed our experiments with 40 different developers, our results might not also hold for other developers since they may have different perceptions about the code smells analyzed in our study (Mäntylä and Lassenius, 2006; Schumacher et al., 2010; Santos et al., 2013).

6 CONCLUSION

This paper presented the results of a study that evaluated the accuracy of ML-algorithms on the detection of code smells for different developers. This investigation is important because machine learning has been considering a promising way to customize the code smell detection. Nevertheless, to the best of our knowledge, there are no study that evaluate the performance of ML-algorithms on customizing the detection for developers with different perceptions about code smells.

We used 6 ML-algorithms to detect smells for 40 software developers that differently detected instances of 4 code smell types. Altogether, we counted with 600 evaluations, that supported our study. With such data we ran two different experiments aiming at analyzing the accuracy and efficiency of the analyzed algorithms.

According to our results, the ML-algorithms were not able to detect smells with high accuracy for the developers that participated of our study. In this context, the quantity of examples considered on training may difficult the learning. However, given that different developers have different perceptions about code smells, and the context may influence in their decision, the detection approaches must be able to customize with a limited portion of data to customize the detection. After all, in a real scenario it is not reasonable that a developer evaluates manually hundreds of code snippets before in order to customize a detection approach. This way, other strategies to speed up the customization performed by the ML-algorithms must be investigated.

Independently of the accuracy levels, we observed

that the ML-algorithms are sensitive to the smell type and the developer. For instance, while the SMO presented the higher accuracies on detecting *Long Method* and *Feature Envy* smells, such algorithm presented lowest accuracies on detecting *God Class* and *Data Class* instances. An, almost, inverted behaviour was verified by the Random Forest algorithm. Similarly, even when an algorithm presented a mean high accuracy on detecting a given smell, his performance was not consistent on on detecting such anomalies for the different developers.

Finally, we will make the dataset used in our experiments available in order to help other studies in smell detection. We had no knowledge of other datasets with a large portion of evaluations manually validated by different developers over a same set of code snippets. Thus, the availability of this dataset represents another contribution of this paper.

REFERENCES

Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pages 181–190. IEEE.

Amorim, L., Costa, E., Antunes, N., Fonseca, B., and Ribeiro, M. (2015). Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, ISSRE '15, pages 261–269, Washington, DC, USA. IEEE Computer Society.

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

Fontana, F. A., Ferme, V., Marino, A., Walter, B., and Martenka, P. (2013). Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. *2013 IEEE International Conference on Software Maintenance*, pages 260–269.

Fontana, F. A., Mäntylä, M. V., Zanoni, M., and Marino, A. (2015). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.

Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009a). An Exploratory Study of the Impact of Code Smells on Software Change-proneness. *2009 16th Working Conference on Reverse Engineering*, pages 75–84.

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., and Antoniol, G. (2011a). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275.

Khomh, F., Vaucher, S., Guéhéneuc, Y. G., and Sahraoui, H. (2009b). A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 305–314. IEEE.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., and Sahraoui, H. (2011b). Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *J. Syst. Softw.*, 84(4):559–572.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gueheneuc, Y.-G., and Aimeur, E. (2012). SMURF: A SVM-based Incremental Anti-pattern Detection Approach. *2012 19th Working Conference on Reverse Engineering*, pages 466–475.

Mäntylä, M. V. and Lassenius, C. (2006). *Subjective evaluation of software evolvability using code smells: An empirical study*, volume 11. Springer.

Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill series in computer science. McGraw-Hill, Boston (Mass.), Burr Ridge (Ill.), Dubuque (Iowa).

Moha, N., Gueheneuc, Y.-G., Duchien, L., and a. F. Le Meur (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Oizumi, W., Garcia, A., da Silva Sousa, L., Cafeo, B., and Zhao, Y. (2016). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 440–451, New York, NY, USA. ACM.

Rasool, G. and Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process*, pages n/a–n/a.

Santos, J. A. M., de Mendonça, M. G., and Silva, C. V. A. (2013). An exploratory study to investigate the impact of conceptualization in god class detection. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, pages 48–59, New York, NY, USA. ACM.

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., and Shaw, M. (2010). Building empirical support for automated code smell detection. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*, page 1.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA.