Empir Software Eng (2016) 21:1143–1191
DOI 10.1007/s10664-015-9378-4

Comparing and experimenting machine learning techniques for code smell detection

Francesca Arcelli Fontana · Mika V. Mäntylä · Marco Zanoni · Alessandro Marino

Published online: 6 June 2015

Abstract Several code smell detection tools have been developed providing different results, because smells can be subjectively interpreted, and hence detected, in different ways. In this paper, we perform the largest experiment of applying machine learning algorithms to code smells to the best of our knowledge. We experiment 16 different machine-learning algorithms on four code smells (Data Class, Large Class, Feature Envy, Long Method) and 74 software systems, with 1986 manually validated code smell samples. We found that all algorithms achieved high performances in the cross-validation data set, yet the highest performances were obtained by J48 and Random Forest, while the worst performance were achieved by support vector machines. However, the lower prevalence of code smells, i.e., imbalanced data, in the entire data set caused varying performances that need to be addressed in the future studies. We conclude that the application of machine learning to the detection of these code smells can provide high accuracy (>96 %), and only a hundred training examples are needed to reach at least 95 % accuracy.

Keywords Code smells detection · Machine learning techniques · Benchmark for code smell detection

Communicated by: Tim Menzies

F. Arcelli Fontana · M. Zanoni (✉) · A. Marino
University of Milano-Bicocca, Milano, Italy
e-mail: marco.zanoni@disco.unimib.it

F. Arcelli Fontana
e-mail: arcelli@disco.unimib.it

A. Marino
e-mail: a.marino4@campus.unimib.it

M. V. Mäntylä
University of Oulu, Oulu, Finland
e-mail: mika.mantyla@oulu.fi

M. V. Mäntylä
Aalto University, Helsinki, Finland

1 Introduction

Code smells were informally identified by Fowler and Beck (1999) as symptoms of possible code or design problems. Code smell detection has become an established method to discover source code (or design) problems to be removed through refactoring steps, with the aim to improve software quality and maintenance.

In the literature, the evidence whether code smells are harmful is conflicting. Some recent works are reporting that smells do not really affect change effort or fault-proneness (e.g., Olbrich et al. 2010; Sjøberg et al. 2013; Hall et al. 2014). For example, in Sjøberg et al. (2013) the file size is a better predictor of change effort than code smells. However, the advantage that code smells have over file size is that they come with specific refactorings to remove them. In the same way Sjøberg et al. (2013) points out that file size and code smells are significantly correlated, we think also that the reduction of file size can be achieved by refactoring out code smells. After all, reducing the file size is not a trivial problem.

At the same time, other works outline different results. For example, Moser et al. (2008) provide empirical, industrially based evidence that refactoring, in general and for removing code smells, increases development productivity and improves quality factors; Zazworka et al. (2011) have investigated questions regarding the impact of God Classes on software quality and they observed that God Classes are in general more change prone and in some cases more defect prone; Deligiannis et al. (2004) in their study observed that a design without a God Class will result in more completeness, correctness and consistency compared to designs with a God Class; Li and Shatnawi (2007) investigated the relationship between six code smells and the probability of error in classes in an industrial system, and found the presence of Shotgun Surgery to be connected with a statistically significant higher probability of defects; Yamashita (2014) investigated the relationship between code smells and the incidence of *problems* during software maintenance, and she recommends, based on her results, that future studies should focus on the interaction effect between code smells, between code smells and other design properties, and between code smells and the program size.

Probably, the fact that we can find in the literature so different results is due to the need to achieve, first of all, consistency in the definition of code smells. Until commonly used smell definitions are established, we are not able to achieve a mature knowledge of the impact of smells on software systems (Bowes et al. 2013).

This fact has obviously a great impact also on code smell detection. In fact, code smells can be subjectively interpreted (Mäntylä and Lassenius 2006), the results provided by detectors are usually different (Arcelli Fontana et al. 2012), the agreement in the results is scarce, and a benchmark for the comparison of these results is not yet available. For example, Bowes et al. (2013) describe the inconsistent measurement of Message Chains smell through the comparison of two tools for code smell detection, and Arcelli Fontana et al. (2012) describe the comparison of four code smell detection tools on six versions of a medium-size software project for six code smells and provide an assessment of the agreement, consistency and relevance of the results produced. Usually, detection rules are based only on the computation of a set of metrics, e.g., well-known object-oriented metrics or metrics defined ad hoc for the detection of a particular smell. The metrics used for the detection of a code smell by different tools can be different, according to their detection rules. Moreover, even if the metrics are the same, the thresholds of the metrics can change. By changing these values, the number of detected smells can increase or decrease accordingly. Another problem regards the accuracy of the results; many false positive smells can be detected, not representing real problems (and hence smells to be refactored), because information related to the context, the domain, the size and the design of the analyzed system is not usually considered (Ferme et al. 2013). Other detection tools are not based on metrics computation, but are able to detect the smell through the opportunities of applying refactoring steps to remove it (Tsantalis et al. 2009).

For all these reasons, and in particular the intrinsic informal and subjective definition of code smells, another way to provide a better-targeted detection would be to make it subjective to the particular user or community. Machine learning technology can be exploited to make a tool learn how to classify code smells by examples, easing the build of automatic code smell detectors.

The solution proposed in this work exploits supervised machine learning techniques, to support a learn-by-example process to construct code smell detection rules able to evaluate new candidates. Most techniques are able to provide a confidence value, telling how much results conform to the learned model. Some algorithms provide also human-readable rules, which have been analyzed to understand which combination of metrics has more influence on the detection and on the discovery of the inferred threshold values.

The application of machine learning to the code smell detection problem needs a formalization of the input and output of the learning algorithms, and a selection of the data to be analyzed and the algorithms to use in the experimentation. A large set of object-oriented metrics, covering different aspects of software design, have been computed on a large repository of heterogeneous software systems (Tempero et al. 2010). Metrics represent the independent variables in the machine learning approach. A set of code smells to detect has been identified, representing the dependent variables. For each code smell, a set of example instances have been manually evaluated and labeled as correct or incorrect (affected or not by a code smell). The selection and labeling phase of example instances plays an important role in machine learning techniques. Our approach selects the example instances by applying stratified random sampling on many projects, guided by the results of a set of pre-existing code smell detection tools and rules, called *Advisors*. This methodology ensures a homogeneous selection of instances on different projects and prioritizes the labeling of instances with a higher chance of being affected by a code smell. The selected instances are used to train a set of machine learning algorithms, to perform experiments evaluating the performance of different algorithms and to search for the best setting of their parameters.

The classification process was performed on four code smells (Data Class, God Class, Feature Envy, and Long Method) and 32 different machine learning algorithms (16 different algorithms plus their combination with a boosting technique). The experimented algorithms obtained high performances, regardless of the type of code smell. This paper extends our previous work (Arcelli Fontana et al. 2013b), where we described our preliminary results. The main contributions of this paper are:

- a methodology for the application of machine learning to address code smell detection tasks;
- an extensive experimentation (on 74 systems) for selecting the best algorithm and the respective parameters, for the detection of each of the considered smells.

The paper is organized through the following sections: in Section 2 we introduce related work on code smell detection techniques or approaches based on machine learning techniques; in Section 3 we introduce the smells considered in this work; in Section 4 we describe all the steps of our machine learning approach for code smell detection; in Section 5 we provide the results of our experimentations; in Section 6 we introduce some threats to validity and limitations of our approach; in Section 7 we discuss the application of machine learning for code smell detection and describe the next future developments of this research; finally, in Section 8 we make our conclusions.

2 Related Work

The past work on code smell detection can be mainly divided into two main categories. Firstly, there are rule-based approaches that rely mostly on metrics, plus, in certain cases, other rules related to the code structure and naming. Secondly, there are approaches using machine learning techniques, which are largely metrics based, such as the one presented in this paper. There are benefits and drawbacks in both approaches.

One can argue that the rule-based approaches are more sophisticated as they use more sources of information than the metrics based ML-approach, e.g., naming (Moha et al. 2010), structural rules (Moha et al. 2010), or even version history (Palomba et al. 2013). One could also think that it makes them better in code smell detection; yet, to the authors' best knowledge this has not been studied. On the other hand, rule-based approaches rely on human created rules that must be manually specified. For example, DÉCOR (Moha et al. 2010) requires that the rules are specified in the form of domain specific language and this specification process must be undertaken by domain experts, engineers or quality experts. Naturally, this rule creation requires effort from these individuals that could be spent in some other tasks. Whether the metrics based ML-approaches require less effort than rule-based approaches is however not clear, and it depends on two factors; (a) how complex rules one needs for the rule based approaches, and (b) how many training samples are needed for the metrics based machine learning approaches. At the moment, we are not aware of any studies comparing the approaches effort-wise. However, what remains a clear benefit for the metrics based ML-approach is the reduction of cognitive load required from the engineers. The rule-based approach requires that the engineers create specific rules of defining each smell. For the machine learning based approached the rule creation is left for the ML-algorithms requiring the engineers only to provide information whether a piece of code has a smell or not.

We do not think any code smell detection is superior to another and we think that several approaches may benefit code smell prevention, detection and fixing. To illustrate why multiple approaches are needed for code smells, we can look into Anti-spam techniques. Over 30 different approaches are presented for spam prevention in Wikipedia[1]; yet, most users still end up with several spam messages per day. We believe it is the same for code smells and it is unlikely that any single technique will completely solve the code smell problem.

Next, we present the prior works in the literature exploiting machine learning techniques for code smell detection. Maiga et al. (2012) introduce SVMDetect, an approach to detected anti-patterns, based on support vector machines (SVM). The subjects of their study are the Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife antipatterns, extracted from three open-source programs: ArgoUML, Azureus, and Xerces. Maiga and Ali (2012) extend the previous paper by introducing SMURF, which takes into account practitioners' feedback.

Khomh et al. (2009) propose a Bayesian approach to detect occurrences of the Blob antipattern on open-source programs (GanttProject v1.10.2 and Xerces v2.7.0). Khomh et al. (2011) present BDTEX (Bayesian Detection Expert), a Goal Question Metric approach to build Bayesian Belief Networks from the definitions of antipatterns and validate BDTEX with Blob, Functional Decomposition, and Spaghetti Code antipatterns on two open-source programs.

Yang et al. (2012) study the judgment of individual users by applying machine learning algorithms on code clones.

Kreimer (2005) proposes an adaptive detection to combine known methods for finding design flaws Large Class and Long Method on the basis of metrics with learning decision

[1] http://en.wikipedia.org/wiki/Anti-spam_techniques

trees. The analyses were conducted on two software systems: IYC system and the WEKA package.

As we can see, the principal differences of the previous works respect to our approach are that they did their experimentations by considering only 2 or 3 systems and they usually experiment only one machine learning algorithm. In our approach, we focus our attention on 4 code smells, we consider 74 systems for the analysis and our validation and we experiment 16 different machine learning algorithms (J48 Pruned, J48 Unpruned, J48 Reduced Error Pruning, JRip, Random Forest, Naïve Bayes, SMO RBF Kernel, SMO Poly Kernel, LibSVM C-SVC Linear Kernel, LibSVM C-SVC Polynomial Kernel, LibSVM C-SVC Radial Kernel, LibSVM C-SVC Sigmoid Kernel, LibSVM ν-SVC Linear Kernel, LibSVM ν-SVC Polynomial Kernel, LibSVM ν-SVC Radial Kernel, LibSVM ν-SVC Sigmoid Kernel), and considering also a boosted variant of the algorithms, the total number rises to 32. Additionally, we tested several hundreds of different parameter settings. In Table 1, we compare the previous approaches according to the detected smells or antipatterns, the used algorithms and number of analyzed systems.

3 Code Smells Definitions

In this work, we chose a set of four code smells to experiment our detection approach. We considered code smells among the ones having the highest frequency (Zhang et al. 2011), that may have the greatest negative impact (Olbrich et al. 2010) on software quality, and having detection rules defined in the literature or implemented in available tools (Arcelli Fontana et al. 2012). We decided to focus our attention on two smells at method level and two smells at class level as reported in Table 2, which correspond to four smells among the most frequent ones, as identified in a study on the frequency of 17 code smells on 76 systems (Arcelli Fontana et al. 2013a).

In Table 2, we outline also the impact of the considered smells on some quality dimensions, as outlined by Marinescu (2005) and extended by us to other smells. Below, the definitions of the selected code smells are reported. These definitions have been created starting from the literature, and basing on our previous experiences in code smell detection; they are taken as a reference for the whole work.

Table 1 Comparison prior work and this paper

Study	Smells	Algorithms	Feature selection	Boosting	Systems
Kreimer (2005)	2: Large Class, Long Method	1: Decision Tree	No	No	2
Khomh et al. (2009)	3: Blob, Functional Decomposition, and Spaghetti Code	1: Bayesian	No	No	2
Maiga et al. (2012) + Maiga and Ali (2012)	4: Blob (Large Class), Functional decomposition, Spaghetti code, Swiss Army Knife	1: SVM	n/a[a]	No	3
Yang et al. (2012)	Code Clones	n/a	n/a	No	n/a
This paper	4: Feature Envy, Data Class, Long Method, Large Class	16: see text,	No	Yes	74

[a] The figure in the paper indicates a smaller set of metrics was selected. However, no information of this feature selection process is given. Thus, the feature selection may have been even performed manually by humans. Regardless, the approach is impossible to replicate due to lack of information

Table 2 Selected code smells

Code smell	Affected entity	Impacted OO quality dimensions
Data Class	Class	Encapsulation, Data Abstraction
God/Large/ Brain Class	Class	Coupling, Cohesion, Complexity, Size
Feature Envy	Method	Data Abstraction
Long/God Method	Method	Coupling, Cohesion, Complexity, Size

3.1 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.

3.2 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.

3.3 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), to use more attributes from other classes than from its own class, and to use many attributes from few different classes.

3.4 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.

For each code smell, we identified in the literature a set of available detection tools and rules, which are able to detect them (see Section 4.3).

4 Towards a Machine Learning Approach

The application of machine learning to the code smell detection problem needs a formalization of the input and output of the learning algorithm and a selection of data and algorithms to be used in the experimentation. Figure 1 summarizes our data sets, code smell detection approaches and the resulting measures of our research.

The following points and the flow graph in Fig. 2 summarize the principal steps of our approach, while the remainder of the section describes them:

1. Collection of a large repository of heterogeneous software systems.

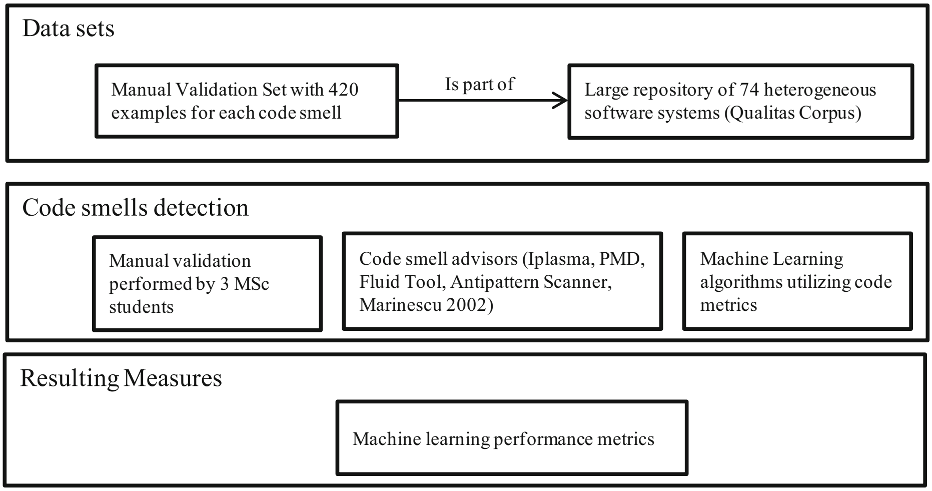

Data sets
Manual Validation Set with 420 examples for each code smell
Is part of
Large repository of 74 heterogeneous software systems (Qualitas Corpus)
Code smells detection
Manual validation performed by 3 MSc students
Code smell advisors (Iplasma, PMD, Fluid Tool, Antipattern Scanner, Marinescu 2002)
Machine Learning algorithms utilizing code metrics
Resulting Measures
Machine learning performance metrics

Fig. 1 Machine learning approach

2. Metrics Extraction: extract a large set of object-oriented metrics from systems at class, method, package and project levels. The metrics are considered as independent variables in the machine learning approach.
3. Choice of tools, or rules, for their detection; they are called *Advisors* in the following.
4. Application of the chosen Advisors on the systems, recording the results for each class and method.

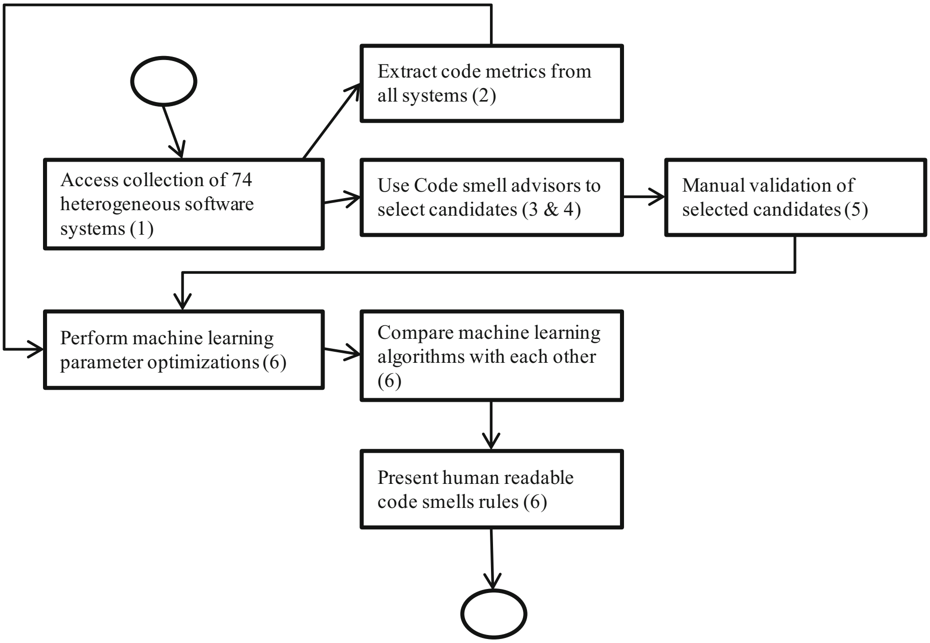

Extract code metrics from all systems (2)
Access collection of 74 heterogeneous software systems (1)
Use Code smell advisors to select candidates (3 & 4)
Manual validation of selected candidates (5)
Perform machine learning parameter optimizations (6)
Compare machine learning algorithms with each other (6)
Present human readable code smells rules (6)

Fig. 2 Flow graph of the research steps taken

5. Labeling: following the output of the Advisors, the reported code smell candidates are manually evaluated, and they are assigned different degrees of gravity.
6. Experimentation: The manual labeling is used to train supervised classifiers, whose performances (e.g., precision, recall, learning curves) will be compared to find the best one.

4.1 Collection of Software Systems

For our analysis, we considered the Qualitas Corpus (QC) of systems collected by Tempero et al. (2010). The corpus, and in particular the collection we used, 20120401r, is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. The systems selected for the analysis are 74. The reason of the selection is that the systems must be compilable to correctly compute the metrics values. We manually added all the missing third party libraries to each system, allowing the resolution of class dependencies.

The selected 74 systems have different sizes and belong to different application domains. Table 22 in the Appendix reports the size measures, application domains and release dates for each selected project; Table 3 reports the overall size measures of the selected projects.

A sufficiently high number of systems is fundamental to have a machine learning process that does not depend on a specific dataset, allowing to generalize the obtained results. At the best of our knowledge, the number of systems analyzed is the largest available, for code smell (or antipattern) detection with machine learning algorithms.

4.2 Metrics Extraction

We computed a large set of object-oriented metrics that are considered as independent variables in our machine learning approach. The metrics were computed on all the 74 systems of the QC.

The selected metrics are at class, method, package and project level; the set of metrics is composed of metrics needed by the exploited Advisors, plus standard metrics covering different aspects of the code, i.e., complexity, cohesion, size, coupling. We chose well-known metrics that are widely used in the literature (Bansiya and Davis 2002; Dubey et al. 2012; Chidamber and Kemerer 1994; Lorenz and Kidd 1994; Aggarwal et al. 2006). The chosen metrics are reported in Table 4, classified under six quality dimensions of object-oriented software. Metrics having a "*" in the name are customized versions of standard metrics, or slight modifications of existing metrics; metrics with a "§" suffix have been defined specifically for the detection of the Message Chain code smell (not considered in this paper). The extended names are reported in Table 23 in the Appendix and the descriptions of all metrics are reported on the web (http://essere.disco.unimib.it/reverse/MLCSD.html).

Furthermore, *Custom Metrics* (see Table 5) have been defined, to catch other structural properties of the source code. Their definitions are mainly based on the combinations of modifiers (e.g., public, private, protected, static, final, and abstract) on attributes and methods

Table 3 Summary of projects characteristics

Number of systems	Lines of code	Number of packages	Number of classes	Number of methods
74	6,785,568	3420	51,826	404,316

Table 4 Selected metrics

Size	Complexity	Cohesion	Coupling	Encapsulation	Inheritance
LOC	CYCLO	LCOM5	FANOUT	LAA	DIT
LOCNAMM*	WMC	TCC	ATFD	NOAM	NOI
NOM	WMCNAMM*		FDP	NOPA	NOC
NOPK	AMWNAMM*		RFC		NMO
NOCS	AMW		CBO		NIM
NOMNAMM*	MAXNESTING		CFNAMM*		NOII
NOA	WOC		CINT		
	CLNAMM		CDISP		
	NOP		MaMCL§		
	NOAV		MeMCL§		
	ATLD*		NMCS§		
	NOLV		CC		
			CM		

(e.g., the number of public and static attributes, or the number of abstract methods). Table 24 in the Appendix lists the full names of the Metrics.

All metrics have been computed through a tool we developed, which parses the source code of Java projects through the Eclipse JDT library. The tool is called "Design Features and Metrics for Java" (DFMC4J). It is designed with the aim of being integrated as a library in other projects. The user can request information on specific entities or on a category of entities (e.g., all the methods of a system, all classes of a system). We tested the computation of all the metrics we defined and computed. We used, as a reference test system, the "Personal Lending Library", a small sample application to manage a personal library of books (Kline 2013). The reliability of the tool has been described in detail in (Ferme 2013).

4.3 Choice of a Set of Code Smells Detection Rules or Tools

The application of supervised learning needs a training set, containing labeled instances. In our case, the label must specify if (or how much) a class (or method) is affected by a code smell. The set of correct label assignments is also called "oracle". As we have no external benchmark data yet, this oracle must be created by manual code analysis.

As we want to work on large and heterogeneous datasets, the creation of an oracle including all the extracted source code elements is not possible without massive human resources. This is a common situation, which leads to create datasets using a sampling approach. The simpler

Table 5 Custom metrics

NODA	NOPVA	NOPRA	NOFA
NOFSA	NOFNSA	NONFNSA	NOSA
NONFSA	NOABM	NOCM	NONCM
NOFM	NOFNSM	NOFSM	NONFMABM
NONFNSM	NONFSM	NONAM	NOSM
NOPLM	NOPRM	NOPM	NODM

Table 6 Code smells and advisors

Code smell	Advisors: detection tools or rules
God Class	iPlasma, PMD
Data Class	iPlasma, Fluid Tool, Antipattern Scanner
Long Method	iPlasma, PMD, Marinescu (2002)
Feature Envy	iPlasma, Fluid Tool

method is random sampling. It prescribes that the subjects to be selected should be chosen randomly from all the available ones.

In this domain, random sampling has a major drawback: code smells density in code is low, so the composition of a randomly sampled training set will have a great chance to contain only a few instances affected by code smells. Machine learning with such an unbalanced training set is very likely to produce bad results. In fact, many preprocessing techniques in the machine learning field are used to provide better-balanced datasets.

In our case, we chose to use a stratified sampling of the available instances, based on a set of hints, called *Advisors*. An Advisor is a deterministic rule, implemented locally or in an external tool, that gives a classification of a code element (class or method), telling if it is a code smell or not. The idea is that Advisors should approximate the label better than the random choice, and by aggregating their suggestions we should be able to sample more code elements affected by code smells.

Following this idea, we formulated some requirements, to determine how to select available Advisors from the literature:

- Different Advisors (for the same smell) should use different rules or approaches as much as possible; in this way, we avoid possible naive correlations among similar rules.
- An Advisor can be implemented as an available external tool. A rule implemented by an automatic detection tool has some advantages: it eliminates the possibility of misinterpretation of the rule definition, and reuses functionalities having (possibly) a wide diffusion and agreement. We considered only freely available tools.
- Advisors implemented in external tools must be available for batch computation, and they should export data in a parsable and documented format.
- Advisors can be also defined by research papers. In this case, the whole detection rule must be clearly described in the original source, to be implemented accordingly.

We did an analysis of the literature related to code smell detection, and related freely available tools. The Advisors selected for each code smell are reported in Table 6. Some well-known tools have not been considered, because it is not possible to run them in batch without a manual configuration of the projects to analyze, or they are not freely available, or they use detection rules that are too similar to other selected ones. For example, the JDeodorant (Tsantalis et al. 2009) tool has not been considered, because it is not possible (without customizations) to run it in batch without a manual configuration of the project that has to be analyzed.

In Table 6, we have considered as Advisors: PMD[2] and iPlasma (Marinescu et al. 2005), which are two free tools; Fluid Tool (Nongpong 2012) and AntiPattern Scanner (Wieman 2011), which have been described in research papers, but the tools are not available. Moreover,

[2] http://pmd.sourceforge.net/

we have considered a detection rule for the Long Method smell defined by Marinescu (2002). In these three last cases we implemented the detection rules, following the respective references. We choose to map two smells detected in iPlasma (God Class and Brain Class) to the God Class code smell in our approach, because they identify very similar concepts. The same has been done for Brain Method, detected by iPlasma, and Long Method.

4.4 Labeling

In this section, the usage of Advisors for code elements sampling is explained.

4.4.1 Sampling

Advisors, as already introduced, are used to provide hints of the presence of a code smell on a particular entity. Being them based on external tools or well-defined deterministic rules, we can apply them, and store their evaluation of all classes and methods. We consider an advice as *positive* when an Advisor reports an element as affected by a code smell, and *negative* otherwise.

Given that we have at least two Advisors for each code smell (and three Advisors for two smells), a possible way to provide a large oracle would be to aggregate the Advisor values, e.g., considering as code smells all classes or methods having at least half positive advices, or a number of positive advices higher than a given threshold. The problem is that the real performances of each single Advisor are unknown, i.e., we have to take into consideration that Advisors are subjected to error. In addition, we do not want to bias our oracle towards the rules implemented in the Advisors.

To comply with our requirements, we set up a stratified sampling method using the following procedure.

Consider having a separate dataset for each code smell, containing only the elements eligible for being affected by the smell (e.g., classes for God Class, methods for Long Method). Consider also that the value of each Advisor is available for all the elements of the dataset. The sampling procedure is organized in this way:

- Each dataset element is annotated with two values:

– the name of the project containing it;
– a number *N*, counting the number of Advisors reporting a positive evaluation, i.e., telling that they consider the element affected by the smell;

- Dataset elements are grouped by project and *N*. Groups are then sorted by project and *N*.
- Cycling the groups in the defined order, an instance *I* is randomly sampled and removed from each group. If a group is empty, it is skipped.
- Instance *I* is then evaluated, and added to the training set.
- Cycling restarts from the first group, until a target number of positive instances is found.

An overview of the procedure is depicted in Fig. 3. The procedure gives the same probability of selection to groups of instances belonging to different projects, and having different likelihoods (given by the Advisors) of being affected by a code smell. In this way, we increase the chance of building a dataset that represents different application domains (if the input projects belong to different domains), and with a sufficient number of affected entities, but keeping different characteristics (exploiting the *N* grouping parameter).

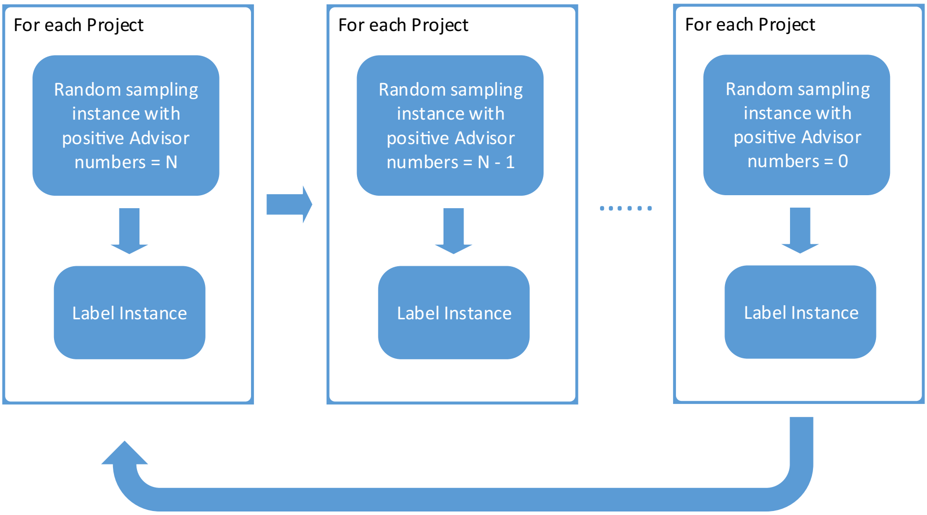

For each Project
Random sampling instance with positive Advisor numbers = N
Label Instance
For each Project
Random sampling instance with positive Advisor numbers = N - 1
Label Instance
......
For each Project
Random sampling instance with positive Advisor numbers = 0
Label Instance

Fig. 3 Labeling process

Finally, the obtained training set is normalized in size, by randomly removing (if needed) negative instances, until the balance of the training set is of $\frac{1}{3}$ positive (and $\frac{2}{3}$ negative) instances. This operation is performed because, usually, only a very small fraction of classes and methods is affected by a particular code smell. This fact results in highly unbalanced datasets, where the large majority of the considered elements are not affected by code smell. Machine learning algorithms tend to perform badly on very unbalanced dataset, so this kind of dataset preparation is popular (Guéhéneuc et al. 2004).

Using this procedure, we obtained four datasets (one for each code smell), each one composed of 140 positive instances and 280 negative instances (420 instances in total).

4.4.2 Label Assignment Criteria

The labeling is done by performing a severity classification of code smells based on an ordinal scale. The idea of a severity classification has been applied to software defects over the years and its industrial adoption is widespread. Several authors have recently worked on the prediction of defect severity by means of machine learning techniques (Menzies and Marcus 2008) (Lamkanfi and Demeyer 2010) (Tian et al. 2012). In this study, the code smell severity can have one of these 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 usage of code smell severity can have two possible benefits. First, the ranking of classes (or methods) by the severity of their code smells can help software developers prioritize their work on the most severe code smells. Developers might not have enough time to fix all the code smells that can be automatically detect. Second, the use of the code smell severity for labelling purposes provides more information than a more traditional binary classification. This information can be exploited during the learning phase, or collapsed back to a binary classification by grouping together the values, e.g., {0}→INCORRECT, {1, 2, 3}→CORRECT. It is important to understand that different machine learning algorithms have different capabilities, and choosing an appropriate labeling helps to achieve better results. In fact, selecting an algorithm which does not exploit the ordinal information in the class attribute is similar to perform binary classification over the (four in our case) possible values; as a consequence, each class value defines a reduced training set, with respect to the binary classification. For these reasons, we did not exploit the severity classification in our experiments, while we keep it for future work. For this work, we mapped severity to a binary class variable, following the schema proposed above.

The labeling evaluation was performed by three MSc students specifically trained for the task. The students independently studied the code smell definitions, and had a 2 h discussion about their comprehension of the definitions. Then they performed a short manual labeling exercise, and discussed their evaluations, to reach an agreement. The output of this setup phase was a set of guidelines for determining the most relevant aspects to consider for each code smell, which determine also the way labels are assigned. The guidelines are reported in the following.

Data Class:

- data classes must not contain complex methods;
- data classes can expose few non-accessor methods, and they must be very simple;
- data classes mainly expose accessor methods;
- the attributes of a data class must be either public or exposed through accessor methods.

God Class:

- god classes tend to access many attributes from many other classes; the number of attributes contained in other classes that are used from the class, considering also attributes accessed using accessors, must be high;
- how much the class centralizes the intelligence of the system;
- god classes usually contain large and complex methods;
- god classes are large;
- god classes usually expose a large number of methods.

Feature Envy:

- feature envies access many foreign attributes; the number of foreign (contained in other classes) attributes, directly used by a method, considering also attributes accessed through accessors, must be high;
- feature envies access more foreign attributes than local ones;
- feature envies mainly use the attributes of a small number of foreign classes.

Long Method:

- long methods tend to be complex;
- long methods access many attributes, a large part of the attributes declared in the class; the number of used variables, considering also attributes accessed through an accessor, must be high;

- long methods contain many lines of code;
- long methods tend to have many parameters.

During the manual labeling phase, each student individually evaluated each selected instance, by inspecting the code with the only support of the Eclipse IDE. No metric values were supplied to the students to evaluate the code, and they did not know the number of Advisors giving a positive evaluation. In the case individual evaluations were in conflict, an agreement was reached by discussion among the students, to decide which label to apply to the instances. For conflict, we mean that there was no total agreement among all students. The procedure was structured to reduce the bias given by the known sensibility of code smell detection based on single developer opinion (Mäntylä et al. 2004).

4.5 Experimentation Setup

We experimented our approach by selecting a set of suitable machine learning algorithms and testing them on the generated datasets, by means of 10-fold cross validation.

For each code smell type, a dataset has been created: two datasets for class-level smells (Data Class and God Class) and two for method-level smells (Feature Envy and Long Method).

In each dataset, each row represents class or method instances, and has one attribute for each metric. In addition, a boolean feature represents the label that shows whether the instance is a code smell or not (obtained by mapping the severity score, as explained in Section 4.4.2). Class and method instances have a different set of metrics; Fig. 4 shows the groups of attributes selected for each instance type.

For each instance (class or method), the metrics of the respective containers are included in the features. The containment relation defines that a method is contained in a class, a class is contained in a package, and a package is contained in a project. The inclusion of the metrics of containers allows exploiting the interaction (if existing) among the features of the classified element and the ones of its containers. The combination of different metrics could better discriminate the class that occurs in the training data, increasing the quality of the classification results as well.

In the set of supervised classification algorithms we selected, we can find algorithms used in the literature for software engineering tasks and algorithms able to produce human-readable models. Moreover, the set of selected algorithms covers many different machine learning approaches, i.e., decision trees, rule learners, support vector machines, bayesian networks. The

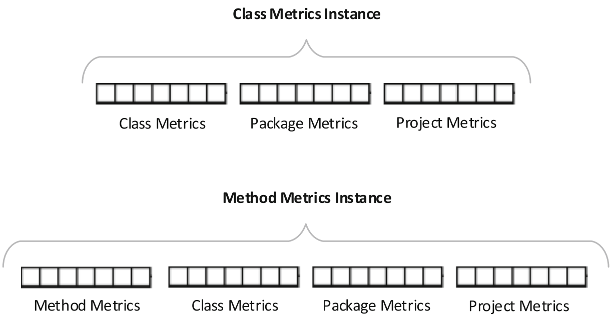


Fig. 4 Class and method metrics instance

algorithms' implementation is the one available in Weka (Hall et al. 2009). In the following, we report the list of the selected algorithms, with a short description:

- 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 listed classifiers were also combined with the AdaBoostM1 (Freund and Schapire 1996) boosting technique. This method combines multiple algorithms of the same type. Through an iterative process, each new model is influenced by the performances of those previously built. AdaBoostM1, at each iteration, builds the input model and assigns a greater weight to those instances that are harder to classify. Therefore, the final classification result depends on the weights given to the input instances.

The cross validation experiment has the goal of finding the best algorithm, in its best setup, with respect to some performance measures. We applied three standard performance measures: accuracy, F-Measure, and Area Under ROC. These three measures express different point of views of the performances of a predictive model, and use a single number to express a whole performance measure. This is in contrast, e.g., with precision and recall, which must be considered together to better understand the performances of a model.

A short description of these estimators is reported below.

Accuracy is the proportion of correctly classified instances in the positive and negative class. This performance index is one of the simplest performance measures for a classification task. Usually, accuracy is never reported alone because, when the positive and negative classes are unbalanced (small number of positive instances and a large number of negative instances) the result undergoes a severe distortion. That never occurs in our experiments, because all datasets are balanced.

The *F-measure* is defined (in the basic version) as the harmonic mean of precision and recall. Precision is the fraction of positive-classified instances that are really positive, whereas recall is the fraction of really-positive instances that are classified as positive. These two performance indexes tend to be in contrast with one another and the purpose of F-Measure is to find a trade-off.

The last performance index is the area under *ROC* (*Receiver Operating Characteristic*). ROC is a plot of true positive rate against false positive rate as the discrimination threshold of the classifier is varied. The area under ROC gets close to value 1 when the discrimination performs better, while a bad classification brings values close to 0.5.

Every algorithm was applied in different configurations to each dataset, by means of 10-fold cross validation, and the performances obtained on each fold were recorded. Then, by means of the Wilcoxon signed rank test (see Section 5.2), the performances were compared, to extract the set of models having the best performances.

5 Experimentations and Results

In this section, we describe the data we collected, the experiments we performed on them, and their outcomes.

5.1 Results of advisors and manual labeling process

Here we present the results of the labeling process that leads to the dataset used as an input for our machine learning algorithms. Section 4.4 previously described the labeling process.

5.1.1 Advisor Results

Table 7 shows the number and the percentage of instances for each dataset, according to the number of Advisors indicating a code smell, i.e., a positive result. It is evident that the number of instances with three positive results of Advisors is very low, because some Advisor rules (in particular for data class) are radically different. Therefore, the probability that all Advisors give positive outcomes is very low. When the number of positive Advisors results decreases, the number of the instances increases, without following a particular law.

As the table shows, the number of the instances is very large when the number of positive results of Advisors is equal to zero. This is due to the distribution of the class/method affected by code smells, which is unbalanced and in favor of the instances that are not code smells. However, the Advisors results are subject to false positive results that may significantly influence the obtained results. For example, the high number of instances detected as Feature Envy with a single Advisor suggests that there may be numerous false positives.

Table 7 Number of instances according to the number of smell detections of Advisor

Number of advisors indicating a smell	Number / percentage of smelly instances on datasets			
	Data class	God class	Feature envy	Long method
3	2 / 0.004 %	102 / 0.199 %	0	166 / 0.044 %
2	286 / 0.587 %	425 / 0.829 %	587 / 0.157 %	1514 / 0.404 %
1	4377 / 8.987 %	1319 / 2.573 %	89,580 / 23.902 %	3693 / 0.985 %
0	44,041 / 90.422 %	49,418 / 96.399 %	284,621 / 74.942 %	369,415 / 98.566 %

Table 8 Manually labeled instances according to the number of positive smell outcomes of Advisors

Number of advisors indicating a smell	Number of SMELLY/NON-SMELLY instances on datasets			
	Data class	God class	Feature envy	Long method
3	2/0	69/1	-	84/0
2	97/22	109/18	136/8	133/15
1	36/112	22/126	56/169	32/116
0	9/152	4/144	37/111	0/166

5.1.2 Manual Labeling of Results

The results of Table 7 are used to produce stratified samples for the manual labeling process, as explained in Section 4.4. Table 8 shows the number of instances manually labeled as affected or not affected by a smell for each dataset, with respect to the number of Advisors indicating a code smell in a particular instance.

The instances selected during the labeling process with three positive results of Advisors have been labeled as smelly, except one for God Class dataset. As expected, when the number of positive results of Advisor decreases, the number of smelly instances decreases. Even when the number of positive results of Advisor is zero, some instances have been labeled as smelly except for Long method, in which all the instances have been labeled as non-smelly. This result indicates for Long Method a strong agreement between the negative results of the Advisors and the results obtained with the manual validation.

Overall, 1986 instances were manually labeled (826 positive and 1160 negative instances), see Table 9. For each code smell, a dataset with 140 positive instances and 280 negative instances (420 instances) was selected. For each project, the positive label and negative instances are randomly sampled until they reach the desired number of labeled instances.

5.2 Training Phase with Search of Best Parameter Setting

The search for the best parameters of the machine learning algorithm is a difficult task in terms of computation complexity, time and efforts. Each algorithm has a set of parameters, each one having its domain, which can be of different types (i.e., continuous, discrete, boolean and nominal), making the entire set of parameters of a single algorithm a large space to explore. Furthermore, some parameters are more relevant than others and their optimal choice cannot be known a priori as they depend on the dataset.

Grid-search (Hsu et al. 2003) is the traditional way of exploring the parameter space and it was adopted. Additionally, there are several other techniques to perform parameter

Table 9 Evaluated instances summary on datasets

Dataset	Candidates	Evaluated	Correct	Incorrect
Data Class	48,706	430	144	286
God Class	51,264	493	204	289
Feature Envy	374,788	517	229	288
Long Method	374,788	546	249	297
Total		1986	826	1160

optimization, e.g., simulated annealing (Dekkers and Aarts 1991), genetic algorithms (Goldberg 1989). The grid search was chosen for two reasons. First, the other methods, which avoid doing an exhaustive parameter search by approximations or heuristics, are not totally safe, i.e., they do not guarantee that unexplored combinations are worse than the experimented ones, and it is not possible to understand the performances of non-optimal solutions. Second, the grid-search can be easily parallelized because each parameter configuration is independent. Many of the advanced methods are iterative processes that can be hardly parallelized. Thus, the grid-search technique is straightforward and very effective. Its only disadvantage is that it is time-consuming.

In the grid search, numeric parameters are discretized in a fixed number of values, equally distributed in a certain range; the possible values of nominal and boolean parameters are assigned to a set of discrete numbers. For each possible combination of parameters, an algorithm is tested by performing 10-fold cross-validation tests with 10 repetitions, one for each different randomization of the dataset (100 runs for each combination). The cross validation procedure has been selected because it is a standard, which guarantees a stratified sampling of the dataset (Bengio and Grandvalet 2004) (Stone 1974) and reduces the overfitting phenomenon (Cohen and Jensen 1997). A high number of repetitions is needed to guarantee a good statistical validity of performance values. Each run produces a set of performance values (i.e., accuracy, precision, recall) that are analyzed. We would like to underline that we treat classifier models as black-box implementations in this regards. The whole parameter selection procedure is set to avoid depending on the particular implementation issues of each algorithm. We simply picked up a widely known and free machine learning library as Weka and applied the selected techniques, which are well known. Our goal is to test known machine learning approaches on this problem, and not defining specific techniques for dealing with this kind of data.

In our experiments, we did not exploit data pre-processing strategies, and in particular we did not try to compare the results obtained through different pre-processing techniques. The only exception are SVM. This class of models is known to be sensible to the scale of data, and for this reason normalization or standardization are often applied in combination with these algorithms. For this reason, we added a normalized and a standardized version of the dataset when applying SVM. These alternatives were treated as additional categorical parameters (e.g., 0: no-process, 1: normalize, 2: standardize) in the parameter selection procedure. Additional pre-processing strategies (e.g., feature selection) were not taken into consideration, and may be experimented in future work; please see further discussion of this in Section Experiments Limitations 6.3.

Once the results of combinations of parameters for each algorithm are computed, it is necessary to choose comparison criteria and a parameter estimator for the comparison of values in order to find the best combination. There are several statistical techniques to compare classifiers performance (Demšar 2006), which are parametric and non-parametric tests. It is not always easy to identify the most suitable test in the context, because, according to the data distribution, a test may be more appropriate than another one.

The selected comparison standard is the Wilcoxon signed rank test (Hollander et al. 2000). It is a non-parametric test, which allows comparing each pair of configuration parameters of the algorithms. It can be used to test if the two algorithms' performances are equal or if one is greater than the other. If the test tells one algorithm configuration has higher performance than the other, the former is the winner and the other algorithm is the loser. If the test tells the two performances are not distinguishable, the algorithms neither win nor lose. Finally, the results of all tests produce a ranking. Configurations are ordered according to the difference between the number of winner and loser configurations. The selected parameter estimator value is accuracy: it represents the percentage of correctly classified instances, both in the positive and negative class. Because all our datasets are balanced, it is possible to use accuracy in order to compare the performance results. The tests have been performed with a significance level equal to 0.05.

Our comparisons were demanding in terms of computation efforts and resources, as some algorithms have a large number of different combinations of parameters and need to perform millions of comparisons before they can produce the ranking result. In order to overcome this issue, we created a filter capable of decreasing the number of comparisons to be tested. All parameter combinations where the average accuracy of a parameter combination was less than the total average accuracy of all parameter combinations were excluded. This allowed us to remove roughly 50 % of the parameter combinations, reducing the number of comparisons by roughly 75 %. With the remaining configurations, we performed first a ranking (wins minus losses) test of all configurations of each algorithm. The winner configuration of each algorithm was chosen to be compared in another ranking test with the winners of all the other algorithms. As a consequence, in the final round, 32 algorithms were compared for each code smell. This selection procedure reduced the number of performed tests to a manageable number.

For some algorithms, the ranking results gave no winners and no losers, whereas, for others, small subsets of configuration results have the same ranking values, making it difficult to determine a true winner. In these cases, the edit distance among parameter configurations has been calculated to select one of the configurations. This value is an integer number that represents the edit distance between each configuration of parameters and the default parameter configuration. The default parameter configuration has been chosen as a reference because default parameter values are well known and more stable than any other configuration. The configuration with the smallest distance form the default represents the best algorithm parameter combination.

5.3 Presentation of the Obtained Performances

Once the best combinations of parameters for each algorithm and dataset are found (32 algorithm for each datasets), the Wilcoxon signed rank tests are performed in order to find the best algorithm for each dataset, through a ranking given by the wins-losses difference between each couple of algorithms. In this section, we report the results of this analysis.

5.3.1 Algorithms – Winners

We report the top 10 ranked algorithms for each code smell in Tables 10, 11, 12 and 13. For each algorithm, we report the number of significant wins-losses difference against the other 31, and the respective percentage. Moreover, we add the Cliff delta effect size measure, in terms of value and magnitude. The effect size is computed considering, for each algorithm, its 100 accuracy values (treatment group) against the 3100 accuracy values of the remaining algorithms (control group).

The best algorithm on the Data Class dataset (Table 10) is B-J48 Pruned.[3] This algorithm wins 29 times against the other algorithms, with 94 % of victories. Random Forest and B-JRip are the second and third best algorithms, with 26 victories. The number of wins decreases up to the tenth algorithm that has 14 victories. The top 2 algorithms have a "large" effect size, meaning that there is a tangible advantage of using them instead of the remaining 30.

[3] B-J48 Pruned is the Pruned variant of J48, with AdaBoostM1 applied. All algorithms with a "B-" prefix in the name are boosted algorithms.

Table 10 The first 10 best algorithms on data class

Classifier	Wilcoxon victories		Cliff Delta effect size	
	Number	Percentage	Value	Magnitude
B-J48 Pruned	29	94 %	0,48	large
Random Forest	26	84 %	0,49	large
B-JRip	26	84 %	0,43	medium
B-J48 Unpruned	24	77 %	0,40	medium
B-Random Forest	24	77 %	0,37	medium
J48 Pruned	23	74 %	0,37	medium
J48 Unpruned	20	65 %	0,32	small
JRip	17	55 %	0,28	small
B-J48 Reduced Error Pruning	16	52 %	0,27	small
J48 Reduced Error Pruning	14	45 %	0,25	small

Naïve Bayes is the best algorithm on the God Class dataset (Table 11) with 24 (77 %) victories. All the three variants of J48 are the second best algorithms with 21 victories, and JRip, B-J48 Pruned, B-J48 Reduced Error Pruning and Random Forest are the third best algorithms with 20 victories. The last two best algorithms are B-Naïve Bayes and B-Random Forest with 19 victories. Only the first algorithm has a “medium” effect size, all the others are reported as “small”. On this dataset, it seems that the choice of the algorithm has no great influence on the results.

As regards the Feature Envy dataset (Table 12), the winning algorithm is B-J48 Pruned, with 26 victories, followed by B-J48 Unpruned, with 25 victories. B-Random Forest, B-JRip and J48 Unpruned algorithms are classified at the third place with 17 victories. The first nine ranked algorithms have “medium” effect size; the remaining one has “small” effect size. These top 10 algorithms appear to be similarly better than the ones out of table.

For Long Method (Table 13) the results did not identify a single winning algorithm configuration. B-J48 Pruned/Unpruned, B-Random Forest, B-JRip are the co-winner

Table 11 The first 10 best algorithms on god class

Classifier	Wilcoxon victories		Cliff delta effect size	
	Number	Percentage	Value	Magnitude
Naïve Bayes	24	77 %	0,33	medium
J48 Pruned	21	68 %	0,29	small
J48 Unpruned	21	68 %	0,29	small
J48 Reduced Error Pruning	21	68 %	0,28	small
Random Forest	20	65 %	0,29	small
B-J48 Pruned	20	65 %	0,29	small
B-J48 Reduced Error Pruning	20	65 %	0,28	small
JRip	20	65 %	0,25	small
B-Random Forest	19	61 %	0,22	small
B-Naïve Bayes	19	61 %	0,21	small

Table 12 The first 10 best algorithms on feature envy

Classifier	Wilcoxon victories		Cliff delta effect size	
	Number	Percentage	Value	Magnitude
B-J48 Pruned	26	84 %	0,46	medium
B-J48 Unpruned	25	81 %	0,43	medium
B-JRip	23	74 %	0,44	medium
B-Random Forest	23	74 %	0,42	medium
J48 Unpruned	23	74 %	0,37	medium
Random Forest	22	71 %	0,39	medium
J48 Reduced Error Pruning	20	65 %	0,35	medium
B-J48 Reduced Error Pruning	19	61 %	0,34	medium
J48 Pruned	18	58 %	0,35	medium
SMO Poly Kernel	16	52 %	0,28	small

algorithms with 25 victories. Random Forest follows with 24 victories, and B-J48 Reduced Error Pruning. The first 6 algorithms have "medium" effect size, while the remaining are "small".

Table 14 shows the best algorithms and Table 15 algorithms with at least one top ten ranking. From the tables we can see that J48 algorithm and its variants perform very well. Overall it, B-J48 Pruned is the best algorithm with 2 victories, with one 3rd and one 8th place. The second best algorithm is Random Forest with a second place, two 5th places and a 6th place. B-J48 Unpruned and B-JRip performed well on three datasets, but did not enter the top ten for God Class, so we consider their overall performances lower than the B-J48 Pruned and Random Forest. In the first 10 best algorithms on all code smell datasets, SVM algorithms are not present, except for SMO Poly Kernel algorithm, only on Feature Envy and Long Method, on the 10th position with a low number of wins. Table 14 sums up the choice of the best algorithms.

Table 13 The first 10 best algorithms on long method

Classifier	Wilcoxon victories		Cliff delta effect size	
	Number	Percentage	Value	Magnitude
B-J48 Unpruned	25	81 %	0,46	medium
B-Random Forest	25	81 %	0,43	medium
B-J48 Pruned	25	81 %	0,40	medium
B-JRip	25	81 %	0,40	medium
Random Forest	24	77 %	0,42	medium
B-J48 Reduced Error Pruning	23	74 %	0,36	medium
JRip	22	71 %	0,32	small
J48 Pruned	18	58 %	0,33	small
J48 Unpruned	16	16 %	0,32	small
SMO Poly Kernel	13	42 %	0,21	small

Table 14 Best algorithms chosen for each code smell

Best classifiers selected			
Data class	God Class	Feature Envy	Long Method
B-J48 pruned	Naïve Bayes	B-J48 Pruned	B-J48 Unpruned

5.3.2 Algorithm Results - Cross Validation Accuracy, F-measure, Area under ROC

The best algorithms configurations have been experimented on each code smell dataset, to evaluate the performance of the algorithms. Three performance estimators have been selected to compare the experimented results: Accuracy, F-measure and Area under ROC.

Tables 16, 17, 18 and 19 report, for each code smell dataset, the performance estimator values of the algorithms with best configuration that have been produced with a 10-fold cross-validation after 10 repetitions, one for each different randomization of the dataset (100 runs for each algorithm). The average has been calculated on performance estimator values of all runs, to have a single performance value with a good statistical validity.

The cross validation results on Data Class (Table 16), God Class (Table 17) and Long Method (Table 19) all have very high accuracy values that vary between 95.15–99.02 %, 93.12–97.55 %, 95.93–99.43 % respectively. For Feature Envy (Table 18), the accuracy values have higher variation between 85.50 and 96.84 %. Regarding the best algorithms, the results are similar to the one presented in the previous section, i.e., J48 family of algorithms and Random Forest are the top performing algorithms for all code smells while the SVM algorithms are the worst performers. There appears to be no clear benefit in using a boosting algorithm (algorithms name starting with B-). Sometimes the boosting algorithm improves performance but in the other times it makes the performance worse. The standard deviations of accuracy are low for all the algorithms and they increase when they get close to the lowest accuracy values. Furthermore, the values of F-Measure and Area under ROC are high and their standard deviations are close to zero

Table 15 Algorithms with at least one top ten rank and their ranks (note: ties exist in the table)

Classifier	Data class	God class	Feature envy	Long method
B-J48 Pruned	1	6	1	3
B-J48 Reduced Error Pruning	9	7	8	6
B-J48 Unpruned	4	–	2	1
B-JRip	3	–	3	4
B-Naïve Bayes	–	10	–	–
B-Random Forest	5	9	4	2
J48 Pruned	6	2	9	8
J48 Reduced Error Pruning	10	4	7	-
J48 Unpruned	7	3	5	9
JRip	8	8	–	7
Naïve Bayes	–	1	–	–
Random Forest	2	5	6	5
SMO Poly Kernel	–	–	10	10

Table 16 Data class – Cross validation results and Entire data set results

Classifier	Cross validation - Manually validated data set						Entire data set (Qualitas Corpus)		
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code smell	Percentage of Code Smell	Kappa B-J48 Pruned (Data Class)
B -J48 Pruned	99.02 %	1.51	99.26 %	1.15	0.9985	0.0064	2230	4.58 %	–
B-J48 Unpruned	98.67 %	1.79	98.99 %	1.86	0.9984	0.0064	2158	4.43 %	0.864
B-J48 Reduce Error Pruning	98.07 %	2.47	98.55 %	1.36	0.9951	0.0103	2118	4.35 %	0.895
B-JRip	98.83 %	1.60	99.12 %	1.21	0.9959	0.0111	1864	3.83 %	0.808
B-Random Forest	98.57 %	1.68	98.94 %	2.08	0.9993	0.0017	1899	3.90 %	0.744
B-Naïve Bayes	97.33 %	2.29	98.02 %	2.01	0.9955	0.0091	2141	4.40 %	0.765
B-SMO RBF Kernel	97.14 %	2.61	97.86 %	2.11	0.9782	0.0286	3697	7.59 %	0.496
B-SMO Poly Kernel	96.90 %	2.25	97.65 %	2.35	0.9862	0.0173	3388	6.96 %	0.463
B-LibSVM C-SVC Linear Kernel	95.14 %	2.73	96.34 %	2.08	0.9796	0.0230	3388	6.96 %	0.356
B-LibSVM C-SVC Polynomial Kernel	95.79 %	2.67	96.83 %	1.94	0.9858	0.0202	3388	6.96 %	0.437
B-LibSVM C-SVC Radial Kernel	95.95 %	2.79	96.96 %	2.01	0.9871	0.0172	3237	6.65 %	0.451
B-LibSVM C-SVC Sigmoid Kernel	95.05 %	3.10	96.30 %	2.34	0.9745	0.0255	6005	12.33 %	0.274
B-LibSVM ν-SVC Linear Kernel	96.10 %	2.75	97.06 %	1.70	0.9863	0.0179	4643	9.53 %	0.378
B-LibSVM ν-SVC Polynomial Kernel	96.48 %	2.53	97.33 %	1.24	0.9888	0.0171	3258	6.69 %	0.42
B-LibSVM ν-SVC Radial Kernel	96.45 %	2.68	97.34 %	1.71	0.9844	0.0195	3083	6.33 %	0.429
B-LibSVM ν-SVC Sigmoid Kernel	94.81 %	3.11	96.13 %	1.95	0.9532	0.0342	2755	5.66 %	0.354
J48 Pruned	98.55 %	1.84	98.91 %	1.39	0.9864	0.0219	2634	5.41 %	0.801
J48 Unpruned	98.38 %	1.87	98.79 %	1.90	0.9873	0.0208	1914	3.93 %	0.831
J48 Reduced Error Pruning	97.98 %	2.46	98.46 %	1.40	0.9839	0.0211	2634	5.41 %	0.801
JRip	98.17 %	2.18	98.62 %	2.07	0.9809	0.0241	2364	4.85 %	0.61
Random Forest	98.95 %	1.51	99.29 %	2.05	0.9996	0.0014	2176	4.47 %	0.766
Naïve Bayes	96.12 %	2.95	97.04 %	1.95	0.9938	0.0099	4344	8.92 %	0.529
SMO RBF Kernel	97.05 %	2.38	97.78 %	2.11	0.9686	0.0286	4158	8.54 %	0.45
SMO Poly Kernel	96.60 %	2.76	97.41 %	2.13	0.9912	0.0138	3591	7.37 %	0.41
LibSVM C-SVC Linear Kernel	95.76 %	2.79	96.86 %	2.00	0.9853	0.0146	3422	7.03 %	0.38
LibSVM C-SVC Polynomial Kernel	95.98 %	2.77	97.02 %	2.12	0.9871	0.0140	3155	6.48 %	0.406
LibSVM C-SVC Radial Kernel	96.00 %	2.63	97.01 %	1.97	0.9915	0.0100	2874	5.90 %	0.443
LibSVM C-SVC Sigmoid Kernel	95.79 %	2.83	96.87 %	1.65	0.9831	0.0162	3152	6.47 %	0.364
LibSVM ν-SVC Linear Kernel	95.57 %	2.86	96.70 %	2.27	0.9813	0.0180	3005	6.17 %	0.371

Table 16 (continued)

Classifier	Cross validation - Manually validated data set						Entire data set (Qualitas Corpus)		
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code smell	Percentage of Code Smell	Kappa B-J48 Pruned (Data Class)
LibSVM ν-SVC Polynomial Kernel	95.98 %	2.71	97.02 %	1.12	0.9863	0.0144	3024	6.21 %	0.387
LibSVM ν-SVC Radial Kernel	96.52 %	2.81	97.39 %	2.12	0.9896	0.0143	2831	5.81 %	0.509
LibSVM ν-SVC Sigmoid Kernel	95.90 %	2.65	96.96 %	1.79	0.9822	0.0169	3127	6.42 %	0.366

In view of the obtained results, it is possible to conclude that all the best algorithm configurations have high performances, regardless of the type of code smell. The SVM algorithms tend to give worse performances than the other algorithms, regardless of the type of code smell and of the boosting technique. The algorithms J48, JRip, Naive Bayes (except for the code smell Feature Envy) and Random Forest gave the best performances. The application of the boosting technique on the algorithms does not always improve their performances, and in some cases, it makes them worse. Despite this, the performances of the algorithms are very high.

5.3.3 Learning Curves

A learning curve depicts an improvement in performance on the vertical axis when there are changes in another parameter, in our case the number of training samples. Figures 5, 6, 7 and 8 show the machine learning curves of the top 10 algorithms with the best configurations for each code smell (see Tables 10, 11, 12 and 13 for the top 10 algorithms on each dataset). Each curve plots the accuracy vs. the number of training examples. Each point of the curves represents the average of the runs obtained by a 10-fold cross-validation with 10 repetitions, one for each different randomization of the dataset (100 different runs). Each cross-validation was performed with different sizes of training dataset. Twenty small datasets were created for each code smell by starting from 20 instances and increasing them by the same amount until 420 instances were reached (the complete dataset). Each dataset has the same ratio of affected instances of the larger datasets and the instances belong to different projects, in order to ensure a homogeneous distribution of the datasets. The first point on the X-axis is equal to 18, because the 10-fold cross-validation procedure splits each fold into two testing instances and 18 training in stances. For each small dataset, the number of training instances is 90 % of the size of the datasets. For each code smell dataset, the obtained results are reported below.

The learning curves of the algorithms on Data Class have a non-monotonic increasing trend. As it can be seen in Fig. 5, the accuracies start to approach a constant value when the number of training instances is between 216 and 234. When these thresholds are overcome, the accuracies tend to have a flat but slightly oscillatory trend, having more than 96 % of accuracy values. The obtained results indicate that the learning rate of the curves are very similar to one another and the number of training instances provided by the algorithms is enough to guarantee a good learning process.

The learning curves of the algorithms on God Class show an oscillatory trend that tends to fade away when the number of training instances increases. As it can be observed from Fig. 6,

Table 17 God Class – Cross validation results and entire data set results

Classifier	Cross validation - manually validated data set					Entire data set (Qualitas Corpus)			
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code Smell	Percentage of Code Smell	Naive Bayes (God Class)
B -J48 Pruned	97.02 %	2.82	97.75 %	2.14	0.9923	0.0125	4309	8.41 %	0.778
B-J48 Unpruned	97.02 %	2.88	97.75 %	2.00	0.9925	0.0135	4076	7.95 %	0.764
B-J48 Reduce Error Pruning	97.26 %	2.64	97.94 %	2.18	0.9861	0.0230	3547	6.92 %	0.739
B-JRip	96.90 %	3.15	97.67 %	2.39	0.9916	0.0137	3024	5.90 %	0.703
B-Random Forest	96.95 %	2.86	97.70 %	2.67	0.9890	0.0167	3637	7.09 %	0.779
B-Naïve Bayes	97.54 %	2.65	97.70 %	2.72	0.9871	0.0186	3437	6.70 %	0.756
B-SMO RBF Kernel	94.62 %	3.34	95.98 %	2.89	0.9838	0.0186	3937	7.68 %	0.558
B-SMO Poly Kernel	94.33 %	3.58	95.75 %	2.48	0.9799	0.0221	4791	9.35 %	0.539
B-LibSVM C-SVC Linear Kernel	94.64 %	3.52	95.97 %	2.79	0.9829	0.0209	3930	7.67 %	0.548
B-LibSVM C-SVC Polynomial Kernel	94.21 %	3.61	95.66 %	3.04	0.9792	0.0235	6340	12.37 %	0.498
B-LibSVM C-SVC Radial Kernel	94.36 %	3.80	95.76 %	2.40	0.9801	0.0222	4839	9.44 %	0.545
B-LibSVM C-SVC Sigmoid Kernel	94.57 %	3.30	95.93 %	2.68	0.9837	0.0207	2374	4.63 %	0.53
B-LibSVM ν-SVC Linear Kernel	94.52 %	3.68	95.88 %	2.02	0.9819	0.0231	4596	8.97 %	0.533
B-LibSVM ν-SVC Polynomial Kernel	93.12 %	4.01	94.84 %	2.17	0.9766	0.0241	5744	11.20 %	0.493
B-LibSVM ν-SVC Radial Kernel	94.10 %	3.15	95.57 %	2.72	0.9778	0.0241	5440	10.61 %	0.495
B-LibSVM ν-SVC Sigmoid Kernel	94.02 %	3.51	95.52 %	2.50	0.9818	0.0201	3028	5.91 %	0.472
J48 Pruned	97.31 %	2.51	97.98 %	1.89	0.9783	0.0250	3547	6.92 %	0.739
J48 Unpruned	97.31 %	2.51	97.98 %	1.93	0.9783	0.0250	2308	4.50 %	0.61
J48 Reduced Error Pruning	97.29 %	2.52	97.94 %	1.89	0.9742	0.0263	3547	6.92 %	0.739
JRip	97.12 %	2.70	97.81 %	2.48	0.9717	0.0285	3547	6.92 %	0.739
Random Forest	97.33 %	2.64	97.98 %	2.24	0.9927	0.0151	3695	7.21 %	0.814
Naïve Bayes	97.55 %	2.51	98.14 %	2.20	0.9916	0.0148	4936	9.63 %	-
SMO RBF Kernel	95.43 %	3.26	96.62 %	2.40	0.9427	0.0407	4782	9.33 %	0.533
SMO Poly Kernel	95.71 %	3.14	96.83 %	2.40	0.9459	0.0396	5603	10.93 %	0.531
LibSVM C-SVC Linear Kernel	95.02 %	3.37	96.31 %	2.41	0.9903	0.0123	2199	4.29 %	0.491
LibSVM C-SVC Polynomial Kernel	95.76 %	3.08	96.89 %	2.23	0.9892	0.0124	5239	10.22 %	0.453
LibSVM C-SVC Radial Kernel	95.76 %	2.97	96.87 %	2.62	0.9924	0.0093	4780	9.32 %	0.548
LibSVM C-SVC Sigmoid Kernel	95.12 %	3.22	96.38 %	2.08	0.9904	0.0136	2429	4.74 %	0.528
LibSVM ν-SVC Linear Kernel	95.48 %	3.24	96.65 %	1.93	0.9908	0.0135	2806	5.47 %	0.549

Table 17 (continued)

Classifier	Cross validation - manually validated data set					Entire data set (Qualitas Corpus)			
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code Smell	Percentage of Code Smell	Naive Bayes (God Class)
LibSVM ν-SVC Polynomial Kernel	95.62 %	3.29	96.78 %	2.01	0.9913	0.0121	4338	8.46 %	0.471
LibSVM ν-SVC Radial Kernel	95.76 %	3.02	96.87 %	2.32	0.9925	0.0091	4948	9.65 %	0.536
LibSVM ν-SVC Sigmoid Kernel	94.90 %	3.53	96.23 %	2.42	0.9903	0.0141	2908	5.67 %	0.552

B-J48 Unpruned, J48 Reduced Error Pruning and B-J48 Unpruned have the learning curves with the greatest fluctuations, whereas Naive Bayes is the algorithm with the most constant fluctuation. The course of the accuracies becomes more stable when the number of training instances is about 306. When this threshold is overcome, the oscillatory trend of the learning curves grows until it reaches the accuracy values that are between 96.5 and 97 %.

The learning curves of the algorithms on Feature Envy (Fig. 7) have a growing but oscillatory trend. Excluding the trend of the learning curves until the first 54 training instances, they are all very similar in shape and trend, except for the learning curve of the SMO Poly Kernel algorithm. In this case, the learning curve keeps growing but with an oscillatory trend until it reaches the maximum value of the training instances, whereas the other algorithms tend to stabilize when the number of the training instances is about 288. When this threshold is overcome, the accuracies of these algorithms tend to have a flat but oscillatory trend with more than 94 % of accuracy values.

The learning curves of the algorithms on Long Method (Fig. 8) tend to have a growing but oscillatory trend. After the algorithms have learned about 90 training instances, all the learning curves become similar in shape and trend, except for the learning curves of the algorithms SMO Poly Kernel Error Pruning and J48 Pruned. The learning curve of B-J48 Reduced Error Pruning is different from the others because the accuracy values decrease when they get close to 162 training instances. They subsequently keep growing until they reach 306 training instances and remain constant with slight variations. Indeed, the learning curve of B-J48 Pruned slightly differs from the other learning curves when the values of the training instances range from 90 to 180. During this range, the accuracy values slightly decrease and then increase progressively.

In view of the obtained results, it is possible to conclude that the learning curves of the top 10 algorithms have different trends in dependence of the involved code smell dataset. For each code smell, the rates of the learning curves are very similar to one another (except for SMO Poly Kernel on Feature Envy and Long Method dataset) and the number of the training instances that have been used to train the algorithms seems to be high enough to guarantee a good learning process.

5.3.4 Applying the Algorithms to Entire Data Set

Tables 16, 17, 18 and 19 report the results of applying the best algorithm configuration to the entire Qualitas Corpus dataset. Note: for the entire data set there is no manual validation, so the truth of the code smells in the entire data set is unknown. The results tell us the number and

Table 18 Feature envy – cross validation results and Entire data set results

Classifier	Cross validation - manually validated data set					Entire data set (Qualitas Corpus)			
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code Smell	Percentage of Code Smell	Kappa B-JRip
B -J48 Pruned	96.62 %	2.78	97.41 %	2.16	0.9900	0.0144	16,729	4.46 %	0.728
B-J48 Unpruned	96.50 %	2.96	97.37 %	2.37	0.9899	0.0137	14,908	3.98 %	0.791
B-J48 Reduce Error Pruning	95.90 %	3.11	96.90 %	2.24	0.9866	0.0185	16,300	4.35 %	0.736
B-JRip	96.64 %	2.84	97.44 %	2.16	0.9891	0.0150	12,645	3.37 %	–
B-Random Forest	96.40 %	2.70	97.29 %	2.73	0.9886	0.0154	14,804	3.95 %	0.781
B-Naïve Bayes	91.50 %	4.20	93.56 %	2.93	0.9527	0.0391	94,426	25.19 %	0.16
B-SMO RBF Kernel	93.88 %	3.20	95.40 %	3.17	0.9369	0.0379	113,340	30.24 %	0.152
B-SMO Poly Kernel	92.05 %	3.50	94.06 %	3.07	0.9541	0.0364	40,279	10.75 %	0.352
B-LibSVM C-SVC Linear Kernel	92.57 %	3.60	94.46 %	3.01	0.9668	0.0279	61,059	16.29 %	0.29
B-LibSVM C-SVC Polynomial Kernel	91.76 %	3.92	93.82 %	2.82	0.9706	0.0267	42,443	11.32 %	0.318
B-LibSVM C-SVC Radial Kernel	92.05 %	4.24	94.06 %	2.93	0.9707	0.0260	36,041	9.62 %	0.319
B-LibSVM C-SVC Sigmoid Kernel	88.57 %	4.12	91.59 %	3.08	0.9199	0.0487	58,607	15.64 %	0.253
B-LibSVM ν-SVC Linear Kernel	92.02 %	3.98	94.05 %	3.19	0.9691	0.0240	57,130	15.24 %	0.322
B-LibSVM ν-SVC Polynomial Kernel	91.86 %	3.69	93.88 %	2.05	0.9710	0.0230	46,451	12.39 %	0.292
B-LibSVM ν-SVC Radial Kernel	91.95 %	3.89	93.97 %	2.66	0.9723	0.0253	42,372	11.31 %	0.304
B-LibSVM ν-SVC Sigmoid Kernel	89.07 %	4.13	91.97 %	2.43	0.9080	0.0494	66,365	17.71 %	0.219
J48 Pruned	95.95 %	2.77	96.91 %	2.16	0.9647	0.0283	11,280	3.01 %	0.914
J48 Unpruned	96.12 %	2.71	97.04 %	2.17	0.9661	0.0280	11,280	3.01 %	0.914
J48 Reduced Error Pruning	95.93 %	2.80	96.89 %	2.10	0.9646	0.0287	13,049	3.48 %	0.879
JRip	95.67 %	3.13	96.69 %	2.34	0.9584	0.0335	13,588	3.63 %	0.815
Random Forest	96.26 %	2.86	97.19 %	2.57	0.9902	0.0122	13,294	3.55 %	0.836
Naïve Bayes	85.50 %	6.09	89.17 %	2.46	0.9194	0.0481	106,900	28.52 %	0.113
SMO RBF Kernel	93.83 %	3.39	95.36 %	3.15	0.9309	0.0382	85,255	22.75 %	0.188
SMO Poly Kernel	95.45 %	3.61	96.58 %	2.80	0.9484	0.0410	51,444	13.73 %	0.328
LibSVM C-SVC Linear Kernel	94.86 %	3.10	96.16 %	2.87	0.9803	0.0227	37,536	10.02 %	0.421
LibSVM C-SVC Polynomial Kernel	93.64 %	3.45	95.26 %	2.55	0.9785	0.0213	37,777	10.08 %	0.364
LibSVM C-SVC Radial Kernel	94.14 %	3.27	95.62 %	3.04	0.9802	0.0191	27,594	7.36 %	0.403
LibSVM C-SVC Sigmoid Kernel	89.95 %	4.23	92.57 %	2.43	0.9587	0.0275	51,191	13.66 %	0.307
LibSVM ν-SVC Linear Kernel	94.67 %	3.66	96.00 %	4.59	0.9819	0.0200	26,055	6.95 %	0.461

Table 18 (continued)

Classifier	Cross validation - manually validated data set					Entire data set (Qualitas Corpus)			
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code Smell	Percentage of Code Smell	Kappa B-JRip
LibSVM ν-SVC Polynomial Kernel	93.62 %	3.82	95.24 %	2.15	0.9784	0.0206	32,989	8.80 %	0.397
LibSVM ν-SVC Radial Kernel	94.05 %	3.38	95.55 %	2.76	0.9793	0.0193	26,221	7.00 %	0.411
LibSVM ν-SVC Sigmoid Kernel	90.02 %	4.09	92.62 %	2.59	0.9592	0.0267	52,813	14.09 %	0.299

percentage of instances affected by the code smell. Table 20 reports, for each code smell, the algorithm with the best cross-validation performance and the algorithms reporting the highest and lowest percentage of code smells. With every algorithm, the percentage of code smells detected on the entire dataset is reported.

In view of the obtained results, it is possible to conclude that the percentages of code smell instances vary according to the machine-learning algorithm. The results show that J48, JRip and Random Forest detect an average lower percentage of code smell instances than the other algorithms. The differences between the percentages of code smell instances detected by the boosted algorithms and the corresponding non-boosted version significantly vary for each code smell. Additionally, we saw that despite very small differences in the algorithms' performance in the cross validation, the differences in the entire data set are larger, as the highest code smell percentage can be ten times larger than the smallest one. Differences can be attributed to the lower prevalence of code smells in the entire data set.

5.4 Discussion About the Extracted Rules

The J48 and JRip algorithms provide, as output, human understandable detection rules, that are expressed through combinations of conditions or boolean logical propositions. The application of these machine learning algorithms has the advantage of inferring the threshold values of the metrics in the detection rule. These techniques calibrate threshold values over a labeled dataset. With respect to the definition of thresholds by experts, this approach decreases the cases of false negative code smells in which metric values are close to thresholds used in the detection rules, but not enough to classify the instance as affected by code smell.

JRip provides a boolean condition based on the combination of a particular set of metrics values. If the condition is verified, JRip detects the instance as affected by the code smell. J48 provides a decision tree, where the leaves represent the class labels (affected or not affected by smell) and the branches signify the conjunctions of metrics (nodes of the tree) leading to those class labels. Each branch has a condition concerning a metric and an associated threshold value that determines the navigability of the tree. Since, in our case, J48 produced relatively simple decision trees (with only one leaf having TRUE label) we present only the path leading to this label with boolean logic. All the other paths end in the FALSE label.

In the remainder of this section, the decision trees and the propositional logic rules with the best configurations on each code smell, produced respectively by J48 Pruned and JRip, are reported. It is worth noting that we did not select necessarily the best algorithm for each code

Table 19 Long method - cross validation results and entire data set results

Classifier	Cross validation - manually validated data set						Entire data set (Qualitas Corpus)		
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code smell	Percentage of code smell	Kappa B-J48 pruned
B -J48 Pruned	99.43 %	1.36	99.49 %	1.00	0.9969	0.0127	4539	1.21 %	–
B-J48 Unpruned	99.20 %	1.18	99.63 %	0.99	0.9969	0.0126	4884	1.30 %	0.883
B-J48 Reduce Error Pruning	99.19 %	1.31	99.39 %	0.87	0.9967	0.0100	4939	1.32 %	0.909
B-JRip	99.03 %	1.26	99.50 %	0.94	0.9937	0.0144	4863	1.30 %	0.919
B-Random Forest	99.23 %	1.17	99.57 %	1.91	0.9998	0.0006	5196	1.39 %	0.89
B-Naïve Bayes	97.86 %	2.37	98.35 %	2.02	0.9950	0.0084	49,521	13.21 %	0.292
B-SMO RBF Kernel	97.00 %	2.49	97.75 %	2.38	0.9930	0.0116	19,989	5.33 %	0.424
B-SMO Poly Kernel	98.67 %	1.76	99.00 %	2.17	0.9852	0.0208	19,886	5.31 %	0.379
B-LibSVM C-SVC Linear Kernel	96.86 %	2.54	97.64 %	2.11	0.9938	0.0104	11,143	2.97 %	0.461
B-LibSVM C-SVC Polynomial Kernel	96.52 %	2.70	97.41 %	1.79	0.9927	0.0102	21,296	5.68 %	0.388
B-LibSVM C-SVC Radial Kernel	95.93 %	3.16	96.95 %	2.08	0.9901	0.0117	47,177	12.59 %	0.396
B-LibSVM C-SVC Sigmoid Kernel	96.74 %	2.89	97.54 %	2.01	0.9933	0.0099	16,454	4.39 %	0.445
B-LibSVM ν-SVC Linear Kernel	96.74 %	2.83	97.55 %	1.85	0.9936	0.0104	10,148	2.71 %	0.493
B-LibSVM ν-SVC Polynomial Kernel	96.90 %	2.42	97.69 %	0.88	0.9934	0.0107	22,431	5.98 %	0.385
B-LibSVM ν-SVC Radial Kernel	96.76 %	2.77	97.57 %	1.32	0.9936	0.0094	14,829	3.96 %	0.434
B-LibSVM ν-SVC Sigmoid Kernel	96.67 %	2.69	97.50 %	1.87	0.9849	0.0203	16,092	4.29 %	0.523
J48 Pruned	99.10 %	1.38	99.32 %	1.04	0.9930	0.0151	5048	1.35 %	0.912
J48 Unpruned	99.05 %	1.51	99.28 %	1.54	0.9925	0.0168	5048	1.35 %	0.912
J48 Reduced Error Pruning	98.40 %	2.02	98.80 %	1.13	0.9868	0.0222	4939	1.32 %	0.909
JRip	99.02 %	1.62	99.26 %	1.79	0.9884	0.0181	5322	1.42 %	0.886
Random Forest	99.18 %	1.20	99.54 %	1.62	0.9998	0.0011	5735	1.53 %	0.806
Naïve Bayes	96.24 %	2.39	97.09 %	1.72	0.9921	0.0086	49,521	13.21 %	0.292
SMO RBF Kernel	97.57 %	2.02	98.17 %	1.61	0.9732	0.0235	19,886	5.31 %	0.446
SMO Poly Kernel	98.67 %	1.76	99.00 %	1.69	0.9852	0.0208	24,240	6.47 %	0.379
LibSVM C-SVC Linear Kernel	97.07 %	2.35	97.79 %	1.58	0.9980	0.0035	20,366	5.43 %	0.478
LibSVM C-SVC Polynomial Kernel	97.24 %	2.17	97.94 %	1.60	0.9956	0.0059	19,777	5.28 %	0.402
LibSVM C-SVC Radial Kernel	96.38 %	2.19	97.22 %	1.63	0.9915	0.0091	76,715	20.47 %	0.343
LibSVM C-SVC Sigmoid Kernel	97.21 %	2.13	97.90 %	1.24	0.9970	0.0053	13,611	3.63 %	0.52
	97.31 %	2.22	97.97 %	1.88	0.9978	0.0044	18,589	4.96 %	0.478

Table 19 (continued)

Classifier	Cross validation - manually validated data set						Entire data set (Qualitas Corpus)		
	Accuracy	Std. dev.	F measure	Std. dev.	Area under ROC	Std. dev.	Code smell	Percentage of code smell	Kappa B-J48 pruned
LibSVM ν-SVC Linear Kernel									
LibSVM ν-SVC Polynomial Kernel	97.36 %	2.13	98.03 %	0.90	0.9958	0.0056	19,730	5.26 %	0.411
LibSVM ν-SVC Radial Kernel	97.43 %	2.09	98.05 %	1.32	0.9972	0.0045	21,537	5.75 %	0.437
LibSVM ν-SVC Sigmoid Kernel	97.38 %	2.17	98.03 %	1.53	0.9979	0.0037	20,319	5.42 %	0.467

smell, but the highest performing setup of JRip and J48, without boosting. According to the configuration of the selected parameters, the boosting technique generates 10 different decision trees (or rule sets), which is too much to be reported in the text. Furthermore, J48 Pruned and JRip have so high performance values that the detection rules deserve to be shown. For details on metrics, see the definitions reported on the web page (http://essere.disco.unimib.it/reverse/MLCSD.html).

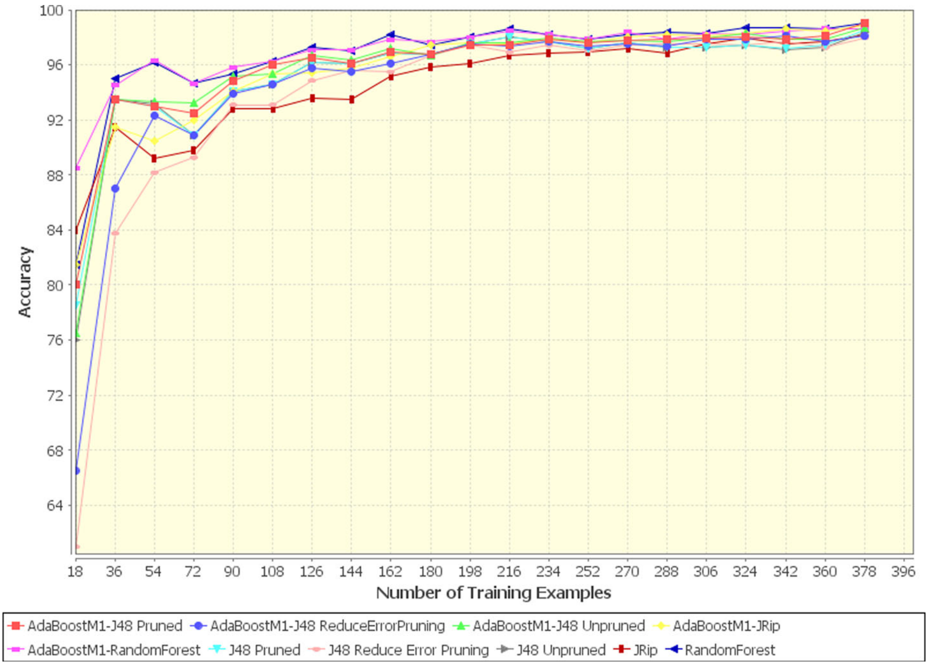


Fig. 5 Learning curves of first 10 best algorithms on Data Class

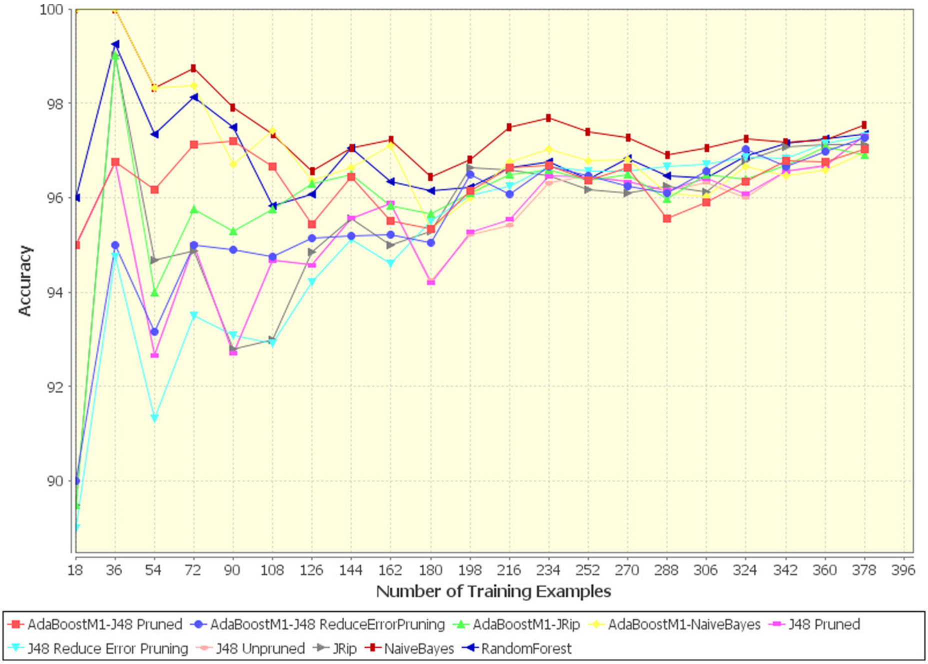


Fig. 6 Learning curves of first 10 best algorithms on God Class

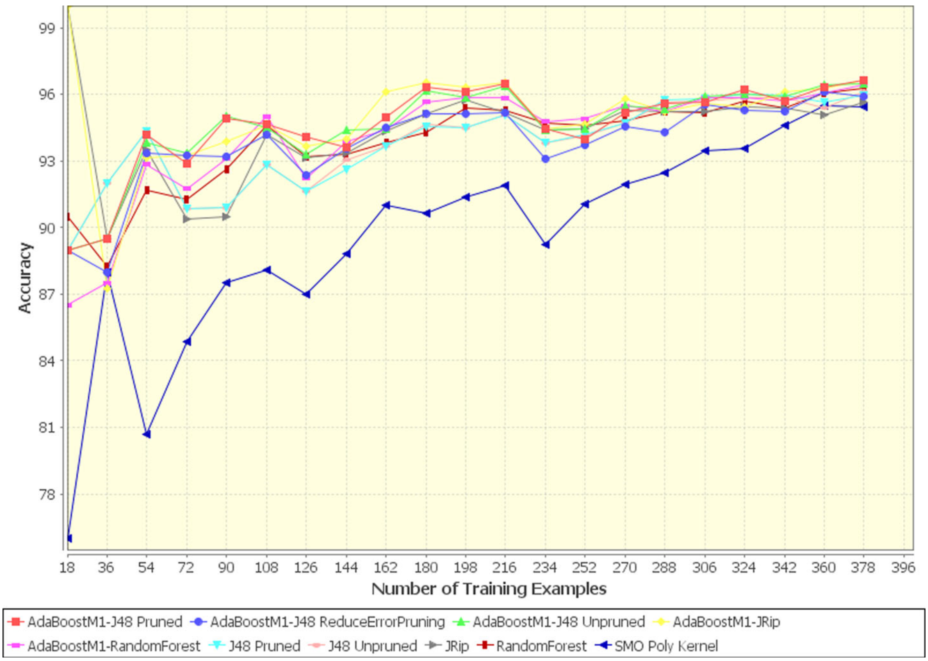


Fig. 7 Learning curves of first 10 best algorithms on Feature Envy

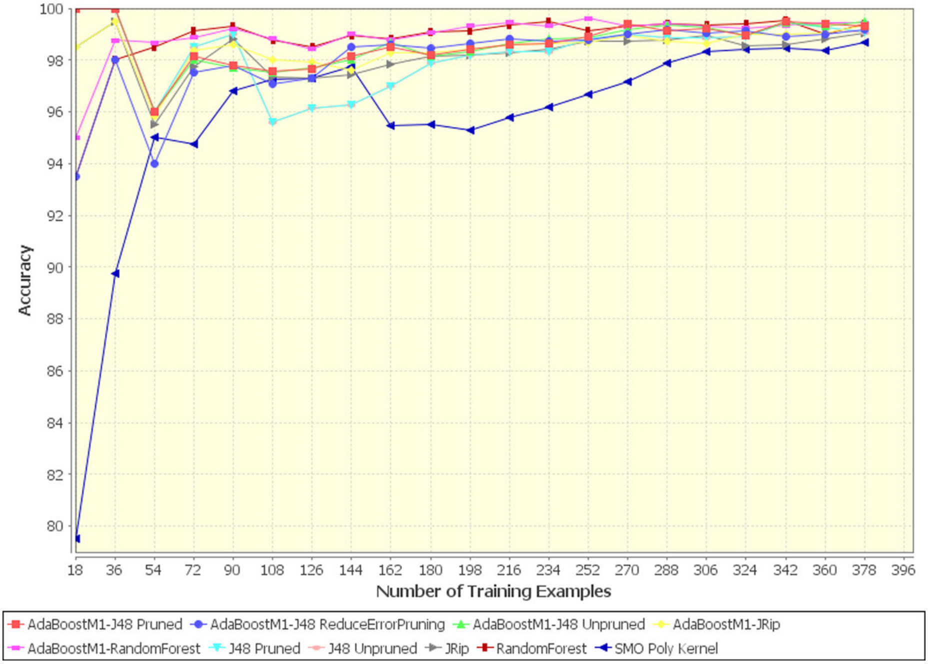


Fig. 8 Learning curves of first 10 best algorithms on Long Method

5.4.1 Data Class

For Data Class, J48 Pruned produces a decision tree expressed in Boolean logic as:

$$\text{NOAM} > 2 \ \text{ and } \ \text{WMCNAMM} \leq 21 \ \text{ and } \ \text{NIM} \leq 30.$$

The rule detects Data Class if the class has more than 2 accessor/mutator methods (NOAM metric), it is not very complex (WMCNAMM metric) and does not have a very high number of inherited methods (NIM metric). The first two conditions of the detection rule are partially included in the conceptual definition of Data Class reported in Section 3, and the threshold values fall approximately in the expected range. The last condition does not have an apparent connection with the definition of the code smell. The threshold value of the NIM metric is about 70 % of the maximum value assumed by the metric in the entire dataset. Furthermore,

Table 20 Summary of code smells detected on the entire dataset

Code smell	Winner algorithm	Highest % algorithm	Lowest % algorithm
Data Class	B-J48 Pruned (4.58 %)	B-LibSVM C-SVC Sigmoid Kernel (12.33 %)	B-JRip (3.83 %)
God Class	Naïve Bayes (9.63 %)	B-LibSVM C-SVC Polynomial Kernel (12.37 %)	LibSVM C-SVC Sigmoid Kernel (4.29 %)
Feature Envy	B-JRip (3.37 %)	B-SMO RBF Kernel (30.24 %)	J48 Pruned (3.01 %)
Long Method	B-J48 Pruned (1.21 %)	B-LibSVM C-SVC Polynomial Kernel (20.47 %)	B-J48 Pruned (1.21 %)

this metric is represented in the decision tree as the ultimate choice to discriminate whether the instance is affected by a code smell or not, i.e., as a leaf in the tree. As a consequence, this metric identifies a part of the domain that is very limited and becomes discriminative only for few instances. Its presence can be justified by statistic dependence between the values of the metrics and, in particular, it could be caused by the presence of some training instances bringing noise in the dataset.

For Data Class, JRip produces the following detection rule:

$$(\text{WOC} < 0.352941 \text{ and } \text{NOAM} \geq 4 \text{ and } \text{RFC} \leq 41)$$

$$\text{or } (\text{CFNAMM} = 0 \text{ and } \text{NOAM} \geq 3) \text{ or } (\text{AMW} \leq 1 \text{ and } \text{NOPVA} \geq 3).$$

The detection rule detects Data Class when at least one of the following conditions are verified: (*1*) the class reveals data rather than providing services (WOC metric), it has more than three accessor methods (NOAM metric) and it has a response set which is not extremely high (RFC metric); (*2*) the class does not call foreign methods, except for accessor/mutator methods (CFNAMM metric) and it has more than two accessors methods; (*3*) the class is not very complex (AMW metric) and it has more than three private attributes (NOPVA metric). The first condition of the detection rule is partially in the conceptual definition of Data Class reported in Section 3, and the threshold values of the metrics approximately meet the expected size, except for RFC. The presence of this metric can be justified again by statistic dependence or noise in the dataset. The same conclusions made for NIM metric and J48 Pruned on Data Class are valid for this metric. The second and third conditions are in the conceptual definition of Data Class and the threshold values of the metrics fall approximately in the expected range. The detection rule, as a whole, can represent a good example of rule that is inferred by a machine learning algorithm.

Comparison of the rules created by JRip and J48 pruned show that rules by J48 pruned uses only 3 metrics where as JRip utilizes 6 different metrics. Conceptually they make the same findings. First, the classes have accessor methods above a certain threshold. Second, the classes are not very complex.

5.4.2 God Class

As regards God Class, J48 Pruned produces a very simple decision tree:

$$\text{WMCNAMM} > 47.$$

The detection rule detects God Class if the class is complex. The threshold value of the WMCNAMM metric respects approximately the expected value. The rule detects the only feature that mostly characterizes a God Class: complexity. The detection rule of JRip for God Class is

$$\text{WMCNAMM} \geq 48,$$

which is exactly equal to that produced by J48 Pruned for God Class.

5.4.3 Feature Envy

For Feature Envy, J48 Pruned produces the following decision tree:

$$\text{ATFD(method)} > 4 \text{ and } \text{LAA} < 0.458571 \text{ and } \text{NOA} \leq 16.$$

The detection rule detects the code smell if the method directly uses many attributes of other classes (ATFD metric on method), it uses far more attributes from other classes than its own (LAA metric) and the class that contains the method can have a discrete number (16) of own attributes (NOA). The first two conditions of the detection rule are in the conceptual definition of Feature Envy reported in Section 3 and the threshold values of the metrics fall approximately in the expected range. There is no apparent relationship between the condition on NOA metric and the definition of the code smell. Its threshold value is more than 75 % of the maximum value of the metric on all systems. Consequently, this metric identifies a part of the domain that is very limited and becomes discriminative only for a few instances. Moreover, this condition is represented by a leaf in the decision tree.

For Feature Envy, JRip produces a detection rule based on the value of the following metrics:

$$(\mathrm{ATFD}{\geq}5 \text{ and } \mathrm{LAA} < 0.3125) \text{ or } (\mathrm{ATFD}{\geq}9) \text{ or } (\mathrm{FDP}{\leq}3 \text{ and } \mathrm{NMO}{\leq}1).$$

The detection rule detects the code smell when at least one of the following conditions are verified: (*1*) the method directly uses many of other classes attributes (ATFD metric), and it uses far more attributes from other classes than its own (LAA metric); (*2*) the method directly uses many attributes (ATFD); the foreign attributes used by the method belong to limited set of classes and the class that contains the method has at most one overridden method. In general, the detection rule is in the conceptual definition of Feature Envy reported in Section 3 and the threshold values of the metrics approximately fall approximately in the expected range, except for NMO metric. The condition on NMO is verified only for the values 0 or 1. For this reason, the condition on NMO metric should not have a significant impact on the detection rule, and could be eliminated in order to make the rule clearer and more readable.

Comparison between J48 Pruned and JRip on Feature Envy shows that both algorithms share two metrics ATFD and LAA. The former measures the number of attributes used from other classes and both algorithms have exactly the same threshold for the value (>4). The latter measure is the share of foreign attributes used. The threshold for this metric is different between algorithms 0.459 and 0.313 for J48 Pruned and JRip respectively. JRip produces additional rules that are not present in the J48 Pruned.

5.4.4 Long Method

For Long Method, J48 produces a decision tree:

$$\mathrm{LOC}(\mathrm{method}) > 79 \text{ and } \mathrm{CYCLO} > 9.$$

The detection rule detects the code smell if the method size is large (LOC metric) and it is complex (CYCLO metric). Both conditions reflect the conceptual definition of Long Method reported Section 3 and the threshold values of the metrics fall approximately in the expected range.

The detection rule of JRip for Long Method is very similar to the one produced by J48:

$$\mathrm{LOC} \ \mathrm{method}{\geq}80 \text{ And } \mathrm{CYCLO}{\geq}8.$$

The only difference concerns the threshold value of CYCLO metric: this value is equal to 8 for JRip, whereas it is equal to 10 for J48 Pruned. JRip imposes a slightly more restrictive rule for this smell.

5.4.5 Summary

Both algorithms produced similar detection rules that conceptually matched the smell descriptions. The thresholds and rules are equal for God Class and nearly equal for Long Method. For

Feature Envy and Data Class the produced detection rules had some differences in addition to similarities. For those smells there were also some rules that were not related to the concepts of the code smell but existed only due to properties of the data used for training. Table 21 summarizes the extracted rules and their properties.

Some rules included metrics that are not semantically related to the smell definition. This happens for Data Class and Feature Envy. We investigated this aspect by comparing the results obtained by the rule reported above with the ones obtained by applying a revised version of the respective rule, not containing the unrelated metric.

For Data Class, both J48 and JRip introduced unrelated metrics in the rule. We revised the rules in this way:

- J48: NOAM>2 and WMCNAMM≤21; (we removed NIM≤30)
- JRip: (WOC<0.352941 AND NOAM≥4) OR (CFNAMM=0 AND NOAM≥3) OR (AMW≤1 AND NOPVA≥3); (we removed RFC≤41).

We applied the same revision procedure to the rule for Feature Envy:

- J48: ATFD (method)>4 and LAA<0.458571; (we removed NOA≤16)
- JRip: (ATFD≥5 and LAA<0.3125) or (ATFD≥9) or (FDP≤3); (we removed NMO≤1).

The revised rules for JRip in both cases decrease the rule precision, moving answers from true negatives to false positives. For Data Class, the difference is 14 answers (−0.08 precision). For Feature Envy, the difference is 53 answers (−0.03 precision).

For J48, there is no similarity in the effect, instead. In the case of Data Class, the revised rule moves 8 answers from negative to positive, slightly increasing recall and decreasing precision. The effect on Feature Envy is instead very positive: 24 answers are moved from false negative to true positive, increasing both precision and recall; F-Measure is increased of 0.1 as a result, reaching 0.93.

The effect of these simple revised rules is not consistent, so we cannot draw generalized conclusions about the extracted rules and the role of the unrelated metrics. We plan a larger assessment of the role of these metrics in the context of rule extraction in future work.

Table 21 Comparison of rules created by J48 Pruned and JRip

Code smell	Rules By J48 pruned	Rules by JRip	Comment
Data class	NOAM>2 and WMCNAMM≤21 and NIM≤30	(WOC<0.352941 and NOAM≥4 and RFC≤41) or (CFNAMM=0 And NOAM≥3) or (AMW≤1 And NOPVA≥3)	NIM and RFC are not conceptually part of Data Class smell.
God class	WMCNAMM≥48	WMCNAMM≥48	Both algorithms produce an equal rule
Feature Envy	ATFD(method)>4 and LAA<0.458571 and NOA≤16	(ATFD≥5 And LAA<0.3125) Or ATFD≥9 Or (FDP≤3 And NMO≤1)	NOA and NMO are not conceptually part of Feature Envy smell.
Long Method	LOC(method)≥80 And CYCLO≥10	LOC method≥80 And CYCLO≥8	Both algorithms produce a nearly equal rule

6 Threats to Validity

This section discusses the threats to validity of our machine learning approach. 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.

6.1 Threats to Internal Validity

The main threat to internal validity is the difficulty of deciding when a code smell candidate is actually a real code smell or not. The manual evaluation of code smells is subject to a certain degree of error. 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. This could cause a distortion of the training set. It is not possible to know a priori if the distortion produced better or worse performances during the experiments, while it surely lowers the performances in the general case. We managed this threat by aggregating the evaluation of three raters, through the method described in Section 4.4.2.

6.2 Threats to External Validity

One of the most important threats to external validity is the difficulty to have a training set that completely represents the code smell domain. As explained in Section 3, for each project, code smell candidates were selected with random sampling, and stratifying the choice according to the number of positive results of code smell Advisors. This criterion increases the probability of selecting instances affected by code smells. On the other hand, the sampling method could cause a distortion during the building of the training set, because the selection criterion is only partly random. Furthermore, the selected systems are open source, so they cannot be considered representative of the panorama of all the possible existing systems. This problem can lead to lack of generalization of the machine learning algorithms, because models tend to fit the characteristics that are peculiar to the training set, but they are not tested on other cases. As for the possibility to generalize results, the datasets contain only Java projects, so results could be transferred only to projects written in the same language. In particular, the programming language affects the values of many metrics, and often the design style.

We only used open source systems that might affect the generalizability of our results to industrial context. Early works suggest that open source may produce poorer code quality than commercial systems (Stamelos et al. 2002). However, more recent studies suggest that overall there is no difference between the code quality of open source and commercial systems (Spinellis 2008). But in more detail it seems that “pure open source” produces better software structure than commercial system or open source with commercial involvement (Spinellis 2008; Capra et al. 2011). The Qualitas Corpus (Tempero et al. 2010) has both “pure open source” systems and open source systems where there has been commercial involvement. Thus, we feel the threat of using only open source systems is small.

6.3 Experiments Limitations

The experiments have some limitations. First, only four code smells have been selected because the approach requires a manual validation for each code smell. This task requires a lot of time, because it is necessary to inspect the code of each instance (class or method) and evaluate its interactions.

Another limitation is the amount of data collected to train the machine learning algorithms. In this work, the number of the instances is sufficiently large to train the algorithms, but having a larger dataset would allow to have more reliable data. Larger datasets allow having a validation dataset, used as an additional test for the best-performing algorithms.

A limitation in the manual labeling process is the use of students instead of professionals. The use of students has been discussed extensively in the context of human subject experiments in software engineering (Höst et al. 2000; Tichy 2000; Carver et al. 2003; Runeson 2003; Berander 2004; Svahnberg et al. 2008). The general conclusion seems to be that students can be used in experiments as substitutes for professionals as long as they are trained for the task and the results are used to establish trends. Our students were trained for the task, thus, the first criterion is fulfilled. We also think that the use of students has no effect on whether machine learning approaches work for code smell detection in general. However, the second criterion means that the results regarding to the exact code smell detection rules, see Section 5.4, need to be further experimented with professionals, as their hints or drivers on code smell detection might be different from the students'.

Often in machine learning feature selection is used to improve the performance. In our study, we exploited no possibilities of feature selection. There is no go-to solution for feature selection in the general case that would be equally good to all of our algorithms. Thus, doing it would create yet another layer of complexity to the machine learning experimentation process by adding more variables to test and compare. Furthermore, we already obtained very good results with the simpler models among the tested ones, so we do not feel that we need to optimize the process more at this stage.

In the past, various papers show how the performance of the machine-learning algorithms we utilized can be enhanced by feature selection, e.g., SVM's (Navot et al. 2006; Weston et al. 2000; Chen and Lin 2006), Decision trees (C4.5) (Hall and Holmes 2003), Naive Bayes (Hall and Holmes 2003). Random Forest algorithm includes a feature selection by default and it might have provided some advantage in our paper. The number of features used by Random Forest was *log_2(#predictors)+1* for Data Class and God Class, 20 for Feature envy and 16 for Long Method. For SVMs, we performed data normalization or standardization as explained in Section 5.2. We recognize that finding the best feature selection algorithm to combine with machine learning algorithms is an active research topic and we encourage future works on this topic. The need for such future work is amplified by the fact that none of the past works on code smell detection have done or provided proper reports for their feature selection, see Table 1. Lack of feature selection has also been present in other machine learning benchmarking studies in software engineering (Lessmann et al. 2008).

Finally, the last limitation is the number of Advisors used as hints to select the instances to manually evaluate. As outlined in Section 4, only some of the detectors described in the literature are available or the detection rules they exploit are clearly defined and few detectors provide an easy and accessible way to export the detection results.

7 Discussion and Future Developments

All of our algorithms produced similar performances see Tables 16, 17, 18 and 19. The performances were also good or very good with respect to the machine learning measures used (Accuracy, F-measure, ROC). This confirms that using machine learning algorithms for code smell detection is an appropriate approach, or at least, is appropriate for the experimented four code smells. The winning machine learning algorithms have significant differences with the remaining ones, with a medium effect size on average.

However, performances are already so good that we think it does not really matter in practice what machine learning algorithm one chooses for code smell detection. This result is similar to (Lessmann et al. 2008) who investigated machine learning algorithms for defect prediction and found that the differences between algorithms are small. Detailed comparisons between the results by us and Lessmann show differences. In our study, decision tree algorithms offered one of the best performances, but in their study decision trees did not do so well. This difference can probably be partially attributed to the different decision trees used: we used J48 and JRIP while they used C4.5, ADT and CART. On the other hand, SVM's did quite well in their study but performed poorly in our study, which could be due to lack of feature selection for SVMs in our study. If one wants a single answer on which algorithm to use based on the two studies, the answer is Random Forest. It was among the very top algorithms in our study and it was the best algorithm in (Lessmann et al. 2008). Furthermore, machine learning competitions from other domains have showed that Random Forest tends to work quite well in various circumstances, see www.kaggle.com

On a more general level, our results and the ones by (Lessmann et al. 2008) suggest that future comparison of machine learning on software engineering data should not be assessed on algorithmic performance alone as the differences between algorithms might not be so meaningful in practice. The effort used to setup the algorithm by practitioners might be a better factor that needs to be considered in future studies. For example, some algorithms like SVM's would require an additional feature selection step to match the performance of the best performing algorithms in our study. Another dimension is the understandability of the algorithmic results. Decision tree algorithms extract understandable rules for code smells (see Section 5.4) that can offer additional benefits to humans as the rules can be used in developer training and knowledge sharing between developers.

An additional future research area would be to combine the machine learning based approaches with the fixed rule based approaches such as ones presented in (Moha et al. 2010) and (Palomba et al. 2013). After all, machine learning approaches reduce the cognitive load of the individuals as the computer creates the rules for code smells based on input samples. Unfortunately, for some topics such as code naming it is impossible to use machine learning algorithms to create such rules. Thus, approaches that utilize machine learning when possible and complement them with fixed rules based approaches seems like a highly important future work. In a similar way, another opportunity for future work is to analyze the rules extracted decision trees (see Section 5.4), and compare them to other rules proposed in the literature, focusing on the threshold values and on the choice of the metrics. Machine learning may be used as a way to gather a first version of a detection rule, to be refined with the inclusion of developer's knowledge.

In Section 5.3.4, we pointed out that despite the rather small differences of machine learning algorithm performance in the manually validated data set, we found much larger variances when we applied the algorithms to the entire data set. The lowest code smell percentage proposed by an algorithm in the entire data set varied from 1.2 to 4.3 % depending on the code smell. The highest code smell percentages proposed by an algorithm were much higher varying from 12.3 to 30.2 %, see Table 20. This suggests that machine learning performance suffers when there is low prevalence in findings, i.e., imbalanced data set. In our training set, 33.3 % percent of instances had a code smell. In the entire data set, we do not know the prevalence of code smells as it has not been manually labeled. However, if we use the code smell percentage of the winner algorithm as a surrogate we get prevalence values from 1.2 to 9.6 %. Support for the idea that lower prevalence decreases machine learning algorithm performance has been reported previously (McKnight et al. 2002). In more detail, (McKnight et al. 2002) shows through simulation that when prevalence is less than 10 % then the performances of machine learning algorithms suffer significantly, and this matches our situation in the entire Qualitas Corpus data set. Furthermore, the problem of the imbalance data has only recently gained wider attention in the machine learning community, e.g., (He and Garcia 2009) and (Sun et al. 2007). Thus, future work should be performed on code smells to see how the problem of imbalanced data can be assessed.

In addition to extending the experiments using more/other features, as code smells, algorithms, and Advisors, we have identified different possible future developments in the following contexts:

Code Smell severity classification. The code smell severity classification can help software developers in prioritizing their work on most severe code smells and provides more information than a more traditional binary classification. According to the number of severity type, the severity classification requires a manual validation on a large dataset to be exploited by machine learning algorithms. In this work we produced a severity classification for our training set, which we will exploit in future experiments.

Social platforms to collect data. To collect more instances, it is possible to create social platform in which experts in this area share the manually evaluated instances belonging to different systems. The social platform allows having a larger dataset, statistically more reliable because it reduces the bias given by the known sensibility of code smell detection based on a single developer opinion (Mäntylä and Lassenius 2006). Furthermore, this solution decreases the probability of making errors and enables to perform a careful rechecking of the already instances evaluated. Additionally, the social platform can educate developers on code smells.

Host online the performance of the algorithms. To better validate the machine learning approach, an extensive comparison with other rules and tools will be performed, hosting online the code smell dataset to find the best possible code smell predictor. The competition results can be used as a baseline for a community effort in finding the best possible code smell predictor, and then can be extended with other tools, e.g., JDeodorant (Tsantalis et al. 2009), Stench Blossom (Murphy-Hill and Black 2010), DÉCOR-Ptidej (Moha et al. 2010).

8 Conclusions

In this paper, we compare and benchmark code smell detection with supervised machine learning techniques that are able to evolve with the availability of new data. The approach can be easily used to detect other code smells, depending on the availability of training data.

Our machine learning applies stratified sampling of the instances, creating strata using two criteria: the project containing instances and the results of a set of code smell detection tools and rules (called Advisors). This methodology ensures a homogeneous distribution of instances on different projects and prioritizes the labelling of instances with a higher chance of being affected by code smells. Selected instances are used to train the machine learning algorithms. Then a parameter optimization technique is applied, to find the best algorithm with respect to a set of standard performance measures. Finally, experiments are performed to evaluate the performance values of the algorithms on code smell dataset. The classification process was tested through a set of experiments on four code smells (Data Class, God Class, Feature Envy, Long Method) and 32 different machine learning algorithms (16 different algorithms and their application of boosting technique). The experimented algorithms obtain high performances, regardless of the type of code smell. The SVM algorithms tend to give worse performances than the other algorithms, regardless of the type of code smell and of the boosting technique. Future studies should look whether the difference remains if one performs feature selection for all algorithms. The algorithms J48, JRip, Naive Bayes (except for the code smell Feature Envy) and Random Forest gave the best performances.

For each code smell, the best algorithms obtain the following performance: B-J48 Pruned scores 99.02 % of accuracy on Data Class, Naïve Bayes scores 97.55 % of accuracy on God Class, B-JRip scores 96.64 % of accuracy on Feature Envy and B-J48 Pruned scores 99.43 % of accuracy on Long Method. For each code smell, the best algorithms score at least 97 % of F-Measure. The application of the boosting technique on the algorithms does not always improve their performances, and in some cases, it makes them worse. The machine learning curves of the best algorithms indicate that the learning rates of the curves are very similar to one another, according to the type of code smell. The number of the training instances used to train the algorithms seems to be high enough to guarantee a good learning process.

Through our experimental analysis and results, we obtained:

- a large dataset containing manually validated code smell instances, useful for benchmarking code smell detectors;
- a dataset containing the gathered metrics, useful for other software quality assessment tasks;
- a dataset containing the results of the exploited detection tools and rules (*Advisors*), useful to make comparisons among them and new detection results;
- the possibility to host online competitions to find the best code smell predictors; the competition results can then be used as a baseline for a community to compare the best code smell predictors.

All datasets are available for download at http://essere.disco.unimib.it/reverse/MLCSD.html.

Appendix

Table 22

Table 22 Projects characteristics

System	LOC	NOPK	NOCS	NOM	Domain	Release date
aoi-2.8.1	136,533	22	799	688	3D/graphics/media	01/03/2010
argouml-0.34	284,934	125	2361	18,015	diagram generator/data visualization	12/15/2011
axion-1.0-M2	28,583	13	223	2989	Database	07/11/2003
castor-1.3.1	213,479	149	1542	11,967	Middleware	01/03/2010
cobertura-1.9.4.1	58,364	19	107	3309	Testing	03/03/2010
colt-1.2.0	75,688	24	525	4143	SDK	09/10/2004
columba-1.0	109,035	183	1188	6818	Tool	09/18/2005
displaytag-1.2	20,892	18	128	1064	diagram generator/data visualization	12/27/2008
drawswf-1.2.9	38,451	33	297	2742	3D/graphics/media	06/08/2004

Table 22 (continued)

System	LOC	NOPK	NOCS	NOM	Domain	Release date
drjava-sTable-20100913-r5387	130,132	29	225	10,364	IDE	09/13/2010
emma-2.0.5312	34,404	26	262	1805	Testing	06/12/2005
exoportal-v1.0.2	102,803	382	1855	11,709	diagram generator/data visualization	05/16/2006
findbugs-1.3.9	146,551	54	1631	10,153	Testing	08/21/2009
fitjava-1.1	2453	2	60	254	Testing	04/07/2004
fitlibraryforfitnesse-20100806	25,691	108	795	4165	Testing	08/06/2010
freecol-0.10.3	163,595	36	1244	8322	Games	09/27/2011
freecs-1.3.20100406	25,747	12	131	1404	Tool	04/06/2010
freemind-0.9.0	65,687	43	849	5788	diagram generator/data visualization	02/19/2011
galleon-2.3.0	12,072	35	764	4305	3D/graphics/media	04/15/2006
ganttproject-2.0.9	58,718	54	959	5518	Tool	03/31/2009
heritrix-1.14.4	9424	45	649	5366	Tool	05/10/2010
hsqldb-2.0.0	171,667	22	465	7652	Database	06/07/2010
itext-5.0.3	117,757	24	497	5768	diagram generator/data visualization	07/22/2010
jag-6.1	24,112	15	255	145	Tool	05/25/2006
jasml-0.10	6694	4	48	245	Tool	05/23/2006
jasperreports-3.7.3	260,912	66	1571	17,113	diagram generator/data visualization	07/20/2010
javacc-5.0	19,045	7	102	808	parsers/generators/make	10/20/2009
jedit-4.3.2	138,536	28	1037	656	Tool	05/09/2010
jena-2.6.3	117,117	48	1196	99	Middleware	06/01/2010
jext-5.0	34,855	37	485	2169	diagram generator/data visualization	07/07/2004
jFin_DateMath-R1.0.1	7842	11	58	541	SDK	02/19/2010
jfreechart-1.0.13	247,421	69	960	1181	Tool	04/20/2009
jgraph-5.13.0.0	53,577	32	399	2996	Tool	09/28/2009
jgraphpad-5.10.0.2	33,431	22	426	1879	Tool	11/09/2006
jgrapht-0.8.1	28,493	19	299	1475	Tool	07/04/2009
jgroups-2.10.0	126,255	21	1093	8798	Tool	07/12/2010
jhotdraw-7.5.1	104,357	64	968	7232	3D/graphics/media	08/01/2010
jmeter-2.5.1	113,375	110	909	8059	Testing	09/29/2011
jmoney-0.4.4	9457	4	190	713	tool	09/29/2003
jparse-0.96	16,524	3	65	780	parsers/generators/make	07/17/2004
jpf-1.0.2	18,172	10	121	1271	SDK	05/19/2007
jruby-1.5.2	199,533	77	2023	17,693	programming language	08/20/2010
jspwiki-2.8.4	69,144	40	405	2714	Middleware	05/08/2010
jsXe-04_beta	1448	7	100	703	Tool	04/25/2006
jung-2.0.1	53,617	40	786	3884	diagram generator/data visualization	01/25/2010
junit-4.10	9065	28	204	1031	Testing	09/29/2011
log4j-1.2.16	34,617	22	296	2118	Testing	03/30/2010
lucene-3.5.0	214,819	133	1908	12,486	Tool	11/20/2011

Table 22 (continued)

System	LOC	NOPK	NOCS	NOM	Domain	Release date
marauroa-3.8.1	26,472	30	208	1593	Games	07/25/2010
megamek-0.35.18	315,953	37	2096	13,676	Games	08/31/2010
mvnforum-1.2.2-ga	92,696	29	338	5983	Tool	08/17/2010
nekohtml-1.9.14	10,835	4	56	502	parsers/generators/make	02/02/2010
openjms-0.7.7-beta-1	68,929	48	515	379	Middleware	03/14/2007
oscache-2.4.1	11,929	13	66	629	Middleware	07/07/2007
picocontainer-2.10.2	12,103	15	208	1302	Middleware	02/25/2010
pmd-4.2.5	71,486	88	862	5959	Testing	02/08/2009
poi-3.6	299,402	133	233	19,618	Tool	12/15/2009
pooka-3.0-080505	68,127	28	813	4551	Tool	05/05/2008
proguard-4.5.1	82,661	33	604	5154	Tool	07/08/2010
quartz-1.8.3	52,319	44	280	2923	Middleware	06/22/2010
quickserver-1.4.7	18,243	16	132	1278	Middleware	03/01/2006
quilt-0.6-a-5	8425	14	66	641	Testing	10/20/2003
roller-4.0.1	78,591	80	567	5715	Tool	01/13/2009
squirrel_sql-3.1.2	8378	3	153	689	Database	06/15/2010
sunflow-0.07.2	24,319	21	191	1447	3D/graphics/media	02/08/2007
tomcat-7.0.2	283,829	106	1538	15,627	Middleware	08/11/2010
trove-2.1.0	8432	3	91	585	SDK	08/14/2009
velocity-1.6.4	5559	33	388	2957	diagram generator/data visualization	05/10/2010
wct-1.5.2	69,698	85	606	5527	Tool	08/22/2011
webmail-0.7.10	14,175	16	118	1092	Tool	10/07/2002
weka-3.7.5	390,008	75	2045	17,321	Tool	10/28/2011
xalan-2.7.1	312,068	42	1171	10,384	parsers/generators/make	11/27/2007
xerces-2.10.0	188,289	40	789	9246	parsers/generators/make	06/18/2010
xmojo-5.0.0	31,037	9	110	1199	Middleware	07/17/2003

Table 23

Table 23 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
Coupling	FANOUT	–	Class, Method
	FANIN	–	Class
	ATFD	Access to Foreign Data	Method
	FDP	Foreign Data Providers	Method
	RFC	Response for a Class	Class
	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
Encapsulation	NOAM	Number of Accessor Methods	Class
	NOPA (NOAP)	Number of Public Attribute	Class
	LAA	Locality of Attribute Accesses	Method
Inheritance	DIT	Depth of Inheritance Tree	Class
	NOI	Number of Interfaces	Project, Package
	NOC	Number of Children	Class
	NMO	Number of Methods Overridden	Class

Table 23 (continued)

Quality dimension	Metric label	metric name	Granularity
	NIM	Number of Inherited Methods	Class
	NOII	Number of Implemented Interfaces	Class

Table 24

Table 24 Custom 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 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

References

Aggarwal KK, Singh Y, Kaur A, Malhotra R (2006) Empirical study of object-oriented metrics. J Object Technol 5(8):149

Arcelli Fontana F, Braione P, Zanoni M (2012) Automatic detection of bad smells in code: an experimental assessment. J Object Technol 11(2), p. 5:1

Arcelli Fontana F, Ferme V, Marino A, Walter B, Martenka P (2013a) Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains. Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013), 260–269

Arcelli Fontana F, Zanoni M, Marino A, Mantyla MV (2013b) Code smell detection: towards a machine learning-based approach. In: Fontana A (ed) Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013), IEEE, Eindhoven, The Netherlands, 396–399. doi:10.1109/ICSM.2013.56

Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Trans Softw Eng 28(1):4–17. doi:10.1109/32.979986

Bengio Y, Grandvalet Y (2004) No unbiased estimator of the variance of k-fold cross-validation. J Mach Learn Res 5:1089–1105

Berander P (2004) Using students as subjects in requirements prioritization. Proceedings of the International Symposium on Empirical Software Engineering (ISESE'04), 167–176

Bowes D, Randall D, Hall T (2013) The inconsistent measurement of message chains. Proceedings of the 4th International Workshop on Emerging Trends in Software Metrics (WETSoM 2013). IEEE, San Francisco, CA, USA, 62–68

Capra E, Francalanci C, Merlo F, Rossi-Lamastra C (2011) Firms' involvement in open source projects: a trade-off between software structural quality and popularity. J Syst Softw 84(1):144–161

Carver J, Jaccheri L, Morasca S, Shull F (2003) Issues in using students in empirical studies in software engineering education. Proceedings of the Ninth International Software Metrics Symposium (METRICS 2003). IEEE, Sydney, Australia, 239–249

Chen Y-W, Lin C-J (2006) Combining SVMs with various feature selection strategies. Feature extraction. Springer Berlin Heidelberg, 315–324

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493. doi:10.1109/32.295895

Cohen PR, Jensen D (1997) Overfitting explained, Preliminary Papers of the Sixth International Workshop on Artificial Intelligence and Statistics. Self published. Printed proceedings distributed at the workshop, 115–122 http://w3.sista.arizona.edu/~cohen/Publications/papers/cohen-ais96b.pdf

Dekkers A, Aarts E (1991) Global optimization and simulated annealing. Math Program 50:367–393

Deligiannis I, Stamelos I, Angelis L, Roumeliotis M, Shepperd M (2004) A controlled experiment investigation of an object-oriented design heuristic for maintainability. J Syst Softw 72(2):129–143

Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30

Dubey SK, Sharma A, Rana A (2012) Comparison of software quality metrics for object-oriented system. Int J Comput Sci Manag Stud 12:12–24

Ferme V (2013) JCodeOdor: a software quality advisor through design flaws detection, Master's thesis, University of Milano-Bicocca

Ferme V, Marino A, Arcelli Fontana F (2013) Is it a real code smell to be removed or not? Presented at the RefTest 2013 Workshop, co-located event with XP 2013 Conference, 15

Fowler M, Beck K (1999) Refactoring: improving the design of existing code. 1–82

Freund Y, Schapire R (1996) Experiments with a new boosting algorithm. In: Proceedings of the thirteenth International Conference on Machine Learning (ICML 1996), Bari, Italy, 148–156

Goldberg D (1989) Genetic algorithms in search, optimization, and machine learning, 1st edition. Addison-Wesley, Ed. Longman Publishing Co., Inc

Guéhéneuc Y-G, Sahraoui H, Zaidi F (2004) Fingerprinting design patterns. Proceedings. 11th Working Conference on Reverse Engineering (WCRE 2004), IEEE, Victoria, BC, Canada, 172–181. doi:10.1109/WCRE.2004.21

Hall MA, Holmes G (2003) Benchmarking attribute selection techniques for discrete class data mining. IEEE Trans Knowl Data Eng 15(6):1437–1447

Hall M, Frank E, Holmes G (2009) The WEKA data mining software: an update. SIGKDD Explor Newsl 11(1):10–18. doi:10.1145/1656274.1656278

Hall T, Zhang M, Bowes D, Sun Y (2014) Some code smells have a significant but small effect on faults. ACM Trans Softw Eng Methodol 23(4):33

He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans Knowl Data Eng 21(9):1263–1284

Hollander M, Wolfe DA, Chicken E (2000) Nonparametric statistical methods, 3rd edn. Wiley, New York, pp 39–55, 84-87

Höst M, Regnell B, Wohlin C (2000) Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. Empir Softw Eng 5(3):201–214

Hsu C, Chang C, Lin C (2003) A practical guide to support vector classification. vol. 1, no. 1, pp. 5–8. URL: http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf. Accessed: 9 March 2015

Khomh F, Vaucher S, Guéhéneuc Y-G, Sahraoui H (2009) A bayesian approach for the detection of code and design smells. Proceedings of the 9th International Conference on Quality Software (QSIC 2009). IEEE, Jeju, pp 305–314. doi:10.1109/QSIC.2009.47

F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," Journal of Systems and Software, vol. 84, no. 4, pp. 559–572, Apr. 2011, Elsevier Inc. doi:10.1016/j.jss.2010.11.921.

Kline RM. Library Example. [Online]. Available: http://www.cs.wcupa.edu/~rkline/java/library.html. Accessed: 23 September 2013

Kreimer J (2005) Adaptive detection of design flaws. Electron Notes Theor Comput Sci 141(4):117–136. doi:10.1016/j.entcs.2005.02.059

Lamkanfi A, Demeyer S (2010) Predicting the severity of a reported bug. In: 7th working conference on Mining Software Repositories (MSR 2010). IEEE, Cape Town, 1–10. doi:10.1109/MSR.2010.5463284

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496

Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128. doi:10.1016/j.jss.2006.10.018, ISSN 0164–1212

Lorenz M, Kidd J (1994) Object-oriented software metrics: a practical guide. Prentice-Hall, Ed

Maiga A, Ali N (2012) SMURF: a SVM-based incremental anti-pattern detection approach. Proceedings of the 19th Working Conference on Reverse Engineering (WCRE 2012). IEEE, Kingston, pp 466–475. doi:10.1109/WCRE.2012.56

Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc Y-G, Antoniol G, Aïmeur E (2012) 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, 278–281

Mäntylä MV, Vanhanen J, Lassenius C (2004) Bad smells-humans as code critics. Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004), IEEE, Chicago Illinois, USA, 399–408

Mäntylä MV, Lassenius C (2006) Subjective evaluation of software evolvability using code smells: an empirical study. Empir Softw Eng 11(3):395–431

Marinescu R (2002) Measurement and quality in object-oriented design. Politechnica University of Timisoara http://loose.upt.ro./download/thesis/thesis.zip

Marinescu R (2005) Measurement and quality in object-oriented design. Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), IEEE, Budapest, Hungary, 701–704. doi:10.1109/ICSM.2005.63

Marinescu C, Marinescu R, Mihancea P, Ratiu D, Wettel R (2005) iPlasma: an integrated platform for quality assessment of object-oriented design. Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005). IEEE, Budapest, Hungary, 77–80 http://loose.upt.ro./download/papers/marinescu-iPlasma.pdf

McKnight LK, Wilcox A, Hripcsak G (2002) The effect of sample size and disease prevalence on supervised machine learning of narrative data. Proc AMIA Symp. 519–22

Menzies T, Marcus A (2008) Automated severity assessment of software defect reports. Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008). IEEE, Beijing, Beijing, China 346–355. doi:10.1109/ICSM.2008.4658083

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

Moser R, Abrahamsson P, Pedrycz W, Sillitti A, Succi G (2008) A case study on the impact of refactoring on quality and productivity in an agile team. In: Meyer B, Nawrocki J, Walter B (eds) Balancing agility and formalism in software engineering. Lecture Notes in Computer Science, vol 5082. Springer Berlin Heidelberg, pp. 252–266

Murphy-Hill E, Black AP (2010) An interactive ambient visualization for code smells. In: Proceedings of the 5th international symposium on Software visualization (SOFTVIS'10). ACM, Salt Lake City, Utah, USA, 5–14. doi:10.1145/1879211.1879216

Navot A, Gilad-Bachrach R, Navot Y, Tishby N (2006) Is feature selection still necessary?. In: Subspace, latent structure and feature selection (pp. 127–138). Springer Berlin Heidelberg

Nongpong K (2012) Integrating 'Code Smells' detection with refactoring tool support. University of Wisconsin-Milwaukee http://dc.uwm.edu/etd/13

Olbrich SM, Cruzes DS, Sjoberg DIK (2010) Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010). IEEE, Timişoara, Romania, pp 1–10 doi:10.1109/ICSM.2010.5609564

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

Runeson P (2003) Using Students as Experimental Subjects -An analysis of Graduate and Freshmen PSP Student Data. Proceedings of the 7th international conference on empirical assessment in software engineering. Keele University, UK, pp 95–102

Sjøberg DIK, Yamashita AF, Anda BCD, Mockus A, Dybå T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 39(8):1144–1156

Spinellis D (2008) A tale of four kernels. In: Proceedings of the 30th international conference on Software engineering (ICSE 2008). ACM, Leipzig, Germany, pp. 381–390 doi: 10.1145/1368088.1368140

Stamelos I, Angelis L, Oikonomou A, Bleris GL (2002) Code quality analysis in open source software development. Inf Syst J 12(1):43–60

Stone M (1974) Cross-validatory choice and assessment of statistical predictions. J R Stat Soc 36(2):111–147

Sun Y, Kamel MS, Wong AK, Wang Y (2007) Cost-sensitive boosting for classification of imbalanced data. Pattern Recogn 40(12):3358–3378

Svahnberg M, Aurum A, Wohlin C (2008) Using students as subjects-an empirical evaluation. Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM '08). ACM, Kaiserslautern, Germany, pp 288–290

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

Tian Y, Lo D, Sun C (2012) Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: 19th Working Conference on Reverse Engineering. IEEE, Ontario, Canada, 215–224. doi:10.1109/WCRE.2012.31

Tichy WF (2000) Hints for reviewing empirical work in software engineering. Empir Softw Eng 5(4):309–312

Tsantalis N, Member S, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Trans Softw Eng 35(3):347–367

Weston J, Mukherjee S, Chapelle O, Pontil M, Poggio T, Vapnik V (2000) Feature selection for SVMs. In NIPS (Vol. 12, pp. 668–674)

Wieman R (2011) Anti-pattern scanner: an approach to detect anti-patterns and design violations. Delft University of Technology. LAP Lambert Academic Publishing

Yamashita A (2014) Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. J Empir Softw Eng 19(4):1111–1143

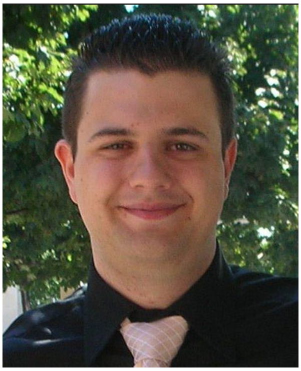
Yang J, Hotta K, Higo Y, Igaki H, Kusumoto S (2012) Filtering clones for individual user based on machine learning analysis. Proceedings of the 6th International Workshop on Software Clones (IWSC 2012). IEEE, Zurich, pp 76–77. doi:10.1109/IWSC.2012.6227872

Zazworka N, Shaw MA, Shull F, Seaman C (2011) Investigating the impact of design debt on software quality. Proceedings of the 2nd Workshop on Managing Technical Debt. ACM, Waikiki, Honolulu, HI, USA, pp. 17–23 doi:10.1002/smr.521

Zhang M, Hall T, Baddoo N (2011) Code Bad Smells : a review of current knowledge. J Softw Maint Evol Res Pract 23(3):179–202. doi:10.1002/smr.521

Francesca Arcelli Fontana received the Diploma and PhD degrees in computer science from the University of Milano. She was at University of Salerno and University of Sannio, Faculty of Engineering, as assistant professor in software engineering. She is currently an associate professor at the Department of Computer Science of the University of Milano-Bicocca. Her research interests include software evolution, software metrics, software maintenance and reverse engineering. She is a member of the IEEE and the IEEE Computer Society.

Mika Mäntylä is a professor of Software Engineering at University of Oulu, Finland. He received a D. Sc. degree in 2009 in software engineering from the Helsinki University of Technology, Finland His research interests include empirical software engineering, software testing, human cognition, defect data, and software evolution & maintenance. He has previously worked as a post-doc at Lund University, Sweden and as an assistant professor at Aalto University, Finland. His previous studies have appeared in journals such as IEEE Transaction on Software Engineering, Empirical Software Engineering, and Information and Software Technology. For more information http://mikamantyla.eu/.



Marco Zanoni received the MS and Ph.D. degrees in computer science from the University of Milano-Bicocca. He is a post-doc research fellow of the Department of Informatics, Systems and Communication of the University of Milano-Bicocca. His research interests include design pattern detection, software architecture reconstruction and machine learning.

Alessandro Marino received the MS degree in computer science from the University of Milano-Bicocca. In his Master thesis he investigated code smell detection using machine learning techniques. Since November 2013, he is a IT consultant in Reply, working on Systems Integration and Digital Services.