Experience Report: Evaluating the Effectiveness of Decision Trees for Detecting Code Smells

Lucas Amorim, Evandro Costa
Instituto de Computação
Universidade Federal de Alagoas
Maceió – AL, Brazil, 57072-900
lucas@ic.ufal.br, evandro@ic.ufal.br

Nuno Antunes
CISUC, Department of Informatics Engineering
University of Coimbra
Polo II, 3030-290 Coimbra – Portugal
nmsa@dei.uc.pt

Baldoino Fonseca, Márcio Ribeiro
Instituto de Computação
Universidade Federal de Alagoas
Maceió – AL, Brazil, 57072-900
baldoino@ic.ufal.br marcio@ic.ufal.br

***Abstract*—Developers continuously maintain software systems to adapt to new requirements and to fix bugs. Due to the complexity of maintenance tasks and the time-to-market, developers make poor implementation choices, also known as *code smells*. Studies indicate that code smells hinder comprehensibility, and possibly increase change- and fault-proneness. Therefore, they must be identified to enable the application of corrections. The challenge is that the inaccurate definitions of code smells make developers disagree whether a piece of code is a smell or not, consequently, making difficult creation of a universal detection solution able to recognize smells in different software projects. Several works have been proposed to identify code smells but they still report inaccurate results and use techniques that do not present to developers a comprehensive explanation how these results have been obtained. In this experimental report we study the effectiveness of the *Decision Tree* algorithm to recognize code smells. For this, it was applied in a dataset containing 4 open source projects and the results were compared with the manual oracle, with existing detection approaches and with other machine learning algorithms. The results showed that the approach was able to effectively learn rules for the detection of the code smells studied. The results were even better when genetic algorithms are used to pre-select the metrics to use.**

Keywords*—*Software Quality, Code Smells, Decision Tree, Genetic Algorithm

I. Introduction

Software development and maintenance are continuous activities that have a never-ending cycle [1]. While developers commit changes on a software system to fix bugs or to implement new requirements, they sometimes introduce *code smells*, which represent symptoms of poor design and implementation choices [2]. Example of a *code smell* is the *Message Chain* [2], which occurs when the realization of a functionality of a class requires a long chain of method invocations between objects of different classes.

Previous studies have found that code smells hinder comprehensibility [3], and possibly increase change- and fault-proneness [4], [5]. For instance, a *Message Chain* is conjectured to impact change- and fault-proneness because of the high number of indirections. Therefore, *code smells* need to be carefully detected in order to enable correction actions can be planned and performed to deal with them.

The main challenge to detect *code smells* lies on their inaccurate definitions that make developers disagree whether a piece of code is a smell or not, consequently, making difficult the creation of a universal detection solution able to recognize smells in different software projects. This results that different tools classify the same code differently in terms of smells and also it is frequent that developers disagree in the existing of a certain code smell due to their different backgrounds and experience. To make matters worse, developers usually do not have the expertise and experience necessary to detect these smells at the naked eye, leaving to automated tools the responsibility of reporting them.

Several approaches have been proposed to identify *code smells* to alert developers of their presence. Namely, [6], [7], [8], [9], [10] propose techniques that rely on declarative rule specification to identify parts of the source code that need to be improved. In [11], [12] the authors propose a technique that, instead of manually specifying the rules for detecting each code smell type, such rules are extracted from instances of maintainability smells by using Machine Learning techniques, such as, Bayesian Belief Networks (BBNs) [13] and Support-vector Machine [14].

However, the existing approaches still present inaccurate results and use techniques that do not present to developers a comprehensive explanation how these results have been obtained. Such explanation is important to increase the confidence of developers, while deciding to agree (or not) with the smells identified by a specific approach. Conversely, *Decision Tree* algorithms [15] generate models that can be easily inspected by a developer or software engineer.

In this practical experience report we **evaluate the use of the decision tree algorithm to detect code smells in software projects**. This algorithm can work with the values of software metrics (e.g. number of lines of code, methods, parameters, etc) generating this way rules based on these metrics. Such rules can easily be validated, discussed, learned and even improved by the team members.

For this, we applied the `C5.0` algorithm (state of the art in terms of decision tree algorithm) [16] in the software metrics of 4 Java projects, namely Eclipse [17], Mylyn [18], ArgoUML [19] and Rhino [20]. These projects are part of a dataset in which the code smells have been previously identified and annotated. We compared the results obtained against the annotated oracle and other detectors, such as approaches based on general rules and based on other machine learning techniques. The results of our experiments show clearly that the decision tree algorithm used can be an effective way to

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

extract rules for the classification of code smells, mainly when used together with genetic algorithms for metrics selection. The approach was able to reach values of Precision, Recall and F-measure [21] greater than the ones reached by using existing approaches.

The results obtained were even better when a genetic algorithm was used to automatize the process of selecting the relevant metrics. This is a interesting finding, since in the future it can be used to contribute towards the definition of a *full-fledged* solution for the automated detection of code smells.

The structure of the paper is as follows. Section II presents, based on a concrete example, the motivation for this experimental study. Section III presents the research study and its goals, while Section IV presents and discusses the results obtained. Section V presents the threats to the validity of the findings of this study and Section VI reviews the related work. Finally, Section VII concludes the papers.

II. Motivating Example

In this section we describe a motivating example to show the difficulties faced by developers when using existing tools do detect *code smells*. In such example we concentrate the identification in four well known and easy to understand *smells*:

- **Blob** (also called *God Class*) refers to class that tends to centralize the intelligence of the system [22];
- **Long Parameter List** (LPL) is a parameter list that is too long and thus difficult to understand [23];
- **Long Method** (LM) is a method that is too long, so it is difficult to understand, change, or extend [24];
- **Feature Envy** (FE) means that a method is more interested in other class(es) than the one where it is currently located [24];

We used the tools *PMD* 5.1.1 [25], *CheckStyle* 5.7 [26], *JDeodorant* 5.0 [27] and *inFusion* 1.8.5 [28] to recognize these four kinds of smells in the *GanttProject* 2.0.10 [29]. Table I presents the amount of *code smells* recognized by each tool.

Table I. Results obtained by different tools.

		Analyzed Code Smells			
		Blob	LPL	LM	FE
Tools	inFusion	17	×	×	12
	JDedorant	43	×	108	49
	Checkstyle	×	1	13	×
	PMD	36	0	14	×

We notice that the agreement among the tools is low. For instance, while the tool *JDeodorant* identified 108 (LM), the *Checkstyle* recognized only 13 (LM). This divergence is influenced by two factors: (i) the different understanding of the developers about *smells* implies different strategies to detect them. For example, the tools *inFusion* [28] and *iPlasma* [30] adopt the metrics proposed by [22] to detect *God Classes*. In other hand, *JDeodorant* adopts the metric described in [24]; and, (ii) even in cases where the tools use the same metrics to detect a specific *smell*, the threshold used to define such metrics can be different. Indeed, in *Long Parameter List* detection, the *Checkstyle* and *PMD* use the number of parameters of each method (named as `NOParam`) to detect this *smell*. However, while *Checkstyle* considers *smells* all methods with `NOParam > 7`, PMD uses this thresholds tuned to 10, by default.

Although these tools enable developer to define manually the thresholds to detect *smells*, the definition of these thresholds is a hard and time-consuming task [31]. In this way, we performed a practical experiment in order to evaluate the performance of the Decision Tree algorithm to identify code smells automatically (see Section III).

III. Research Study

The challenge in the identification of code smells is that their definitions are based on inaccurate concepts that make developers disagree whether a piece of code is a smell or not. It is necessary solutions able to help developers in the smells detection in different projects.

This way, the main goal of this study is to investigate if **decision tree classification algorithms are a effective way of generating metric-based classification rules able to recognize smells**. Thus, our aim is to evaluate the proposed technique with regards to the effectiveness of the generated rules in performing the classification. In practice, the detection rules will take the form of decision trees which represent the classification model extracted based on the information present in the training datasets.

In addition, we investigate how the use of automated metric selection would impact the results of decision tree algorithms. Feature selection is useful to improve machine learning algorithms both by reducing training times and increasing the generalization ability. They also are useful for simplification of models, making them easier to understand by the user, which is already one of the main benefits of decision trees.

In summary, this study intends to contribute towards answering to the following questions:

Q1: Are Decision Tree Induction algorithms adequate to recognize code smells?

Q2: Are Decision Tree Induction algorithms able to perform better than general rules based approaches?

Q3: How do Decision Tree Induction algorithms perform when compared with other machine learning techniques?

Q4: Is the use of genetic algorithms an effective way to perform feature selection for code smell detection?

An experiment was designed in which a decision tree classification algorithm was evaluated and compared with existing approaches to identify code smells. The followed procedure was based on a set of steps, which are described in the following paragraphs.

The step **(1)** was to define the subjects to be used during the evaluation. This includes selecting the software, analyzing the existing code smells and software metrics in each Class. As detailed in Section III-A, the gathered dataset contains, for a total of about `7952` Classes, information about the existence or not of each of the `12` different code smells considered as well as information about the `62` software metrics used.

Next step **(2)** was to apply the decision tree algorithm. The tool **C5.0** was used, as it is regarded as state of the art in the domain. Details on the algorithm are presented in Section III-C. To asses the statistical validity of the analysis, *10-fold crossover validation was followed*.

In the step **(3)** the following measurements were collected: the average classification error rate, the average size of the decision trees and the quantities of true positives, false positives, false negatives. Based on this, we were able to calculate the values of the metrics used (Section III-B).

Afterwards, step **(4)** was to apply the existing code smell detection tools, i.e. the approaches based on general rules (Section III-D), and gather the respective results. Four tools regarded as state of the art in detection of code smells by static analysis were used.

Similarly, step **(5)** was to use other machine learning approaches to detect code existence of code smells, and gather the respective results. Two algorithms were used: Support-vector Machine (SVM) and Bayesian Beliefs Neworks (BBNs), as detailed in Section III-E.

Step **(6)** was to apply the genetic algorithms. It was also compared with a second prototype implemented, one which combines a genetic algorithm with the decision trees to provide metric selection (see Section III-F).

The final step **(7)** was the analysis. Basically, using the gathered the results, the values of the metrics were computed (see Section III-B). The results obtained **(3)** were then compared with the manual classification values and with the ones obtained in **(4)**, **(5)**, and **(6)**.

The following subsections present the details on the experimental approach and steps preformed. The results obtained and the respective discussion is presented in Section IV.

A. *Subject Selection*

In our research study, we used a data set containing information about various metrics calculated for each code piece (in our case, a code piece is a *Class*, as in Object Oriented Programming) and code smells detected in four open source software projects developed in the Java programming language: Eclipse [17] 3.3.1, Mylyn 3.1.1 [18], ArgoUML [19] 0.26 and Rhino 1.6 [20].

The **Eclipse** project consists of an Integrated Development Environment (IDE) that supports many languages through the use of plugins, it is a mature project with more than 3.5MLOCs (million lines of code) distributed along more than 10,000 Classes. **Mylyn** is an Eclipse plugin designed to help developers deal with information overloading, it is also developed by the Eclipse community and consists of more than 200KLOCs and more than 1500 Classes.

The **ArgoUML** project is a popular UML modeling tool with about 300KLOCs and 2000 Classes. **Rhino** is the smallest project that we analyzed, it is an implementation in JavaScript (used in Mozilla Firefox, for example) whose development is managed by the Mozilla Foundation, and consists of only 70KLOCs and 200 Classes.

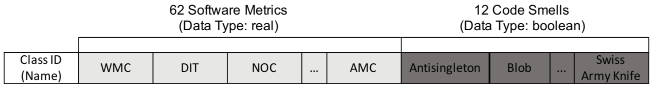
The information about the existence of each Bad Smell in the Classes within the code of each project above was derived from the dataset used in [5], from the **Ptidej** group researchers, and available at [32]. This dataset contains information about the 12 bad smells considered in this study: *Antisingleton, Blob, Class Data Should Be Private, Complex Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chains, Refused Parent Bequest, Speculative Generality, and Swiss Army Knife*. Information regarding each of these bad smells can be found in a paper from the same group [33].

To obtain the software metrics, two tools were used: CKJM [34] and POM [35]. Both tools are able to calculate metrics for Classes by analyzing the project's .jar files, which, in our case, were obtained in the project's download page.

The CKJM and POM were used to calculate 18 and 44 metrics respectively for each Class contained in the .jar files, resulting in 62 columns in our dataset. Examples of the metrics used are the number of lines of code (LOC), number of parameters (NOParam), Depth of Inheritance Tree (DIT) (the complete list of metrics is available at https://goo.gl/T2ifbC). Some metrics are calculated by both tools but we decided to keep both versions since they are calculated differently in each tool, and thus, produce different values.

It is important to observe that some code smells are more related with some metrics than with others, and thus, the models used to detect them give more importance to some metrics than others, and consequently are also more sensitive to some of those metrics.

Finally, in order to create a single dataset for our experiments, we merged the information of both datasets using the Class names as matching key. Some Classes found in the code smells dataset were not found in CKJM or POM outputs and vice-versa, therefore they were be removed. Thus, only the 7952 Classes with consistent data among the datasets were considered for the experiments preformed. Figure 1 shows the structure of the final dataset.



	62 Software Metrics (Data Type: real)					12 Code Smells (Data Type: boolean)			
Class ID (Name)	WMC	DIT	NOC	...	AMC	Antisingleton	Blob	...	Swiss Army Knife

Figure 1. Schema for the dataset used in this study.

B. *Effectiveness Metrics*

To characterize the effectiveness of the quality of the models generated, we decided to adopt three metrics widely used in domains such as information retrieval, machine learning and other domains that involve binary classification [36]: **Precision**, **Recall** and **F-Measure**.

$$\text{Precision} = \frac{TP}{FP + TP}$$

$$\text{Recall} = \frac{TP}{FN + TP}$$

Where:

TP: True Positives is the number of positive instances *correctly* classified as positive.

FP: False Positives is the number of negative instances *incorrectly* classified as positive.

FN: False Negatives is the number of positive instances *incorrectly* classified as negative.

F-Measure is no more than the *harmonic mean* between Precision and Recall, and it is represented by:

$$\text{F-Measure} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

C. Decision Tree Algorithm

The decision tree classification algorithm has the *very valuable feature of being easily inspected by a developer or software engineer* involved in the project that is the target of the approach. This means that the models can be validated, discussed, learned and even improved by the team members. This represents a major advantage when compared to many other machine learning techniques that generate "black box" models.

The decision tree algorithm is a statistical model generation technique that uses *supervised training* for data classification and prediction. This means that the classes of each instance in the training set are known. The produced classification model presents a tree structure, where the highest level node is the root node, each internal node (non-leaf) is a test attribute and each terminal node (leaf) has a class label [37]. After learning the model parameters, the tree is able to classify an instance whose class is still unknown, according to the path from the root to a leaf.

Inside the decision tree algorithm family, the `C5.0` algorithm is the most advanced one and the one that produces better results. The algorithm is trained using the values of software metrics (e.g. number of lines of code, methods, parameters, etc) about the Classes of the pre-annotated training dataset. This way, the rules produced are based on these metrics.

In our proposal, we have used a training dataset containing various metrics calculated for each code piece (in our case, a code piece is a Class[1], as in Object Oriented Programming) from the studied projects in addition to the information of whether that Class contains each of the studied Code Smells. This way, the `C5.0` algorithm is able to generate Decision Trees (as the one in Figure 2) through which it is now possible to determine if a certain Class contains a certain Code Smell as long as its metrics are known.

Figure 2 shows the example of a model to detect the existence or not of a specific type of code smell in a class.

As we can observe, the non-leaf tree nodes represent software metrics, while the leaf nodes represent the model prediction as to the existence of the code smell analyzed. The complete set of metrics consists of a total of 62 metrics. The `C5.0` algorithm tends to allocate the most significant metrics for the detection of the Code Smell on the tree top, while the less significant metrics appear as the tree deepens and gets more specialized.

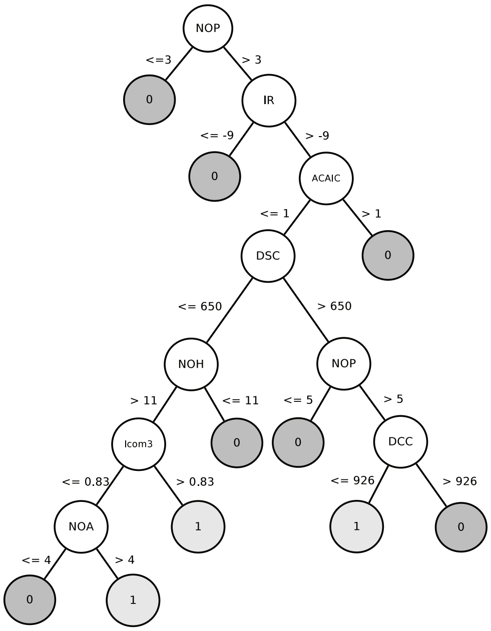


Figure 2. A decision tree for detecting the "Swiss Army Knife" smell.

Our preliminary experiments showed that the `C5.0` algorithm is not always able to select the *optimal* most significant metrics for the construction of the tree. However, after the set of metrics given as input is readjusted, removing some of them, the algorithm is able to obtain better and more precise decision trees. Later we will show how the process of selecting the relevant metrics can be automatized by adding a genetic algorithm to the approach (see Section III-F).

D. Rule Based techniques

Several techniques exist for the automated detection of code smells but most of them are based on static code analysis and reports the code smells based on fixed-threshold rules. We selected to be used in our experiment four state of the art tools able to report code smells in Java code: inCode [38], inFusion [28], iPlasma [30] e PMD [25].

The tools inCode [38] and inFusion [28] are commercial tools both developed by the same company (Intooitus) and claim to offer the automated detection of code smells such as "God Class", "Data Class", "Code Duplication", "Data Clumps", among others. The tool *inCode* is simpler and is considered a light-weight version of *inFusion*.

The tool *iPlasma* is a research product that is still not available commercially and is described in [30] as an integrated environment for the analysis of the quality of object oriented software systems. It is able to detect code smells such as "God Class", "Refused Parent Request", among others.

Finally, the tool PMD, described as source analyzer, finds all sort of programming problems and best practices not being following. It also is able to find some code smells such as "God Class", "Excessive Public Count", "Excessive Parameter List", among others.

All these tools work with a set of rules based on software metrics to detect code smells. These rules are usually pre-

[1] Class with capital C is not to be confused a class as in classification.

defined, but they can also be customized by the developing teams. However, depending on the smells, the rules may be inter-dependent, which makes the customization harder.

E. Other Machine Learning techniques

Existing works have used machine learning techniques in order to identify code smells with higher accuracy. In our experiment, we compared the effectiveness of the *Decision Tree* algorithm to recognize code smells with two state of the art machine learning techniques: Support-vector Machine (SVM) [39] and Bayesian Beliefs Network (BBN) [40], [41].

SVM is a technique based on statistical theory of supervised learning that uses *kernel* functions to perform an optimal separation of data into two categories by a hyperplane [39].

A BBN is composed by a network structure (in the form of nodes and arcs) and conditional-probability tables describing the decision processes between each node [40], [41]. The conditional probabilities are learned by using historical data. The network ensures the qualitative validity of the approach while appropriate conditional tables ensure that the model is well-calibrated and is quantitatively valid.

F. Decision Tree together with Genetic Algorithm

In our experiment, we evaluated if the effectiveness of the Decision Tree algorithm can be improved by combining it with Genetic Algorithm. In particular, the genetic algorithm is responsible for selecting the *features* (in our case, software metrics) to be used by the decision-tree algorithm to recognize code smells. In a nutshell, this combination works as follows (see Figure 3).

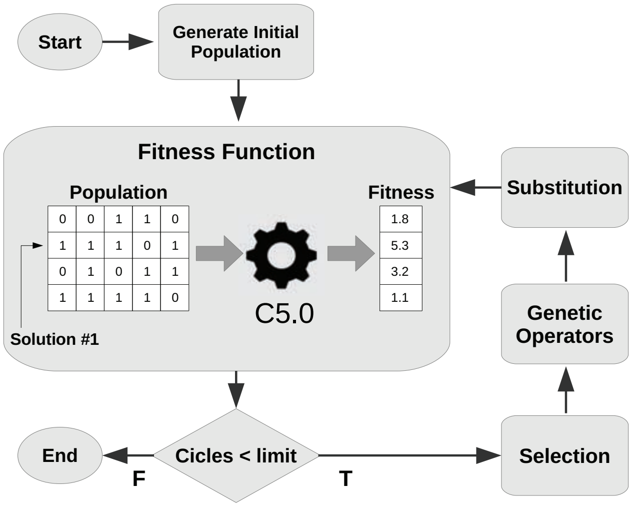


Figure 3. Overall representation of the combination Decision Tree and Genetic Algorithm.

In our approach, each solution is a binary vector representing the total number of metrics available (the set of 62 metrics put together in Section III-A), and where each bit represents whether a certain metric is going to be used or not by the classifier algorithm. These solutions will be evolved, seeking for better subset of metrics. The initial population is created containing a configurable number of random solutions. In each of these random solutions every metric has configurable probability of being selected or not.

For each solution, (i.e. for each set of metrics) a decision tree is built by executing the Decision Tree algorithm. After building the decision-trees, they are used to identify code smells in software projects and we use the *error rate* of the identification to evaluate the *fitness level* of each one. If the *fitness* of any decision-tree satisfies a algorithm stopping criterion, the learning process is stopped and the tree with the best *fitness* is used to detect code smells.

Otherwise, the process continues by applying evolutionary operations on the set of metrics. First, the sets of metrics associated to the decision-trees with higher *fitness* are selected and genetic operations (such as, *mutation* and *crossover*) are applied to these selected set of metrics. Finally, the resulting sets of metrics are used to create a new population and, then, to generate new decision-trees.

IV. Results and Discussion

The following subsections present and discuss the results obtained and the lessons learned during this study. Due to space constraints we do not present the detailed results. Interested readers can obtain them at https://goo.gl/T2ifbC.

A. Detection Trees Effectiveness

Table II presents the overall results obtained for the decision tree algorithm `C5.0`. For each of the code smells considered, we present the average values of precision, recall and F-Measure achieved by the models generated.

Table II. Results obtained by the C5.0 Algorithm.

Code Smell	Precision	Recall	F-measure
AntiSingleton	0.735	0.611	0.667
Blob	0.606	0.424	0.499
ClassDataShouldBePrivate	0.779	0.616	0.688
ComplexClass	0.871	0.842	0.856
LargeClass	0.826	0.737	0.779
Lazyclass	0.930	0.910	0.920
LongMethod	0.821	0.827	0.824
LongParameterList	0.972	0.959	0.965
MeassageChains	0.718	0.581	0.642
RefusedParentRequest	0.862	0.829	0.845
SpeculativeGenerality	0.535	0.518	0.526
SwissArmyKnife	0.725	0.685	0.704

The analysis of these results tries to answer to the research question **Q1** presented on Section III. The results indicate that most cases the algorithm used is able to generate rules effective in the detection of code smells in the used dataset.

In general terms, the F-Measure values are above 0.67 except for two of the code smells considered (*Blob* and *SpeculativeGenerality*).

From the results obtained for *Blob* and *SpeculativeGenerality*, we conclude that the characteristics of the code smells probably are not the best to be detected based on the software metrics (or at least, not based on the ones used), or cannot be generalized by the techniques used. In fact, in both cases the recall was less than 0.5 which means that less than half of the existing code smells of these types were detected. In the case of *SpeculativeGenerality*, almost half of the reported cases were false positives. This is related to the ambiguous nature of the code smell.

On the other end of the spectrum, code smells like *Lazyclass* and *LongParameterList* present very good results. These are code smells that are less subjective (as their own name shows) and thus are easier to learn and classify.

This discrepancy of the results also comes to reaffirm the need of techniques that are able to learn from other projects and generalize, helping serving as support for developers to find parts of their code that can be improved.

Although the results seem promising, it is very hard to draw definitive conclusions only based on the absolute results of metrics like precision, recall and F-Measure. This way, the following sections analyze these results with other scenarios that help in establishing comparison.

B. Comparison with Rule-Based Techniques

Table III presents the results obtained by the rule-based techniques evaluated. Like in the previous analysis, we present the average values of precision, recall and F-Measure. We also provide the difference (Δ) between this F-Measure to ease the analysis, with the better results signaled with a $+$ and the worse results signaled with $-$.

Table III. RESULTS FOR RULE-BASED TOOLS.

Tool	Code Smell	Precision	Recall	F-measure (Δ)
inCode	**Blob**	0.589	0.177	0.272 (-0.183)
	LongParameterList	0.591	0.458	0.516 (-0.455)
	MeassageChains	0.922	0.650	0.726 (+0.088)
InFusion	**LongParameterList**	0.260	0.630	0.370 (-0.601)
iPlasma	**Blob**	0.711	0.280	0.402 (-0.053)
	RefusedParentRequest	0.445	0.070	0.544 (-0.298)
PMD	**Blob**	0.554	0.497	0.524 (+0.069)
	CDSBP	0.341	0.352	0.346 (-0.340)
	LM	0.752	0.311	0.440 (-0.373)
	LPL	0.922	0.095	0.173 (-0.798)

These results aim at answering to the research question **Q1** presented on Section III. The first observation is that the number of different types of code smells supported by each tool is rather small, with the four tools reporting only 6 distinct code smells of the ones annotated in our *dataset*. We can also observe that in the case of code smells reported by different tools, the results are very contrasting, highlighting the need but also the difficulty of selecting just one tool to use.

From the total of 12 results, the decision tree algorithms presents better F-Measure in 10. Additionally, we can see that the in the two cases in which the tools performed better, the difference is rather small ($+0.05$ and $+0.07$), while in the other way around the differences are much larger. Although we do not highlight the difference in the table, comparing the values for recall and precision reveal a similar pattern.

The observed results indicate that the algorithm is able to learn and to extract models that perform more effectively than pre-defined sets of rules.

C. Comparison with Intelligent Techniques

Tables IV and V present average values of precision, recall and F-measure obtained by the techniques support-vector machine and Bayesian Beliefs Networks, respectively. In addition, We describe the difference (Δ) between this F-Measure and the ones obtained by *Decision Tree* algorithm, with the better results signaled with a $+$ and the worse results signaled with $-$.

Table IV. RESULTS OBTAINED BY THE SUPPORT-VECTOR MACHINE ALGORITHM.

	Precision	Recall	F-measure (Δ)
AntiSingleton	0/0	0.000	0/0 (-0.667)
Blob	0.830	0.084	0.153 (-0.346)
ClassDataShouldBePrivate	0.736	0.247	0.370 (-0.318)
ComplexClass	0.884	0.569	0.692 (-0.164)
LargeClass	0.804	0.270	0.404 (-0.375)
Lazyclass	0.783	0.644	0.707 (-0.213)
LongMethod	0.791	0.639	0.707 (-0.117)
LongParameterList	0.859	0.597	0.704 (-0.261)
MeassageChains	0.792	0.560	0.656 (+0.014)
RefusedParentRequest	0.730	0.301	0.426 (-0.419)
SpeculativeGenerality	0/0	0.000	0/0 (-0.526)
SwissArmyKnife	0.600	0.056	0.102 (-0.602)

As shown in Table IV, from the total of 12 results, the decision tree algorithms presents better F-Measure in 11. Although we do not highlight the difference in the table, comparing the values for recall and precision reveal a similar pattern. In order to increase the confidence level of such results, we apply a statistic test, named *t-test* [21]. The t-test was used to verify if the values of the precision, recall and f-measure resulting of the Decision Tree algorithm differ (or not) of the ones generated by using the technique SVM. By applying the t-test to the means of the precision, recall and f-measure resulting of these two techniques we obtained the *p-values* 0.09, 0.0001 and 0.0008, respectively. According to [21], to represent a significant difference between two means, normally, the *p-value* should be lower than `0.05`. Thus, we conclude that the difference between the means of the recall and f-measure generated by the use of Decision Tree is significant when compared with the ones generated by the use of the SVM. However, we note that mean of the precision values do not present a difference statistically significant. In this way, we confirm that the Decision Tree present a better recall and f-measure when comparing with the SVM, but we can not confirm that Decision Tree reach a better precision.

Table V. RESULTS OBTAINED BY THE BAYESIAN BELIEFS NETWORKS ALGORITHM

	Precision	Recall	F-measure (Δ)
AntiSingleton	0.240	0.266	0.252 (-0.415)
Blob	0.463	0.346	0.396 (-0.103)
ClassDataShouldBePrivate	0.338	0.231	0.274 (-0.414)
ComplexClass	0.649	0.718	0.682 (-0.174)
LargeClass	0.235	0.484	0.316 (-0.463)
Lazyclass	0.285	0.938	0.437 (-0.483)
LongMethod	0.626	0.431	0.511 (-0.313)
LongParameterList	0.493	0.327	0.393 (-0.572)
MeassageChains	0.510	0.382	0.437 (-0.205)
RefusedParentRequest	0.215	0.940	0.350 (-0.495)
SpeculativeGenerality	0.025	0.890	0.049 (-0.477)
SwissArmyKnife	0.059	0.389	0.102 (-0.602)

The results described in Table V indicate that the Decision Tree algorithm reaches better F-measure in all cases when comparing with the technique BBN. The same observation can be seen to the values of precision and recall. However, it is necessary to apply the t-test to verify whether such observations can be confirmed statistically. In this case, the t-test was used to verify if the values of the precision, recall and f-measure resulting of the Decision Tree algorithm differ (or not) of the ones generated by using the technique SVM. By applying the t-test to the means of the precision, recall and f-measure resulting of these two techniques we obtained the *p-values* 0.000001, 0.028 and 0,000002, respectively. Thus, we conclude that the difference between the means of the precision, recall and f-measure generated by the use of Decision Tree is significant when comparing with the ones generated by the use of the BBN. In this way, we confirm that the Decision Tree present a better precision, recall and f-measure when comparing with the SVM.

D. Using Genetic Algorithms for Metric Selection

To answer to the research question **Q4** presented on Section III, we executed the combined approach presented in Section III-F and we compared the obtained results with the ones presented in Section IV-A.

Table VI presents the difference (Δ) between the values of precision, recall and F-Measure, again with the better results signaled with a $+$ and the worse results signaled with $-$.

As we can observe, the results improve in (almost) all the cases when using the the genetic algorithm for metric selection. In fact, the results for precision and F-measure are always higher, while the values for recall are only slightly smaller in two cases. Regarding the magnitude of the improvement, in average the results improved $+0.03$, but in several cases the improvement was higher than $+0.10$.

The improvements in precision were much higher than the improvements in the recall. This has origin in the fact that the function used (error rate) can be biased by the prevalence of the classes. This might be also an indication to, in future studies, use different fitness functions that are not bias by prevalence or skew. Also, in different scenarios, other fitness functions may be convenient depending on the characteristics of the model that we ought to maximize.

Finally, another improvement of the technique that is not visible on the table is the reduction of the size of the trees obtained. In practice, the models become smaller, faster to apply and easier to understand or modify by developers. This facilitates that developers may discuss and learn with the models, as well as improving them.

These observations indicate that also in this domain *feature selection* is an important task and that the use of a genetic algorithm may be an effective way to do it in an automated fashion, reducing the intervention of the human.

Table VI. IMPROVEMENTS (Δ) INTRODUCED BY GEN. ALGORITHM

	Δ Precision	Δ Recall	Δ F-Measure
AntiSingleton	+0.045	+0.022	+0.031
Blob	+0.078	+0.011	+0.028
ClassDataShouldBePrivate	+0.058	+0.025	+0.037
ComplexClass	+0.004	+0.015	+0.010
LargeClass	+0.049	+0.107	+0.081
LazyClass	+0.026	+0.027	+0.027
LongMethod	+0.027	+0.043	+0.035
LongParameterList	+0.007	+0.016	+0.012
MessageChains	+0.055	-0.018	+0.011
RefusedParentRequest	+0.006	+0.013	+0.010
SpeculativeGenerality	+0.128	-0.006	+0.042
SwissArmyKnife	+0.025	+0.093	+0.057

V. THREATS TO THE VALIDITY OF THE EXPERIMENT

After analyzing the results of our experiments, there are some points that threat its validity and that must be discussed.

Representativeness of the Dataset

The generalization of the results depends on the representativeness of the software used. We belive that the four projects used provide enough diversity and a sizable sample which provides some support to the data. Eclipse is open source and maintained by a very large and active community of software developers. Not all are so famous, but all are used by large communities.

Quality of the Data

The quality of the results obtained depends on the quality of the data used. The dataset comes from an independent work has origin in the work of [5]. It contains annotations about the presence of 12 code smells in each class. Obviously, the degree of subjectiveness of the problem and human nature of the classification task make it prone to have mistakes. However, the study was preformed by experts, which provides us with a confidence that the classification mistakes might be in acceptable values.

Size of the Sample

We are aware that the amount of the data used in the experiment may be too small, besides the limitations described above. But the main goal of this work was to understand if the decision tree algorithms are able to learn classification rules and how effective these rules are when compared with the existing alternatives. For further evaluations or to propose methodologies or tools, it will be necessary to gather and classify larger amounts of data.

Imprecise definition of Code Smells

Due to the different nomenclatures used to refer to the same code smell by different authors, during the experiments with rule based tools in Section IV-B. Thus, it was necessary to establish equivalencies between the names to allow a comparison of the results. To do this, the textual description of the smells reported by each tool were examined to find matches with the 12 types of smells existing in the dataset. This is a task based on the interpretation of text, which leads to ambiguities and decision, but which we tried to minimize in order to reduce the impact on the data.

VI. RELATED WORK

There are several proposals that have recently focused on recognizing the *code smells* described in [2]. These proposals range from automatic to guided manual approaches.

For instance, [6], [8], [9], [10] propose techniques that rely on declarative rules specification to detect smells. In these settings, rules are manually defined to identify the key symptoms that characterize a smell. These symptoms are described using quantitative metrics, structural, and/or lexical information. [7] analyses some of these tools, and, comparatively, investigate them against their usefulness in assisting the main stakeholders of the software development process when assessing the code quality of a software. In particular, [7] analyzes the tools *Checkstyle*, *DECOR*, *inCode*, *inFusion*, *JDeodorant*, *PMD* and

Stench Blossom in order to evaluate its ability to expose the pieces of code which are most affected by structural decay, and the relevance of tool responses with respect to future software evolution. The authors conclude that such tools are certainly useful to recognize smells, but they report that the manual specification of the rules to be used by these tools is a difficult and time-consuming tasks.

In this way, some authors [11], [12], [13], [42] propose automatic approaches that, instead of manually specifying the rules for detecting each code smell type, such rules are inferred by using intelligent techniques and software information. For instance, in [13] rules are extracted from instances of maintainability smells by using Bayesian Belief Networks or Support-vector machine [14]. [42] proposes a technique, named Historical Information for Smell deTection (HIST), to detect source code smells based on change history information extracted from versioning systems. The results shown in [42] indicate that HIST is able to identify code smells that cannot be identified through other approaches based on code analysis.

Although such automatic approaches have provided solutions able to recognize smells with higher accuracy and lower time-consuming, they still generate inaccurate results and do not present a comprehensive explanation how their results have been obtained. It is important that the applied reasoning to achieve such results be comprehensive to the developer to increase his confidence in the identified smells and he can contribute to the detection process by providing feedback. Studies [14] indicate that the developer's feedback can improve smells identification.

VII. Conclusion

In this paper, we presented a practical experience report that evaluates the use of *Decision Tree* classification algorithm to recognize *code smells* in different software projects. Such study contributes to the state-of-the-art on *code smells* detection by analyzing the use of the *Decision Tree* algorithm to recognize them in an automated fashion.

Our study evaluated the performance of the *Decision Tree* algorithm to detect 12 kind of smells into 4 open source projects. Then, we compared the Precision, Recall and F-measure results generated from application of the Decision Tree with the ones showed by general rules based approaches. We also compared the Decision Tree results with the ones generated by intelligent techniques based approaches. In both experiments, the *Decision Tree* reaches better performance in most cases.

As future work we will apply the *Decision Tree* in other software projects and we will investigate how this technique can be improved by using the developer's feedback and software information, such as, changes, faults and tests history.

Acknowledgment

This work has been partially supported by the project DEVASSES – *DEsign, Verification and VAlidation of large-scale, dynamic Service SystEmS*, funded by the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no `PIRSES-GA-2013-612569`.

References

[1] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London., 1985.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] M. Abbes, F. Khomh, Y. G. Gueheneuc, and G. Antoniol, "Anempirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering*, ser. CSMR 11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 181–190.

[4] F. Khomh, M. D. Penta, and Y. G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering*, ser. WCRE 09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 75–84.

[5] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[6] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–359.

[7] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, 2012.

[8] J. Padilha, J. Pereira, E. Figueiredo, J. Almeida, A. Garcia, and C. Sant'Anna., "On the effectiveness of concern metrics to detect code smells: An empirical study." in *Proc. of the 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014)*, 2014.

[9] T. Tourwé and T. Mens, "Identifying Refactoring Opportunities Using Logic Meta Programming," in *European Conference on Software Maintenance and Reengineering*, 2003.

[10] T.-W. Kim, T.-G. Kim, and J.-H. Seu, "Specification and automated detection of code smells using ocl," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 4, July 2013.

[11] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, 2012.

[12] M. Kessentini, W. Kessentini, and A. Erradi, "Example-based design defects detection and correction," *Journal of Automated Software Engineering*, 2011.

[13] Y. Guéhéneuc and H. Sahraoui, "A bayesian approach for the detection of code and design smells foutse," in *9th International Conference on Quality Software*, 2011.

[14] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, and E. Aimeur, "SMURF: A SVM-based Incremental Anti-pattern Detection Approach," in *2012 19th Working Conference on Reverse Engineering (WCRE)*. IEEE, pp. 466–475.

[15] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining*, ser. Practical Machine Learning Tools and Techniques. Morgan Kaufmann, Jan. 2011.

[16] R. C. Barros, M. P. Basgalupp, A. C. P. L. F. de Carvalho, and A. A. Freitas, "A survey of evolutionary algorithms for decision-tree induction," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 42, no. 3, pp. 291–312, MAY 2012.

[17] F. Eclipse, "The eclipse foundation open source community website," Available in http://www.eclipse.org, 2013, last access in October 2013.

[18] ——, "Eclipse mylyn open source project," Available in http://www.eclipse.org/mylyn, 2013, last access in October 2013.

[19] Tigris, "Argouml," Available in http://argouml.tigris.org, 2013, last access in October 2013.

[20] M. D. Network and individual contributors, "Rhino — mdn," Available in http://www.mozilla.org/rhino, 2013, last access in October 2013.

[21] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[22] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[23] D. Fowler, Martin; Beck, K.; Brant, J.; Opdyke, W.;Roberts, *Refactoring: improving the design of existing code*. Pearson Education India, 2002.

[24] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, 2012.

[25] InfoEther, "Pmd," Available in http://pmd.sourceforge.net/, 2013, last access in October 2013.

[26] CheckStyle, "Dowload checkstyle," Available in http://checkstyle.sourceforge.net/, 2014, last access in March 2014.

[27] JDeodorant, "Dowload jdeodorant," Avalable in http://www.jdeodorant.com/, 2014, last access in March 2014.

[28] Intooitus, "infusion — intooitus - source code analysis and quality assesment tools," Available in http://www.intooitus.com/products/infusion, 2013, last access in October 2013.

[29] GranttProject, "Dowload granttproject," Available in http://www.ganttproject.biz/, 2014, last access in March 2014.

[30] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design," in *In ICSM (Industrial and Tool Volume*. Society Press, 2005, pp. 77–80.

[31] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating software refactoring with appropriate resolution order of bad smells," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 265–268.

[32] P. Team, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," Available in http://www.ptidej.net/downloads/replications/emse10/, 2010, last access in October 2013.

[33] F. Khomh, M. Di Penta, and Y. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, Oct 2009, pp. 75–84.

[34] M. Jureczko and D. Spinellis, *Using Object-Oriented Design Metrics to Predict Software Defects*, ser. Monographs of System Dependability. Wroclaw, Poland: Oficyna Wydawnicza Politechniki Wroclawskiej, 2010, vol. Models and Methodology of System Dependability, pp. 69–81.

[35] Y.-G. Guehneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," *2013 20th Working Conference on Reverse Engineering (WCRE)*, vol. 0, pp. 172–181, 2004.

[36] N. Antunes and M. Vieira, "On the Metrics for Benchmarking Vulnerability Detection Tools," in *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*. Rio de Janeiro, Brazil: IEEE, 2015.

[37] J. Han and M. Kamber, *Data mining: concepts and techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[38] Intooitus, "incode — intooitus - source code analysis and quality assesment tools," Available http://www.intooitus.com/products/incode, 2013, last access in October 2013.

[39] C. Cortes and V. Vapnik, "Support vector machines and other kernel-based learning methods," *Machine Learning*, vol. 20, 1995.

[40] R. G. Cowell, R. J. Verral, and Y. K. Yoon, "Modeling operational risk with bayesian networks," *Journal of Risk and Insurance*, vol. 4, 2007.

[41] P. Szolovits, "Methods of information in medicine," *Uncertainty and Decision in Medical Informatics.*, 1995.

[42] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2013, pp. 268–278.