Comparison of Machine Learning Methods for Code Smell Detection Using Reduced Features

Kanita Karađuzović-Hadžiabdić
Faculty of Engineering and Natural Sciences,
Computer Sciences and Engineering Program,
International University of Sarajevo
Sarajevo, Bosnia and Herzegovina
kanita@ius.edu.ba

Rialda Spahić
Equinor ASA Research Center
Trondheim Rotvoll, Norway
rias@equinor.com

***Abstract*—We examine a machine learning approach for detecting common Class and Method level code smells (Data Class and God Class, Feature Envy and Long Method). The focus of the work is selection of reduced set of features that will achieve high classification accuracy. The proposed features may be used by the developers to develop better quality software since the selected features focus on the most critical parts of the code that is responsible for creation of common code smells. We obtained a high accuracy results for all four code smells using the selected features: 98.57% for Data Class, 97.86% for God Class, 99.67% for Feature Envy, and 99.76% for Long Method.**

Keywords—code smells, machine learning, feature selection

I. Introduction

Code smells are signals of bad coding and design practices. They are an established concept for patterns that cause problems for software development and maintenance and are closely related to object-oriented principles. Design patterns are definitions of good software design practices. Gamma et al. [1], often also referred to as the Gang of Four introduced the first concept of design patterns in 1994. With the rise of design patterns, software performance antipatterns were documented in 1998. The antipatterns are conceptually similar to design patterns and are named as antipatterns due to their misuse that produces negative consequences. Smith and Williams [2] document common mistakes made during software development as well as bad coding practices. Together with performance antipatterns detection, refactoring opportunities are identified as applicable design patterns that remove the detected antipatterns. Code smells indicate code quality. What makes code smells different from performance antipatterns is that they are unknown to be worst practice, but rather indicate bad code and potential future problems in software quality. Each code smell has certain characteristics that determine to what extent is software quality harmed. Some of the most severe code smells are Data Class, God Class, Feature Envy, Long Method, Blob, Functional Decomposition and the famous Spaghetti Code. In this study, we focus on two Class level code smells: Data Class and God Class, and two Method level code smells: Feature Envy and Long Method.

Data Class is a class that only consists of fields for getters and setters. This class is often manipulated by other classes as they serve only as data holders and have no meaningful purpose. *God Class* is a class that serves as a central intelligence of a system. This kind of class is often too complex, consists of too much code that should not be placed there and serves different functionalities that should be separated. God class uses most of the other resources from the system and contains a lot of information. *Long Method* is a code smell that refers to methods that tend to centralize the functionality of a class and ends up being too complex and difficult to understand. Such methods use large amounts of data from other classes. *Feature Envy* is a classical code smell where a class uses fields of other classes in their methods, rather than using the fields that belong to its own class. This kind of coding increases coupling of the class, and as a result increases the code maintenance costs.

What kind of impact do the smelly classes have on the overall performance of software depends on the level of actual harmfulness. If the impact of a smelly code is severe on the software performance, does the code smell evolve into an antipattern? To answer this question, the first step is to identify the most relevant discriminative features (or attributes) that can be used to detect code smells. Selected features may then be compared to the antipatterns to find what coding practices are most likely to produce code smells in a software system.

Findings based on research performed by Hozano et al. [3] conclude that developers detect code smells in significantly different ways. There is also a need for accurate and customized detection of code smells. According to [3] there is a disagreement of how developers detect code smells. One of the aims of this work is to identify most relevant features that will result in the mentioned Class Level and Methods Level code smell detection.

978-1-5386-7893-0/18/$31.00 ©2018 IEEE

Until recently, not much research has been done on code smell detection using machine learning methods. Commercial tools for code smell detection and analysis such as InCode [4], Borland Together [5], and JDeodorant [6] were used instead. One of the most recent studies performed on code smell detection using machine learning methods are the works of Fontana et al. [7,8,9]. For detection of Class Level and Method Level code smells, Fontana et al. use 66 and 88 features respectively. In [7], the authors perform binary classification of code smell detection using six machine learning methods. In [8] and in [9], Fontana et al. classify code smells into four categories (no smell, non-severe smell, smell and severe smell) and conduct the classification based on these four categories.

TABLE I. REDUCED FEATURESET FOR CLASS AND METHOD LEVEL CODE SMELLS

Class Level Feature Set		Method Level Feature Set	
Data Class	**God Class**	**Feature Envy**	**Long Method**
Accessor Method no.	No. of Classes	Parameter no.	Max. Nesting Level
No. of Attributes	No. of Methods Overridden	Access to Foreign Data	Lines of Code (LOC)
No. of Inherited Methods	Access to Foreign Data	No. of Message Chain Statements	Cyclomatic Complexity
Response for a Class (RFC)	(LOC)	Locality of Attribute Accesses	No. of Local Variables
No. of Children	LOC of Not Accessor or Mutator Methods	No. of Classes	Called Foreign Not Accessor or Mutator Methods
Weight of Class	Called Foreign Not Accessor or Mutator Methods	No. of Methods with Package Visibility	No. of Methods Overridden
Weighted Methods Count of Not Accessor or Mutator Methods	Tight Class Cohesion		Average Methods Weight of Not Accessor or Mutator Methods
No. of Packages	RFC		No. of Interfaces
No. of Private Methods	Weighted Methods Count		No. of Attributes with Protected Visibility
	Weighted Methods Count of Not Accessor or Mutator Methods		No. of Final Attributes
	No. of Interfaces		No. of Final, Non-Static Attributes
	No. of Protected Attributes		
	No. of Attributes with Package Visibility		
	No. of Final, Non-Static Attributes		
	No. of Standardly Designed Methods		

In this work, we evaluate the performance of 10 machine learning methods in detecting Class Level and Method level code smells. We also examine the most discriminative features to be used in generating Class level and Method level code smells.

In [10], we extended the work of Fontana et al, [7], with an aim to reduce the number of features used in classification of two *Class level* code smells (i.e. Data Class and God Class). In this study, we further the work and also include two *Method level* code smells, Long Method and Feature Envy. We also include four more machine learning methods into our evaluation. To the best of our knowledge, the selection of most relevant features and the effect that they have on the detection accuracy has been performed on Class Level code smells by [10] using six machine learning methods. However, [10] do not perform evaluation nor feature reduction for Method Level code smells, which is the main contribution of this work.

Since the programmers are often under pressure to reduce the delivery time, other than speeding up the task of code smell detection, lower feature number could also encourage programmers (or quality assurance teams) to actually use the proposed machine learning approach to test the presence of code smells. As a result, this could also increase the possibility of code refactoring and thus improved code design which would at the same time automatically lower the cost of future code maintenance.

II. MATERIALS AND METHODS

A. Dataset

The dataset used for this work was obtained from [7]. It contains 1,986 validated code smells obtained from 76 software systems of Qualitas Corpus which is composed of several Java systems categorized into four subsets. Namely, for each analyzed code smell: God Class and Data Class, Feature Envy and Long Method. Each set contains 420 instances, out of which 140 are positive and 280 negative instances. One of the critical issues in classification is the selection of effective discriminative features. We extend the work of Fontana et al. [7] and reduce the number of features used to detect mentioned code smells.

B. Methods

We analyze the performance of 10 machine learning methods for code smell detection. Analyzed methods used were: J48, JRip, Naïve Bayes (NB), Random Forest (RF), Sequential Minimal Optimization (SMO), LibSVM, K-Nearest Neighbours (KNN), Decision Table, Multilayer Perceptron and Voted Perceptron. The first six methods were also used by [7,10] for code smell detection, the latter four methods are added in this work for the evaluation.

III. Results and discussion

We perform binary classification of code smells detection, (i.e. whether the tested code smell exists or not). As already mentioned, Fontana et al. [7] use 66 features for Class level (God Class and Data Class) and 88 features for Method level (Feature Envy and Long Method) code smell detection. We reduce this number to 9 features for Data Class, 15 features for God Class, 6 for Feature Envy and 11 for Long Method as the most critical features for code smell detection. These features are summarized in Table I. Table II shows the obtained machine learning accuracy results for each tested code smell using full set of the features used by Fontana et al. [7] and our proposed reduced feature set. Bolded results show the methods that achieve highest detection accuracy results: 98.57% for Data Class code smell is achieved by the Random Forest method, 97.86% for God Glass code smell is achieved by JRip method, 99.67% for Feature Envy code smell is obtained by Multilayer Perceptron and 99.76% for Long Method code smell is obtained by the Random Forest method.

TABLE II. Comparison of Machine Learning Methods

Machine Learning Method		**Class Level Code Smells**		**Method Level Code Smells**	
		Data Class	***God Class***	***Feature Envy***	***Long Method***
J48	Fontana et al.	97.6	96.4	94.4	98.3
	Reduced Features	98.33	97.38	95.95	99.52
JRip	Fontana et al.	96.6	97.3	96.4	99.2
	Reduced Features	97.14	**97.86**	96.19	99.52
Naïve Bayes	Fontana et al.	81.9	95.4	86.1	93
	Reduced Features	83.81	95.95	89.76	96.43
Random Forest	Fontana et al.	97.8	97.3	95.2	99
	Reduced Features	**98.57**	97.38	96.90	**99.76**
SMO	Fontana et al.	96.9	96.4	93	96.6
	Reduced Features	88.57	96.90	88.33	97.38
LibSVM	Fontana et al.	94.7	76.6	69	69.2
	Reduced Features	82.14	66.67	90.95	85.47
KNN	Full Feature Set	87.14	88.81	85.95	91.67
	Reduced Features	95.71	94.29	94.52	97.86
Decision Table	Full Feature Set	98.10	96.90	95.71	99.05
	Reduced Features	97.86	97.38	95.24	99.29
Multilayer Perceptron	Full Feature Set	95.8042	93.7063	93.007	98.6014
	Reduced Features	97.86	96.43	**99.67**	93.43
Voted Perceptron	Full Feature Set	66.67	64.29	60.95	60.95
	Reduced Features	79.76	69.76	85.29	63.33

All four results are obtained with the selected reduced feature set. By closer inspection of the results, it can be concluded that in general, almost all of the examined methods achieve higher accuracy results when the reduced feature set is used for classification.

IV. Conclusion

In this paper we analyzed the performance of 10 machine learning methods for detection of two Class Level and two Method Level code smells. Most discriminative features for code smell detection were selected and used in the classification. In comparison to the work performed by [7], the selected feature set was considerably reduced for all four analyzed code smells. Furthermore, the results obtained for the four tested code smells are better than those achieved by Fontana et al. [7]. One of the future works is to attempt to determine if and to what extend can a particular feature develop into an antipattern.

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," in *Design Patterns: Elements of Reusable Object-Oriented Software*, USA, Addison-Wesley, 1994.
[2] C. U. Smith and L. G. Williams, "Software Performance AntiPatterns: Common Performance Problems and Their Solutions," Performance Engineering Services and Software Engineering Research, Santa Fe, New Mexico, 2002.
[3] M. Hozano, A. Garcia, B. Fonseca and C. Evandro, "Are you smelling it? Investigting how similar developers detect code smells," *Information and Software Technology,* vol. 93, no. 0950-5849, pp. 130-146, 2017
[4] Incode : http://www.intooitus.com/products/incode
[5] Borland: http://www.borland.com/us/products/together
[6] https://users.encs.concordia.ca/~nikolaos/jdeodorant/
[7] F. A. Fontana, M. Zanoni and A. Marino, "Code Smell Detection: Towards a Machine Learning-based Approach," *2013 IEEE International Conference on Software Maintenance,* pp. 396-399, 2013.
[8] F. A. Fontana, M. V. Mäntylä, M. Zanoni and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering,* vol. 21, no. 3, pp. 1149-1191, 2016.
[9] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Elsevier,* vol. 128, pp. 43-58, 2017.
[10] R. Spahic, K. Karaduzovic-Hadziabdic, “Class Level Code Smell Detection using Machine Learning Methods”, Conference on Computational Methods and Telecommunication in Electrical Engineering and Finance, 2018.