2013 IEEE International Conference on Software Maintenance

Code Smell Detection: Towards a Machine Learning-based Approach

Francesca Arcelli Fontana, Marco Zanoni and Alessandro Marino
Department of Informatics, Systems and Communication
University of Milano-Bicocca
Milano, Italy
Email: {arcelli, marco.zanoni}@disco.unimib.it,
a.marino4@campus.unimib.it

Mika V. Mäntylä
Aalto University
Helsinki, Finland
Email: mika.mantyla@aalto.fil

***Abstract*—Several code smells detection tools have been developed providing different results, because smells can be subjectively interpreted and hence detected in different ways. Usually the detection techniques are based on the computation of different kinds of metrics, and other aspects related to the domain of the system under analysis, its size and other design features are not taken into account. In this paper we propose an approach we are studying based on machine learning techniques. We outline some common problems faced for smells detection and we describe the different steps of our approach and the algorithms we use for the classification.**

***Keywords*—code smells detection, machine learning techniques**

I. Introduction

Many tools for code smells detection have been developed, some of them are available to be used, for other tools one can find the paper on the tool but not the tool and other are commercial tools. The results provided by the detectors are usually different, as we observed in a previous paper [1], code smell detectors do not agree in their answers. Usually detection rules are based only on the computation of a set of metrics, 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 smell can be different, for example for the Large Class smell a tool can use different metrics of cohesion and complexity, while another one only the LOC metric; moreover, also if the metrics are the same, the thresholds of the metrics can be different. By changing this value, the number of smells increases or decreases accordingly. Some tools allow to change and set this value. 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 design of the analyzed system are not usually considered. Detection rules of smells not based on metrics computation are provided by the JDeodorant tool [2], which is able to detect four smells through the opportunities of applying refactoring steps to remove the smells.

Detection rules based on machine learning techniques have not been largely explored, and this is the reason why we decided to exploit them. In the paper we describe the process we follow, the empirical analysis we carried out and the machine supervised learning techniques which we experimented on the Qualitas Corpus repository [3]. For all the reasons reported above and the intrinsic informal and subjective definition of smells, a benchmark platform has not been yet developed. Through our experimental analysis on a set of 76 systems of the Qualitas Corpus, we aim to provide also a significative dataset to be used as a basis for future benchmarks of code smells detectors.

The paper is organized through the following Sections: in II we introduce some related works; In III we briefly outline some common problems encountered through our experiences in code smells detection; in IV we describe our approach based on machine learning techniques and finally in V we provide some concluding remarks.

II. Related Work

We found few works in the literature exploiting machine learning techniques for code smell detection.
Maiga et al. [4] introduce SVMDetect, an approach to detected anti-patterns, based on support vector machines. The subject of their study are Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife antipatterns on three open-source programs: ArgoUML, Azureus, and Xerces. Khomh et al. [5] 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. Maiga and Ali [6] introduce SMURF, an approach to detect antipatterns, based on a machine learning technique with support vector machines and taking into account practitioners' feedback. Yang et al. [7] study the judgment of individual users by applying machine learning algorithms on code clones.

As we can see the principal differences of the previous works respect to our approach is that the above papers are principally focused on the detection of antipatterns and not of code smells of Fowler [8], they did their experimentations by considering only 2 or 3 systems and they usually experiment only one machine learning algorithm. While in our approach we focus our attention on 4 code smells, we consider 76 systems for the analysis and our validation and we experiment 6 different machine learning algorithms.

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.56

III. Considerations ons code smell detectors results

In a previous paper [1] we presented a comparison of four code smell detection tools and an assessment of the agreement, consistency and relevance of the answers produced. We investigated whether different tools for code smell detection, based on different algorithms, agree on their results or not. We considered smells analyzed by at least two different tools: Duplicated Code (analyzed by Checkstyle[1] and inFusion [9]), Feature Envy and God Class (analyzed by JDeodorant [2] and inFusion), Large Class and Long Parameter List (analyzed by Checkstyle and PMD[2]), Long Method (analyzed by Checkstyle, PMD and JDeodorant). We computed the kappa statistics of the tools, which is basically an attempt to balance the amount of accordance between the tools and the amount of accordance due to randomness (tools returned the same results, but not for the same reason). Our experiments outlined that different detectors for the same smell do not meaningfully agree in their answers. Nevertheless, they can detect problematic regions of code which are relevant for the future evolution of software. We observed that we have the best agreement in the results for the God Class detection and then for Large Class and Long Parameter List smells.

In another analysis we have done [10], we outline another problem regarding the reliability of the results of the detectors, in particular related to the recall values. With this aim, we have manually checked all the results provided by the tool iPlasma[3] on the detection of two smell, God and Data Class, on twelve systems of the Qualitas Corpus. The results of this analysis revealed that there is a high number of false positive results, detected smells that are not real smells and they have not to be refactored. These smells have been excluded because they represent domain or design dependent smells. We found many false positives with a total percentages of real God classes and Data classes of 23.38% and 13.67% respectively.

Starting from these observations on the low agreement on the results provided by different code smells detectors and on the high number of false positive smells detected by one tool (we have obviously to check if the same holds with other detectors), in the next Section we describe our proposal for code smells detection based on machine learning techniques.

IV. Towards a machine learning based 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 the data to be used and the algorithms to use in the experimentation. The following points summarize the principal steps of our approach, while the remainder of the section describes them.

- Collection of a large repository of heterogeneous software systems.
- Choice of a set of code smells and tools, or rules, for their detection.
- Application of the chosen tools/rules on the systems, recording the results for each class and method.

[1]http://checkstyle.sourceforge.net/

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

[3]http://loose.upt.ro/iplasma/index.htm

Table I. Advisors

Code smell	Reference Tool/Detection rules
God Class	iPlasma (God Class, Brain Class), PMD
Data Class	iPlasma, Fluid Tool [13], Anti-Pattern Scanner [12]
Long Method	iPlasma (Brain Method), PMD, Marinescu detection rule [14]
Feature Envy	iPlasma, Fluid Tool

- Labeling: following the output of the code smell detection tools, the reported code smell candidates are manually evaluated, and they are assigned different degrees of gravity.
- Experimentation: The manual labeling is used to train a supervised classifier, whose performance (precision, recall, etc…) will be compared with the other tools.

For our analysis, we considered the Qualitas Corpus of systems collected by Tempero et al. [3]. The corpus, and in particular the collection we use, 20120401r, is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. We selected 76 systems of different size and computed a large set of object-oriented metrics that are considered as independent variables in our machine learning approach. The selected metrics are at class, method, package and project level; the set of metrics is composed of metrics needed by the exploited detection rules, plus standard metrics covering different aspects of the code, i.e., complexity, cohesion, size, coupling. The metrics were then detected on each system and recorded. To be able to correctly compute the metric values, most of the 76 systems were completed (e.g., adding missing libraries) in order to make them compile.

The first step towards the application of machine learning has been to identify a set of code smells defined in the literature, by choosing the ones having the high frequency [11], that may have the greatest negative impact [12] on the software quality, and which can be recognized by some of the available detection tools [1]. In this study, code smells represent the dependent variable of a machine learning task. At class level, we decided to detect God/Large Class and Data Class, while at method level, we detect Long/God/Brain Method and Feature Envy. For each code smell we identified in the literature a set of available detection tools and rules, which are able to detect them. We selected as many heterogeneous detection rules as possible, favoring the detection rules implemented by tools, because they should be more reliable and have a larger user base. The detection rules selected for each code smell are reported in I. Other tools have not been considered because it is not possible to run them in batch without a manual configuration of the projects to analyze.

Our proposal is to detect code smells using the selected tools and rules on the chosen 76 systems. The detection output will be used as a hint to select a class (or method) to be manually evaluated, producing a label for each class (or method). The labelling process involves the inspection of the code of the selected class or method, supported also by graphical code representations, like dependencies, call, or hierarchy graphs. This information is used to support the decision of which label to assign.

The labelling is done with a severity classification of code smells. The idea of severity classification has been applied for software defects for years and its industrial adaption is widespread. Recently, several authors have worked in predicting defect severity with machine learning techniques [15], [16], [17]. Our initial idea is that code smell severity classification can have one of the four possible values:

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

The code smell severity classification can have two possible benefits. First, ranking of classes (or methods) by the severity of their code smells can aid software developers in prioritizing their work on the most severe code smells. Often developers have to work under time-constraints and they might not have time to fix all the code smells we can automatically detect. Second, the usage of the code smell severity classification for the labelling provides more information than a more traditional binary classification, as for machine learning purposes the code smell severity classification will be interpreted as an ordinal scale. This information can be exploited during the learning phase, or collapsed back to a binary classification by grouping together the values, e.g., $\{0\} \rightarrow INCORRECT$, $\{1, 2, 3\} \rightarrow CORRECT$. 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.

The manual evaluation will be performed also on a random sample of the remaining classes and methods, to avoid the introduction of a bias in the evaluation.

After this phase the training set will be analyzed using some machine learning methods we already selected. We chose the following classifiers, in the version available in the Weka tool [18]: Support Vector Machines (SMO, LibSVM), Decision Trees (J48), Random Forest, Naïve Bayes, JRip, applied directly and through boosting approaches. The choice of the algorithms was made by selecting different algorithms representing different approaches, and possibly already exploited in the literature for similar purposes; a preference was made for algorithms able to give a human-readable explanation of their learning.

Machine learning algorithms need a dataset input format. We decided to create two datasets, one for class-level smells, and another one for method-level smells. In each dataset, each row represents a class or method, and has one feature for each metric gathered on the subject. In addition, a boolean feature represents the label telling if the subject is a code smell instance or not.

Before the application of machine learning, it is necessary to process the values of the metrics to avoid possible distortions due to different scales of values; to this end, normalization and standardization will be applied.

Table II. PERFORMANCE RESULTS FOR DATA CLASS CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.976	0.976	0.981
Random Forest	0.978	0.979	0.998
Naïve Bayes	0.819	0.824	0.955
JRip	0.966	0.967	0.970
SMO	0.969	0.969	0.961
LibSVM	0.947	0.947	0.927

Table III. PERFORMANCE RESULTS FOR GOD CLASS CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.964	0.964	0.947
Random Forest	0.973	0.974	0.989
Naïve Bayes	0.954	0.955	0.981
JRip	0.973	0.974	0.973
SMO	0.964	0.964	0.954
LibSVM	0.766	0.726	0.655

We are experimenting each classifier through k-fold cross validation, in different configurations, to find the best one, in terms of the standard performance measures, e.g., precision and recall. For each classifier, we will produce a learning curve, to determine which algorithm learns more quickly, in addition to the absolute performances. Prior work has found that the criteria of code smells vary somewhat between people [19]. Thus, algorithms with a short learning curve are needed as they are quickly able to match the individual's personal criteria. Furthermore, the learning curve lets us to understand if adding new examples to the training set would be useful. The manual validation of code smells, in fact, is costly, and the effort should better be measured.

Some algorithms, like J48, Random Forest and JRip, will provide also human-readable rules; we are analyzing them to understand which (combination of) metrics have more influence on the detection.

The best classifier's results will be compared with the state of the art tools and rules, to measure the effectiveness of the classification approach. Finally, it might be that no single algorithm is superior in code smell detection. From website http://www.kaggle.com/, that hosts online data analysis competitions, we can find that winners often use several machine learning methods in combination [20]. Thus, we will also experiment with combining several machine learning algorithms if no clear winner arises.

A. Preliminary Results

In Tables II, III, IV, V we show the results of the machine learning algorithms used with their default parameters for each type of code smell. We use 10-fold cross-validation to assess the performance of predictive models. Each code smell data set contains 140 positive instances and 280 negative instances (420 instances), having a ratio of 33% of positive instances. The labeling was performed by three MsC students specifically trained for the task. The instances were taken from all the 76 analyzed systems.

Although the algorithms do not use optimized parameters, the results are very interesting. J48, Random Forest, JRip and

Table IV. PERFORMANCE RESULTS FOR FEATURE ENVY CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.940	0.941	0.943
Random Forest	0.952	0.952	0.991
Naïve Bayes	0.861	0.861	0.945
JRip	0.964	0.964	0.969
SMO	0.930	0.930	0.911
LibSVM	0.690	0.586	0.536

Table V. PERFORMANCE RESULTS FOR LONG METHOD CODE SMELL

Classifier	Accuracy	F-measure	ROC Area
J48	0.983	0.983	0.984
Random Forest	0.990	0.999	0.999
Naïve Bayes	0.930	0.932	0.968
JRip	0.992	0.993	0.993
SMO	0.966	0.967	0.964
LibSVM	0.692	0.590	0.539

SMO have accuracy values greater than 90% for all datasets, and on average they have the best performances. Naïve Bayes has slightly lower performances on Data Class and Feature Envy than on the other two smells. LibSVM performances are lower than the others (by far lower in three cases out of four). LibSVM belongs to the same family of predictive models of SMO, so the main reason of its lower performances is to be found in the default setting of the parameters. We expect better results with this class of models when we will experiment with different parameters.

V. CONCLUDING REMARKS

In this paper we outlined some common problems of code smell detectors and we described the approach we are following based on machine learning-techniques. We aim through our experimental analysis and the results we obtained to:

- provide a large dataset containing: 1) the source code, 2) gathered source code metrics, 3) results of the used detection tools and rules, and 4) manually validated information of the code smells using code smell severity classification
- the dataset can be used to: 1) make statistically significant comparisons between the tools and algorithms, 2) host online competitions on the dataset to find the best possible code smell predictors. 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).
- provide a new code smell detector, able to evolve with the availability of new data; the same detector can be easily used to detect other code smells, when training data will be available.

REFERENCES

[1] F. Arcelli Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, pp. 5:1–38, aug 2012. [Online]. Available: http://www.jot.fm/contents/issue_2012_08/article5.html

[2] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[3] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Proceedings of the 17th Asia Pacific Software Engineering Conference*. IEEE Computer Society, December 2010, pp. 336–345.

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

[5] 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, 2011, the Ninth International Conference on Quality Software.

[6] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *19th Working Conference on Reverse Engineering (WCRE 2012)*. Kingston, Ontario, Canada: IEE, October 2012, pp. 466–475.

[7] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Filtering clones for individual user based on machine learning analysis," in *Proceedings 6th International Workshop on Software Clones (IWSC 2012)*. Zurich, Switzerland: IEEE Computer Society, June 2012, pp. 76–77.

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

[9] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, 2012.

[10] V. Ferme, A. Marino, and F. Arcelli Fontana, "Is it a real smell to be removed or not," in *Presented at the RefTest 2013 Workshop, co-located event with XP 2013 Conference*, June 2013.

[11] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011.

[12] S. Olbrich, D. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance (ICSM 2010)*, 2010, p. 10.

[13] K. Nongpong, "Integrating "code smells" detection with refactoring tool support," Ph.D. dissertation, University of Wisconsin Milwaukee, August 2012. [Online]. Available: http://dc.uwm.edu/etd/13

[14] R. Marinescu, "Measurement and quality in objectoriented design," Ph.D. dissertation, Department of Computer Science, "Politehnica" University of Timisoara, 2002.

[15] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.

[16] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 1–10.

[17] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 215–224.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.

[19] M. Mantyla, "An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005, pp. 10 pp.–.

[20] P. Aldhous, "Specialist knowledge is useless and unhelpful," *New Scientist*, no. 2893, pp. 28–29, 2012.