ABSTRACT

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Nowadays, the complexity of software systems is growing fast and software companies are required to continuously update their source code. Those continuous changes frequently occur under time pressure and lead developers to set aside good programming practices and principles in order to deliver the most appropriate but still immature product in the shortest time possible. This process can often result in the introduction of so-called technical debt design problems likely to have negative consequences during the system maintenance and evolution.

To overcome these limitations, machine-learning (ML) techniques are being adopted to detect code smells. Usually a supervised method is exploited, i.e., a set of independent variables (a.k.a. predictors) are used to determine the value of a dependent variable (i.e., presence of a smell or degree of the smelliness of a code element) using a machine-learning classifier (e.g., Logistic Regression).

In order to empirically assess the actual capabilities of ML techniques for code smell detection, Arcelli Fontana et al.conducted a large-scale study where 32 different ML algorithms were applied to detect four code smell types, i.e., Data Class, Large Class, Feature Envy and Long Method. The authors reported that most of the classifiers exceeded 95% both in terms of accuracy and of F-Measure, with J48 and RANDOM FOREST obtaining the best performance. The authors see in these results an indication that “using machine learning algorithms for code smell detection is an appropriate approach” and that “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”.

In our research, we have observed important limitations of the work by Arcelli Fontana et al. that might affect the generalizability of their findings. Specifically, the high performance reported might be due to the way the dataset was constructed: for each type of code smell analyzed, the dataset contains only instances affected by this type of smell or non-smelly instances, with a non-realistic balance of smelly and non-smelly instances and a strongly different distribution of the metrics between the two groups of

instances, which is far from reality. During software maintenance and evolution, software systems need to be continuously changed by developers in order to (i) implement new requirements, (ii) enhance existing features, or (iii) fix important bugs. Due to time pressure or community-related factors, developers do not always have the time or the willingness to keep the complexity of the system under control and find good design solutions before applying their modifications. As a consequence, the development activities are often performed in an undisciplined manner, and have the effect to erode the original design of the system by introducing the so-called technical debt. Code smells, i.e., symptoms of the presence of poor design or implementation choices in the source code, represent one of the most serious forms of technical debt. Indeed, previous research found that not only they strongly reduce the ability of developers to comprehend the source 2code, but also make the affected classes more change- and fault-prone. Thus, they represent an important threat for maintainability effort and costs.

In past and recent years, the research community was highly active on the topic. On the one hand, many empirical studies have been conducted with the aim of understanding (i) when and why code smells are introduced, (ii) what is their evolution and longevity in software projects, and (iii) to what extent they are relevant for developers. On the other hand, several code smell detectors have been proposed as well. Most of them can be considered as heuristics-based: they apply a two-step process where a set of metrics are firstly computed, and then some thresholds are applied upon such metrics to discriminate between smelly and non-smelly classes.

They differ from each other for (i) the specific algorithms used to identify code smells (e.g., a combination of metrics or through the use of more advanced methodologies like Relational Topic Modeling) and (ii) the metrics exploited (e.g., based on code metrics or historical data). Although it has been showed that such detectors have reasonable performance in terms of accuracy of the recommendations, previous works highlighted a number of important limitations that might preclude the use of such detectors in practice.

In particular, code smells identified by existing detectors can be subjectively interpreted by developers. At the same time, the agreement between them is low. More importantly,

most of them require the specification of thresholds to distinguish smelly code components from non-smelly one: naturally, the selection of thresholds strongly influence their accuracy. For all these reasons, a recent trend is the adoption of Machine Learning (ML) techniques for approaching the problem.

In this scenario, a supervised method is exploited: a set of independent variables (a.k.a., predictors) are used to predict the value of a dependent variable (i.e., the smelliness of a class) using a machine learning classifier (e.g., Logistic Regression). The model can be trained using a sufficiently large amount of data available from the project under analysis, i.e., within-project strategy, or using data coming from other (similar) software projects, i.e., cross-project strategy. These approaches clearly differ from the heuristics-based ones, as they rely on classifiers to discriminate the smelliness of classes rather than on predefined thresholds upon computed metrics.

CHAPTER 2

LITERATURE SURVEY

2.1. Detecting Code Smells using Machine Learning Techniques

ABSTRACT:

Code smells are symptoms of poor design and implementation choices weighing heavily on the quality of produced source code. During the last decades several code smell detection tools have been proposed. However, the literature shows that the results of these tools can be subjective and are intrinsically tied to the nature and approach of the detection.

In a recent work the use of Machine-Learning (ML) techniques for code smell detection has been proposed, possibly solving the issue of tool subjectivity giving to a learner the ability to discern between smelly and non-smelly source code elements. While this work opened a new perspective for code smell detection, it only considered the case where instances affected by a single type smell are contained in each dataset used to train and test the machine learners. In this work we replicate the study with a different dataset configuration containing instances of more than one type of smell. The results reveal that with this configuration the machine learning techniques reveal critical limitations in the state of the art which deserve further research. Index Terms—Code Smells; Machine Learning; Empirical Studies; Replication Study .

THE REFERENCE WORK

In the authors analyze three main aspects related to the use of machine-learning algorithms for code smell detection: (i) performance of a set of classifiers over a sample of the total instances contained in the dataset, (ii) analysis of the minimum training set size needed to accurately detect code smells, and (iii) analysis of the number of code smells detected by different classifiers over the entire dataset.

In this paper, we focus on the first research question of the reference work. In the following subsections we detail the methodological process adopted in

A. Context Selection

The context of the study was composed of software systems and code smells. The authors have analyzed systems from the Qualitas Corpus, release 20120401r, one of the largest curated benchmark datasets to date, specially designed for empirical software engineering research. Among 111 Java systems of the corpus, 37 were discarded because they could not be compiled and therefore code smell detection could not be applied. Hence, the authors focused on the remaining 74 systems. For each system 61 source code metrics were computed at class level and 82—at method level. The former were used as independent variables for predicting class-level smells Data Class and God Class, the latter for predicting method-level smells Feature Envy and Long Method:

God Class.

It arises when a source code class implements more than one responsibility; it is usually characterized by a large number of attributes and methods, and has several dependencies with other classes of the system.

Data Class.

This smell refers to classes that store data without providing complex functionality.

Feature Envy.

This is a method-level code smell that appears when a method uses much more data than another class with respect to the one it is actually inLong Method. It represents a large method that implements more than one function. The choice of these smells is due to the fact that they capture different design issues, e.g., large classes or misplaced methods.

B. Machine Learning Techniques Experimented

In six basic ML techniques have been evaluated: J48, JRIP, RANDOM FOREST, NAIVE BAYES, SMO, and LIBSVM. As for J48, the three types of pruning techniques available in WEKA were used, SMO was based on two kernels (e.g., POLYNOMIAL and RBF), while for LIBSVM eight different configurations, using C-SVC and V-SVC, were used. Thus, in total 16 different ML techniques have been evaluated. Moreover, the ML techniques were also combined with the DABOOSTM1 boosting technique, i.e., a method that iteratively uses a set of models built in previous iterations to manipulate the training set and make it more suitable for the classification problem, leading to 32 different variants. An important step for an effective construction of machinelearning models consists of the identification of the best configuration of parameters: the authors

applied to each classifier the Grid-search algorithm, capable of exploring the parameter space to find an optimal configuration.

C. Dataset Building

To establish the dependent variable for code smell prediction models, the authors applied for each code smell the set of automatic detectors . However, code smell detectors cannot usually achieve 100% recall, meaning that an automatic detection process might not identify actual code smell instances (i.e., false negatives) even in the case that multiple detectors are combined. To cope with false positives and to increase their confidence in validity of the dependent variable, the authors applied a stratified random sampling of the classes/methods of the considered systems: this sampling produced 1,986 instances (826 smelly elements and 1,160 nonsmelly ones), which were manually validated by the authors in order to verify the results of the detectors. As a final step, the sampled dataset was normalized for size: the authors randomly removed smelly and non-smelly elements building four disjoint datasets, i.e., one for each code smell type, composed of 140 smelly instances and 280 non smelly ones (for a total of 420 elements). These four datasets represented the training set for the ML techniques above.

D. Validation Methodology

To test the performance of the different code smell prediction models built, the authors applied 10-fold cross validation: each of the four datasets was randomly partitioned in ten folds of equal size, such that each fold has the same proportion of smelly elements. A single fold was retained as test set, while the remaining ones were used to train the ML models. The process was then repeated ten times, using each time a different fold as the test set. Finally, the performance of the models was assessed using mean accuracy, F-Measure, and AUC-ROC over the ten runs.

E. Limitations and Replication Problem Statement

The results achieved in reported that most of the classifiers have accuracy and F-Measure higher than 95%, with J48 and RANDOM FOREST being the most powerful ML techniques. These results seem to suggest that the problem of code smell detection can be solved almost perfectly through ML approaches, while other unsupervised techniques (e.g., the ones based on detection rules) only provide suboptimal recommendations. However, we identified possible reasons for these good results:

selection, and (iii) validation methodology on the results of our study. At the same, we aim at addressing these issues, thus defining new prediction models for code smell detection.

CHAPTER 3

METHODOLOGY

3.1 EXISTING SYSTEM

The goal of the empirical study reported in this paper was to analyze the sensitivity of the results achieved by our reference work with respect to the metric distribution of smelly and non-smelly instances, with the purpose of understanding the real capabilities of existing prediction models in the detection of code smells.

The perspective is of both researchers and practitioners: the former are interested in understanding possible limitations of current approaches in order to devise better ones; the latter are interested in evaluating the actual applicability of code smell prediction in practice.

DISADVANTAGE:

- Subjective of developers which respect to code smell detection by such tools
- Scarce agreement between different detectors

Difficulties in finding good thresholds to be used for detection

3.2 PROPOSED MODEL

- Stage 1: Code smell classification
- Stage-2: Evaluation of machine learning model
- Navie bayes
- Random forest
- Decision tree classifier
- Stage-3: Model variance test

ADVANTAGE:

- performance of a set of classifiers over a sample of the total instances contained in the dataset,