Expert Systems with Applications 36 (2009) 10000–10003

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

An expert system for determining candidate software classes for refactoring

Yasemin Kosker *, Burak Turhan, Ayse Bener

Dept. of Computer Engineering, Bogazici University, 34342 Istanbul, Turkey

ARTICLE INFO

Keywords:
Refactoring
Software metrics
Naive Bayes
Refactor prediction

ABSTRACT

In the lifetime of a software product, development costs are only the tip of the iceberg. Nearly 90% of the cost is maintenance due to error correction, adaptation and mainly enhancements. As Lehman and Belady [Lehman, M. M., & Belady, L. A. (1985). *Program evolution: Processes of software change*. Academic Press Professional.] state that software will become increasingly unstructured as it is changed. One way to overcome this problem is refactoring. Refactoring is an approach which reduces the software complexity by incrementally improving internal software quality. Our motivation in this research is to detect the classes that need to be rafactored by analyzing the code complexity. We propose a machine learning based model to predict classes to be refactored. We use Weighted Naïve Bayes with InfoGain heuristic as the learner and we conducted experiments with metric data that we collected from the largest GSM operator in Turkey. Our results showed that we can predict 82% of the classes that need refactoring with 13% of manual inspection effort on the average.

1. Introduction

Refactoring is an approach to improve the design of a software without changing its external behaviour which means it always gives the same output with the same input after the change is applied (Fowler, Beck, Brant, Opdyke, & Roberts, 2001). As the project gets larger, the complexity of the classes increase and the maintenance becomes harder. Also, it is not easy or practical for developers to refactor a software project without considering the cost and deadline of the project. In general software refactoring compose of these phases (Zhao & Hayes, 2006):

- Identify the code segments which need refactoring.
- Analyze the cost/ benefit effect of each refactoring.
- Apply the refactorings.

Since developers carry out these processes, a proper tool support can decrease the cost and increase the quality of the software. There are some commercial tools that enables refactoring, however there is still a need for process automation (Simon & Lewerentz 2001). The objective of refactoring is to reduce the complexity of certain code segments such as methods or classes. Developers refactor a code segment in order to make it simpler or decrease its complexity such as extracting a method and then calling it. A code segment's complexity can increase due to its size or logic as well as its interactions with other code segments (Zhao & Hayes, 2006).

In this paper we focus on the automatic prediction of refactoring candidates for the same purposes mentioned above. We treat refactoring as a machine learning problem and try to predict the classes which are in need of refactoring in order to decrease the complexity, maintenance costs and bad smells in the project. We have inspired by the prediction results of Naïve Bayes and Weighted Naïve Bayes learners in defect prediction research (Turhan & Bener, 2007). In this research we use class level information and define the problem as two way classification: refactored and not-refactored classes. We then try to estimate the classes that need refactoring.

The rest of the paper is organized as follows. Section 2 presents related work. In Section 3 we explain the Weighted Naïve Bayes algorithm. In Section 4 we present our experimental setup to predict classes in need of refactoring. We discuss the evaluation criteria in Section 5. Results are presented and discussed in Section 6, and the conclusion and future work are presented in Section 7.

2. Related work

Welker and Oman (1995) suggested measuring software's maintainability using a Maintainability Index (MI) which is a combination of multiple metrics, including Halstead metrics, McCabe's cyclomatic complexity, lines of code, and number of comments.

Hayes and Zhao (2005) introduced and validated that the RDC ratio (the sum of requirement and design effort divided by code effort) is a good predictor for maintainability. Fowler et al. (2001) suggested using a set of bad smells such as long method to decide when and where to apply refactoring.

* Corresponding author. Tel.: +90 212 3597227; fax: +90 212 2872461.
E-mail address: yasemin.kosker@boun.edu.tr (Y. Kosker).

0957-4174/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2008.12.066

Mens, Tourwé, and Muñoz (2003) designed a tool to detect places that need refactoring and decide which refactoring should be applied. They did so by detecting the existence of "bad smells" using logic queries. Zhao & Hayes (2006) introduced a cost-benefit analysis to prioritize the identified classes with bad smells.

Our approach differs from the above approaches since we treat the prediction of candidate classes for refactoring as a data mining problem. We use Weighted Naïve Bayes (Turhan & Bener, 2007), which is an extension to the well-known Naïve Bayes algorithm in order to predict the classes which are in need of refactoring (Kosker, Bener, & Turhan, 2008).

3. Weighted Naïve Bayes

The Naïve Bayes classifier, currently experiencing a renaissance in machine learning, has long been a core technique in information retrieval (Lewis, 1998). Naïve Bayes models have been used for text retrieval and classification, focusing on the distributional assumptions made about word occurrences in documents. In defect prediction it has so far given the best results in terms of probability of detection and probability of false alarm (which will be defined in Section 5) (Menzies, Greenwald, & Frank, 2007). However, Naïve Bayes makes certain assumptions that may not be suitable for software engineering data (Turhan & Bener, 2007). Naïve Bayes treats attributes as independent and with equal importance. Turhan and Bener (2007) argued that some software attributes are more important than the others. Therefore each metric must be assigned a weight as per its importance. "Weighted Naïve Bayes" approach showed promising outcomes that can generate better results in defect prediction problems with the InfoGain and GainRatio weight assignment heuristics. In this paper, our aim is to implement and evaluate Weighted Naïve Bayes with InfoGain and show that it can be used for predicting the refactoring candidates.

Naïve Bayes classifier is a simple yet powerful classification method based on the famous Bayes' Rule. Bayes' Rule uses prior probability and likelihood information of a sample for estimating posterior probability (Alpaydin, 2004)

$$P(C_i|x) = \frac{P(x|C_i)P(C_i)}{P(x)} \tag{1}$$

To use it as a classifier, one should compute posterior probabilities $P(C_i|x)$ for each class and choose the one with the maximum posterior as the classification result. Class posteriors in Naïve Bayes classification are calculated as follows:

$$P(C_i|x) = -\frac{1}{2}\sum_{j=1}^{d}\left(\frac{x_j^2 - m_{ij}}{S_j}\right)^2 + \log(\hat{P}(C_i)) \tag{2}$$

This simple implementation assumes that each dimension of the data has equal importance on the classification. However, this might not be the case in real life. For example, the cyclomatic complexity of a class should be more important than the count of commented lines in a class. To cope with that problem, Weighted Naïve Bayes classifier is proposed and tested against Naïve Bayes (Ferreira, Denison, & Hand, 2001; Turhan & Bener, 2007). Class posterior computation is quite similar to Naïve Bayes only with the introduction of weights for each dimension. Formula for computing class posteriors in Weighted Naïve Bayes is as follows:

$$P(C_i|x) = -\frac{1}{2}\sum_{j=1}^{d} w_j\left(\frac{x_j^2 - m_{ij}}{s_j}\right)^2 + \log(\hat{P}(C_i)) \tag{3}$$

Introduction of weights brings a flexibility that allows us to favor some dimensions over others but it also raises a new problem: determining the weights. In our case, dimensions consist of different attributes calculated from the source code (see Table 1) and we need some heuristics for determining the weights (or the importance's) of the attributes.

Table 1
Metrics collected from the project.

CyclomaticDensity	HalsteadProgramDifficulty
DecisionDensity	HalsteadProgramLength
EssentialDensity	HalsteadProgramLevel
BranchCount	HalsteadProgramminqEffort
ConditionCount	HalsteadProgrammingTirne
CyclomaticComplexity	HalsteadProgramVolume
DecisionCount	MaintenanceSeverity
EssentialComplexity	CouplinqBetweenObjects
Loc	Fanln
TotalOperands	NutnberOfchildren
TotalOperators	PercentageOfPubData
UniqueOperandsNumber	ResponseForClass
UniqueOperatorsNumber	WeightedMethods

In this study we use InfoGain as the heuristic for weight assignment. InfoGain measures the minimum number of bits to encode the information obtained for prediction of a class (C) by knowing the presence or absence of a feature in data. Concisely, the information gain is a measure of the reduction in entropy of the class variable after the value for the feature is observed

$$InfoGain(x,A) = Entropy(x) - \sum_{a\in A}\frac{|x=A|}{|x|}Entropy(x=a) \tag{4}$$

In the equations "w" denotes the weight of attribute in data set which is calculated with

$$W_d = \frac{Infogain(d)\times n}{\sum Infogain(i)} \tag{5}$$

4. Experimental setup

We collect data from a local GSM operator company. The data contains one project and its three versions. The project is implemented in Java programming language and corresponds to a middleware application. We collected 26 static code attributes including Halstead metrics, McCabe's cyclomatic complexity and lines of code from the project and its versions. The full metric list is given in Table 1 and the class information of all project versions is listed in Table 2.

We can collect the method, class and package metrics with our Metric Parser, Prest (Turhan, Oral, & Bener, 2007), which is developed in Java. We also collect the call graph data which gives us the information of caller and callee methods with Prest. All the relationship between methods, classes and packages are based on the ID's of these objects, which is generated and assigned automatically during the parsing process. Since the ID's are generated automatically, the ID's of the objects differs from each other among each version. So, in order to avoid this problem we use the names of each object while comparing it with other versions. Thus, if a Rename type refactoring is applied to an object during a version upgrade, then our approach could not handle this case. We leave this case out as our future work.

Table 2
Attribute and class information of the project.

Name	# Attributes	# Classes
Trcll1 2.19	26	524
Trcll1 2.20	26	528
Trcll1 2.21	26	528
Trcll1 2.22	26	534