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).

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

We focus on complexity metrics in this paper since the main idea behind class refactoring is to reduce complexity. We have used commonly used refactoring types such as Extract Interface, Push Members Down and Extract Superclass to decrease the overall complexity of the classes while preserving their external behaviour and increasing the readability and maintainability of the code and design. Halstead metrics are computed directly from operators and operands in the code. Halstead length (total number of operators and operands), vocabulary (total number of unique operators and operands), and effort are used to represent the length, understandability, and complexity of each class as well cyclomatic complexity and Loc (Hayes and Zhao, 2005 ; Halstead, 1977).

It is very hard to understand or modify one part of a class before reading, modifying and testing it (Zhao & Hayes, 2006). Also, new functionalities are added to the project by using a method in a class, generally with the usage of some other methods in that class. Therefore the overall complexity increases together with the maintenance costs of the system. In order to trace changes in the class structure, we propose a class based approach to find out which classes are more likely to be refactored first.

We developed our code in Matlab in order to estimate the classes that should be refactored during each version upgrade since there is no data for refactored classes. So, we made an assumption that if the cyclomatic complexity, essential complexity or total number of operands decreases from the beginning of the project, then these classes are assumed to be refactored. We did not know which metrics directly affected the refactoring decision. However, we assumed that complexity metrics affected the refactoring decision during the version upgrades. Table 3 shows the ID's of the classes, which we assumed to be refactored according to their cyclomatic complexity value changes. We show cyclomatic complexity values of a class for each version of the project. Not Applicable (N/A) means that no complexity comparison is made for that version. There are totally seven classes (see Table 3) which are refactored during the development of the project according to our assumption described above. We normalized the data in Trcll1 datasets since it is a complex project at the application layer. It is refactored and changed frequently during the development of the project. In Fig. 1 we provide the algorithm which is used during the construction of the data sets.

After collecting refactored class data, we apply Weighted Naïve Bayes for automatic prediction of candidate classes. Thus, we can recommend the developers to refactor these classes first.

We have designed the experiments to achieve high degree of internal validity by carefully studying the effect of independent variables on dependent variable in a controlled manner (Mitchell & Jolley, 2001). In order to carry on statistically valid experiments, datasets should be prepared carefully. A common technique is working with two data sets which are namely test and train instead of entire data. Generally, these sets are constructed randomly by dividing whole data into two sets. Here, a problem arises due to the nature of random selection, which is that it is not guaranteed to have a good representation of the real data by doing a single sampling. To cope with that problem, *k*-fold cross validation is used. In k-fold cross validation, data is divided into k equal portions and training process is repeated for k times with $k-1$ folds used as train data and one fold is used as test data (Turhan & Bener, 2007). We chose *k* as 10 so at each run, we use 9 folds as train data and one fold as test data. This whole process is repeated 10 times with the shuffled data. Moreover, since both the train and test data should have a good representation of the real data, the ratio among the refactored and not-refactored samples should be preserved. We have used stratified sampling so when dividing the data into 10 folds, we made sure that each fold preserves the refactored/not-refactored samples ratio.

Table 3
Refactored classes.

Class ID	Trcll1 2.19	Trcll1 2.20	Trcll1 2.21	Trcll1 2.22
2151	3	1	N/A	N/A
	3	N/A	1	N/A
	3	N/A	N/A	1
2157	5	3	N/A	N/A
	5	N/A	3	N/A
	5	N/A	N/A	3
2359	16	6	N/A	N/A
	16	N/A	3	N/A
	16	N/A	N/A	3
2839	6	3	N/A	N/A
	6	N/A	3	N/A
	6	N/A	N/A	1
2848	3	1	N/A	N/A
	3	N/A	1	N/A
	3	N/A	N/A	1
4665	14	10	N/A	N/A
5150	22	17	N/A	N/A

5. Evaluation criteria

We evaluated the accuracy of our predictor with probability of detection (pd) and probability of false alarm (pf) measures (Men-

```
   Change all zeros by 1 × $10^{-6}$ in the data set
   Take the logarithm of all values in the data set. .
   if   Cyclomatic  Complexity  of  the  class  in  the  current  version   >
Cyclomatic Complexity of the class in the next version
   then
       the class is marked as refactored
       attribute value in the Trcll1 datasets is set to 2
   else
       the class is marked as not-refactored
       attribute value in the Trcll1 datasets is set to 1
```

Fig. 1. Algorithm to construct the datasets.

Table 4
Confusion matrix.

	Estimated	
Real	Defective	Non-defective
Defective	A	C
Non-defective	B	D

Table 5
Results for Trcll1 project.

Project	InfoGain + WNB (%)		NB(%)	
	pd	Pf	pel	Pf
Trcll1 2.19	63	16	49	17
Trcll1 2.20	90	12	38	13
Trcll1 2.21	93	11	92	12
Avg	32	13	76	14

zies et al., 2007). Pd is the measure of detecting real refactored classes over all real refactored ones and pf is the measure of detecting classes as refactored that are not actually refactored over all not-refactored classes. Higher pd values and lower pf values reflects the accuracy of the predictor. The confusion matrix used for calculating pd and pf is shown in Table 4

$$pd = \frac{A}{A+C} \tag{6}$$

$$pf = \frac{B}{B+D} \tag{7}$$

6. Results

The results of our experiments show that in the first unstable version of a software our predictor detects the classes that need to be refactored with 63% accuracy (Table 5). In the second version which we can call that the first stable version the predictor's performance increases to 90%. We can conclude that the learning performance improves as we move to more stable versions and learn more about the complexity of the code. Our results also show that learning complexity related information on the code, i.e. weight assignment, considerably improves the learning performance of the predictor as evidenced by the IG+WNB pd of 82 (avg) versus NB pd of 76 (avg). We also observe that as we move to later versions false alarm rates decrease (from pf:16 to pf:11) with our proposed learner. Low pf rates prevent software architects from manual analysis of classes which are not needed to be refactored. In the three versions of a complex code such as Trcll1 project we can predict 82% of the refactored classes with 13% of manual inspection effort on the average.

Our concern for external validity is the use of limited number of datasets. We used one complex project and its three versions. To overcome ceiling effects we used 10-fold cross validation.

7. Conclusion and future work

The developers and the architects make refactoring decisions based on their years of experience. Therefore refactoring process becomes highly human dependent, subjective and costly. Process automation and tool support can help reduce this overhead cost as well as increase consistency, efficiency, and effectiveness of the code reviewing and refactoring decision process.

In this paper presented an empirical study of refactoring prediction. We addressed the problem as a machine learning problem and we used Weighted Naïve Bayes with InfoGain weighting heuristic for predicting the candidate refactorings of classes. We used three data sets from Trcll1 project and run our model on these data sets. We have seen that our algorithm works better as it learns in terms of higher pd rates and lower pf rates. We have seen that using oracles to predict which classes to refactor considerably decreases the manual effort for code inspection (note that avg pf is 13), identifies the complex and problematic pieces of the code and hence makes the maintenance less costly and trouble free process.

Our future direction would be to collect more refactor data and repeat our experiments. We also plan to try other heuristics to further lower the pf rate. We will also include different types of refactoring such as Rename to be automatically handled by our model.

References

Alpaydin, E. (2004). *Introduction to machine learning*. MIT Press.

Ferreira, J. T. A. S., Denison, D. G. T., & Hand, D. J. (2001). *Weighted Naive Bayes Modelling for Data Mining.*

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2001). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Halstead, M.H. (1977). *Elements of software science, operating, and programming systems series.*

Hayes, J., & Zhao, L. (2005). Maintainability prediction: A regression analysis of measures of evolving systems. In *Proceedings of the IEEE 21th international conference on software maintenance*.

Kosker, Y., Bener, A., & Turhan, B. (2008). Refactoring prediction using class complexity metrics. In *ICSOFT 2008, Porto, Portugal, July 2008.*

Lehman, M. M., & Belady, L. A. (1985). *Program evolution: Processes of software change*. Academic Press Professional.

Lewis, D. (1998). Naive (Bayes) at forty: The independence assumption in information retrieval. In *Proceedings of ECML-98, 10th european conference on machine learning.*

Mens, T., Tourwé, T., & Muñoz, F. (2003). Beyond the refactoring browser: Advanced tool support for software refactoring. In *Proceedings of the international workshop on principles of software evolution.*

Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering.*

Mitchell, M., & Jolley, J. (2001). *Research design explained*. New York: Harcourt.

Simon, F. S. F., & Lewerentz, C. (2001). Metrics based refactoring. In *Proceedings of the european conference software maintenance and reengineering.*

Turhan, B., & Bener, A.B. (2007). Software defect prediction: Heuristics for Weighted Naive Bayes. In *ICSOFT 2007.*

Turhan, B., Oral, A.D., & Bener, A.B. (2007). Prest – A tool for pre-test defect prediction. Boğaziçi University Technical Report.

Welker, K., Oman, P.W. (1995). Software maintainability metrics models in practice. *Journal of Defense Software Engineering.*

Zhao, L., & Hayes, J.H. (2006). Predicting classes in need of refactoring: An application of static metrics. In *Proceedings of the workshop on predictive models of software engineering (PROMISE), associated with ICSM 2006.*