REFACTORING PREDICTION USING CLASS COMPLEXITY METRICS

Yasemin Köşker, Burak Turhan and Ayşe Bener
Department of Computer Engineering, Boğaziçi University, Bebek, İstanbul, Turkey

Keywords: Weighted Naïve Bayes, Refactoring, Software Metrics, Naïve Bayes, Defect Prediction, 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, adoptation and mainly enhancements. As Belady and Lehman (Lehman and Belady, 1985) 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 and 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 and Hayes, 2006):

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

Since this processes can be done by developers, a proper tool support can decrease the cost and increase the quality of the software. There are some commercial tools which enables refactoring, however there is still a need for process automation (Simon and Lewerentz, 2001). Developers refactor a code segment 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 and 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 and 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. Results are presented and discussed is section 5, and the conclusions are given in section 6.

Köşker Y., Turhan B. and Bener A. (2008).
REFACTORING PREDICTION USING CLASS COMPLEXITY METRICS.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 289-292
DOI: 10.5220/0001893802890292

2 RELATED WORK

Welker (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 et al. (Hayes and Zhao, 2006) 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 (Fowler, Beck, Brant, Opdyke and Roberts, 2001) suggested using a set of bad smells such as long method to decide when and where to apply refactoring. Mens, Tourwé and Muñoz (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. 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 and 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.

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). In defect prediction it has so far given the best results in terms of probability of detection and false alarm (which will be defined in Section 5) (Menzies, Greenwald and Frank, 2007). However, Naïve Bayes makes certain assumptions that may not be suitable for software engineering data (Turhan and Bener, 2007). Naïve Bayes treats attributes as independent and with equal importance. Turhan and Bener argued that some software attributes are more important than the others (Turhan and Bener, 2007). 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 (Alpaydın, 2004).

$$P(C_i \mid x) = \frac{P(x \mid C_i)P(C_i)}{P(x)} \quad (1)$$

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.

$$P(C_i \mid x) = -\frac{1}{2}\sum_{j=1}^{d}\left(\frac{x_j^t - m_{ij}}{s_j}\right)^2 + \log\left(\hat{P}(C_i)\right) \quad (2)$$

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 (Turhan and Bener, 2007), (Ferreira, Denison and Hand 2001). Class posterior computation is quite similar to Naïve Bayes only with the introduction of weights for each dimension and the formula in Weighted Naïve Bayes is as follows:

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

Introduction of weights brings a flexibility that allows us to favour 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 and we need some heuristics for determining the weights (or the importance's) of the attributes. 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.

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

In the equations "w" denotes the weight of attribute in data set which is calculated with Equation 5.

$$w_d = \frac{InfoGain(d) \times n}{\sum InfoGain(i)} \quad (5)$$

4 EXPERIMENTAL SETUP

We collect data from a local GSM operator company. The data contains one project and its 3 versions. The project is implemented in Java 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 with our Metric Parser, Prest (Turhan, Oral and Bener, 2007) which is written in Java. The class information of all project versions is listed in Table 1.

Table 1: 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

In order to estimate the classes that should be refactored during each version upgrade, 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. Since we did not know the affect of metrics on refactoring decision we assumed that complexity metrics affected the refactoring decision during the version upgrades. 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. After collecting refactor data, we apply Weighted Naïve Bayes for automatic prediction of candidate classes.

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 and 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 and Bener, 2007). We chose k as 10 in our experiments. 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.

5 RESULTS

Table 2: Confusion Matrix.

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

We evaluated the accuracy of our predictor with probability of detection (pd = A/(A+C)) and probability of false alarm (pf = B/(B+D)) measures (Menzies, Greenwald and Frank, 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 2.

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 3). 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 tne 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.

Table 3: Results for Trcll1 project.

Project	InfoGain + WNB (%)		NB (%)	
	pd	pf	pd	pf
Trcll1 2.19	63	16	49	17
Trcll1 2.20	90	12	88	13
Trcll1 2.21	93	11	92	12
Avg	82	13	76	14

6 CONCLUSIONS 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 we 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 3 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 may also try other heuristics to further lower the pf rate.

ACKNOWLEDGEMENTS

This research is supported in part by Bogazici University research fund under grant number BAP-06HA104 and by Turkcell A.Ş

REFERENCES

Simon, F. S., F., Lewerentz, C., 2001. Metrics based refactoring. *Proc. European Conf. Software Maintenance and Reengineering.*

Fowler, M., Beck, K., Brant,J., Opdyke,W., Roberts,D., 2001. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

Lewis, D., 1998. Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval. Proceedings of ECML-98, 10th European Conference on Machine Learning.

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.*

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

Mens, T., Tourwé, T., Muñoz, F., 2003. Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring. *Proceedings of the International Workshop on Principles of Software Evolution.*

Fowler, M., Beck, K., Brant,J., Opdyke,W., Roberts,D., 2001. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

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

Turhan, B., Bener, A.B., 2007. Software Defect Prediction: Heuristics for Weighted Naive Bayes. *ICSOFT 2007*

Alpaydin, E., 2004. *Introduction to Machine Learning,* MIT Press.

Lehman, M.M., Belady, L.A., 1985. *Program evolution: processes of software change*, Academic Press Professional.

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

Ferreira, J.T.A.S., Denison, D.G.T., Hand, D.J., 2001. Weighted naive Bayes modelling for data mining.

Mitchell, M., Jolley, J., 2001. *Research Design Explained,* New York:Harcourt.