Predicting Source Code Quality with Static Analysis and Machine Learning

Vera Barstad, Morten Goodwin, Terje Gjøsæter

Faculty of Engineering and Science, University of Agder
Serviceboks 509, NO-4898 Grimstad, Norway
vera.barstad@gmail.com
{morten.goodwin, terje.gjosater}@uia.no

Abstract

This paper is investigating if it is possible to predict source code quality based on static analysis and machine learning. The proposed approach includes a plugin in Eclipse, uses a combination of peer review/human rating, static code analysis, and classification methods. As training data, public data and student hand-ins in programming are used. Based on this training data, new and uninspected source code can be accurately classified as "well written" or "badly written". This is a step towards feedback in an interactive environment without peer assessment.

1 Introduction

Analysis of source code has been a topic of interest for many years. Both static and dynamic solutions are developed [1]. Previous relevant research mainly focuses on fault-prediction in the code, but there is a lack of research on identifying both well written and badly written code. It is not a trivial task to detect the quality of the code, since it depends on both functional requirements, structural requirements, and complexity.

This paper introduces an approach that use static code analysis and classification methods. For static code analysis, 15-20 rules are used, and the classifier is supposed to classify well written or badly written source code based on training data provided by peer reviews, public datasets and student hand-ins in programming. The proposed approach is based on and extends the implemented proof of concept plugin by Barstad et.al [2] which has the possibility for several users to add files from different projects for review, and perform peer review on selected files.

Motivation

A comprehensive study of 21 papers in the field of automated code analysis indicates that this research direction is very promising but far from trivial [1], and that there exist several plugins aimed on providing a way of rating source code [3, 4]. For

This paper was presented at the NIK-2014 conference; see http://www.nik.no/.

example, there exists an Eclipse-based software fault prediction tool using the naïve Bayes algorithm [5]. While previous research mainly focuses on fault-prediction, this paper propose an approach with static code analysis as well as machine learning based classification used to automatically identify both well written and badly written code.

Problem Definition

We are investigating if it is possible to identify well written and badly written source code by using peer review/human rating, static code analysis and review of the source code by classification. We focus on finding badly written code because detecting bad student assignments is more important.

The rest of the paper is organized as follows : Section 2 describes related research in the area, section 3 present the training data used for the classifications. Section 4 describes the approach, while the results are shown in section 5. Our conclusion are given in section 6, while future work and acknowledgement are given in section 7 and 8.

2 State of the Art

Code quality is essential for good software development, and the quality of the source code depends on several aspects. One of them is the functional quality which reflects how well it fulfils its functional requirement and specifications. Another aspect is the structural quality which refers to non-functional requirements like robustness and maintainability, this can be evaluated through analysis of the inner structure of the code. Code review, static code analysis, metrics and environment are all relevant to this paper.

One of the important issues about code quality is how easy it is for humans to read and understand the code, and an important measurement for this is the complexity measurement. A measurement to calculate the cyclomatic complexity of a program was developed by McCabe [6] in 1976. This measurement uses the control flow graph of the program to compute the number of linearly independent paths through the source code. Another complexity measurement is defined by Healsted [7]. This work was based on studying the complexities of both programming languages and written languages. A code smell is an indicator that a program possibly contains a problem. Code smells are usually not bugs since they are technically correct and do not prevent the program from functioning. Instead they indicate weakness in the design or an increasing risk of failures in the future. In the research performed by Mäntylä et al. [8], a taxonomy of bad smells in code was created. They categorized similar bad smells and performed an empirical study which provided an initial correlation between the smells. Another study in this field was performed by Yamashita and Counsell [9], this research investigated the potential of code smells to reflect system-level indicators of maintainability evaluations.

Inspection of source code can be performed with either manual or automated approaches. An introduction to Code Review and Static code analysis is presented by Barstad et al. [2]. Feature location is the process of identifying locations in the source code that implements a functional requirement of a software system. B. Dit et al. [10] presents a survey of feature location techniques. In this paper 89 articles

were reviewed and classified.

Code Review and Static Code Analysis

Static code analysis gives an opportunity to check the quality of source code without executing the program. There exists several ways of analyzing code without, or nearly without, human participation. Among them is information retrieval based tools (IR) that complement traditional static and dynamic analysis by exploiting the natural language found within a program's text. D. Binkley [11] surveys current (2007) work on source code analysis. In addition, Wong and Gokhale[12] proposed a static and dynamic distance metric to determine the distance between features of a software system. In their work Fehnker et al. [13] looked at common software bugs and potential false alarms.

Software Metrics and Public Datasets

Research has been performed based on the metrics defined by available public data sets[1]. Research performed by Menzies et al. [14] uses these data sets and has shown that it is possible to use static code attributes to automatically detect predictors for software quality. An Eclipse plugin for detection of design patterns through static and dynamic analysis was developed by De Lucia et al. [15].

Classification Methods

In machine learning, classification is the task of identifying which class a new and unlabelled item belongs to, based on previous knowledge. Methods used for text classification and static analysis including Bayesian classification [5, 16] , n-gram [17] , Support Vector Machine (SVM) [18, 19] and Decision Tree [20].

Development Environment

Eclipse is the main development environment (IDE) for many Java developers. It is a multi-language software IDE that has an opportunity for customizing the environment by using plugins [21]. A plugin is a component that allows the developer to customize and extend the IDE. Extensions are supposed to simplify the development process and give additional possibilities for the developer. By extracting information about design patterns from software systems, insight on the structure of the software and its internal characteristics can be provided. A short paper about the Automatic and Collaborative Code Review Plugin was published [2] [2] that shows the possibility to implement a plugin which facilitates identification of well written and badly written code. The paper proposed methods and algorithms on how to use static analysis and classification to automatically identify well written and badly written code.

Since software development is a collaborative process, there is a need for computer-based collaborative tools. Integrating such tools into the IDE, and enabling them with awareness of development processes and artifacts, may help reduce context switches between tools inside and outside the IDE, and make the connection between development and collaboration more seamless. Depending of

[1] http://promise.site.uottawa.ca/SERepository/datasets-page.html

[2] by the authors of this paper

different purposes, collaborative tools may have somehow different implementations and capabilities [4, 22, 23].

3 Data

To classify source code with an unknown class, there is a need of training data labelled with the appropriate labels: "well written" code and "badly written" code. To achive this, data from the following sources are collected as training data.

* PROMISE1: Available public dataset from the PROMISE workshop website named jm1 with continuous values [3]
* PROMISE2: Public dataset jm1 dataset with discrete values
* STUDENT: Student hand-ins in basic programming course at University of Agder

Public Datasets

Both PROMISE1 and PROMISE2 datasets are labeled true/false indicating if the source code has a probability of being faulty or not, and it has 21 featur values that can be calculated by a textual analysis of the source code. The cyclomatic complexity and the number of operators and operands in the code are examples of features. From the 21 features, 15 features were selected for further study. The reason for selecting these features was that they can be calculated using JHawk [4] or easily be implemented in a real live application. A list of the selected features are shown in Table 1. A limitation of using the jm1 dataset from promise as training data, is that in this dataset each method are labelled true/false according to if they are fault-prone or not. It might not be completely correct to assume that a fault-prone method is badly written, or that a not fault-prone method is well written, but in this stage of the project we have chosen to use the jm1 promise dataset as training data for the classification because there is a lack of human resources to perform peer reviews.

The PROMISE1 dataset were imported into the MySql database, and the continuous feature values were converted from continuous values into a discrete range by analysing a plotted graph and calculating the cumulative distribution for each feature. The reason for doing this was to investigate if the process of discretizing the values gave better results when classifying the dataset. An example of the conversion for the total operators and operands feature(C2) is shown in Table 2, the output from this process is the PROMISE2 dataset.

Several combinations of the features were selected for classification. A quantile-quantile(QQ) plot is a graphical technique used to determine if two datasets has a common distribution. By looking at the QQ plots of the features, the ones that look most discriminating was selected. Examples of the graphs are shown in Figure 1.

From the figure we can see that feature C4 are more discriminating than feature C9, and that feature C4 and C5 have a common distribution, while feature C4 and C6 does not have a common distribution. After inspecting all features, a feature set including C1,C4,C6,C10,C11,C12,C13,C14 and C14 was selected as one of the combinations.

[3] http://promise.site.uottawa.ca/SERepository/datasets-page.html

[4] http://www.virtualmachinery.com/jhawkprod.html

Feature	Description
C1	McCabe Cyclomatic complexity
C2	Halstead total operators + operands
C3	Halstead Volume
C4	Halstead program length
C5	Halstead difficulty
C6	Halstead intelligence
C7	Halstead effort
C8	Halstead bug
C9	Halstead time estimator
C10	Halstead line cont
C11	Halstead lines of comments
C12	Halstead lines of blank lines
C13	Unique operators and operands
C14	Total operators
C15	Total operands

Table 1: Selected features

From Value	To Value	Discretized Value	Cumulative Percentage
0	6.99	A	5.04 %
7	9.99	B	10.45 %
10	13.99	C	14.67 %
14	18.99	D	20.32 %
19	24.99	E	25.65 %
25	30.99	F	30.45 %
31	36.99	G	35.04 %
37	43.99	H	40.15 %
44	59.99	I	50.27 %
60	80.99	J	60.04 %
81	113.99	K	70.19 %
114	165.99	L	80.11 %
166	289.99	M	90.06 %
290	∞	N	100.00 %

Table 2: Converting data feature C2

Student Hand-ins in Programming

In addition to the public data set, student hand-ins in basic programming were collected for the year 2013-2014 together with the teachers solution to each hand-in(STUDENT dataset). An example of the statical analysis generated for some of the hand-ins are shown in Figure 2. As shown in the figure, feature C12, "number of blank lines", are not available in JHawk, so this feature has been implemented manually. A metric is a value generated by the static analysis for each feature, for example the count of lines in a given method. An example of the metrics and the classification of the methods are shown in Table 3

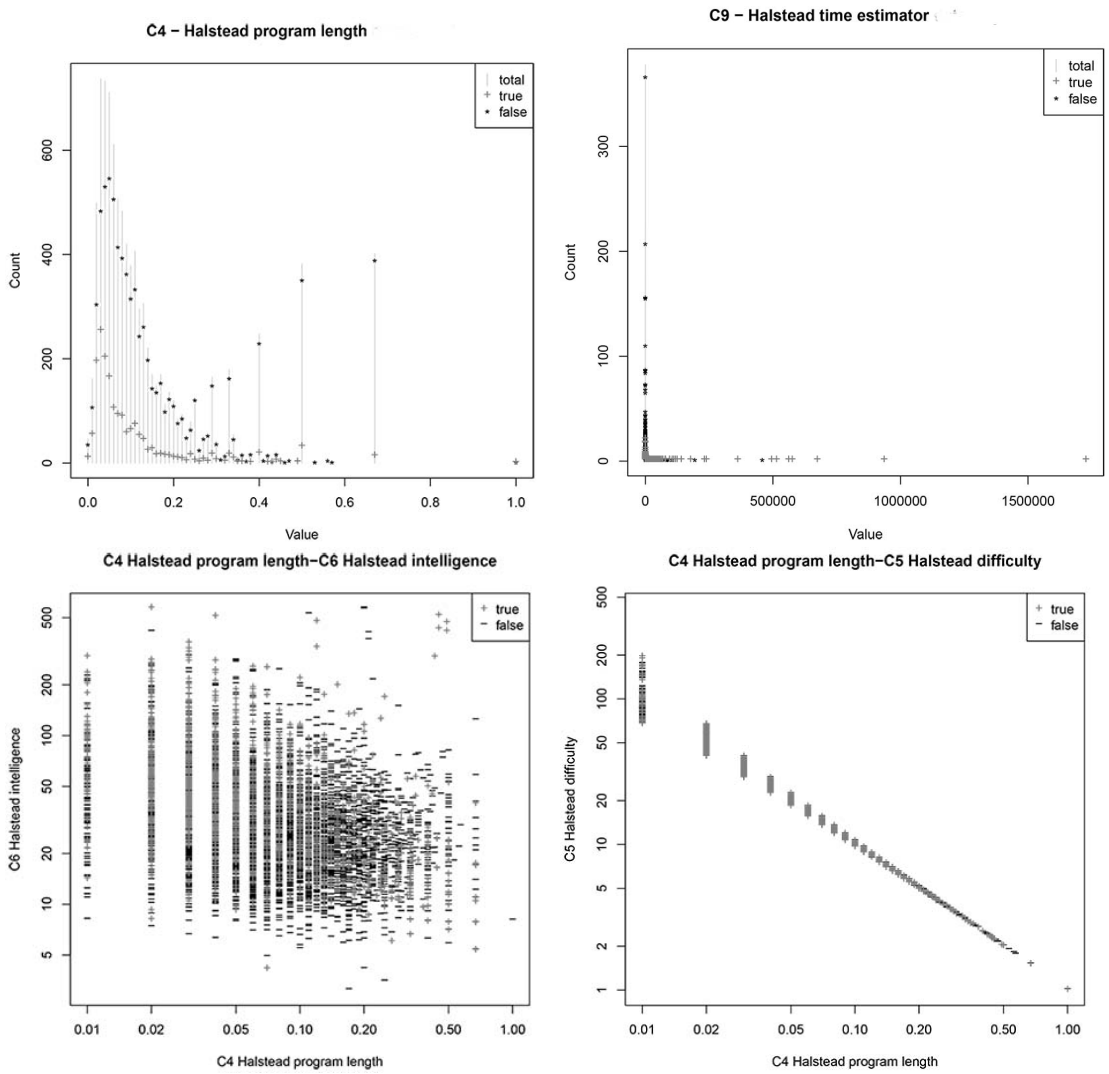
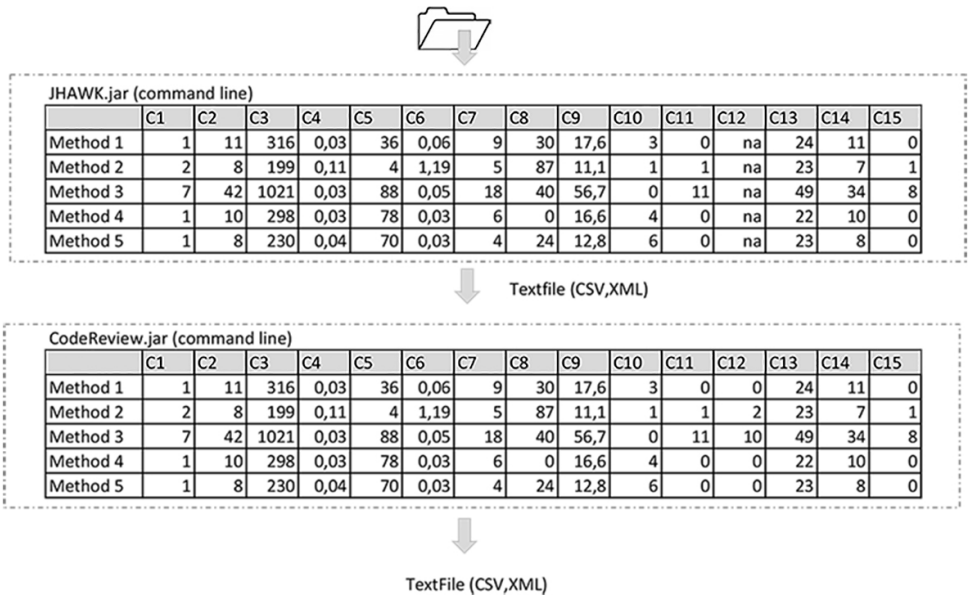


Figure 1: Example of feature graphs and QQ plots



	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
Method 1	1	11	316	0,03	36	0,06	9	30	17,6	3	0	na	24	11	0
Method 2	2	8	199	0,11	4	1,19	5	87	11,1	1	1	na	23	7	1
Method 3	7	42	1021	0,03	88	0,05	18	40	56,7	0	11	na	49	34	8
Method 4	1	10	298	0,03	78	0,03	6	0	16,6	4	0	na	22	10	0
Method 5	1	8	230	0,04	70	0,03	4	24	12,8	6	0	na	23	8	0

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
Method 1	1	11	316	0,03	36	0,06	9	30	17,6	3	0	0	24	11	0
Method 2	2	8	199	0,11	4	1,19	5	87	11,1	1	1	2	23	7	1
Method 3	7	42	1021	0,03	88	0,05	18	40	56,7	0	11	10	49	34	8
Method 4	1	10	298	0,03	78	0,03	6	0	16,6	4	0	0	22	10	0
Method 5	1	8	230	0,04	70	0,03	4	24	12,8	6	0	0	23	8	0

Figure 2: Static Analysis

4 Approach

To further investigate the possibility of code analysis, we base ourselves on the proof of concept implementation from previous work [2], together with an implementation of the selected classification methods in Python. JHawk is used to generate the static analysis for the source code, and some metrics are generated by the plugin. The plugin offers peer code review, and the implementation in Python the automated classification of source code quality. Figure 3 shows an overview of the architecture of the approach.

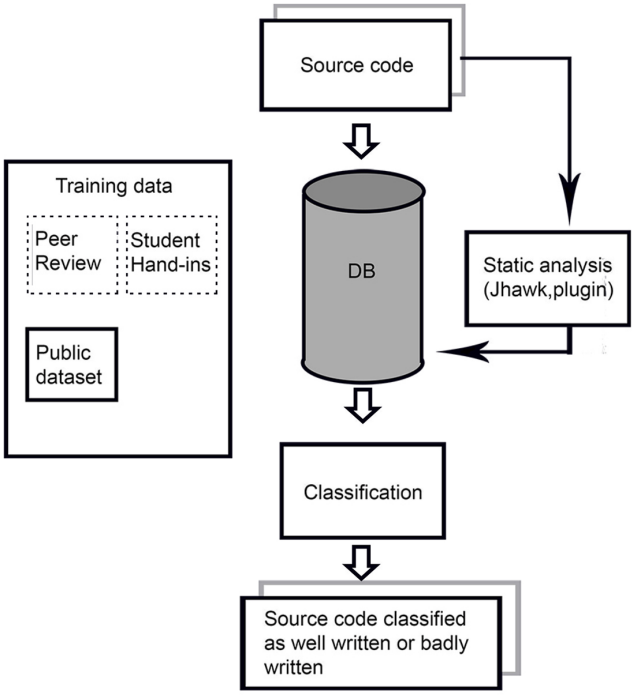


Figure 3: Architectural overview

Tools

The approach uses a MySql database for storing the collected training data and the results from static analysis. Some of the static analysis is implemented by the plugin, the rest is performed using JHawk. The classification is implemented in Python, the selected methods are k nearest neighbour (KNN), naïve Bayes (NB) and decision tree (DTree).

Static Analysis

The plugin extracts features based on static analysis. Static analysis gives us an opportunity to analyse the source code without executing the program. The source code to be analysed is given as an input to JHawk by command line, and a text file containing the static metrics is generated. These metrics are stored in the MySql database together with a textual representation of the source code to be analysed. The classification of the methods as well written or badly written are based on

this metrics together with some metrics generated by the plugin. An figure of the generated metrics are shown in Figure 2. Before the metrics are stored in the database, the metrics are converted to discrete values as described in Table 1.

Classification

Classification is the task of deciding which category a new instance belongs to based on the available training data. By using the generated metrics from the static analysis, and the available training data, it is possible to classify the methods; this process is shown in Figure 4.

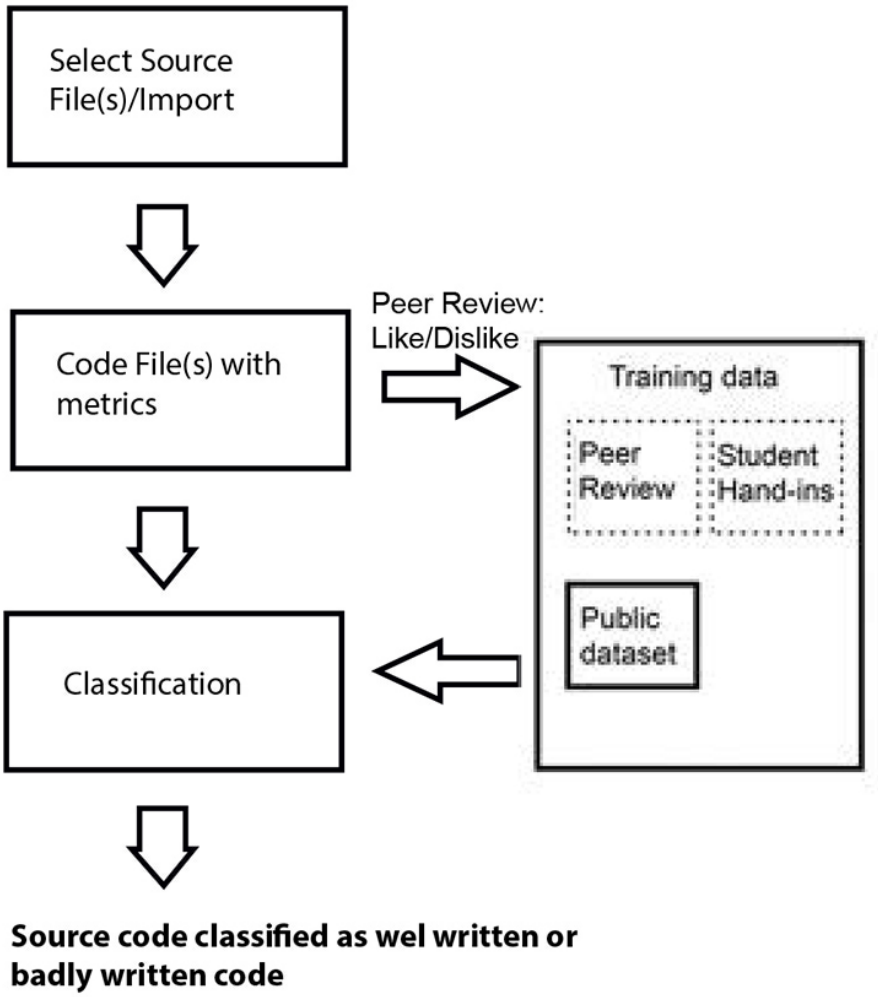


Figure 4: Classification process

An example of the classification of the source code class methods (see Figure 2) are shown in Table 3.

	Prediction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
Method 1	Well written	1	11	316	0.03	36	0.06	9	30	17.6	3	0
Method 2	Badly written	2	8	199	0.11	4	1.19	5	87	11.1	1	1
Method 3	Well written	7	42	1021	0.03	88	0.05	18	40	56.7	0	11
Method 4	Well written	1	10	298	0.03	78	0.03	6	0	16.6	4	0
Method 5	Well written	1	8	230	0.04	70	0.03	4	24	12.8	6	0

Table 3: Example of classification of student hand-ins process

5 Results and Discussion

The public dataset was classified with k nearest neighbour (KNN), naïve Bayes(NB) and decision tree(DTree), the results of this classifications are shown in Table 4[5].

Dataset	Features	Method	Badly written Pre	Badly written Rec	Well written Pre	Well written Rec	Total	
PROMISE1	21	KNN	**0.228**	0.356	0.906	0.837	77.95	%
		DTree	**0.364**	0.366	0.856	0.855	76.40	%
		NB	**0.084**	0.402	0.971	0.822	80.61	%
PROMISE1	15	KNN	**0.227**	0.348	0.903	0.836	77.68	%
		DTree	**0.374**	0.362	0.849	0.856	76.06	%
		NB	**0.087**	0.415	0.972	0.823	80.72	%
PROMISE1	9	KNN	**0.287**	0.423	0.910	0.848	79.43	%
		DTree	**0.339**	0.336	0.847	0.848	75.24	%
		NB	**0.323**	0.361	0.869	0.849	76.76	%
PROMISE2	9	KNN	**0.274**	0.409	0.909	0.846	79.11	%
		DTree	**0.364**	0.350	0.845	0.853	75.54	%
		NB	**0.506**	0.323	0.757	0.870	71.02	%
PROMISE2	15	KNN	**0.292**	0.419	0.907	0.848	79.28	%
		DTree	**0.361**	0.355	0.850	0.853	75.90	%
		NB	**0.523**	0.315	0.739	0.871	69.90	%

Table 4: Classification by K Nearest Neighbour(KNN), naïve Bayes (NB) and decision tree (DTree), CrossValidation = 10, Pre = Precision, Rec = Recall

As a baseline for the experiments the 21 features from PROMISE1 was used for classifications. At a first glance at the results in Table 4, it seems like NB performed best (80.72 %) on the original dataset with continuous values using 15 features, but when inspecting precision and recall for each class this might not be true. NB was able to classify 8.7 % of the badly written methods correctly and 97.2 % of the well written methods. By using the PROMISE2 dataset the precision for the badly written methods were improved at the expense of the precision for the well written methods. When optimizing the classification to find the badly written methods, NB was able to classify 52.3 %) of the badly written methods correctly and 73.9 % of the well written methods by using the PROMISE2 dataset and 15 features.

Including student hand-ins in programming as training data might give even better results. When automatically classifying these methods as well written/badly written by the first classification, we can manually inspect the source code with peer

[5]The classifications are also performed with CV = 25 and 50, the differences are small

review and use this data as training data for the next classification. An example of classifying student handins are shown in Table 3.

6 Conclusion

In this paper we have proposed an approach for classification of source code quality based on static metrics and training data from public datasets and student hand-ins in programming. Several Classification methods are used including naïve Bayes, knn and decision tree. When optimizing for finding the badly written methods, NB outperforms the other classifiers. The classifiers are better able to identify well written code than badly written code, a reason for this might be that the number of well written methods are larger than the number of badly written methods in the training data. When building up better training data, this results may improve. From this result, we conclude that it is possible to accurately identify the quality of the source code by using static analysis and classification.

7 Future work

By the time of writing, the approach handles only the promise datasets as training data, and use KNN, NB and DTree as classification methods. We have concrete plans to classify the STUDENT dataset by the quality of the code and to give immediate feedback to the students when they deliver their hand-in. We plan to use Support Vector Machine (SVM) to classify the source code based on available training data. The training data will be improved to get better results for both well written and badly written methods.

8 Acknowledgements

The paper is based on the work performed in a Master Thesis in ICT at University of Agder which facilitating identification of source code quality. This thesis is part of the Test Centred Learning Tool (TCLT) project at University of Agder [6]. The TCLT project aims at enabling students to work in a industry-like test-driven environment and to automatically get response on assignments handed in.

References

[1] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[2] V. Barstad, V. Shulgin, M. Goodwin, and T. Gjøsæter, "Towards a Collaborative Code Review Plugin," in *Proceedings of the 2013 NIK conference*, 2013, pp. 37–40.

[3] M. Tang. (2010) Caesar: A Social Code Review Tool for Programming Education. Massachusetts Institute of Technology. Accessed: 07/01/2014. [Online]. Available: http://groups.csail.mit.edu/uid/other-pubs/masont-thesis.pdf

[6] http://tclt.uia.no

[4] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson, "Jazzing up Eclipse with collaborative tools," in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, 2003, pp. 45–49.

[5] C. Catal, U. Sevim, and B. Diri, "Practical development of an Eclipse-based software fault prediction tool using Naive Bayes algorithm," *Expert Systems with Applications*, 2011.

[6] T. J. McCabe, "A Complexity Measure," vol. SE-2, no. 4, December 1976.

[7] M. H. Halstead, *Elements of Software Science*. Elsevier North-Holland, Inc, 1977.

[8] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," in *Proceedings of the International Conference on Software Maintenance(ICSM03)*, editor, Ed., 2003.

[9] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," vol. 86, pp. 2639–2653, 2013.

[10] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[11] D. Binkley, "Source code analysis: A road map," *Future of Software Engineering FOSE*, vol. 7, pp. 104–119, 2007.

[12] W. E. Wong and S. Gokhale, "Static and dynamic distance metrics for feature-based code analysis," *The Journal of Systems and Software*, vol. 74, pp. 283–295, 2005.

[13] A. Fehnker, R. Huuck, S. Seefried, and M. Tapp, "Fade to Grey: Tuning Static Program Analysis," *Electronic Notes in Theoretical Computer Science*, vol. 66, pp. 17–32, 2010.

[14] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[15] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "An Eclipse plug-in for the Detection of Design Pattern instances through Static and Dynamic Analysis," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010, pp. 1–6.

[16] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design smells," in *QSIC '09. 9th International Conference on Quality Software*, 2009, pp. 305–314.

[17] W. Cavnar and J. Trenkle, "N-Gram-Based Text Categorization," in *Proceedings of SDAIR-93, 3rd annual Symposium on Document Analysis and Information Retrival)*, 1994, pp. 161–175.

[18] C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998. [Online]. Available: http://dx.doi.org/10.1023/A%3A1009715923555

[19] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code?: Automatic classification of source code archives," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 632–638.

[20] P. Knab, M. Pinzger, and A. Bernstein, "Predicting Defect Densities in Source Code Files with Decision Tree Learners," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 119–125. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138012

[21] G. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *Software, IEEE*, vol. 23, pp. 76–83, 2006.

[22] G. Booch and B. A, "Collaborative Development Environments," *Advances in Computers*, vol. 59, 2003.

[23] M. Coccoli, L. Stanganelli, and P. Maresca, "Computer Supported Collaborative Learning in software engineering," in *Global Engineering Education Conference (EDUCON), 2011 IEEE*, 2011, pp. 990–995.