Using Machine Learning to Identify Code Fragments for Manual Review

Miroslaw Staron
Chalmers | Univ. of Gothenburg
Gothenburg, Sweden
miroslaw.staron@gu.se
ORCID: 0000-0002-9052-0864

Mirosław Ochodek
Poznan Univ. of Technology
Poznan, Poland
miroslaw.ochodek@cs.put.poznan.pl
ORCID: 0000-0002-9103-717X

Wilhelm Meding
Ericsson AB
Gothenburg, Sweden
wilhelm.meding@ericsson.com

Ola Söder
Axis Communications
Lund, Sweden
ola.soder@axis.com

***Abstract*—Code reviews are one of the first quality assurance tasks in continuous software integration and delivery. The goal of our work is to reduce the need for manual reviews by automatically identify which code fragments should be further reviewed manually. We conducted an action research study with two companies where we extracted code reviews and build machine learning classifiers (AdaBoost and Convolutional Neural Network — CNN). Our results show that the accuracy of recognizing code fragments that require manual review, measured with Matthews Correlation Coefficient, was 0.70 in the combination of our own feature extraction and CNN. We conclude that this way of combining automation with manual code reviews can improve the speed of reviews while providing organizations with the possibility to support knowledge transfer among the designers.**

***Index Terms*—code reviews, continuous integration, machine learning**

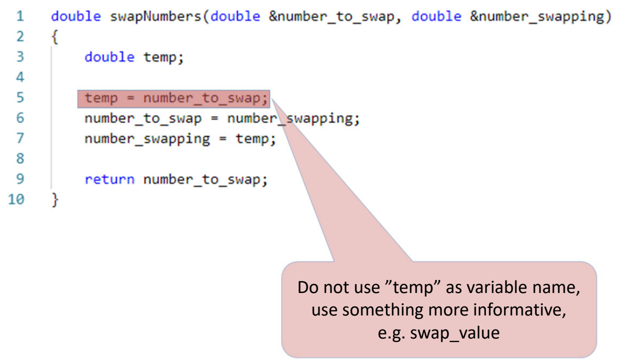
I. Introduction

The paradigm shift from simple Agile practices to Continuous Integration (CI) brought a different dynamics into software development [1], [2]. One of the dynamics was the evolution from using code walkthroughs and inspections to code fragments reviews (patches) in modern tools e.g. Gerrit.

Keeping high quality of the code base when reviewing code in patches, entails the use of many quality assurance steps between the check-ins of the code by the designers and its integration into the main branch. These steps include static analysis of the code (e.g. using Lint) and manual code reviews [3]. Despite their obvious benefits, these methods have several downsides. Manual reviews are time-consuming, effort-intensive and often reviewer-dependent [4].

Therefore, in this paper, we address the research problem of how to automatically process a large number of code patches submitted in CI flows, extract the rules for coding guidelines automatically and recommend code fragments/lines for manual reviews from the perspective of software designers. In particular, we set off to address the following research question: *How to automatically identify lines which require manual review in CI flows?*

The underlying challenges with the automatic identification of lines which require manual review are illustrated by an example presented in Figure 1. In the example, a code fragment is submitted to Gerrit's code review system and has been reviewed. The review comment shows that one line has been identified as problematic.



```
1   double swapNumbers(double &number_to_swap, double &number_swapping)
2   {
3       double temp;
4
5       temp = number_to_swap;
6       number_to_swap = number_swapping;
7       number_swapping = temp;
8
9       return number_to_swap;
10  }
```

Do not use "temp" as variable name, use something more informative, e.g. swap_value

Fig. 1. Underlying challenges of identifying code fragments for manual review – understanding whether the comment is positive or negative, and identifying the actual code fragment which is commented on.

The reviewer has reacted on the use of `temp` variable in the function swapping the value of two variables. The comment suggests changing the name of the variable, which means the code fragment (line 5 in this example) is the line that should be changed. Today, software designers use scripts/tools to find similar lines, which means that effort is needed to develop scripts for each comment in the code. The example illustrates two challenges: how to extract the information about which fragment is considered to be problematic (not all code review systems provide the tools to extract which lines were commented) and how to establish whether the comment's sentiment.

To address this question we conduct an action research project with two companies [5], in two cycles – designing the system and applying that in practice. We evaluated the tool using the standard machine learning metrics and through workshops with the companies. Our contribution is a method that automatically tags lines which need manual review (line 5 in this example) based on the historical review comments and highlight these lines for the reviewers.

978-1-7281-9532-2/20/$31.00 ©2020 IEEE
DOI 10.1109/SEAA51224.2020.00085

II. Action research design

The design of our study follows guidelines and good practices for action research studies in software engineering [6] and [7]. In this study, we combined the development of software with the development of knowledge at the collaborating companies. Table I presents a summary of the design of our action research study.

In the first cycle, we focused on the design of the workflow of the analysis of code reviews in the company's tools—design the tools for data extraction and provide an automated data analysis flow. We used two open source projects as study objects – Wireshark[1] and Gromacs[2]. Both tools are used in industrial applications and have a significant number of code review comments (over 5,000 for Wireshark and over 1,000 for Gromacs). In the second cycle, we focused on the improvement of accuracy as well as designing and evaluating the data analysis flow. We also conducted workshops with companies to address the most challenging organizational aspects of this new method.

Company A develops embedded software products for the commercial market. The products are matured and the development team has over a decade of experience of agile software development and have been using continuous integration for over five years. We chose the company as it provided us with the possibility to analyze their code for software integration. The code is a mixture of Python and a proprietary integration language. This means that the code needs several preprocessing steps before it can be combined, which renders the static analysis and compiler-based approaches unusable in this context. From that perspective, we see the analysis of this code as the hardest scenario for automated analysis, which means that we can generalize the results to simpler contexts.

Company B is a large infrastructure provider from Sweden. The studied organization within this company has over 100 developers who work in a combination of Agile and Lean principles. They develop an embedded software product that has been on the market for over ten years and has a stable, mature code base. The organization has adopted practices of CI for over five years. The studied code base was in C and C++.

Both companies use a similar set-up of modern software development tools, including Jenkins for continuous integration and Gerrit for code reviews. The size of the teams is similar too. However, at these companies, the testing tools and equipment are different, as the products are different.

III. Results

A. Cycle 1: Code review extraction system

In the **diagnosing** phase, we decided to use Gerrit as the data source, bag-of-words as the algorithm to extract features from the code and to use keyword-based sentiment analysis for the classification of code fragments.

[1]http://www.wireshark.org
[2]http://www.gromacs.org

In the **action planning** phase, we decided to focus on individual lines of code instead of code fragments. This was dictated by the fact that the literature in the area of code measures showed that size is an important factor for measurement [8], [9]. In order to reduce the effect of the size, we chose single lines as code fragments.

Our **action taking** was to design the extraction system which is presented in Figure 2, the code is available in our GitHub repository[3].

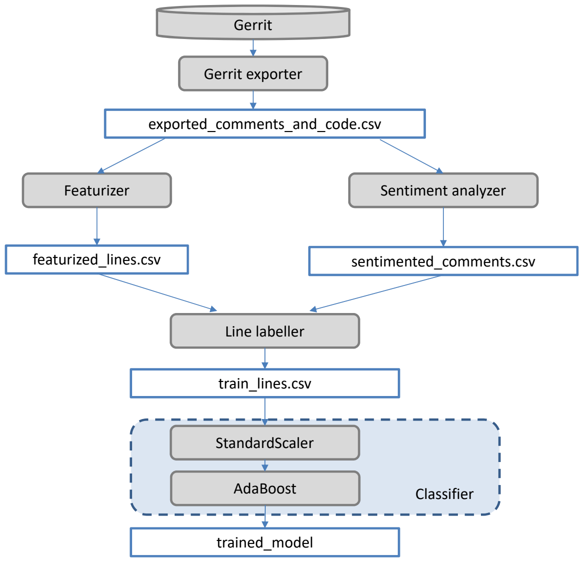


Fig. 2. Overview of the processing workflow. Tools and algorithms are presented in the grey background while the resulting or intermediate files are presented in a white background.

The algorithm starts with the extraction of `JSON` data from a Gerrit instance. The extraction algorithm goes through all merged or rejected changes and identifies which lines of code were commented on. It then creates a large comma-separated file where each line that was commented on is stored together with the text of the comment (`exported_comments_and_code.csv`). If the code fragment contains multiple lines, the comment is duplicated for each of the lines. If the comment has multiple lines, the line breaks are removed so that the comment text can be stored in one line.

The resulting file is an input to two other tools—featurizer and sentiment analyzer. The featurizer tool processes the lines of code and extracts code features from them. The sentiment analyzer processes the comments by looking for predefined keywords and decides whether the comments are positive or negative. The results of both these tools are used as input to the line labeller, which merges these two files into one large data table, used for the classifier. In Figure 2, we show the use of AdaBoost classifier combined with the StandardScaler

[3]https://github.com/miroslawstaron/auto_code_reviewer

TABLE I
SUMMARY OF ACTION RESEARCH CYCLES

Cycle	Diagnosing	Action planning	Action taking / Executing	Evaluating	Learning
1	Refined the research question to be *How to extract the information about coding guidelines violations from Gerrit?*	The plan was to design a tool and test it on open source and industrial projects.	We extracted the data from four different projects: Wireshark, Gromacs, Company A and Company B.	The evaluation showed that there is a potential, but we need to refine the methodology.	We learned which types of lines were to be extracted and how to classify the lines into good and bad.
2	How to automatically identify the coding violations based on the information extracted from Gerrit?	We expanded the study to use both neural networks and more advanced machine learning methods. We also planned for closer work with Company A to study the organizational aspects of our proposed method.	We studied the same product as in the first cycle, but we used token-based feature extraction and neural networks.	Token-based feature extraction and sentiment analysis increased the MCC score by 0.39 (to 0.70).	The accuracy of the algorithm needs to be complemented with good recommendations for the organization to adopt it.

algorithm. The AdaBoost uses multiple decision trees for the classification (100 trees with max. 20 levels each). The StandardScaler algorithm normalizes each feature column-wise so they have mean equal zero and unit variance.

In the **action evaluation** phase, we applied this workflow on four projects —Wireshark, Gromacs, Company A and B.

For the evaluation metrics, we used the Matthews Correlation Coefficient and F1-score. The first metric is important to choose as we wanted to measure the balance between the correctly classified code fragments and the ones classified incorrectly (for both decision classes). The F1-score allows us to balance the quality of the predictions in terms of precision and recall (how many relevant cases were captured and how many non-relevant ones were included). The results from the evaluation are presented in Figure 3.

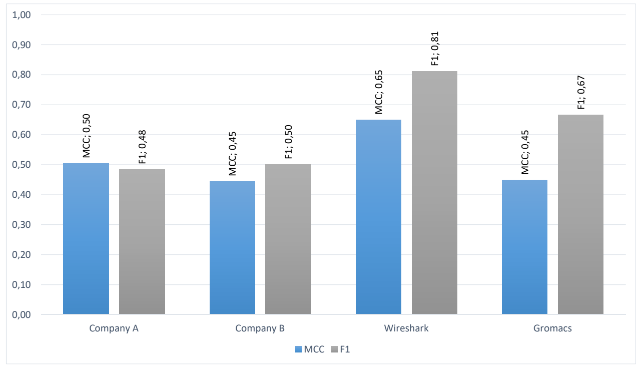


Fig. 3. Comparison of the performance of algorithms for different data sets, with the same feature extraction algorithm.

The **specifying learning** phase was focused on understanding the causes behind the differences in the performance between the projects, and triggered the discussion on how to improve the proposed prediction workflow. Our further manual examination of a selection of code fragments showed that we need to collect more data from the reviews and that we also need to be more refined in the analysis of the sentiment of the comments. When reading the feature lists for each source code base, we also noticed that there is a difference in the number of features extracted. Therefore, we decided that we need to evaluate the use of two different feature extraction techniques in Cycle 2—a token-based featurizer and the natural-language-processing-based bag-of-words featurizer.

B. Cycle 2: Automated recommendation system

In the **diagnosing** phase, we explored the challenge related to the accuracy of the classification algorithms. Therefore, we set off to investigate the research question of how to automate the entire process and whether using neural networks would improve accuracy. Since we knew that neural networks are black-box classifiers, we complemented them with the k-means algorithm to find similar lines of code and therefore similar comments after a specific line was identified to require manual review.

In the **action planning** phase we, decided that we would once again collect the data from Company A and focus on that company as there we had access to the practitioners who could verify our classifications. Their code base was also deemed to be the hard-case scenario for our algorithm as the code was a mix of Python, Bash, and a proprietary build scripting language in each code file/fragment. The scenario was the hardest as there is no compiler for that language and it requires pre-processing and therefore no static analysis tool was possible in the CI-flow.

In the **action taking** phase, we exchanged the feature extraction algorithm from Bag-of-Words to token-based feature extraction. Both algorithms analyze the entire code base and use tokens as features. The main difference is in the selection of which tokens to use. The Bag-of-Words algorithm chooses the tokens based on the frequency of their appearance in the text. Selecting the most frequently appearing tokens as features could help in reducing the sparseness of the features matrix, however, it could also lead to introduction of noise – different lines belonging to different decision classes have the same representation in the features space. To mitigate this problem, we designed our own token-based featurizer which selects the tokens as features that allow distinguishing lines belonging to different decision classes. The resulting features matrix is, therefore, more sparse when using a token-based featurizer instead of Bag-of-Words. However, this algorithm guarantees that two lines which are different have different representation in their feature vector.

In order to deal with the challenges of having larger feature vectors, we also exchanged the classifier from AdaBoost to Convolutional Neural Network with one LSTM Bidirectional Layer.

In the **action evaluation**, we extended the data set from cycle 1 with more data points. The results of the classification for the validation set are presented in Table II.

TABLE II
SUMMARY OF THE CLASSIFICATION RESULTS FOR THE VALIDATION DATA SET FROM COMPANY A.

Cycle	True positives	True negatives	False positives	False negatives
1: Bag of words	12,210	216	170	844
2: Token-based	19,161	3,298	1,136	1,041

In the phase of **specifying learning**, we discussed these results with the review team at Company A. The accuracy numbers were good and we could explain how the algorithm deals with the harder cases, e.g. identifying lines that are not sorted alphabetically (an important property for the reviewed program code in the company).

C. Summary of the results: How to automatically identify lines which require manual review in CI flows?

To summarize the results from our work, we can examine the performance of different set-ups which we evaluated in cycle 1 and cycle 2 of our action research cycle. The combination which provides the best performance in terms of both true positives and true negatives is the combination used in the second cycle of our study—Convolutional Neural Network, our own token-based featurizer and using comments to classify each line as good or bad. The summary is presented in Figure 4.

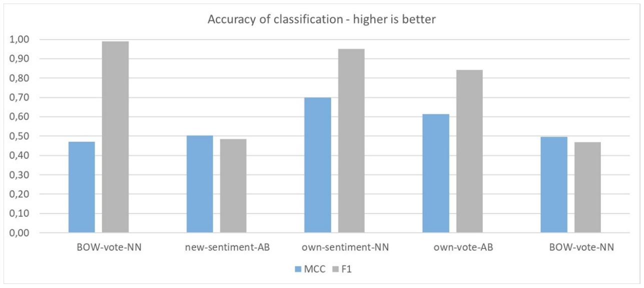


Fig. 4. Overall comparison of the different combinations of the classification algorithms and feature extraction methods for Company A.

This means that the answer to the research question from our study is: In order to automatically identify lines which require manual review in CI flows, we need to:

- use previous comments as input to classify the lines—transformed into a feature vector using sentiment analysis,
- a token-based feature extraction mechanism to extract features from the source code, and
- use Convolutional Neural Network as a classifier.

What we learned during the project was that we should focus on providing recommendations for which lines and fragments should be manually reviewed, rather than providing the suggestions for improvements. This is because we saw that organizational learning is stimulated by this kind of recommendations rather than replaced by the automation.

IV. Conclusions

In this paper, we addressed the problem of how to automatically identify lines which require manual review in CI flows. This challenge is important in modern software development organizations as the continuous inflow of code commits require reviews and overflow the designers. This, in turn, takes effort from the experienced designers [4] and causes delays in the integration process [10].

By conducting an action research study with two companies, we could design an automated workflow that uses previous code review comments as a basis for identifying code fragments. We also constructed our own feature extraction method from the source code. The results showed that we could identify the lines that require manual review with high accuracy, with the Matthews Correlation Coefficient of 0.70.

This work opens up for new possibilities: we can move to the analyses of larger code fragments to enlarge the context provided for the designers when reviewing the code, or we can identify challenging code fragments and use them for internal knowledge transfer activities.

Acknowledgments

This research has been conducted at Software Center, Chalmers, University of Gothenburg, Poznan University of Technology, Axis Communications, and Ericsson.

References

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
[2] M. Meyer, "Continuous integration and its tools," *IEEE software*, vol. 31, no. 3, pp. 14–16, 2014.
[3] F. Zampetti, G. Bavota, G. Canfora, and M. Di Penta, "A study on the interplay between pull request review and continuous integration builds," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 38–48.
[4] ZDNet, "Linus Torvalds: I'm not a programmer anymore," *ZDNet magazine*, 2019.
[5] M. Staron, *Action Research in Software Engineering, Theory and Applications*. Springer, 2020.
[6] P. S. M. d. Santos and G. H. Travassos, "Action research use in software engineering: An initial survey," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 414–417.
[7] M. Staron, *Action Research in Software Engineering: Theory and Applications*. Springer, 2020.
[8] M. A. Al Mamun, C. Berger, and J. Hansson, "Effects of measurements on correlations of software code metrics," *Empirical Software Engineering*, pp. 1–55, 2019.
[9] Y. Gil and G. Lalouche, "On the correlation between size and metric validity," *Empirical Software Engineering*, vol. 22, no. 5, pp. 2585–2611, 2017.
[10] M. Staron, W. Meding, O. Söder, and M. Bäck, "Measurement and impact factors of speed of reviews and integration in continuous software engineering," *Foundations of Computing and Decision Sciences*, vol. 43, no. 4, pp. 281–303, 2018.