Automatic classification of software issues via code maintainability metrics

Marco di Biase
Software Improvement Group
Amsterdam, The Netherlands
m.dibiase@sig.eu

Magiel Bruntink
Software Improvement Group
Amsterdam, The Netherlands
m.bruntink@sig.eu

Arie Van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.VanDeursen@tudelft.nl

***Abstract*—Understanding the purpose of software issues is a challenging problem in software engineering. Even more so is to automatically detect their goal given some information about them. Several studies provided techniques to automatically classify changes using details taken from source code metadata and issue trackers.**

In this work, we investigate if we can classify issues using source code as our only source of information by extracting maintainability metrics and computing the differences at issue level. We use a manually classified dataset of more than 4,000 issues belonging to five Java Open-Source projects. We conduct an initial evaluation by using these values to train a machine learning algorithm. We validate our approach first on the dataset itself, and then we test our approach against a set of issues extracted from GitHub using GHTorrent.

I. Introduction

Change classification in software systems is a known problem as software products continuously evolve and increase in complexity [1]. Issue classfication is still a responsibility for the reporter; this is both time-consuming and error-prone. Reporters, which often are users of the software system, generally have a shallow understanding of the technicalities of the software artifact. When something is not correctly working they file a new issue most likely as a bug. Previous research back up this claim: Herzig *et al.* found that more than 40% of the issues are not classified correctly [2]. This misclassification can cause a plethora of issues: for instance, a misclassified issue is correlated with its reassignment [3]. Given a change for a system, being able to automatically classify its purpose would provide a real-world benefit for scenarios like the one just mentioned.

Previous studies already explored the field of automatic software changes classification. Hindle *et al.* [4] propose a technique to categorise large changes into software maintenance categories using a set of features extracted from the software projects. Thung *et al.* [5] propose a classification that automatically puts defects into three categories, part of the IBM's Orthogonal Defect Classification, using a mix of data coming from source code and bug reports. Somasundaram and Murphy [6] propose an automatic classification which is able to categorize issues using the bug report textual description.

In spite of the innovative techniques proposed to classify software changes in a different set of categories, these are done always using data coming from the issue tracker. The classification of an issue becomes more challenging in scenarios in which one would consider having available only code artifacts. In fact, when issue tracker informations are not available, the aforementioned approaches show severe limitations that hinder their applicability. This is the case, for instance, when dealing with closed source code. Coming up with data relative to the issue tracker or the versioning system is, in fact, often unfeasible.

To fill this gap, in this paper we propose a novel approach on how to classify issues belonging to software products, using fine-grained differences at issue level in maintainability metrics. Firstly, we set our research goal to understand if different categories of issues show differences in maintainability metrics when analysing their fixes. In addition to that, we use the resulting dataset to evaluate how effectively maintainability metrics can be used to automatically classify issues in software projects. This approach can provide an abstraction relatively to the classification of issues by using only source code, without relying on any data coming from the issue tracker.

Using a manually pre-classified set of issues categorized by Herzig *et al.* [2], we set up our study aiming at discovering the differences in the computed metrics of an aggregated change-set (*i.e.*, the the set of commits that fix an issue, aggregated by their source code modifications). We analyse more than 2,000 issues belonging to five popular and widely studied Java Open-Source projects. We train a Machine-Learning algorithm to automatically classify these issue, using the aforementioned metrics as features. Then, after evaluating the performances of various approaches, we seek to externally validate our findings. By extracting the same maintainability metrics from other issues belonging to Java Open-Source projects on GitHub, we estabilish the confidence level that our approach has when assessing the performance of the classificator.

II. Research method

This section defines the overall goal of our study, motivates our research questions, and outlines our research method.

A. Research Questions

The ultimate goal of our study is to understand the primary purpose of aggregated code changes that are fixing an issue, given only its source code. We define *aggregated code changes* as *the set of commits that fix an issue, aggregated by their*

source code modifications. The challenge is that we do not use description messages or any of additional info regarding these changes.

Our analysis of past literature that studied automatic classification of code changes provides insight as how to categorise these in maintenance tasks [4]. However, such studies make use of a plethora of informations that are not strictly related to the source code itself (*i.e.*, author identity, word distribution, *etc.*). Therefore, the first research question that we aim to answer is is:

RQ1. Can aggregated source code changes be automatically classified based on maintainability metrics?

Given the relevance that this has for our final purpose, the natural next step is to test the automatic classification. For this reason, we research the extent of the approach described in RQ1 in classifying aggregated code changes according to our strategy. The accuracy of an automatic classifier is an essential step to provide real-world benefit. Our second research question is then:

RQ2. Based on the Machine Learning approach, is the automatic classification of aggregated code changes effective?

B. Selection of subject systems

To conduct our study, we limited our scope to Open-Source project written in *Java*. This is because we want a narrower sample to gather more meaningful results. Our hypotheses here is that different languages might have differences in metrics, thus introducing some level of inconsistency. In fact, Zhang *et al.* [7] reports that the most influential context factor that impacts the distribution of maintanability metrics is the programming language.

The work by Herzig *et al.* [2] provides a reliable dataset of manually classified issues in five different Java projects.[1] We filter out only the issues belonging to the categories:

- *BUG*, that classifies corrective maintenance tasks
- *RFE*, that classifies adaptive maintenance task that implement a new functionality
- *IMPROVEMENT*, that classifies perfective maintenance that improve an existing functionality

This is because the total number of issues for each category not belonging to these three are far outnumbered. Using this predefined dataset we build the aggregated code changes that fix each issue. Using the log of commit messages provided by the version control system, we filter data using the unique issue identifier. Commits that contain this id are added to our set of issues. Our dataset consist of 4,346 issues: 2,230 are classified as BUG, 1,071 are classified as IMPROVEMENT and 1,045 are classified as RFE.

C. Extracting aggregated changesets from classified issues

Once we select the sample of issues with their aggregated code changes, we extract the maintainability metrics. To this end, we use Software Improvement Group (SIG) System Analysis Toolkit (SAT). SAT is a source code analysis tool that retrieves maintainability metrics of a system given its codebase. Metrics are based on the SIG Maintainabilty Model, and maintainability measurement are compliant to the ISO/IEC 25010 standard [8].

To gather all the metrics we checkout the projects source code from their git repository. For each issue in our dataset, given the commits that fix that issue, we extract the maintainability metrics.

D. Automated classification of source code maintainability metrics

In the first research question we set to investigate to what extent and with which accuracy source code maintainability metrics can be used to automatically categorize aggreagated software changes. To this end, we use automatic classification techniques that are capable of extracting common features given the differences between categories.

We set the detail of our delta to file level, but then we aggregate our results at issue level. In fact, our final purpose is to classify what we have defined as the *aggregated code changes* defined in Section II-A.

Having a large dataset to answer RQ1 (comprising more than 4,000 issues over five Java Open-Source projects), we make use of supervised machine learning techinques to build the classificator. In particular, we tested two different classes of supervised classifiers: (1) probabilistic classifiers, such as naive Bayes or naive Bayes Multinominal, and (2) decision tree algorithms, such as J48 and Random Forest. These classes make different assumptions on the underlying data, as well as have different advantages and drawbacks in terms of execution speed and overfitting.

E. External validation of our approach

To build an external validation set, our source of data is GHTorrent [9], a dataset that mirrors the content of GitHub, one of the most popular host for Open-Source software. We select only Java systems that: 1) are not forked from other projects 2) have at least 50 stars 3) have at least 10 issues 4) the issues have at least one commit referenced . This filtering choice is to discard small-scale projects that might not be representative for our target, as well as to remove issues that have no reference to any commit or activity. We extract 1847 projects and 222,034 issues, that we random sample to gather a significant set of issues. We set a range of systems, rather than the number of issues, as variable to use for the sampling. This is because results in the sample sets give a better accuracy when sampling issues from a limited set of systems. Moreover, this allows to speed up the metric extraction phase.

III. Conclusions

In our proposed research talk, we plan to discuss about our work on automatic issue classification using source code maintanability metrics. Through our presentation, we would like to show our first results and discuss with the audience with the goal of gathering useful feedback and discuss potential improvement and future work.

[1] https://www.st.cs.uni-saarland.de/softevo/bugclassify

REFERENCES

[1] M. M. Lehman. *Laws of software evolution revisited*, pages 108–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[2] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[3] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Not my bug! and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 395–404. ACM, 2011.

[4] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 30–39, May 2009.

[5] Ferdian Thung, David Lo, and Lingxiao Jiang. Automatic defect categorization. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 205–214. IEEE, 2012.

[6] Kalyanasundaram Somasundaram and Gail C Murphy. Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India software engineering conference*, pages 125–130. ACM, 2012.

[7] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E. Hassan. How does context affect the distribution of software maintainability metrics? In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 350–359, Washington, DC, USA, 2013. IEEE Computer Society.

[8] Iso/iec 25010:2011. https://www.iso.org/standard/35733.html.

[9] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 12–21, Piscataway, NJ, USA, 2012. IEEE Press.