Software Metrics as Identifiers of Defect Occurrence Severity

GORAN MAUŠA, University of Rijeka
TIHANA GALINAC GRBAC, Juraj Dobrila University of Pula
LUCIJA BREZOČNIK, VILI PODGORELEC and MARJAN HERIČKO, University of Maribor

Successful prediction of defects at an early stage is one of the main goals of software quality assurance. Having an indicator of the severity of defect occurrence may bring further benefit to allocation of testing resources. This paper is a part of a project aimed at identifying the role of different software metrics in order to improve the software quality assurance activities. In a preliminary case study we analyzed the relationship between the software metrics and defects using fundamentally different approaches for feature selection. Our case study showed that some metrics do not indicate defect occurrence, several of them exhibit moderate level of correlation, and the choice of the appropriate metrics is biased by the choice of feature selection technique. The next step would be to integrate the finding from different approaches and various datasets to develop a hybrid method for a precise definition of software metrics and their threshold levels that are good indicators of defect occurrence.

1. INTRODUCTION

Software quality assurance is in need of finding indicators for early corrective activities in the software development life-cycle. Code smell is a term used to encompass the potential indicators of deeper problems in a software [Fowler 2018]. Unlike software defects, the code smells usually do not encompass the kind of bugs that prevents the software from functioning properly. Instead, they are used to describe technical incorrectness or other weaknesses in design that may deteriorate software development or increase the risk of future defects [Vidal et al. 2018]. Obviously, the code smells and software defects are interlinked and both pose a threat to software correctness, validity and performances.

The code smells may be divided into three main groups, depending on the granularity level of their analysis: application-level, class-level, and method-level smells [Fowler et al. 1999]. At each level, different software metrics may be extracted and the open research question is which of them are important for the detection of code smells. Existing metrics most often measure static code attributes like cohesion, coupling, complexity, encapsulation, inheritance, and size. These metrics have been widely used to build models for software defect prediction [Basili et al. 1996]. However, obtaining good prediction models requires the selection of appropriate software metrics and the definition and validation of their thresholds for a particular programming language and/or project type. Nowadays, we have two main branches of threshold derivation: statistical approaches and approaches based on machine-

This work has been supported in part by CEEPUS Mobility grant programme (CII-HU-0019-12-1617-M-98633), Bilateral project (BI-HR/18-19-036) and by the University of Rijeka Research Grant 18.10.2.1.01.
Corresponding author's address: G. Mauša, Faculty of engineering, Vukovarska 58, 51000 Rijeka, Croatia; email: gmausa@riteh.hr.

In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

learning techniques. The problem that occurs is that different code smell detection tools, as well as metrics tools/programs, give significantly different results, resulting in different perception of code smells [Paiva et al. 2017].

This paper is a continuous effort on analyzing code smells and their impact on software quality [Gradišnik et al. 2019b; Gradišnik et al. 2019a]. The aim of this paper is to address the variability of object-oriented software metrics' importance for detecting the problematic code by investigating the relationship between faults and code smells, i.e. software components with technical debt. The relationship is analyzed on 5 subsequent releases of Eclipse JDT open source project written in Java by using four different approaches:

(1) statistical approach based on Spearman's correlation analysis,
(2) feature selection algorithm based on optimization algorithm Binary Particle Swarm Optimization,
(3) univariate classification algorithm based on Logistic regression, and
(4) threshold derivation algorithm based on Bender method.

The structure of the paper is the following: a brief description of studies that motivated our approach is presented in section 2, the algorithms of the four different approaches is given in section 3, the results of the case study are shown in section 4, while the discuccion along with threats to validity and the conclusion is given in sections 5 and 6.

2. BACKGROUND

A vast number of collected and stored attributes or features is nowadays presenting more disadvantages than advantages. One of the biggest reasons for that statement lies in the problem of building a suitable classifier based on the whole set of features, and related significant time complexity [Brezočnik et al. 2018]. To overcome such problems, researchers often employ some feature selection methods.

Too many independent variables can also have negative effects on model's fault-proneness prediction, making the model more dependent on the data set currently in use and therefore less general [Briand et al. 2000]. Selecting the appropriate measures to be used in the model requires a strategy of minimizing the number of independent variables in the model. This paper investigates the usage of static code attributes as independent variables and univariate forward and backward step-wise selection principles in choosing the appropriate ones. By finding the subset of the most relevant measures, we might also gain deeper understanding of the relations between software metrics and defect occurrence. The model performances are evaluated using widely used performance measures such as accuracy, sensitivity and false alarm rate [Zhou and Leung 2006; Guo et al. 2004]. A defect prediction model should identify as many fault prone modules as possible while avoiding false alarms [Lessmann et al. 2008].

Some studies analyzed the relationship between software metrics and defects and found that certain metrics do exhibit strong association. An early study by Shatnawi et al. [Shatnawi 2010] and a recent study by Arar et al. [Arar and Ayan 2016] analyzed several object-oriented metrics like coupling between objects ('CBO'), response for class ('RFC'), weighted method per class ('WMC'), depth of inheritance tree ('DIT') and number of children ('NOC') . The results of their analysis showed that 'CBO', 'WMC' and 'RFC' metrics are the ones that have stronger association to software defects that the remaining 'DIT' and 'NOC' metrics. The replicated study by Arar et al. [Arar and Ayan 2016] included also other metrics like lack of cohesion between objects ('LCOM'), maximum and average cyclomatic complexity ('MAXCC', 'AVCC') and lines of code ('LOC') and showed that these metrics may also prove to be significant for successful prediction models. However, the object-oriented programs can hold several types of software complexity [Seront et al. 2005] and hence, our project goal is to examine a larger number of software metrics aiming to find the appropriate Java code smell identification metrics.

3. SOFTWARE METRICS ANALYSIS

The goal to find relationship between static code attributes and defect occurrence in the program code. Four different techniques are described in the following subsections.

3.1 Correlation Analysis

Correlation analysis is a standard statistical approach to establish whether there exists a relationship between two variables. As the values within the variables are usually distributed normally, Pearson's correlation is the standard technique. However, in software engineering this is rarely the case, so the Spearman's correlation, a non-parametric variant, is used to express correlation [D'Agostino and Pearson 1973]. Spearman's correlation coefficient (r_s) measures the power of a monotonic relationship between paired data in a sample s, that consists of n observations, using the following expression [Myers and Well 1991]:

$$r_s = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n^3 - n} \tag{1}$$

Parameter d_i presents the difference between the ranks of two variables at each observation i, yielding the values in range $-1 \leq r_s \leq 1$ with interpretation similar to Pearson's correlation [Lehamn et al. 2005]. Positive and negative values indicate correlation, whereas values close to zero indicate absence of correlation. The more distant the value is from zero, the stronger the correlation is, and it is interpreted according to the five levels of correlation strength, as represented in Table I [Evans 1996].

Table I. Correlation strength levels

$\lvert r_s \rvert$	**Interpretation**
0.00-0.19	Very poor correlation
0.20-0.39	Low correlation
0.40-0.59	Moderate correlation
0.60-0.79	Strong correlation
0.80-1.00	Very strong correlation

3.2 Binary Particle Swarm Optimization and Feature Selection

Algorithm BPSO+C4.5 [Brezočnik, Lucija 2017] uses a Binary Particle Swarm Optimization algorithm (BPSO) [Kennedy 1997] as a basis and extends it with Feature Selection mechanism. Since particles in the BPSO+C4.5 algorithm are moving in a binary search space, their initialization is done with binary values 0 and 1. Each particle represents a feasible solution in an n-dimensional search space. For example, a particle 1001, presents 4-dimensional space where only the first and the last features are included.

The particles are moving through iterations following two extremes. First one is individual extreme $pBest_{\mathrm{i}}$, which comprises the best-obtained position of each particle i in the n iterations $pBest_{\mathrm{i}} = (pBest_i^1, pBest_i^2, \ldots, pBest_i^n)$. The second extreme is called global extreme $gBest$ which comprises the best-found solution so far in the entire swarm $gBest = (gBest^1, gBest^2, \ldots, gBest^n)$.

The movement of the i-th particle from *old* to *new* position in the search space is controlled by velocity vector $V_i = [v_i^1, v_i^2, \ldots, v_i^n]$ and by position vector $X_i = [x_i^1, x_i^2, \ldots, x_i^n]$. Velocity is calculated according to the equation 2 and consists of three main parts:

$$v_{id}{}^{new} = \omega \times v_{id}{}^{old} + c_1 r_1 (pBest_{id}{}^{old} - x_{id}{}^{old}) + c_2 r_2 (gBest_d{}^{old} - x_{id}{}^{old}) \tag{2}$$

where ω is the inertia weight, c_1 and c_2 are positive constants and r_1 and r_2 are two random functions in the range [0,1]. Inertia component $\omega \times {v_{id}}^{old}$ is responsible for controlling the velocity of each particle, cognitive component $c_1 r_1({pBest_{id}}^{old} - {x_{id}}^{old})$ or particle memory tends to direct particles to their personal best positions, and the social component $c_2 r_2({gBest_d}^{old} - {x_{id}}^{old})$ which tends to direct particles to the globally best position found so far in the entire swarm. After the velocity update, position adjustment is carried out with equation 3:

$$x_{id}^{new} = \begin{cases} 1, & \text{if } \frac{1}{1+e^{-v_{id}^{new}}} > U(0,1) \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

where e is the base of the natural logarithm and $U(0,1)$ is the uniform distribution.

3.3 Univariate Logistic Regression

Logistic regression is a statistical modeling method used to estimate the probability of occurrence of an event, referred to as the dependent variable. Depending on the number of independent variables that are fitted to a logistic function, there are multivariate and univariate models, the later containing only one variable. Logistic regression is suitable for binomial dependent variables and hence it is an appropriate model for software defect prediction. In comparison to other related statistical techniques it is more flexible and robust [Hamadicharef et al. 2008; Tabachnick et al. 2007]. It does not assume normal distribution or equal variance within independent variables, nor linear relationship between the dependent and independent variable. These characteristics make logistic regression a commonly used classification algorithm.

The logistic regression models the probabilities of two different classes, ensuring the sum of probabilities equals 1 using the following equation [Hastie et al. 2009]:

$$P(X) = \frac{e^{\beta_0 + \beta_1 \cdot X}}{1 + e^{\beta_0 + \beta_1 \cdot X}} \tag{4}$$

where β_0 is the free coefficient and β_1 is the regression slope coefficients for dependent variable X. In case of software defect prediction, X represents a software metric and $P(X)$ is the fault-proneness probability. The coefficients β_0 and β_1 define the curvature of the non-linear logistic regression output curve and they are estimated by the maximum likelihood procedure.

A likelihood ratio chi-square test is usually used to assess the statistical significance of each independent variable in the model. The null-hypothesis is: there is no relationship between the logistic regression model and the dependent variable, i.e. the true coefficients are zero. Finally, if the level of statistical significance *p-value* is below 0.05, the dependent variable is considered to be significant for prediction. In univariate logistic regression model, each software metric is estimated separately and this is another way to find the ”appropriate indicators of software quality”.

3.4 Bender Method

Bender method is a threshold derivation method based on the univariate logistic prediction model [Arar and Ayan 2016], graphically presented in Figure 1. The method itself consists of three phases: sampling the data according to the stratified 10-fold cross-validation, training the univariate logistic regression model, and computing the threshold level for all the metrics which are statistically significant for prediction in the logistic model. The fourth phase is the testing phase in which threshold-based defect prediction models are built and their effectiveness is evaluated in terms of the geometric mean (GM) accuracy measure. In these models, an instance is declared fault-prone should a metric level go beyond the calculated threshold.

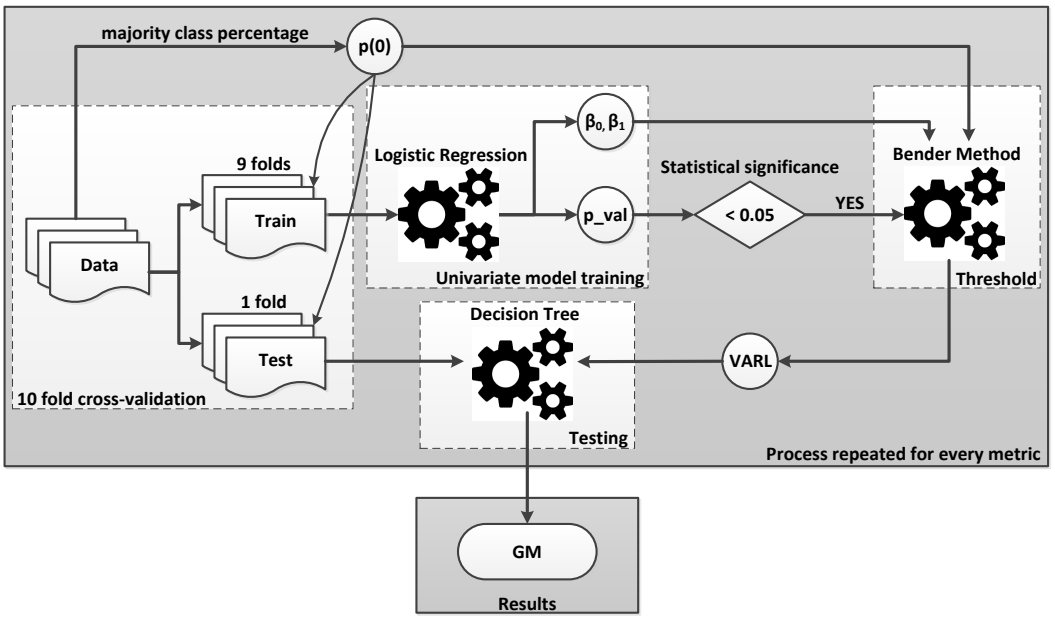

Fig. 1. Flow chart of Bender Method and its evaluation process

The univariate logistic regression model described in previous subsection is used in the same way in this method. After performing the analysis of statistical significance, the free coefficient β_0 and the regression slope coefficient β_1 are passed along with the the base probability $p(0)$ to compute the threshold level. As shown in figure 1, the percentage of majority class is used as the base probability [Arar and Ayan 2016], thus taking into account the probability that a randomly selected instance belongs to the majority class. The value of an acceptable risk level, i.e. the threshold value for each metric THR is computed using the following equation [Bender 1999]:

$$THR = \frac{1}{\beta_1}(ln(\frac{p_0}{1-p_0}) - \beta_0), \tag{5}$$

The effectiveness of calculated threshold levels for each metric is evaluated in the last phase. A simple decision tree is built using the obtained threshold levels to classify the software modules from the testing dataset according to the following equation:

$$Y_i = \begin{cases} faulty & if X_i > THR \\ non-faulty & if X_i \leq THR \end{cases}, \tag{6}$$

where i represents the $i-th$ software module in the testing dataset, X_i represents the actual value of software metric and Y_i represents the prediction output. After comparing the prediction results against the known values of defect proneness in the testing set, GM is calculated as:

$$GM = \sqrt{TPR * TNR} \tag{7}$$

where true positive rate (TPR) and true negative rate (TNR) represent the accuracy of positive (faulty) and negative (non-faulty) class, respectively.

$$TPR = \frac{TP}{TP+FN}, \quad TNR = \frac{TN}{TN+FP} \tag{8}$$

4. RESULTS

This section presents preliminary results of a case study used to compare the four fundamentally different approaches which we used for scoring the importance of software metrics for defect prediction. Each of these four approaches has been used to perform some sort of feature selection, i.e. to find the important indicators of software quality. Only the BPSO method is designed to perform pure feature selection based on wrapper method, whereas other methods are used for different purposes. The aim of this comparison is to estimate their degree of overlapping and open new research questions about this issue.

4.1 Datasets

The datasets used in this case study are five consecutive releases of Eclipse JDT open source project (2.0, 2.1, 3.0, 3.1 and 3.2), systematically collected by the BuCo Analyzer tool [Mausa and Grbac 2016]. The datasets contain 50 different software metrics and the number of defects for each java source code file in the project. The full list of metrics and their descriptions may be found in [Mauša and Grbac 2017]. The software metrics that are computed describe static attributes like size, complexity, various object-oriented principles, design characteristics, programming style and more. The datasets are also available on-line for the whole research community[1].

4.2 Comparison results

Table II presents full list of metrics from software defect prediction datasets and how well they scored in four different techniques used to analyze their importance for defect prediction. The correlation analysis computed the Spearman correlation coefficients (r_s), the BPSO algorithm returned the percentage of including a metric (Rate), Logistic regression returned the percante of finding a metric significant for prediction of defects (Rate) and Bender Method yielded the geometric mean accuracy value of how successful a metric's threshold is for finding defects (GM). There is no rule on how to interpret the inclusion rate given by BPSO and Logistic Regression, nor GM value of threshold values calculated by Bender method. In order to have a common baseline of comparison, the standard interpretation levels of correlation analysis, given in Table I, were used for all four approaches. This kind of interpretation seams understandable for interpreting the Rate given by BPSO and Logistic regression and it is in accordance with the interpretation of GM values obtained by Bender method. The interpretation is given in the right sub-column of each technique.

5. DISCUSSION

There are only a few metrics for which all techniques agreed about the level of their relationship with defects. For example, LCOM and cohesion (COH) exhibit low or very low relationship while efferent coupling (FOUT) and specialization index (SIX) exhibit low or very low relationship for all techniques besides the BPSO. On the other hand, a greater number of metrics have been found to have moderate or strong relationship, like: AVCC, HBUG, HEFF, UWCS, RFC, LMC, HVOL, EXT, TCC, NCO and CCOMHLTH. The dissagreement between the techniques is present for metrics like INST, PACK, CBO, MI, CCML, NLOC, F_IN, R_R and HIER where correlation analysis and BPSO indicate lower level whilst Logistic regression and Bender method indicate stronger levels. The disagreement in the opposite direction is present for metrics like S_R, NSUB, where correlation analysis and BPSO found a stronger level of relationship than the other techniques. This means that the choice of the appropriate software metrics is biased by the choice of feature selection.

[1] http://www.seiplab.riteh.uniri.hr/?page_id=834

Table II. Comparison of different techniques for finding software metrics adequate for defect prediction

	Correlation Analysis		BPSO		Logistic Regression		Bender Method	
Metrics	r_s		Rate		Rate		GM	
LOC	0.58	MODERATE	35%	LOW	100%	VERY STRONG	0.66	STRONG
SLOC_P	0.58	MODERATE	55%	MODERATE	100%	VERY STRONG	0.64	STRONG
SLOC_L	0.58	MODERATE	30%	LOW	100%	VERY STRONG	0.64	STRONG
MVG	0.62	STRONG	60%	MODERATE	100%	VERY STRONG	0.62	STRONG
BLOC	0.49	MODERATE	65%	STRONG	100%	VERY STRONG	0.63	STRONG
C_SLOC	0.58	MODERATE	50%	MODERATE	100%	VERY STRONG	0.59	MODERATE
CLOC	0.40	LOW	75%	STRONG	100%	VERY STRONG	0.62	STRONG
CWORD	0.51	MODERATE	70%	STRONG	100%	VERY STRONG	0.61	STRONG
HCLOC	0.67	STRONG	70%	STRONG	4%	VERY LOW	0.00	VERY LOW
HCWORD	0.48	MODERATE	80%	STRONG	40%	LOW	0.12	VERY LOW
No_Methods	0.50	MODERATE	55%	MODERATE	100%	VERY STRONG	0.62	STRONG
LCOM	-0.11	VERY LOW	50%	MODERATE	0%	VERY LOW	0.00	VERY LOW
AVCC	0.47	MODERATE	65%	STRONG	100%	VERY STRONG	0.61	STRONG
NOS	0.53	MODERATE	35%	LOW	100%	VERY STRONG	0.63	STRONG
HBUG	0.53	MODERATE	70%	STRONG	100%	VERY STRONG	0.62	STRONG
HEFF	0.52	MODERATE	60%	MODERATE	100%	VERY STRONG	0.58	MODERATE
UWCS	0.51	MODERATE	75%	STRONG	100%	VERY STRONG	0.61	STRONG
INST	0.28	LOW	30%	LOW	100%	VERY STRONG	0.56	MODERATE
PACK	0.13	VERY LOW	60%	MODERATE	100%	VERY STRONG	0.60	MODERATE
RFC	0.49	MODERATE	50%	MODERATE	100%	VERY STRONG	0.62	STRONG
CBO	0.10	VERY LOW	50%	MODERATE	100%	VERY STRONG	0.48	MODERATE
MI	-0.20	VERY LOW	30%	LOW	100%	VERY STRONG	0.41	MODERATE
CCML	0.45	MODERATE	40%	LOW	100%	VERY STRONG	0.63	STRONG
NLOC	0.52	MODERATE	25%	LOW	100%	VERY STRONG	0.64	STRONG
F_IN	0.10	VERY LOW	55%	MODERATE	100%	VERY STRONG	0.47	MODERATE
DIT	-0.87	VERY STRONG	45%	MODERATE	23%	LOW	0.03	VERY LOW
MINC	-0.49	MODERATE	60%	MODERATE	100%	VERY STRONG	0.35	LOW
S_R	-0.50	MODERATE	65%	STRONG	24%	LOW	0.03	VERY LOW
R_R	0.24	LOW	50%	MODERATE	100%	VERY STRONG	0.56	MODERATE
COH	-0.29	LOW	45%	MODERATE	29%	LOW	0.13	VERY LOW
LMC	0.54	MODERATE	55%	MODERATE	100%	VERY STRONG	0.61	STRONG
LCOM2	0.42	MODERATE	40%	LOW	99%	VERY STRONG	0.45	MODERATE
MAXCC	0.54	MODERATE	35%	LOW	100%	VERY STRONG	0.64	STRONG
HVOL	0.52	MODERATE	50%	MODERATE	100%	VERY STRONG	0.62	STRONG
HIER	0.38	LOW	45%	MODERATE	100%	VERY STRONG	0.63	STRONG
NQU	0.21	LOW	55%	MODERATE	100%	VERY STRONG	0.54	MODERATE
FOUT	0.10	VERY LOW	70%	STRONG	24%	LOW	0.04	VERY LOW
SIX	-0.30	LOW	55%	MODERATE	27%	LOW	0.04	VERY LOW
EXT	0.53	MODERATE	60%	MODERATE	100%	VERY STRONG	0.42	MODERATE
NSUP	0.19	VERY LOW	70%	STRONG	100%	VERY STRONG	0.56	MODERATE
TCC	0.57	MODERATE	60%	MODERATE	100%	VERY STRONG	0.63	STRONG
NSUB	-0.50	MODERATE	35%	LOW	36%	LOW	0.05	VERY LOW
MPC	0.53	MODERATE	40%	LOW	100%	VERY STRONG	0.42	MODERATE
NCO	0.51	MODERATE	50%	MODERATE	100%	VERY STRONG	0.62	STRONG
INTR	-0.07	VERY LOW	80%	STRONG	100%	VERY STRONG	0.48	MODERATE
CCOM	0.53	MODERATE	55%	MODERATE	100%	VERY STRONG	0.65	STRONG
HLTH	0.53	MODERATE	45%	MODERATE	100%	VERY STRONG	0.63	STRONG
MOD	0.10	VERY LOW	55%	MODERATE	97%	VERY STRONG	0.28	LOW

Further analysis of the obtained results have shown that the relationship of certain metrics and defects exhibits uniform level of strength within different datasets. The correlations analysis is a more strict technique, rarely indicating strong and never indicating very strong relationship. Hence, the moderate level of relationship indicated by the correlation analysis has a greater weight that the same level of relationship indicated by the other techniques. The only two metrics that have strong relationship according to the correlation analysis are McCabe complexity (MVG), header comments (HCLOC) and depth of inheritance tree (DIT). It is interesting to notice that HCLOC and DIT exhibit such relationship when analyzed by BPSO and correlation analysis, but opposite levels of strength when analyzed by Logistic regression or Bender method and the agreement of all four techniques is achieved only for MVG.

Unlike the correlation analysis, the uniform Logistic regression technique is a less strict one and it indicated very strong level of relationship for most metrics. The Bender method, which is based upon the Logistic regression, and BPSO are moderately strict techniques. The distribution of the levels of relationship found by these two techniques is closer to a normal distribution than for the other two techniques.

The validity of this small scale case study is strongly affected by the choice of data. This is a first preliminary study, so it is based only on a sample of data from five consecutive releases of an highly object-oriented open source project from Eclipse community. The datasets are chosen from big and complex software projects to resemble as much as possible the industrial projects to cope with construct validity. The external validity is clearly threatened and general conclusions cannot be drawn yet. The aim of this case study was more to open new research questions and motivate the research direction of future work. Projects from different background, like different communities, development methodologies or written in different programming language need to be included to obtain more general conclusions.

The four chosen techniques which were used to analyze the relationship between software metrics and defects are a threat to internal validity. There exists a number of other techniques of similar purpose, but these are chosen because correlation analysis and logistic regression are the well known and widely used ones, Bender method is a rarely used technique with completely different aim (finding threshold levels) and BPSO is a novel hybrid technique for feature selection. At this stage, the conclusions about the relationship levels between analyzed metrics and defects still lack a precise explanation, hence threatening the conclusion validity and the their causality is unknown and open to speculations. That is why we believe this project to be an important one, and giving precise answers to our research objectives may bring significant improvement in understanding the role of different software metrics and improving the software quality assurance activities.

6. CONCLUSION

The algorithms that implement the proposed methodology have been applied to 50 different software metrics that are present in the datasets. The moderate level of relationship is an indication enough that a metric may improve the classification of defective software modules. The metrics for which at least one technique found a stronger level of relationship is thus a potential indicator of code smell. However, this may be misleading since different techniques find different metrics to be important. Hence, it is important to continue this research direction and find stronger evidence to precisely defining software metrics and their threshold levels that are good indicators of code smell. If such metrics are to be found, it would be possible to pay more attention to them in the whole life cycle of software development and reduce the possibility of code smell turning into defects.

REFERENCES

Ömer Faruk Arar and Kürşat Ayan. 2016. Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies. *Expert Systems with Applications* 61 (2016), 106–121.

Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. 1996. A Validation of Object-Oriented Design Metrics As Quality Indicators. *IEEE Trans. Softw. Eng.* 22, 10 (Oct. 1996), 751–761.

Ralf Bender. 1999. Quantitative risk assessment in epidemiological studies investigating threshold effects. *Biometrical Journal: Journal of Mathematical Methods in Biosciences* 41, 3 (1999), 305–319.

Lucija Brezočnik, Iztok Fister, and Vili Podgorelec. 2018. Swarm Intelligence Algorithms for Feature Selection: A Review. *Applied Sciences* 8, 9 (2018). DOI:http://dx.doi.org/10.3390/app8091521

Brezočnik, Lucija. 2017. Feature Selection for Classification Using Particle Swarm Optimization. In *17th IEEE International Conference on Smart Technologies (IEEE EUROCON 2017)*. IEEE, 966–971.

Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. 2000. Exploring the Relationship Between Design Measures and Software Quality in Object-oriented Systems. *J. Syst. Softw.* 51, 3 (May 2000), 245–273.

Ralph D'Agostino and Egon S. Pearson. 1973. Tests for Departure from Normality. Empirical Results for the Distributions of b_2 and $\sqrt{b_1}$. *Biometrika* 60, 3 (1973), 613–622.

James D. Evans. 1996. *Straightforward Statistics for the Behavioral Sciences*. Brooks/Cole Publishing Company.

Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

M. Gradišnik, T. Beranič, S. Karakatič, and G. Mauša. 2019a. Adapting God Class Thresholds for Software Defect Prediction: A Case Study. In *Proceedings of Mipro 2019 SSE*. IEEE, 1–6.

M. Gradišnik, S. Karakatič, G. Mauša, T. Beranič, and M. Heričko. 2019b. Možnosti vpeljave umetne inteligence v proces razvoja programske opreme. In *Proceedings of DSI 2019*. 1–6.

Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. Robust prediction of fault-proneness by random forests. In *15th International Symposium on Software Reliability Engineering*. IEEE, 417–428.

Brahim Hamadicharef, Cuntai Guan, Emmanuel Ifeachor, Nigel Hudson, and Sunil Wimalaratna. 2008. Performance evaluation and fusion of methods for early detection of Alzheimer Disease. In *2008 International Conference on BioMedical Engineering and Informatics*, Vol. 1. IEEE, 347–351.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference and prediction* (2 ed.). Springer.

J Kennedy. 1997. A discrete binary version of the particle swarm algorithm. In *Proc. IEEE Int. Conf. Syst. Man Cybern.*, Vol. 5. IEEE, Orlando, FL, USA, 4104–4108.

A Lehamn, N O'Rourke, and L Stepanski. 2005. *JMP for Basic Univariate and Multivariate Statistics: A Step-by-step Guide*. SAS Institute, Cary, NC.

Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Softw. Eng.* 34, 4 (July 2008), 485–496.

Goran Mausa and Tihana Galinac Grbac. 2016. Assessing the Impact of Untraceable Bugs on the Quality of Software Defect Prediction Datasets. In *Proceedings of SQAMIA 2016*. 47–56.

Goran Mauša and Tihana Galinac Grbac. 2017. Co-evolutionary Multi-Population Genetic Programming for Classification in Software Defect Prediction: an Empirical Case Study. *Applied Soft Computing* 55 (2017), 331 – 351.

Jerome L. Myers and Arnold D. Well. 1991. *Research Design and Statistical Analysis*. HarperCollins.

Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (06 Oct 2017), 7.

Grégory Seront, Miguel Lopez, Valérie Paulus, and Naji Habra. 2005. On the relationship between Cyclomatic Complexity and the Degree of Object Orientation. In *Proc. of QAOOSE Workshop, ECOOP, Glasgow*. 109–117.

R. Shatnawi. 2010. A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems. *IEEE Transactions on Software Engineering* 36, 2 (March 2010), 216–225.

Barbara G Tabachnick, Linda S Fidell, and Jodie B Ullman. 2007. *Using multivariate statistics*. Vol. 5. Pearson Boston, MA. 437–505 pages.

Santiago Vidal, Iñaki berra, Santiago Zulliani, Claudia Marcos, and J. Andrés Díaz Pace. 2018. Assessing the Refactoring of Brain Methods. *ACM Trans. Softw. Eng. Methodol.* 27, 1, Article 2 (April 2018), 43 pages.

Yuming Zhou and Hareton Leung. 2006. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on software engineering* 32, 10 (2006), 771–789.