Improving Code Smell Predictions in Continuous Integration by Differentiating Organic from Cumulative Measures

Md Abdullah Al Mamun, Miroslaw Staron
Christian Berger, Regina Hebig
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
Email: abdullah.mamun@chalmers.se,
[miroslaw.staron, christian.berger,
regina.hebig]@cse.gu.se

Jörgen Hansson
School of Informatics
University of Skövde
Skövde, Sweden
Email: jorgen.hansson@his.se

***Abstract*—Continuous integration and deployment are enablers of quick innovation cycles of software and systems through incremental releases of a product within short periods of time. If software qualities can be predicted for the next release, quality managers can plan ahead with resource allocation for concerning issues. Cumulative metrics are observed to have much higher correlation coefficients compared to non-cumulative metrics. Given the difference in correlation coefficients of cumulative and non-cumulative metrics, this study investigates the difference between metrics of these two categories concerning the correctness of predicting code smell which is internal software quality. This study considers 12 metrics from each measurement category, and 35 code smells collected from 36,217 software revisions (commits) of 242 open source Java projects. We build 8,190 predictive models and evaluate them to determine how measurement categories of predictors and targets affect model accuracies predicting code smells. To further validate our approach, we compared our results with Principal Component Analysis (PCA), a statistical procedure for dimensionality reduction. Results of the study show that within the context of continuous integration, non-cumulative metrics as predictors build better predictive models with respect to model accuracy compared to cumulative metrics. When the results are compared with models built from extracted PCA components, we found better results using our approach.**

Keywords–Software metrics; code smells; effects of measurement types; cumulative metrics; organic metrics; random forest; training-test-split cross-validation; time-series cross-validation; principal component analysis; interactions.

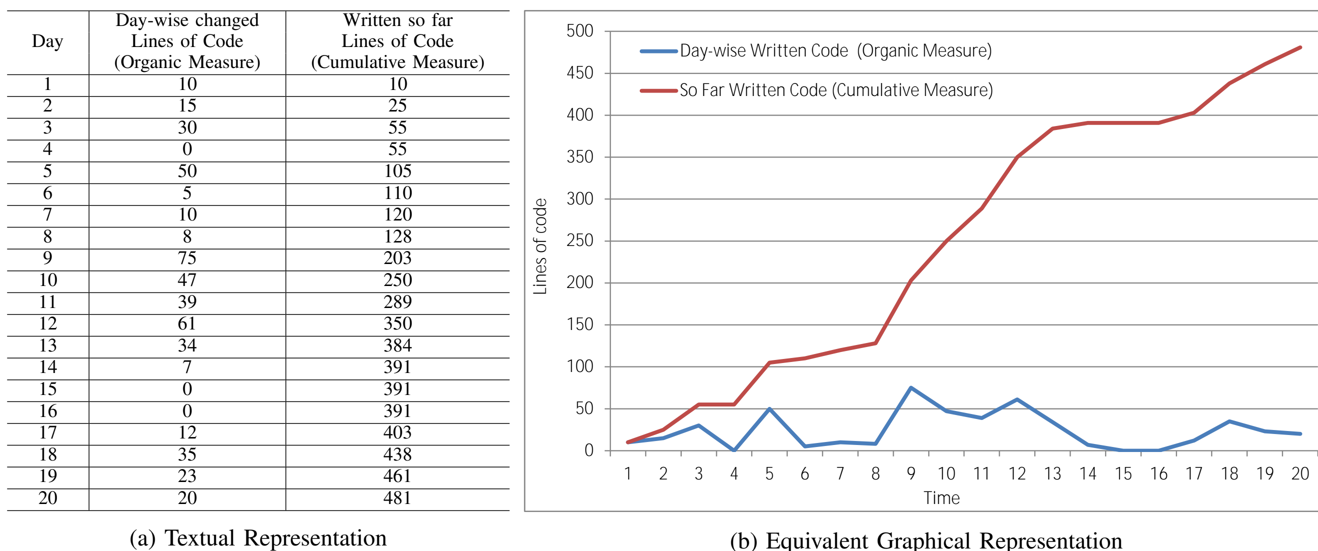
I. INTRODUCTION

Continuous integration and deployment shorten the release cycles and speed up the innovation cycles [1]. Fortunately, Agile software development can support continuous integration and deployment through sprints, time-boxes of one to four weeks, during which a releasable product increment is created [2]. As the release cycles are becoming shorter, it would be helpful for quality managers if they can predict internal and external software quality within a short span of time. For example, if quality managers have tools to predict the maintainability of code base at the end of the current sprint, they can already start allocating hours on the maintainability issue or plan for the next sprint. With the introduction of modern version control systems, we can address the problem of *how to predict internal quality changes between different revisions of software code base?* A version control system stores every revision or commit of a project; such commit-level data are much fine-grained with the possibility to reflect actual software development within a short span of time. Now we ask the question: can such refined-level data explain or predict software qualities at a short time span better compared to the traditionally counted cumulative way of measures?

Software metrics can be measured in various ways, or they have different measurement types. By organic or non-cumulative metrics, we refer to delta measures or code churn measures. If we take the example of Lines Of Code (LOC), the organic measure of LOC for a software revision or commit is the actual number of LOC written since the previous commit. On the other hand, cumulative measurement is the most commonly used technique in software engineering. The cumulative-way of measuring LOC for a commit would be to sum the organic measure of LOC for a specific commit with the cumulative measure of LOC from the previous commit. Therefore, cumulative measures develop as a moving sum. If ten new LOC are added for a commit, the organic measure of LOC for that specific commit would be ten, reflecting the actual change of LOC. However, when measured cumulatively, the value of LOC should be ten plus the total LOC from the previous commit. For each cumulative metric, a corresponding organic metric can be constructed. These measurement types are explained in Section II. A recent study [3] exclusively targeting cumulative and organic measures confirms that correlations of cumulative metrics are much higher than their corresponding organic metrics. Since that study [3] is still in the final minor review round, we are reporting the relevant part of the results in Table I. In addition, in a previous study [4], we had indications that correlations between cumulative metrics are higher than their corresponding organic metrics.

High correlation coefficients between metrics imply high collinearity between them. If the correlation coefficient is as high as 0.9, it can be considered as very strong [5]. Thus, corresponding metrics can be considered collinear meaning they are redundant. Non-collinearity is mentioned as a validation criterion for software metrics [6]. However, some methods for predictive analyses, such as multiple linear regression, rely on the assumption that input features are non-collinear. Thus, it is to be expected that combinations of cumulative metrics (which have higher collinearity) are less fit to be used in predictive models compared to their corresponding organic metrics. However, to our knowledge, to this day no studies have investigated whether organic metrics can lead to better predictions of aspects, such as code smells or bugs compared to their corresponding cumulative metrics. Therefore, we want

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-701-6



Day	Day-wise changed Lines of Code (Organic Measure)	Written so far Lines of Code (Cumulative Measure)
1	10	10
2	15	25
3	30	55
4	0	55
5	50	105
6	5	110
7	10	120
8	8	128
9	75	203
10	47	250
11	39	289
12	61	350
13	34	384
14	7	391
15	0	391
16	0	391
17	12	403
18	35	438
19	23	461
20	20	481

(a) Textual Representation

(b) Equivalent Graphical Representation

Figure 1. Organic and Cumulative Measures.

to investigate this. Code smells are internal software qualities that are associated with code maintainability [7] [8] and using code smells has significantly less internal validity threat in the context of this research because unlike the bugs, we can precisely determine which code smells are originated from which revisions. Thus, code smell is a good candidate as a target for this study. Note that we are using code smell as an interesting example to validate whether there is any difference between the two measurement categories, our objective is not to build the best predictive model for code smells. This study has the following research question:

- RQ: What is the difference between organic and cumulative measures with respect to the correctness of predicting code smells within the context of continuous integration?

In this paper, we analyzed 36,217 revisions of 242 open source Java projects to mine various measures and code smells. Our results show that measurement types of metrics have a significant impact on model accuracies. The findings will help to build predictive models specifically targeted to predict software artifacts within a short period of time. Our findings are also significant to clearly understand the difference between cumulative and organic measures of software artifacts.

Section II of this paper describes the ideas of cumulative and organic measures, followed by related work in Section. III. The design of this study including project selection (IV-A), data collection (IV-B), analysis procedure (IV-C) and threats to validity (IV-D) are discussed in Section IV.

II. Cumulative and Organic Measures

In this section, we explain cumulative and organic measures with a simple example, as illustrated in Fig. 1. The figure illustrates the change of the measure lines of code of a fictional example system developed over 20 days with daily commits. Considering LOC as a measure, the actual changes in values of LOC from each day compared to the day before reflects the organic measurement of LOC. However, if we calculate the total LOC written up to a day by summing up the changes of LOC from day one up to that day, we get the cumulative measurement of LOC. In this example, we have considered "daily commits" or "day" as a unit of time, but it can also be week, sprint, or release.

Correlation of various size and complexity metrics of type cumulative result in high coefficients, which is observed by many studies [9]–[13]. Correlations between organic metrics are significantly lower than their corresponding cumulative metrics. We have observed this phenomenon in earlier research [4] and in a recent study (in submission) specific to this topic, the result of which is presented in Table I. The reported correlation coefficients (τ_b) in Table I are mean values of Kendall τ_b from 11,874 software revisions of 21 open source Java projects. In Table I, if a τ_b value is greater than or equal to 0.9 (i.e., with very strong correlation coefficient), we can consider the corresponding two metrics of the τ_b as collinear, meaning they are redundant and either of them can represent the other. Because their r^2 (coefficient of determination) becomes minimum 0.81, meaning one metric can at minimum explain 81% variability in the other. Based on this, eight metrics out of 12 metrics in the cumulative category in Table I become redundant. On the other hand, from the corresponding organic metrics, we do not see a single pair of metrics for which the correlation coefficient is greater than 0.9. Thus, we can expect that combinations of organic metrics can bring added value to predict code smells. This observation is one of the key motivations for this study where we want to see how these two groups of measures, i.e., cumulative and organic, perform predicting code smells.

III. Related Work

After Fowler et al. [15] first introduced the term bad code smells, there have been many studies investigating this subject. Zhang et al. [16] report on a systematic literature review on code smells and find that most of the studies in this area are focused on the identification or detection of code smells. Automated detection of code smells has become an

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-701-6

TABLE I. KENDALL'S τ [14] CORRELATION COEFFICIENTS (τ_b) FOR CUMULATIVE AND ORGANIC METRICS.
The whole range of τ_b ($-0.1 \leqslant \tau_b \leqslant +1.0$) is labeled into four levels according to strength. Three gray-scale cell colors indicate three levels of τ_b (Very Strong: $0.9 \leqslant abs(\tau_b) \leqslant +1.0$, Strong: $0.7 \leqslant abs(\tau_b) < 0.9$, and Moderate: $0.4 \leqslant abs(\tau_b) < 0.7$). Weak τ_b ($0 \leqslant abs(\tau_b) < 0.9$) is indicated with red font color. A dot (.) in a cell indicates zero value. All cells in the diagonal represent correlation of a metric with itself (always having τ_b = 1) are left blank.

		Cumulative													Organic												
		ncloc	functions	statements	complexity	classes	files	public_api	pub_undoc_api	comment_ln	directories	dup_lines	dup_blocks	dup_files	_ncloc	_functions	_statements	_complexity	_classes	_files	_public_api	_pub_undoc_api	_comment_ln	_directories	_dup_lines	_dup_blocks	_dup_files
Cumulative	ncloc		.97	.98	.97	.93	.93	.94	.87	.86	.71	.63	.64	.64	.03	.	.01	.01	.	.	.01	.02	.01	.	.02	.02	.01
	functions	.97		.96	.98	.93	.92	.94	.88	.86	.71	.64	.65	.65	.03	.01	.02	.01	.01	.	.01	.02	.01	.	.02	.02	.01
	statements	.98	.96		.97	.92	.91	.94	.87	.86	.7	.63	.65	.64	.03	.	.02	.01	.	.	.01	.02	.02	.	.02	.02	.01
	complexity	.97	.98	.97		.92	.91	.93	.87	.86	.71	.63	.65	.65	.03	.	.02	.01	.	.	.01	.02	.01	.	.02	.02	.01
	classes	.93	.93	.92	.92		.98	.92	.86	.84	.73	.63	.63	.64	.03	.	.02	.01	.01	.01	.02	.02	.01	.	.02	.02	.01
	files	.93	.92	.91	.91	.98		.91	.85	.84	.73	.62	.63	.63	.03	.	.02	.02	.01	.01	.03	.03	.01	.	.02	.02	.01
	public_api	.94	.94	.94	.93	.92	.91		.91	.85	.7	.64	.64	.64	.01	.01	.01	.01	.	.	.03	.02	.02	.	.02	.01	.01
	pub_undoc_api	.87	.88	.87	.87	.86	.85	.91		.77	.66	.62	.61	.62	.01	.	.	.01	-.01	.	.02	.02	.	.01	.	.01	.
	comment_ln	.86	.86	.86	.86	.84	.84	.85	.77		.71	.55	.56	.59	.01	.	.01	.01	.	.	.02	.01	.02	.	.02	.02	.02
	directories	.71	.71	.7	.71	.73	.73	.7	.66	.71		.57	.59	.62	.01	-.01	.01	.01	-.01	.01	.	.01	.	.01	.02	.02	.02
	dup_lines	.63	.64	.63	.63	.63	.62	.64	.62	.55	.57		.86	.84	.02	.01	.02	.02	-.01	.	.01	.01	.02	.01	.03	.03	.02
	dup_blocks	.64	.65	.65	.65	.63	.63	.64	.61	.56	.59	.86		.85	.02	.01	.02	.02	.	.	.01	.01	.02	.	.03	.03	.02
	dup_files	.64	.65	.64	.65	.64	.63	.64	.62	.59	.62	.84	.85		.02	.	.02	.03	.	.01	.02	.01	.02	.01	.03	.03	.04
Organic	_ncloc	.03	.03	.03	.03	.03	.03	.01	.01	.01	.01	.02	.02	.02		.75	.89	.82	.55	.52	.68	.64	.49	.24	.25	.25	.26
	_functions	.	.01	.	.	.	.	.01	.	.	-.01	.01	.01	.	.75		.75	.81	.65	.59	.82	.76	.54	.26	.26	.28	.28
	_statements	.01	.02	.02	.02	.02	.02	.01	.	.01	.01	.02	.02	.02	.89	.75		.85	.54	.51	.69	.65	.49	.23	.26	.27	.27
	_complexity	.01	.01	.01	.01	.01	.02	.01	.01	.01	.01	.02	.02	.03	.82	.81	.85		.56	.53	.7	.66	.52	.23	.25	.26	.27
	_classes	.	.01	.	.	.01	.01	.	-.01	.	-.01	-.01	.	.	.55	.65	.54	.56		.88	.67	.64	.46	.4	.25	.27	.29
	_files	.	.	.	.	.01	.01	.	.	.	.01	.	.	.01	.52	.59	.51	.53	.88		.65	.6	.45	.44	.26	.29	.32
	_public_api	.01	.01	.01	.01	.02	.03	.03	.02	.02	.	.01	.01	.02	.68	.82	.69	.7	.67	.65		.89	.51	.31	.24	.26	.26
	_pub_undoc_api	.02	.02	.02	.02	.02	.03	.02	.02	.01	.01	.01	.01	.01	.64	.76	.65	.66	.64	.6	.89		.4	.29	.24	.27	.26
	_comment_ln	.01	.01	.02	.01	.01	.01	.02	.	.02	.	.02	.02	.02	.49	.54	.49	.52	.46	.45	.51	.4		.25	.23	.22	.23
	_directories	.	.	.	.	.	.	.	.01	.	.01	.01	.	.01	.24	.26	.23	.23	.4	.44	.31	.29	.25		.14	.15	.17
	_dup_lines	.02	.02	.02	.02	.02	.02	.02	.	.02	.02	.03	.03	.03	.25	.26	.26	.25	.25	.26	.24	.24	.23	.14		.84	.75
	_dup_blocks	.02	.02	.02	.02	.02	.02	.01	.01	.02	.02	.03	.03	.03	.25	.28	.27	.26	.27	.29	.26	.27	.22	.15	.84		.81
	_dup_files	.01	.01	.01	.01	.01	.01	.01	.	.02	.02	.02	.02	.04	.26	.28	.27	.27	.29	.32	.26	.26	.23	.17	.75	.81	

integral part of many static analysis tools, e.g., SonarQube [17], SpotBugs [18], Jtest [19], JArchitect [20], PMD [21], etc. In a proposed method to identify two code smells (lazy class and temporary field), Munro [22] used five code metrics (LOC, number of methods, weighted methods per class, coupling, and depth of inheritance tree). He used a programmatic approach, meaning studying the characteristics of the code smells he devised rules using the metrics to identify the code smells. Fontana et al. [23] performed an empirical investigation to code smells detection and how frequent certain code smells are in various application domains. They also investigated the Spearman ranks correlations between software metrics and code smells. In an extensive study, Fontana et al. [24] experimented with 16 machine learning algorithms to detect four code smells. They worked on 74 software systems with 1986 validated code smells and found Random Forest and J48 as the best performing algorithms. While some of these studies have used code metrics to detect code smells, we are using code metrics to predict code smells in the future software revisions.

Few studies have focused on the impact of code smells. Monden et al. [7] performed a quantitative study on a legacy system to assess the impact of duplicated code smells on software reliability and maintainability. Yamashita and Moonen [25] identified various factors that affect software maintainability and investigated to what extent code smells reflect those factors. Kapser and Godfrey [26] also studied duplication on software quality. Other studies [27], [28] attempted to reveal relations between code smells and software faults. Another study [29] investigated whether source code files with code smells are more prone to change and found that classes with code smells are more change-prone.

Maneerat and Muenchaisri [30] possibly made the first attempt to predict code smells. They used seven machine learning algorithms to predict seven code smells. However, they used K-folds cross-validation, which is inappropriate for time-series data. According to our experience, the high accuracy of their predictive models is due to the wrong choice of cross-validation technique. When we evaluated our models using K-folds, we get model accuracies greater than 90%. However, we have avoided it as K-folds is methodologically wrong to validate time-series data. A recent study by Gupta et al. [31] build entropy based statistical model to predict six code smells on different versions of Apache Abdera project. Among the three selected entropies, Shannon performed best with a r^2 value 0.567. This study differs from their research in different

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-701-6

ways. First, we have taken a machine learning based approach. Seconds, we are focusing on code smells prediction in the continuous integration environment where the release cycle is very short. Their measure of code smells is cumulative whereas we focus on only new code smells that might be added to the future revisions. Moreover, we have a focus on the difference between cumulative and organic predictors, and our study is based on a large number of randomly selected projects.

IV. METHODOLOGY

This empirical study is designed as a case study. We follow the compiled guideline of software engineering case studies by Runeson and Höst [32]. Research design related terminologies used in this study is also adopted from the same guideline [32]. This case study is “explanatory” according to the classification of Robson [33] which Runeson and Höst interpreted as similar to the type “confirmatory” by Easterbrook et al. [34]. Collected data of this case study is quantitative, and the design of the study is more fixed than flexible, meaning we have a defined set of measurement categories, independent and dependent measures, machine learning algorithm, and cross-validation techniques that we are particularly interested in answering the research question. Triangulation is essential because it increases the precision of empirical research. For studies with quantitative data, triangulation is important as it can compensate for measurement or modeling errors [32]. For triangulation, we have considered data source triangulation [35], meaning, more than one data source or project is used in this case study. Runeson and Höst mentioned three major research methods that are related to case studies and experiment is one of them. Since this study is involved with quantitative data, it has some overlap with experiments, e.g., we have identified independent and dependent variables, and have carefully worked with the instrumentation. Runeson and Höst [32] have also mentioned that quasi-experiments have many characteristics that are common with case studies. Quasi-experiments and controlled experiments are similar except that in quasi-experiment subjects are not randomly assigned to treatments. However, since we have randomly selected the projects and exhaustively created all possible models, like a full factorial design, this study is more of a controlled experiment than a quasi-experiment.

A. Project Selection

Open source software projects on GitHub serve as the data source for this study. There are millions of Java projects on GitHub. GitHub provides REST API for users through which meta-data about projects can be collected. However, it is very limited considering the massive number of projects. Therefore, we have used the GHTorrent [36], a project that gathers meta-data of publicly hosted projects on GitHub. We downloaded GHTorrent’s database dumps of size about 300GB and extracted on a local MySQL database because, GHTorrent’s free online database queries have limitations.

We initially selected 2,188,033 candidate GitHub Java projects from GHTorrent’s database. We selected them in such a way that there are no forked projects to avoid partial duplications. We also exclude projects that are marked as deleted by GHTorrent. Fig. 2 shows the distribution of the 145,980 projects from the 2.2 million candidate projects between 50-500 commits. Since the actual distribution consisting all the projects is exponential, projects with commits less than 50 are not shown for better visual presentation. About 1.3 million of the 2.2 million projects have 5 or fewer commits.

We have randomly selected 1000 projects from the 2.2 million candidate projects. Among them, we found 232 projects that have zero or one commit. In case a project has multiple commits from the same day, we consider the latest commit and ignore others, meaning one commit from a day. We have ignored any project that results in less than 10 commits. Because when computing correlations between metrics using Kendall’s τ, the minimum sample size should be 10, which still has some bias and for an unbiased result the sample size should be 50 [37]. We also found some projects that were no more publicly accessible or lack the GIT master branch, which we ignored. Finally, we have 242 projects that are considered for this study.

B. Data Collection

Software revisions in the Git version control system is a form of archival data, which Lethbridge et al. [39] described as a third-degree technique for data collection. In our case, the source of data is of third-degree. However, we have used the SonarQube [17] tool to process the archival data and generate measures of our interest. For data collection, we have considered the entire commits or revision history in the master branch of a project’s Git repository. However, if there are multiple commits from a single day, we analyzed the latest commit of a day using SonarQube. Therefore, for a project, we have collected data from everyday that has at least a single commit.

Table II shows metrics used for this study. Descriptions of these metrics are taken from the SonarQube’s database and metric definition page [40]. Of them, cumulative metrics are measured by the SonarQube tool, and the organic metrics are calculated from the difference of two consecutive values of the cumulative metrics. In this paper, we refer to an organic version of a cumulative metric, by adding an underscore sign _ at the start of the metric name, e.g., the organic form of the cumulative metric *statements* is *_statements*.

SonarQube’s Java plugin has more than 300 code smells classified into different categories. We have skipped all code smells from minor and info categories because these code smells are less severe and the probability of worst things

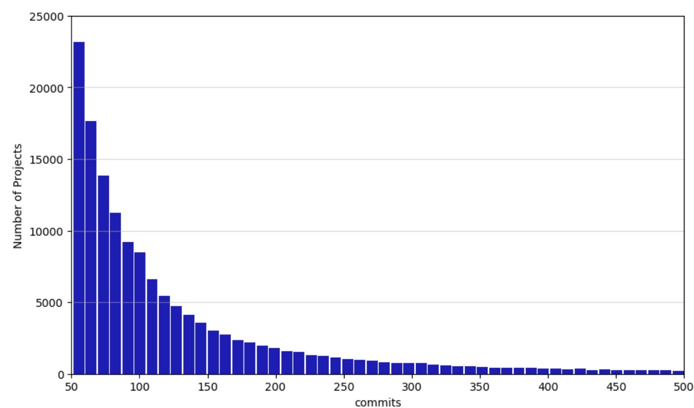


Figure 2. Histogram of candidate GitHub projects from 50 to 500 commits.

Copyright (c) IARIA, 2019.

TABLE II. COLLECTED CODE METRICS. ALL THE 12 METRICS IN THE INDEPENDENT SECTION ARE CUMULATIVE. WE ALSO HAVE CORRESPONDING ORGANIC METRICS OF THESE 12 CUMULATIVE METRICS. THIS PAPER INDICATES AN ORGANIC METRIC WITH A _ SIGN AT THE BEGINNING OF THE METRIC NAME. INTEGER IS THE DATA TYPE OF ALL THESE METRICS.

Variable Type	Metric Name	Description	Short Name
Independent	ncloc	Number of physical lines of code that are not comments (line only containing space, tab, and carriage return are ignored)	ncloc
	classes	Number of classes (including nested classes, interfaces, enums, and annotations)	cls
	files	Number of files	fil
	directories	Number of directories	dir
	functions	Number of methods	func
	statements	Number of statements according to Java language specifications	stmt
	comment_lines	Number of lines containing either comment or commented-out code (Empty comment lines and comment lines containing only special characters are ignored)	com_ln
	cognitive_complexity	A complexity measure of understandability of code [38]	cgn_cmplx
	complexity	Cyclomatic complexity (else, default, and finally keywords are ignored)	cmplx
	duplicated_lines	Number of duplicated lines	dp_ln
	duplicated_blocks	Number of duplicated blocks. To count a block, at least 10 successive duplicated statements are needed. Indentation & string literals are ignored	dp_blk
	duplicated_files	Number of duplicated files	dp_fil
Dependant	_code_smells	Organic measure of identified code smells (named new_code_smells by SonarQube) (Total count of code smells identified for the first time since the last analyzed commit)	_cs

happening due to such code smells is low. We have also ignored code smells that seem to have an obvious linear relation with the single predictors, e.g., a code smell reporting cases when the complexity of a class or method reaches a certain limit, has a high linear relation with the complexity measures. We reviewed the rest of the code smells and selected 35 code smells from the "blocker" and "critical" categories, as listed in Table V in the Appendix.

SonarQube calculates the metric *new_code_smells* (which we have denoted as *_code_smells*) and saves the measure into the database but automatically deletes measures from the earlier runs, if there is any. We have instrumented the database with triggers to automatically retrieve this metric when deleted. Among all metrics, *_code_smells* is used as the dependent variable and rest of the metrics in Table II are used as independent variables or predictors.

C. Analysis Procedure

In Section II, we have discussed the idea, cumulative measures have high collinearity among themselves. Therefore, there is a reason to believe that cumulative measures are collectively weaker as input features for predictive models compared to their corresponding organic measures. Thus, we are interested in investigating whether it makes a difference to use organic measures instead of their cumulative counterparts to predict *_code_smells*. Thus, when building our models, we do not combine predictors or measures from both categories. Therefore, we can denote predictors of a model either cumulative or organic since they always come from the same category because we will not mix them. On the other hand, our target variables *_code_smells* is from the measurement category organic. Therefore, we will denote *_code_smells* as organic.

For prediction, we use random forest regression for this study. Random forest is an ensemble algorithm. It consists of multiple decision trees making a forest where the accuracy of a model is calculated by averaging the accuracies of every single tree in the forest. While a decision tree algorithm generally suffers from overfitting, a random forest prevents overfitting by design. The random forest algorithm is suitable in our case as it can better handle non-normal data compared to many machine-learning algorithms. Moreover, the random forest is known for its quick training time given its effectiveness.

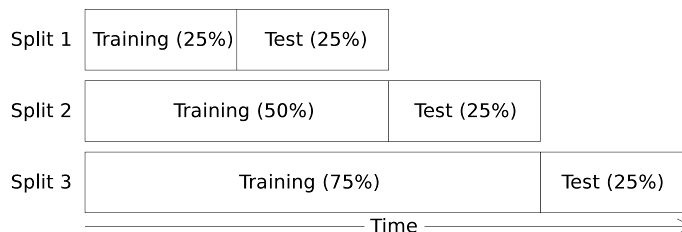


Figure 3. Split of a single dataset in different iterations in a `TimeSeriesSplit` cross-validation technique.

For validation of models, we like to use cross-validation techniques from the popular `Scikit-Learn` Python library. Since, the collected data from the revision history is time-bound, `TimeSeriesSplit` cross-validation from the `Scikit-Learn` library is appropriate in our case. We use the default number of splits (`n_splits = 3`) for our data. In `TimeSeriesSplit` cross-validation, the order of data is important, and unlike general cross-validation, randomization is avoided because, if data is randomized, the training datasets shall have a rough idea about the trend of the future already based on the random data assigned from the later parts of the projects. In our case, we are interested in predicting the future based on the available data (and there is no point in predicting the past). In `TimeSeriesSplit` cross-validation, data is split without disordering time, as shown in Fig. 3.

TABLE III. PCA COMPONENTS. HERE, WE USED SHORT NAMES FOR METRICS, AS MENTIONED IN TABLE. II.

	Cumulative Metrics											
	ncloc	func	stmt	cmplx	cgn_cmplx	cls	fil	com_ln	dir	dp_ln	dp_blk	dp_fil
PC_1	.79	.09	.32	.14	.08	.01	.01	.32	.00	.36	.01	.00
PC_2	-.38	.02	-.29	-.04	-.09	-.01	.00	.40	.00	.77	.01	.00
PC_3	-.21	-.04	.19	.02	.04	-.02	-.01	.84	.00	-.46	-.01	.00
PC_4	.41	.02	-.82	-.16	-.21	.06	.04	.18	.01	-.23	-.01	.00
PC_5	-.05	.14	-.31	.49	.79	-.01	-.02	.00	-.01	-.03	.09	.00
PC_6	.09	-.76	-.01	-.48	.43	-.03	-.03	.02	.01	.07	-.01	.01
PC_7	.00	-.30	-.04	.40	-.11	-.05	.10	-.01	.02	.01	-.85	-.02
PC_8	.02	-.52	-.05	.54	-.33	-.24	-.08	-.01	-.04	-.01	.49	.12
PC_9	-.04	-.16	.04	.13	-.03	.78	.55	.01	.14	.01	.14	.03
PC_10	.00	.08	.00	-.09	.05	-.56	.75	.00	.28	.00	.04	.16
PC_11	.00	.04	.00	-.01	.01	.10	-.30	.00	.58	.00	-.06	.74
PC_12	.00	-.04	.00	.05	-.03	-.03	-.16	.00	.75	.00	.06	-.64

	Organic Metrics											
	_ncloc	_func	_stmt	_cmplx	_cgn_cmplx	_cls	_fil	_com_ln	_dir	_dp_ln	_dp_blk	_dp_fil
PC_1	-.89	-.14	-.31	-.16	-.07	-.02	-.02	-.24	.00	-.09	.00	.00
PC_2	-.20	.08	-.01	.01	-.09	-.01	.01	.42	.01	.88	.02	.01
PC_3	-.24	.25	.04	.14	-.07	-.01	.01	.80	.00	-.47	-.01	.00
PC_4	.33	-.02	-.85	-.25	-.27	.09	.03	.15	.00	-.03	-.01	.00
PC_5	.04	-.67	-.11	-.09	.65	-.09	-.12	.29	-.05	-.01	-.02	-.02
PC_6	.04	-.40	.40	-.63	-.50	.02	-.01	.15	.03	-.06	-.01	.00
PC_7	.05	-.26	-.06	.40	-.38	-.57	-.47	-.01	-.17	-.01	.22	.00
PC_8	.01	.42	.01	-.51	.29	-.32	-.25	-.01	-.04	.00	.55	-.02
PC_9	-.02	-.25	.00	.23	-.09	.34	.33	.02	.10	-.02	.80	.03
PC_10	.02	-.03	-.03	-.02	.01	-.66	.70	.00	.26	.00	-.04	.09
PC_11	.00	.01	.02	-.04	-.01	-.04	.34	.00	-.90	.00	.01	-.27
PC_12	.00	-.01	.00	.03	-.02	-.03	-.01	.00	.28	.01	.01	-.96

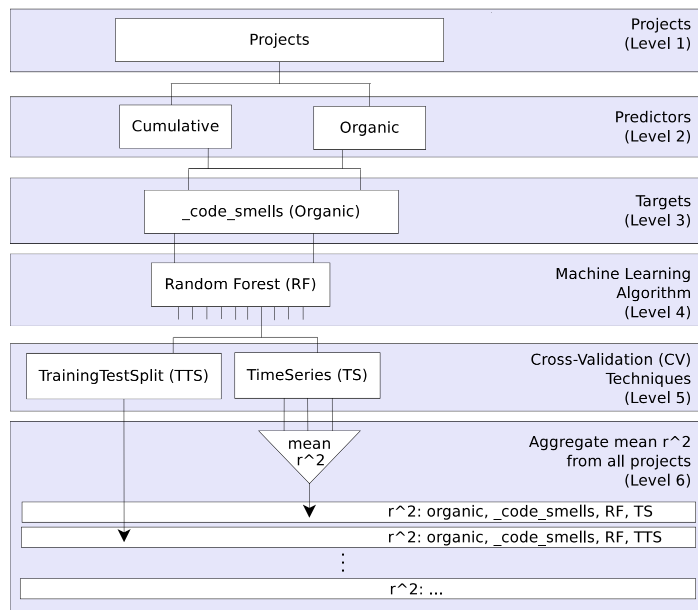


Figure 4. The overall process of model evaluation. At least one sub-level process is expanded from each horizontally highlighted level (i.e., projects, predictors, targets, etc.) Unexpanded sub-processes are indicated with open-ended vertical lines or connectors (e.g., the small vertical lines connected to `Random Forest (RF)` at `Level 4`). Any unexpanded or open-ended vertical connector contains the same expanded sub-process tree from the same level.

A simpler cross-validation technique compared to `TimeSeriesSplit` is `training_test_split`, where the dataset is split only once. For `training_test_split` cross-validation for time-series data, if we avoid randomization when splitting the training and the test sets, we get a coarse-level cross-validation compared to `TimeSeriesSplit`, which is still valid. For `training_test_split`, we will use 70% data for training set and 30% data for test set. K-folds is another popular cross-validation technique, which is similar to time-series cross-validation. However, the k-folds technique has no strict ordering of time, meaning older data corresponding to the earlier commits can be in the test sets and newer data corresponding to the recent commits can appear in the training set. Due to this, it is methodologically wrong to evaluate time-series data with k-folds cross-validation. Therefore, we have avoided it.

The overall process of model evaluation is pictured in Fig. 4. This figure shows how the selected machine learning regressor with two cross-validation methods are used to process the data from the selected projects and calculate the accuracies of regression models. Since, this study has a strong focus to understand the impact of using multiple features from different measurement categories (i.e., cumulative and organic) when predicting the target measures, we want to exhaustively check all possible combinations of the predictors while building machine learning models to predict the target measures. In `Level 2` of Fig. 4, we can select one or multiple predictors to predict the metric in `Level 3`. Then, in `Level 4`, machine learning models are built based on the selections from the prior two levels. Table IV shows all possible number of models that can be generated based on the process shown in Fig. 4, considering all possible lengths of predictors and all possible number of evaluations of such a number of total possible models.

TABLE IV. NUMBER OF MODELS AND EVALUATIONS BASED ON THE STUDY DESIGN.

'Combination length' of predictors (k)	No. of prediction models (nopm)	No. of r^2 values	No. of aggregated r^2 values
1	24	168	48
2	132	924	264
3	440	3,080	880
4	990	6,930	1,980
5	1,584	11,088	3,168
6	1,848	12,936	3,696
7	1,584	11,088	3,168
8	990	6,930	1,980
9	440	3,080	880
10	132	924	264
11	24	168	48
12	2	14	4
Total	8,190	57,330	16,380

Accuracies of the predictive models are calculated as r^2, which is also known as the coefficient of determination. It expresses the proportion of the variance in the dependent measure or target, predictable from the independent measures or predictors. The maximum value of r^2 is `+1`, which implies 100% variability of the target measure has been accounted. It

Copyright (c) IARIA, 2019.

can also be negative suggesting the model has even a worse fit than a horizontal line on the x-axis.

Within the context of this study, 'no. of predictor' (n) = 12, 'no. of category of predictors' (pc) = 2, 'no. of target' (t) = 1, 'no. of regressor' (r) = 1, and 'no. of cross-validation' (cv) = 2. This study has considered generating all possible models and all possible evaluations of the models, as shown in Table IV. Therefore, we build in total 8,190 predictive models and this numerical figure is calculated as $total_nopm = \sum_{k=1}^{12}(\binom{n}{k} \times pc \times t \times r)$. Then, we evaluate these models 57,330 times and generate the same number of r^2 values by $total_norv = \sum_{k=1}^{12}(\binom{n}{k} \times pc \times t \times r \times 4)$. In this equation, the numerical term four comes as we have one r^2 value from `training_test_split` and three r^2 values from `TimeSeriesSplit` cross-validations. Then, aggregating r^2 values for `TimeSeriesSplit`, we have the total 16,380 r^2 values calculated by $total_noarv = \sum_{k=1}^{12}(\binom{n}{k} \times pc \times t \times r \times cv)$ that will be used for results of this study. All evaluations are based on test data sets while models are built with training data sets. The whole model training and evaluation process is automated using Python scripts and MySQL database. When reporting the results, we select the best r^2 value (which means the associated predictive model is best) for each combination length of predictors (k).

To validate our model building and prediction approach, we have performed PCA on both sets of metrics and extracted the maximum possible components, i.e., the total number of metrics in each category, which is 12. The details of the extracted components are shown in Table III. For cumulative metrics, the first two components explain about 98.3% and for cumulative metrics 97.6% variability. Therefore, it would be enough to consider these two components to build models. However, as we have exhaustively built models in the first approach, we want to develop models using PCA by using all the components. For each measurement category, we will build 12 prediction models where the first model would only use the first PCA component and the last model would use all 12 components.

D. Threats to Validity

A general weakness of case study research is generalizability due to the reason that samples are not randomly selected from the population. However, this study has randomly selected the projects from the population.

According to GitHub's statistics for 2017 [41], 6.7 million new users joined the platform, of them 48% are students, 45% are entirely new to programming, and 4.1 million people created their first repository. This means, there are a lot of classroom projects or projects that have little or no code base. Here, we have a trade-off, we want to be inclusive by equally letting all projects the same chance to appear as subjects. But we do not want to include meaningless projects. As we have disregarded all projects that result in less than 10 commits, a lot of such unwanted projects was removed.

Programming languages have different constructs. Therefore, measures of code metrics may vary due to programming languages. To minimize the effects of programming languages, we have selected projects that are mainly labeled as Java projects.

We have selected 35 code smells for this study. This could be seen as a threat as not all code smells are equally severe and we have not tracked which code smells are more present than the others. To minimize this threat, we selected code smells from the top two severity-levels ("blocker" and "critical"). We have removed code smells that are subjective to projects (e.g., code smells related to coding conventions) and that seem to have obvious correlations with the input feature metrics. Since we have collectively predicted the code smells, it would not be possible to differentiate which code smells are more predictable than others. From our results, we have an overall understanding of predictability of the selected code smells as a whole. Therefore, it was important that the severity of the selected code smells do not vary much. From another point of view, since our research question is specific to the context of prediction accuracies between organic and cumulative measures, our approach of not differentiating between the selected code smells do not pose any significant threat to the validity of this study.

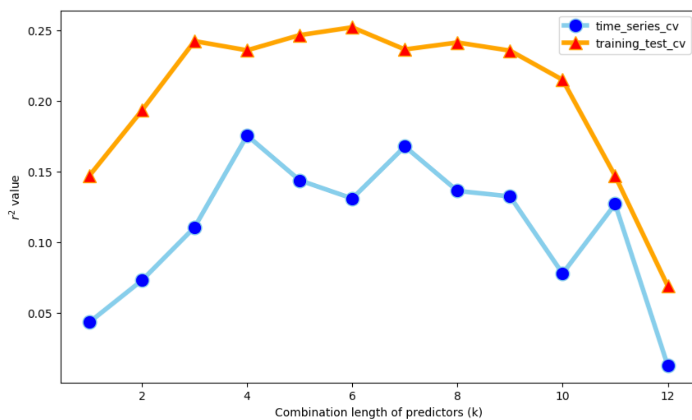
V. Results and Discussions

We are interested in distinguishing the difference between cumulative and organic measures predicting code smells in the continuous integration environment where we want to predict quality change within a short period. Results from our predictions are reported in Fig. 5. All sub-figures in this figure have 12 data points each depicting the best found r^2 value for that specific length and choice of predictor category, target, regressor, and cross-validation from the total available 16,380 r^2 values.

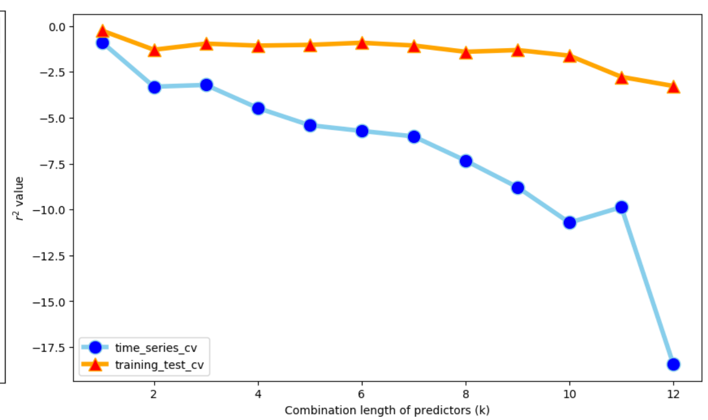
When we look at the model performance from cumulative predictors to *_code_smells* in Fig. 5b, all r^2 values for both cross-validations result in negative values. The maximum r^2 values of -0.25 (for `training_test_split`) and -0.88 (for `TimeSeriesSplit`) come from the single predictor *duplicated_files*. Since we have all negative r^2 in Fig. 5b, results from such models are not useful in practice but we can still interpret the results relatively in comparison to Fig. 5a to understand how prediction accuracies vary when predictor types are different.

For organic metrics predicting *_code_smells*, we see positive prediction accuracies in Fig. 5a. Here, both cross-validation methods yield positive results ($r^2 \geq 0$). It is clear from this sub-figure that the model accuracy increases as the number of predictor increases. In Fig. 5a for `training_test_split`, we see that the maximum r^2 value is 0.25 for a combination of six predictors; then the model accuracy gradually decreases as the number of predictor increases. The six predictors are *_ncloc*, *_statements*, *_classes*, *_comment_lines*, *_duplicated_lines*, and *_duplicated_blocks*. For `training_test_split` cross-validation, the maximum model accuracy r^2 = 0.18 is found for four predictors, which are *_ncloc*, *_directories*, *_duplicated_lines*, and *_duplicated_blocks*.

We see that `TimeSeriesSplit` cross-validation results in lower model performance compared to `training_test_split`. This is most likely because the r^2 value of `TimeSeriesSplit` comes from three r^2 values or three models where the first model is trained with only 25% of the data to evaluate 25% of the data, which is 'Split 1' in Fig. 3. Models corresponding to 'Split 1' not only



(a) Predictor: Organic metrics, Target: *_code_smells*



(b) Predictor: Cumulative metrics, Target: *_code_smells*

Figure 5. Prediction Accuracies using Random Forest Regressor.

have less portion of training data but also the split between training and test is equal. Still, the r^2 graphs corresponding to both cross-validation techniques in two sub-figures in Fig. 5 roughly follow a similar trend concerning shape or slope.

The interactions among predictors are more visible in the `TimeSeriesSplit` graphs than `training_test_split` in both sub-figures in Fig. 5. For example, in Fig. 5a, if we carefully look at the `TimeSeriesSplit` graph, we see that combination lengths 5 and 6 yield lower accuracies than combination lengths 4 and 7. Similar observations are seen for `training_test_split`, e.g., model accuracy for combination length 4 is lower than accuracies for combination lengths 3 and 5. Such observations could most likely be due to the interactions among the predictors. Interactions among predictors are interesting. However, we have not investigated them further, as it falls beyond the scope of this study.

Results from the 24 prediction models where features are extracted through PCA, are presented in Fig. 6. Since we evaluated these models only using `training_test_split` cross-validation technique, we would compare these results with `training_test_split` graphs in Fig. 5a and 5b. We see that our approach outperformed PCA in both cases. For organic metrics predicting *_code_smells*, we get the best r^2 value 0.10 when eight components are used together. However, if we had used maximum two components, we could have achieved 0.05 as the maximum accuracy which is half of 0.10. For cumulative metrics predicting *_code_smells*, all r^2 values are negative. However, the best r^2 is -1.96 compared to -0.25 in Fig. 5b. Interestingly, for PCA, we have similar observations as seen for Random Forest in Fig. 5, i.e., when we use more organic metrics to build predictive models the model accuracy increases. Six metrics for Random Forest in Fig. 5a and 8 components for PCA in Fig. 6 produce the best results.

Based on the observations, we can answer the RQ as organic metrics can better predict *_code_smells* compared to cumulative metrics. Furthermore, as more organic measures are combined to form sets of predictors, the accuracies of the models increase as seen in Fig. 5a. Compared to this, cumulative measures do not contribute to the model accuracies as observed in Fig. 5b. This could be an indication that organic measures collectively contribute more to the predictive models than their corresponding cumulative measures. This could potentially be due to the multicollinearity of the cumulative metrics.

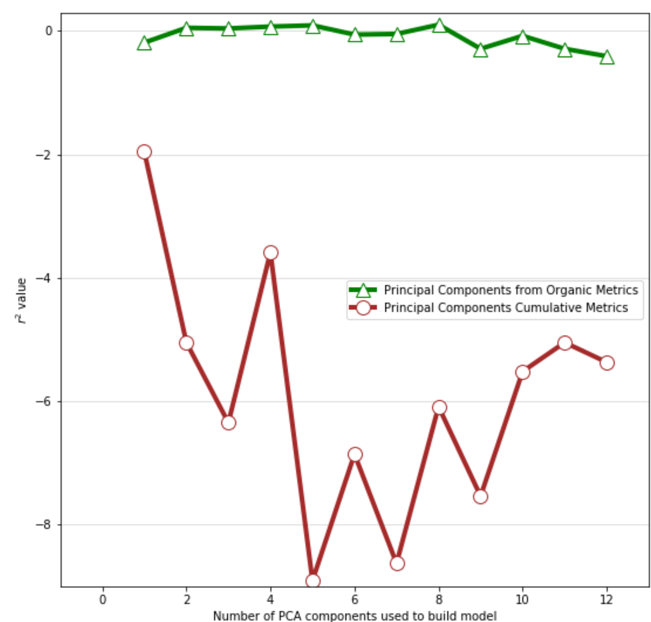


Figure 6. Prediction accuracies of Random Forest Regressors evaluated with `training_test_split` where features are extracted through PCA.

Among the 12 organic metrics used in this study, a combination of six metrics gives the best prediction accuracy (r^2 = 0.25) concerning `training_test_split` as seen in Fig. 5a. Concerning `TimeSeriesSplit` in the same figure, a set of four metrics gives the best prediction accuracy (r^2 = 0.18). The model accuracies are not that high and it can be noted that finding the best predictive model was not an objective of this study. Nevertheless, organic metrics better predict code smells in the continuous integration environment compared to their corresponding cumulative metrics which is the main focus of this study.

Results of this study would help the practitioners, tool developers, and researchers to be aware of the potential of organic metrics to predict code smells in the continuous integration environment. Our results are particularly interesting for the researchers to focus on building improved predictive models for code smells in the continuous integration environment involving more metrics.

VI. Conclusions and Outlook

This empirical study set out to investigate whether organic measures perform better in predicting software artifacts than their corresponding cumulative measures and whether there exists any relationship between measurement types of predictors and a target. Considering code smells as the desired target, we have found that measurement categories play a vital role regarding the accuracies of random forest regression models. Organic measures are found to be much better predicting organic code smells than their corresponding cumulative predictors. Furthermore, organic measures exhibit increased model accuracies when more than one organic measures are combined to form the set of predictors compared to their corresponding cumulative measures. We think, this happens due to the high multicollinearity of the cumulative measures or due to the high correlation among the cumulative measures. We validated our model building and prediction approach by performing PCA to extract components to build models. Using PCA, we have observed similar results but our approach to build models with cumulative and organic metrics as predictors outperformed PCA.

The results of this study are expected to be general within the context of Java projects since it is based on randomly selected projects. Therefore, this study is essential to generally understand the difference between cumulative and organic measures predicting code smells within this context. Further studies are required to understand how model accuracies differ when projects with specific criteria are considered, e.g., size (small vs. large), duration (short vs. long-lived), type (classroom vs. real), programming languages, etc. Our results indicate possible interactions among various predictors which is important knowledge to understand the software metrics better. Therefore, more research can be conducted to investigate the interactions between software metrics. More importantly, further research should be carried out primarily focusing on building improved models to predict code smells in the continuous integration environment using more metrics.

Acknowledgment

The authors would like to thank Dr. Miroslaw Ochodek, Dr. Bartosz Walter, and Dr. Cigdem Gencel for their feedback.

References

[1] J. Bosch, "Speed, Data, and Ecosystems: The Future of Software Engineering," IEEE Software, vol. 33, no. 1, Jan. 2016, pp. 82–88.

[2] L. Rising and N. S. Janoff, "The scrum software development process for small teams," IEEE Software, vol. 17, no. 4, Jul 2000, pp. 26–32.

[3] M. A. A. Mamun, C. Berger, and J. Hansson, "Effects of measurements on correlations of software code metrics," Empirical Software Engineering, 2019.

[4] ——, "Correlations of software code metrics: An empirical study," in Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, ser. IWSM Mensura '17. New York, NY, USA: ACM, 2017, pp. 255–266, [retrieved: Feb, 2019]. [Online]. Available: http://doi.acm.org/10.1145/3143434.3143445

[5] R. Taylor, "Interpretation of the correlation coefficient: a basic review," Journal of diagnostic medical sonography, vol. 6, no. 1, 1990, pp. 35–39.

[6] K. El Emam and N. F. Schneidewind, "Methodology for Validating Software Product Metrics," National Research Council of Canada, Ottawa, Ontario, Canada, Technical Report NCR/ERC-1076, 2000, [retrieved: Feb, 2019]. [Online]. Available: http://nick.adjective.com/work/ElEmam2000.pdf

[7] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in Proceedings Eighth IEEE Symposium on Software Metrics, 2002, pp. 87–94.

[8] M. A. A. Mamun, C. Berger, and J. Hansson, "Explicating, understanding, and managing technical debt from self-driving miniature car projects," in Managing Technical Debt (MTD), 2014 Sixth International Workshop on. IEEE, 2014, pp. 11–18.

[9] S. Henry, D. Kafura, and K. Harris, "On the Relationships Among Three Software Metrics," in Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality. New York, NY, USA: ACM, 1981, pp. 81–88. [Online]. Available: http://doi.acm.org/10.1145/800003.807911

[10] Y. Tashtoush, M. Al-Maolegi, and B. Arkok, "The Correlation among Software Complexity Metrics with Case Study," arXiv:1408.4523 [cs], Aug. 2014, arXiv: 1408.4523. [Online]. Available: http://arxiv.org/abs/1408.4523

[11] S. Saini, S. Sharma, and R. Singh, "Better utilization of correlation between metrics using Principal Component Analysis (PCA)," in 2015 Annual IEEE India Conference (INDICON), Dec. 2015, pp. 1–6.

[12] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship," Journal of Software Engineering and Applications, vol. 02, no. 03, Oct. 2009, p. 137.

[13] M. J. P. v. d. Meulen and M. A. Revilla, "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs," in The 18th IEEE International Symposium on Software Reliability (ISSRE '07), Nov. 2007, pp. 203–208.

[14] M. G. Kendall, "A new measure of rank correlation," Biometrika, vol. 30, no. 1/2, 1938, pp. 81–93. [Online]. Available: http://www.jstor.org/stable/2332226

[15] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[16] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: a review of current knowledge," Journal of Software Maintenance and Evolution: Research and Practice, vol. 23, no. 3, 2010, pp. 179–202. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.521

[17] "SonarQube | Continuous Inspection," [retrieved: Feb, 2019]. [Online]. Available: https://www.sonarqube.org/

[18] "SpotBugs | Find bugs in Java Programs," [retrieved: Mar, 2019]. [Online]. Available: https://spotbugs.github.io/

[19] "Jtest | Java Development Testing for the Enterprise | Parasoft," [retrieved: Mar, 2019]. [Online]. Available: https://www.parasoft.com/products/jtest

[20] "JArchitect :: Achieve Higher Java code quality," [retrieved: Mar, 2019]. [Online]. Available: https://www.jarchitect.com

[21] "PMD | Source Code Analyzer," [retrieved: Mar, 2019]. [Online]. Available: https://pmd.github.io/

[22] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in 11th IEEE International Software Metrics Symposium (METRICS'05), Sept 2005, p. 15.

[23] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains," in 2013 IEEE International Conference on Software Maintenance, Sep. 2013, pp. 260–269.

[24] F. Arcelli Fontana, M. V. Mntyl, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," Empirical Software Engineering, vol. 21, no. 3, Jun. 2016, pp. 1143–1191. [Online]. Available: https://doi.org/10.1007/s10664-015-9378-4

[25] A. Yamashita and L. Moonen, “Do code smells reflect important maintainability aspects?” in 2012 28th IEEE International Conference on Software Maintenance (ICSM), Sep. 2012, pp. 306–315.

[26] C. J. Kapser and M. W. Godfrey, ““cloning considered harmful” considered harmful: patterns of cloning in software,” Empirical Software Engineering, vol. 13, no. 6, Jul 2008, p. 645. [Online]. Available: https://doi.org/10.1007/s10664-008-9076-6

[27] R. Shatnawi and W. Li, “An Investigation of Bad Smells in Object-Oriented Design,” in Third International Conference on Information Technology: New Generations (ITNG’06), Apr. 2006, pp. 161–165.

[28] W. Li and R. Shatnawi, “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution,” Journal of Systems and Software, vol. 80, no. 7, Jul. 2007, pp. 1120–1128. [Online]. Available: http://www.sciencedirect.com/science/article/pii/ S0164121206002780

[29] F. Khomh, M. D. Penta, and Y. G. Gueheneuc, “An Exploratory Study of the Impact of Code Smells on Software Change-proneness,” in 2009 16th Working Conference on Reverse Engineering, Oct. 2009, pp. 75–84.

[30] N. Maneerat and P. Muenchaisri, “Bad-smell prediction from software design model using machine learning techniques,” in 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), May 2011, pp. 331–336.

[31] A. Gupta, B. Suri, V. Kumar, S. Misra, T. Blaauskas, and R. Damaeviius, “Software Code Smell Prediction Model Using Shannon, Rnyi and Tsallis Entropies,” Entropy, vol. 20, no. 5, May 2018, p. 372. [Online]. Available: https://www.mdpi.com/1099-4300/20/5/372

[32] P. Runeson and M. Höst, “Guidelines for Conducting and Reporting Case Study Research in Software Engineering,” Empirical Software Engineering, vol. 14, no. 2, Dec. 2008, pp. 131–164. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9102-8

[33] C. Robson, “Real world research. 2nd,” Edition. Blackwell Publishing. Malden, 2002.

[34] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, Selecting Empirical Methods for Software Engineering Research. London: Springer London, 2008, pp. 285–311. [Online]. Available: https://doi.org/10.1007/978-1-84800-044-5_11

[35] R. E. Stake, The Art of Case Study Research. California, USA: SAGE Publications, Apr. 1995, google-Books-ID: ApGdBx76b9kC.

[36] G. Gousios and D. Spinellis, “GHTorrent: GitHub’s data from a firehose,” in MSR ’12: Proceedings of the 9th Working Conference on Mining Software Repositories, M. W. Godfrey and J. Whitehead, Eds. IEEE, Jun. 2012, pp. 12–21. [Online]. Available: /pub/ghtorrent-githubs-data-from-a-firehose.pdf

[37] J. D. Long and N. Cliff, “Confidence intervals for Kendall’s tau,” British Journal of Mathematical and Statistical Psychology, vol. 50, no. 1, May 1997, pp. 31–41. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2044-8317.1997.tb01100.x

[38] G. A. Campbell, “Cognitive complexity: An overview and evaluation,” in Proceedings of the 2018 International Conference on Technical Debt, ser. TechDebt ’18. New York, NY, USA: ACM, 2018, pp. 57–58. [Online]. Available: http://doi.acm.org/10.1145/3194164.3194186

[39] T. C. Lethbridge, S. E. Sim, and J. Singer, “Studying software engineers: Data collection techniques for software field studies,” Empirical Software Engineering, vol. 10, no. 3, Jul 2005, pp. 311–341. [Online]. Available: https://doi.org/10.1007/s10664-005-1290-x

[40] “SonarQube Metric Definitions,” [retrieved: Feb, 2019]. [Online]. Available: https://docs.sonarqube.org/latest/user-guide/metric-definitions/

[41] “GitHub Octoverse 2017 | Highlights from the last twelve months,” [retrieved: Feb, 2019]. [Online]. Available: https://web.archive.org/web/20180602170102/ https://octoverse.github.com/

APPENDIX

TABLE V. LIST OF CODE SMELLS USED IN THIS STUDY.

Code smell
“main” should not “throw” anything
Class names should not shadow interfaces or superclasses
Short-circuit logic should be used in boolean contexts
Future keywords should not be used as names
String literals should not be duplicated
Modulus results should not be checked for direct equality
“readResolve” methods should be inheritable
Methods & field names should not be the same or differ only by capitalization
Switch cases should end with an unconditional ”break” statement
“if ... else if” constructs should end with “else” clauses
“switch” statements should not be nested
Fields in a “Serializable” class should either be transient or serializable
The Object.finalize() method should not be overriden
Execution of the Garbage Collector should be triggered only by the JVM
Constructors should only call non-overridable methods
Methods returns should not be invariant
“switch” statements should not contain non-case labels
“clone” should not be overridden
Assertions should be complete
“for” loop increment clauses should modify the loops’ counters
Method overrides should not change contracts
Exceptions should not be thrown in finally blocks
Classes should not access their own subclasses during initialization
“Object.wait()” and “Condition.await()” should be called inside a “while” loop
“Cloneables” should implement “clone”
“Object.finalize()” should remain protected (versus public) when overriding
Child class fields should not shadow parent class fields
Factory method injection should be used in “@Configuration” classes
Threads should not be started in constructors
“indexOf” checks should not be for positive numbers
Instance methods should not write to “static” fields
Tests should include assertions
Null should not be returned from a “Boolean” method
Lazy initialization of “static” fields should be “synchronized”
IllegalMonitorStateException should not be caught