On Recommending Actionable Changes For Software Projects

Rahul Krishna
NC State University
i.m.ralk@gmail.com

ABSTRACT

Newer generation of software analytics have placed significant emphasis on data driven decision making. These decisions are usually based on lessons that arise from within a particular project. Some times it is also possible to derive these decisions from across multiple projects. In the past, research efforts have led to the development of XTREE for generating a set of actionable plans within and across projects. We postulated that, each of these plans, if followed will improve the quality of the software project. Our previous work, however, culminated in an open question — do developers make use of these plans? If so, to what extent? This work is an attempt to answer these questions. Specifically, we compare the overlap between changes made by developers and those recommended by XTREE. To this end, we mined several versions of nine popular open source software projects. Our results show that recommendations offered by conventional XTREE overlaps to an extent of 50% with changes undertaken by the developers. Modified versions of XTREE was developed to improve this overlap. And these offer over 75% overlap.

1 INTRODUCTION

2 RELATED WORK

3 PLANNING IN SOFTWARE ENGINEERING

4 EXPERIMENTAL SETUP

4.1 Datasets

The defect dataset used in this study comprises a total of 9 datasets that were mined from GitHub. The projects primarily involve systems developed in JAVA. For these systems, we measure defects at class and method-level granularity.

The procedure used to mine these datasets is illustrated in Fig. 1. All the datasets are described in terms of the attributes of Fig. 3. In order to gather these datasets, we use the following steps:

(1) Find the most popular JAVA projects on GitHub. For these projects, clone them locally and generate a complete log of release dates and commit messages.
(2) For each release, build the project and gather the CK-Metrics. We used the ckjm tool[1] for this.
(3) Use the commit messages between each releases to identify "bug-fixing" commits.
(4) For each "bug-fixing" commit, find the respective files that are changed and mark them as defect prior to the release.

The above process, when repeated for all the projects, generates the required datasets. In Fig. 2, we summarize all the datasets that were mined for experimentation. All these datasets records the

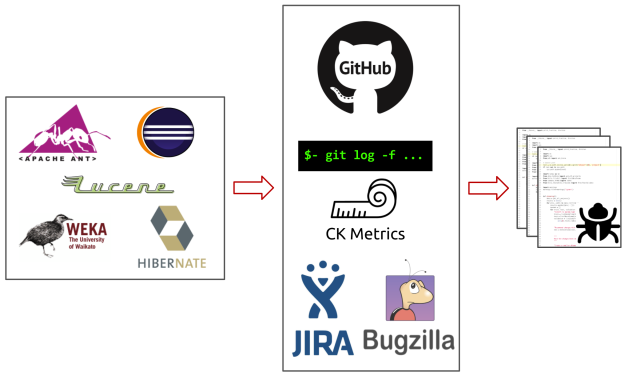

Figure 1: Framework for data generation

Community	Dataset	# of instances		
		# Versions	# samples	Bugs (%)
Java OO	Ant	7	14289	438 (56.01)
	Azureus	20	25818	350 (20.69)
	Ecplise	17	978905	119 (16.90)
	Hibernate	38	20712	303 (17.32)
	JMeter	8	25915	707 (51.31)
	JStock	10	59796	562 (20.19)
	JUnit	8	25855	260 (57.91)
	Lucene	12	58582	367 (57.43)
	Weka	8	29816	1806 (54.40)

Figure 2: The figure lists the defect datasets gathered for this project.

number of known defects for each class using a post-release bug tracking system. The classes are described in terms of 20 OO metrics, including CK metrics and McCabes complexity metrics.

4.2 Evaluation Strategy

Our experimental design is shown in Fig. 4. We divide the project data into two parts the *train set* and the *test test.* The train set could either be data that is available locally within a project, or it could be data from the bellwether dataset. We partition the train set to build both a *planner* and a *verification oracle.*

The following paragraph shows our **principle of separation** which, we assert, is necessary to verify this kind of research:

> *The verification oracle should be built with completely different data to the planner, i.e., the data used for training a planner and constructing the verification oracle are disjoint sets.*

In § **??** we describe the planners and a way to translate these plans into actionable decisions. To assess the effect of acting on these

[1]https://github.com/mjureczko/CKJM-extended

amc	average method complexity	e.g. number of JAVA byte codes
avg cc	average McCabe	average McCabe's cyclomatic complexity seen in class
ca	afferent couplings	how many other classes use the specific class.
class.		
cam	cohesion amongst classes	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
cbm	coupling between methods	total number of new/redefined methods to which all the inherited methods are coupled
cbo	coupling between objects	increased when the methods of one class access services of another.
ce	efferent couplings	how many other classes is used by the specific class.
dam	data access	ratio of the number of private (protected) attributes to the total number of attributes
dit	depth of inheritance tree	
ic	inheritance coupling	number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an case variable.
locm3	another lack of cohesion measure	if m, a are the number of $methods, attributes$ in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $lcom3 = ((\frac{1}{a}\sum_{j}^{a}\mu(a,j)) - m)/(1-m)$.
loc	lines of code	
max cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
mfa	functional abstraction	number of methods inherited by a class plus number of methods accessible by member methods of the class
moa	aggregation	count of the number of data declarations (class fields) whose types are user defined classes
noc	number of children	
npm	number of public methods	
rfc	response for a class	number of methods invoked in response to a message to the object.
wmc	weighted methods per class	
nDefects	raw defect counts	Numeric: number of defects found in post-release bug-tracking systems.
isDefective	defects present?	Boolean: if *nDefects* >0 then *true* else *false*

Figure 3: OO code metrics used for all studies in this paper. Last lines, shown in gray, denote the dependent variables.

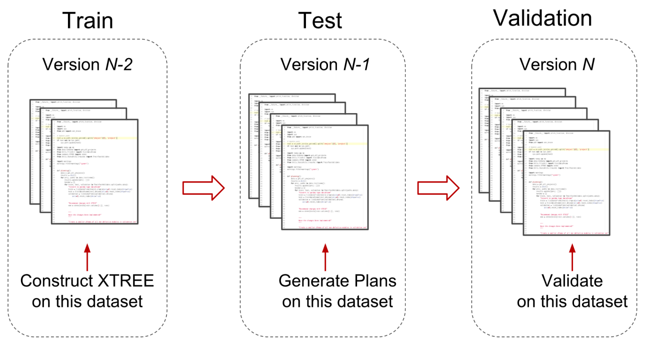

Figure 4: Experimental design

plans, we propose a *verification oracle*. This is necessary because it can be rather difficult to judge the effects of applying the plan. They cannot be assessed just by a rerun of the test suite for three reasons: (1) The defects were recorded by a post release bug tracking system. It is entirely possible it escaped detection by the existing test suite; (2) Rewriting test cases to enable coverage of all possible scenarios presents a significant challenge; and (3) It make take a significant amount of effort to write new test cases that identify these changes as they are made.

To resolve this problem, SE researchers such as Cheng et al. [3], O'Keefe et al. [13, 14], Moghadam [11] and Mkaouer et al. [10] use a *verification oracle* that is learned separately from the primary oracle. The verification oracles assesses how defective the code is before and after some code changes. For their secondary verification oracle, Cheng, O'Keefe, Moghadam and Mkaouer et al. use the QMOOD hierarchical quality model [2]. A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [8].

Hence, for this study, we eschew older quality models like QMOOD.

4.3 Performance Measures

4.3.1 Effectiveness. For planning and construction of a verification oracle, we divide the project data into two parts the *train set* and the *test test*. The train set could either be data that is available locally within a project, or it could be data from the bellwether dataset. We further partition the train set to build both a *planner* and a *verification oracle*. Note that: *The verification oracle should be built with completely different data to the planner.*

After constructing the planner and verification oracle, we (1) deploy the planner to recommend plans; (2) alter the *test* data according to these plans; then (3) apply the verification oracle to the altered data to estimate defects; then (3) Compute the percent improvement, denoted by the following equation:

$$R = (1 - \frac{after}{before}) \times 100\% \tag{1}$$

The value of the measure R has the following properties: (1) If $R = 0\%$, this means "no change from baseline"; (2) If $R > 0\%$, this indicates "improvement"; (3) If $R < 0\%$, this indicates "optimization failure". Ideally, an effective planner should have an improvement of $R > 0$, where larger values indicate better performance.

4.4 Statistics

Changes made to module in a software project are subject to inherent randomness. Researchers have endorsed the use of repeated runs to gather reliable evidence [**?**]. Thus, we repeat the whole experiment independently using a moving window over the various versions of the project (see § 4.2) to provide evidence that the results are reproducible. These repeats provide us with a sufficiently large sample size to statistically compare the performances.

At each position of the moving window, we collect the values of effectiveness R (§ 1) and overlap O (§ **??**). We refrain from performing a cross validation because the process tends to mix the samples from training data (the source) and the test data (other target projects), which defeats the purpose of this study.

To rank the numbers collected above, we use the Scott-Knott test recommended by Mittas and Angelis [9]. Scott-Knott is a top-down clustering approach used to rank different treatments. If that clustering finds an statistically significant splits in data, then some statistical test is applied to the two divisions to check if they are statistically significant different. If so, Scott-Knott recurses into both halves.

To apply Scott-Knott, we sorted a list of l values of § **??** values found in ls different methods. Then, we split l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l, m, n of size ls, ms, ns where $l = m \cup n$:

$$E(\Delta) = \frac{ms}{ls} abs(m.\mu - l.\mu)^2 + \frac{ns}{ls} abs(n.\mu - l.\mu)^2$$

We then apply a statistical hypothesis test H to check if m, n are significantly different. In our case, the conjunction of bootstrapping and A12 test. Both the techniques are non-parametric in nature, i.e., they do not make gaussian assumption about the data. As for hypothesis test, we use a non-parametric bootstrapping test as endorsed by Efron & Tibshirani [4, p220-223]. Even with statistical significance, it is possible that the difference can be so small as to be of no practical value. This is known as a "small effect". To ensure that the statistical significance is not due to "small effect" we use effect-size tests in conjunction with hypothesis tests. A popular effect size test used in SE literature is the A12 test. It has been endorsed by several SE researchers [1, 5, 6, 15**? ?**]. It was first proposed by Vargha and Delany [**?**]. In our context, given the performance measure G, the A12 statistics measures the probability that one treatment yields higher G values than another. If the two algorithms are equivalent, then A12 = 0.5. Likewise if $A12 \geq 0.6$, then 60% of the times, values of one treatment are significantly greater that the other. In such a case, it can be claimed that there is *significant effect* to justify the hypothesis test H. In our case, we divide the data if *both* bootstrap sampling and effect size test agree that a division is statistically significant (with a confidence of 99%) and not a small effect ($A12 \geq 0.6$). Then, we recurse on each of these splits to rank G-scores from best to worst.

5 RESULTS

6 RELIABILITY AND VALIDITY OF CONCLUSIONS

7 CONCLUSIONS AND FUTURE WORK

The results of this paper are biased by our choice of code reorganization goal (reducing defects) and our choice of measures collected from software project (OO measures such as depth of inheritance, number of child classes, etc). That said, it should be possible extend the methods of this paper to other kinds of goals (e.g. maintainability, reliability, security, or the knowledge sharing measures) and other kinds of inputs (e.g. the process measures favored by Rahman, Devanbu et al. [**?**])

7.1 Reliability

Reliability refers to the consistency of the results obtained from the research. It has at least two components: internal and external reliability.

Internal reliability checks if an independent researcher reanalyzing the data would come to the same conclusion. To assist other researchers exploring this point, we offer a full replication package for this study at https://github.com/rahlk/FSS17.

External reliability assesses how well independent researchers could reproduce the study. To increase external reliability, this paper has taken care to clearly define our algorithms. Also, all the data used in this work is available online.

For the researcher wishing to reproduce our work to other kinds of goals, we offer the following advice:

- Find a data source for the other measures of interest;
- Implement another secondary verification oracle that can assess maintainability, reliability, security, technical debt, etc;
- Implement a better primary verification oracle that can do "better" than XTREE at finding changes (where "better" is defined in terms of the opinions of the verification oracle).

7.2 Validity

This paper is a case study that studied the effects of limiting unnecessary code reorganization on some data sets. This section discusses limitations of such case studies. In this context, validity refers to the extent to which a piece of research actually investigates what the researcher purports to investigate. Validity has at least two components: internal and external validity.

Based on the case study presented above, as well as the discussion in § **??**, we believe that defect indicators (e.g. *loc*> 100) have limited external validity beyond the projects from which they are derived. While specific models are externally valid, there may still be general methods like XTREE for finding the good local models.

Our definition of bad smells is limited to those represented by OO code metrics (a premise often used in related work). XTREE, Shatnawi, Alves et al. can only comment on bad smells expressed as code metrics in the historical log of a project.

If developers want to justify their code reorganizations via bad smells expressed in other terminology, then the analysis of this paper must:

- Either wait till data about those new terms has been collected.

- Or, apply cutting edge transfer learning methods [7, 12**?**] to map data from other projects into the current one.

Note that the transfer learning approach would be highly experimental and require more study before it can be safely recommended.

Sampling bias threatens any data mining analysis; i.e., what matters there may not be true here. For example, the data sets used here comes from Jureczko et al. and any biases in their selection procedures threaten the validity of these results. That said, the best we can do is define our methods and publicize our data and code so that other researchers can try to repeat our results and, perhaps, point out a previously unknown bias in our analysis. Hopefully, other researchers will emulate our methods in order to repeat, refute, or improve our results.

REFERENCES

[1] A. Arcuri and L. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE'11*. 1–10.

[2] Jagdish Bansiya and Carl G. Davis. 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Trans. Softw. Eng.* 28, 1 (Jan. 2002), 4–17. https://doi.org/10.1109/32.979986

[3] Betty Cheng and Adam Jensen. 2010. On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO '10)*. ACM, New York, NY, USA, 1341–1348. https://doi.org/10.1145/1830483.1830731

[4] Bradley Efron and Robert J Tibshirani. 1993. *An introduction to the bootstrap*. Chapman and Hall, London.

[5] Vigdis By Kampenes, Tore Dybå, Jo Erskine Hannay, and Dag I. K. Sjøberg. 2007. A systematic review of effect size in software engineering experiments. *Information & Software Technology* 49, 11-12 (2007), 1073–1086.

[6] Ekrem Kocaguneli, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, and Tim Menzies. 2013. Distributed development considered harmful?. In *Proceedings - International Conference on Software Engineering*. 882–890. https://doi.org/10.1109/ICSE.2013.6606637

[7] Rahul Krishna, Tim Menzies, and Wei Fu. 2016. Too much automation? the bellwether effect and its implications for transfer learning. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, New York, New York, USA, 122–131. https://doi.org/10.1145/2970276.2970339

[8] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. [n. d.]. *IEEE Transactions on Software Engineering* ([n. d.]). https://doi.org/10.1109/TSE.2012.83

[9] Nikolaos Mittas and Lefteris Angelis. 2013. Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm. *IEEE Trans. Software Eng.* 39, 4 (2013), 537–551.

[10] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation System for Software Refactoring Using Innovization and Interactive Dynamic Optimization. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 331–336. https://doi.org/10.1145/2642937.2642965

[11] Iman Hemati Moghadam. 2011. *Search Based Software Engineering: Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Multi-level Automated Refactoring Using Design Exploration, 70–75. https://doi.org/10.1007/978-3-642-23716-4_9

[12] Jaechang Nam and Sunghun Kim. 2015. Heterogeneous defect prediction. In *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*. ACM Press, New York, New York, USA, 508–519. https://doi.org/10.1145/2786805.2786814

[13] Mark O'Keeffe and Mel Ó Cinnéide. 2008. Search-based Refactoring: An Empirical Study. *J. Softw. Maint. Evol.* 20, 5 (Sept. 2008), 345–364. https://doi.org/10.1002/smr.v20:5

[14] Mark Kent O'Keeffe and Mel O. Cinneide. 2007. Getting the Most from Search-based Refactoring. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. ACM, New York, NY, USA, 1114–1120. https://doi.org/10.1145/1276958.1277177

[15] Martin J. Shepperd and Steven G. MacDonell. 2012. Evaluating prediction systems in software project estimation. *Information & Software Technology* 54, 8 (2012), 820–827.