Noname manuscript No.
(will be inserted by the editor)

Revisiting Process versus Product Metrics: a Large Scale Analysis

Suvodeep Majumder · Pranav Mody · Tim Menzies

Received: date / Accepted: date

Abstract Numerous methods can build predictive models from software data. But what methods and conclusions should we endorse as we move from analytics in-the small (dealing with a handful of projects) to analytics in-the-large (dealing with hundreds of projects)?

To answer this question, we recheck prior small scale results (about process versus product metrics for defect prediction and the granularity of metrics) using 722,471 commits from 700 Github projects. We find that some analytics in-the-small conclusions still hold when scaling up to analytics in-the large. For example, like prior work, we see that process metrics are better predictors for defects than product metrics (best process/product-based learners respectively achieve recalls of 98%/44% and AUCs of 95%/54%, median values).

That said, we warn that it is unwise to trust metric importance results from analytics in-the-small studies since those change, dramatically when moving to analytics in-the-large. Also, when reasoning in-the-large about hundreds of projects, it is better to use predictions from multiple models (since single model predictions can become very confused and exhibit very high variance).

S. Majumder
Department of Computer Science,
North Carolina State University,
Raleigh, USA
E-mail: smajumd3@ncsu.edu

P. Mody
Department of Computer Science,
North Carolina State University,
Raleigh, USA
E-mail: prmody@ncsu.edu

T. Menzies
Department of Computer Science,
North Carolina State University,
Raleigh, USA
E-mail: tim@ieee.org

1 Introduction

There exist many automated software engineering techniques for building predictive models from software project data [22]. Such models are cost-effective methods for guiding developers on where to quickly find bugs [43,55].

Given there are so many techniques, the question naturally arises: which we should use? Software analytics is growing more complex and more ambitious. A decade ago, a standard study in this field dealt with just 20 projects, or less[1]. Now we can access data on hundreds to thousands of projects. How does this change software analytics? What methods and conclusions should we endorse as we move from analytics in-the small (dealing with a handful of projects) to analytics in-the-large (dealing with hundreds of projects). So reproducing results and findings that were true for analytics in-the small is of utmost importance with hundreds to thousands of projects. Such analytics in-the-large results will help the software engineering community to understand and adopt appropriate methods, beliefs and conclusions.

As part of this study, we revisited the Rahman et al. ICSE 2013 study *"How, and why, process metrics are better"* [59] and Kamei et al. ICSM 2010 study *"Revisiting common bug prediction findings using effort-aware models"* [31] . Both papers were analytics in-the small study that used 12 and 3 projects respectively to see if defect predictors worked best if they used:

– Product metrics showing what was built; e.g. see Table 1.
– Or process metrics showing how code is changed; e.g. see Table 2;

These papers are worth revisiting since it is widely cited[2] and it addresses an important issue. Herbsleb argues convincingly that how groups organize themselves can be highly beneficial/detrimental to the process of writing code [26]. Hence, process factors can be highly informative about what parts of a codebase are buggy. In support of the Herbsleb hypothesis, prior studies have shown that, for the purpose of defect prediction, process metrics significantly out-perform product metrics [37,59,9]. Also, if we wish to learn general principles for software engineering that hold across multiple projects, it is better to use process metrics since:

– Process metrics, are much simpler to collect and can be applied in a uniform manner to software written in different languages.
– Product metrics, on the other hand, can be much harder to collect. For example, some static code analysis requires expensive licenses which need updating every time a new version of a language is released [61]. Also, the collected value for these metrics may not translate between projects since those ranges can be highly specific. Lastly, product metrics are far more verbose and hence harder to collect. For example, for 722,471 commits studied in this paper, data collected required 500 days of CPU (using five machines, 16 cores, 7days). Our process metrics, on the other hand, were an order of magnitude faster to collect.

[1] For examples of such papers, see Table 3, later in this paper.

[2] 232 and 179 citations respectively in Google Scholar, as of Sept 28, 2020.

Since product versus process metrics is such an important issue, we revisited the Rahman et al. and Kamei et al. study. To check their conclusions, we ran an analytics in-the-large study that looked at 722,471 commits from 700 Github projects.

All in all, this paper explores eight hypothesis. We find that in six cases, the analytics-in-the-small are the same as analytics-in-the-large. So what is the value of a paper with 75% agreement with prior work? We assert that this paper makes several important contributions:

- Firstly, in the two cases where we disagree, we strongly disagree with prior work:
  - We find that the use of any learner is not appropriate for analytics-in-large. Our results suggest that any learner that generates a single model may get confused by all the intricacies of data from multiple projects. On the other had, *ensemble learners* (that make the conclusions by polling across many models) know how to generated good predictions from a very large sample.
  - Also, in terms of *what recommendations we would make* to improve software quality, we find that the conclusions achieved via analytics-in-the-large are very different to those achieved via analytic-in-the-small. Later in this paper we compare those two sets of conclusions. We will show that changes to software projects that make sense from analytics-in-the-small (after looking at, any, 5 projects) can be wildly misleading since once we get to analytics-in-the-large, a very different set of attributes are most effective.
- Secondly, in the case where our conclusions are the same as prior work, we have still successfully completed a valuable step in the scientific process: i.e. reproduction of prior results. Current ACM guidelines[3] distinguish replication and reproduction as follows: the former uses artifacts from the prior study while the latter does not. Our work is a *reproduction*[4] since we use ideas from the Rahman et al. and Kamei et al. study, but none of their code or data. We would encourage more researchers to conduct and report more reproduction studies.

Specifically, we ask eight research questions

> **RQ 1:** For predicting defects, do methods that work in-the-small, also work in-the-large?

In a result that agrees with Rahman et al., we find that how we build code is more indicative of what bugs are introduced than what we build (i.e. process metrics make best defect predictions).

> **RQ 2:** Measured in terms of predication variability, do methods that works well in-the-small, also work at at-scale?

[3] https://www.acm.org/publications/policies/artifact-review-and-badging-current

[4] Just to be clear: technically speaking, this paper is a *partial reproduction* of Rahman et al. or Kamei et al. When we tried their methodology, we found in some cases our results needed slightly different approach (see § 3.3).

Table 1: List of product metrics used in this study

Type	Metrics	Count
File	AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, CountDeclClassMethod, CountDeclClassVariable, CountDeclInstanceMethod, CountDeclInstanceVariable, CountDeclMethod, CountDeclMethodAll, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethodProtected, CountDeclMethodPublic, CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict,MaxEssential, RatioCommentToCode, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential	37
Class	PercentLackOfCohesion, PercentLackOfCohesionModified, MaxInheritanceTree, CountClassDerived, CountClassCoupled, CountClassCoupledModified, CountClassBase	7
Method	MaxNesting	1

Rahman et al said that it does not matter what learner is used to build prediction models. **We make the exact opposite conclusion.** For analytics-in-the-large, the more data we process, the more variance in that data. Hence, conclusions that rely on a single model get confused and exhibit large variance in their predictions. To mitigate this problem, it is important to use learners that make conclusions by averaging over multiple models (i.e. ensemble Random Forests are far better for analytics than the Naive Bayes, Logistic Regression, or Support Vector Machines used in prior work).

RQ 3: Measured in terms of granularity, do same granularity that works well in-the-small, also work at at-scale?

Kamei et al. said in their study that although the file level prediction is better than package level prediction when measured using Popt20 the difference is very little and we agree in this result. But when measured via other evaluation measure the difference is significantly different. Thus for analytics-in-the-large, when measured using other criteria it is evident the granularity of the metrics matter and file level prediction show significantly better result that package level prediction.

RQ 4: Measured in terms of stability, are process metrics more/less stable than code metrics, when measured at at-scale?

When measured in-term of stability of performance across last 3 releases by using all other previous releases for training the model, our results agrees with Rahman et al. in all traditional evaluation criteria (i.e. recall, pf, precision). We find that the performance across last 3 releases do not significantly differ in all evaluation criteria except for effort aware evaluation criteria Popt20.

RQ 5: Measured in terms of stasis, Are process metrics more/less static than code metrics, when measured at at-scale?

In this result, we agree with Rahman et al., we can see product metrics are significantly more correlated than process metrics. We measure this correlation in both released based and JIT based setting. Although we can see process metrics have significantly lower correlation than product metrics in both released based and JIT based setting, the difference is lower in case of JIT based setting. Also when lifting process metrics from file level to package level as explored by Kamei et al., we can see a significant increase in correlation in process metrics. This can explain the drop in performance in package level prediction.

RQ 6: Measured in terms of stagnation, Do models built from different sets of metrics stagnate across releases, when measured at at-scale?

Rahman et al. warn that, when reasoning over multiple releases, models can *stagnant* i.e. fixate on old conclusions and miss new ones. For example, if the same kinds of defects occur in release one and two, and some new kinds of detect appear in the second release, the model will catch the old kinds of defects and miss the new ones.

Here we measure the stagnation property of the models built using the metrics. Our results agree with Rahman et al.: we see significantly higher correlation between the predicted probability and learned probability in case of product metrics then process metrics. This signifies models built using product data tends to be stagnant.

RQ 7: Do stagnant models (based on stagnant metrics) tend to predict recurringly defective entities?

In this result we try to evaluate if models built with product and process data tends to predict recurrent defects. Our results concur with Rahman et al. and we see models built with product data tends to predict recurrent defects, while models built with process data does not suffer from this effect.

RQ 8: Measured in terms of metric importance, are metrics that seem important in-the-small, also important when reasoning in-the-large?

Numerous prior analytics in-the-small publications offer conclusions on the relative importance of different metrics. For example, [30], [21], [47], [35], [17] offer such conclusions after an analysis of 1,1,3, 6,and 26 software project, respectively. Their conclusions are far more specific than process-vs-product; rather, these prior studies call our particular metrics are being most important for prediction.

Based on our analysis, we must now call into question any prior analytics in-the-small conclusions that assert that specific metrics are more important than any other (for the purposes of defect prediction). We find that relative importance of different **metrics found via analytics in-the-small is not stable**.

Specifically, when we move to analytics in-the-large, we find very different rankings for metric importance.

The rest of this paper is structured as follows. Some background and related work are discussed in section 2. Our experimental methods are described in section 3. Data collection in section 3.1 and learners used in this study in section 3.2. Followed by experimental setup in section 3.3 and evaluation criteria in section 3.4. The results and answers to the research questions are presented in section 4. Which is followed by threats to validity in section 5. Finally, the conclusion is provided in section 6.

Note that all the scripts and data used in this analysis are available on-line at `http://tiny.cc/revisit` [5].

2 Background and Related Work

2.1 Defect Prediction

This section shows that software defect prediction is a (very) widely explored area with many application areas. Specifically, in 2020, software defect prediction is now a "sub-routine" that enables much other research.

A defect in software is a failure or error represented by incorrect, unexpected or unintended behavior of a system, caused by an action taken by a developer. As today's software grows rapidly both in size and number, software testing for capturing those defects plays more and more crucial roles. During software development, the testing process often has some resource limitations. For example, the effort associated with coordinated human effort across large codebase can grow exponentially with the scale of the project [20].

It is common to match the quality assurance (QA) effort to the perceived criticality and bugginess of the code for managing resources efficiently. Since every decision is associated with a human and resource cost to the developer team, it is impractical and inefficient to distribute equal effort to every component in a software system[11]. Creating defect prediction models from either product metrics (like those from Table 1) or process metrics (like those from Table 2) is an efficient way to take a look at the incoming changes and focus on specific modules or files based on a suggestion from defect predictor.

Recent results show that software defect predictors are also competitive widely-used automatic methods. Rahman et al. [62] compared (a) static code analysis tools FindBugs, Jlint, and PMD with (b) defect predictors (which they called "statistical defect prediction") built using logistic regression. No significant differences in cost-effectiveness were observed. Given this equivalence, it is significant to note that defect prediction can be quickly adapted to new languages by building lightweight parsers to extract product metrics or use common change information by mining git history to build process metrics.

[5] Note to reviewers: Our data is so large we cannot place in Github repo. Zenodo.org will host our data. http://tiny.cc/revisit only contains a sample of our data. we will link that repository to link to data stored at Zenodo.org.

Table 2: List of process metrics used in this study

adev	Active Dev Count
age	Interval between the last and the current change
ddev	Distinct Dev Count
sctr	Distribution of modified code across each file
exp	Experience of the committer
la	Lines of code added
ld	Lines of code deleted
lt	Lines of code in a file before the change
minor	Minor Contributor Count
nadev	Neighbor's Active Dev Count
ncomm	Neighbor's Commit Count
nd	Number of Directories
nddev	Neighbor's Distinct Dev Count
ns	Number of Subsystems
nuc	Number of unique changes to the modified files
own	Owner's Contributed Lines
sexp	Developer experience on a subsystem
rexp	Recent developer experience

The same is not true for static code analyzers - these need extensive modification before they can be used in new languages. Because of this ease of use, and its applicability to many programming languages, defect prediction has been extended in many ways including:

1. Application of defect prediction methods to locate code with security vulnerabilities [69].
2. Understanding the factors that lead to a greater likelihood of defects such as defect prone software components using code metrics (e.g., ratio comment to code, cyclomatic complexity) [42,41] or process metrics (e.g., recent activity).
3. Predicting the location of defects so that appropriate resources may be allocated (e.g., [8])
4. Using predictors to proactively fix defects [4]
5. Studying defect prediction not only just release-level [15] but also change-level or just-in-time [64].
6. Exploring "transfer learning" where predictors from one project are applied to another [36,53].
7. Assessing different learning methods for building predictors [22]. This has led to the development of hyper-parameter optimization and better data harvesting tools [2,1].

2.2 Process vs Product

Defect prediction models are built using various machine learning classification methods such as Random Forest, Support Vector Machine, Naive Bayes, Logistic Regression [73,86,28,88,27,77,36,71,45,67,68,23,85,25,54,56] etc. All these methods input project metrics and output a model that can make pre-

	# Data Sets	Year	Venue	Citations
Using software dependencies and churn metrics to predict field failures: An empirical case study	1	2007	ESEM	2007
Data mining static code attributes to learn defect predictors	8	2006	TSE	1266
Mining metrics to predict component failures	5	2006	TSE	845
Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects	1	2003	TSE	781
Predicting fault incidence using software change history	1	2000	TSE	779
Predicting defects for eclipse	1	2007	ICSE	717
A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction	1	2008	ICSE	580
Empirical analysis of object-oriented design metrics for predicting high and low severity faults,	1	2006	TSE	405
A systematic and comprehensive investigation of methods to build and evaluate fault prediction models	1	2010	JSS	384
Using class imbalance learning for software defect prediction	10	2013	TR	322
Don't touch my code! examining the effects of ownership on software quality	2	2011	FSE	289
How, and why, process metrics are better	12	2013	ICSE	232
Revisiting Common Bug Prediction Findings Using Effort-Aware Models	3	2010	ICSM	179
Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models,	2	2008	EMSE	175
Implications of ceiling effects in defect predictors	12	2008	IPSE	172
Ownership, experience and defects: a fine-grained study of authorship	4	2011	ICSE	171
Change bursts as defect predictors	1	2010	ISSRE	162
Using coding-based ensemble learning to improve software defect prediction	14	2012	SMC	127
Empirical study of the classification performance of learners on imbalanced & noisy software quality data	1	2014	IS	125
The effects of over and under sampling on fault-prone module detection	1	2007	ESEM	118
Which process metrics can significantly improve defect prediction models? an empirical study	18	2015	SQJ	110
An investigation of the relationships between lines of code and defects	1	2009	ICSE	108
Bugcache for inspections: hit or miss	5	2011	FSE	89
An analysis of developer metrics for fault prediction	1	2010	PROMISE	80
Predicting faults in high assurance software	15	2010	HASE	45
Is external code quality correlated with programming experience or feelgoodfactor?	1	2006	XP	23
Empirical analysis of change metrics for software fault prediction	1	2018	CEE	21
A validation of object-oriented design metrics as quality indicators	8	1996	TSE	21

Table 3: Number of data sets explored in recent papers at prominent venues that experiment with process and/or product metrics.

dictions. Fenton et al. [19] says, that a "metric" is an attempt to measure some internal or external characteristic and can broadly be classified into *product* (specification, design, code related) or *process* (constructing specification, detailed design related). The metrics are computed either through parsing the codes (such as modules, files, classes or methods) to extract product (code) metrics or by inspecting the change history by parsing the revision history of files to extract process (change) metrics.

In April 2020, in order to understand the current thinking on process and product methods, we conducted the following literature review. Starting with Rahman et al. [60] and Kamei et al. [31] we used Google Scholar to trace citations forward and backward looking for papers that offered experiments that suggested why certain process or product metrics are better for defect prediction. Initially, we only examined:

- Highly cited papers; i.e. those with at least ten cites per year since publication.
- Papers from senior SE venues; i.e. those listed at "Google Scholar Metrics Software Systems".

Next, using our domain expertise, we augmented that list with papers we considered important or highly influential. Finally, one last paper was added since, as far as we could tell, it was the first to discuss this issue in the context of analytics. This lead in the 27 papers of Table 3.

Within this set of papers, we observe that studies on product metrics are more common that on process metrics (and very few papers experimentally compare both product and process metrics: see Figure 1). The product metrics community [77,71,45,67,68,32,43,91] argues that many kinds of metrics indicate which code modules are buggy:

- For example, for lines of code, it is usually argued that large files can be hard to comprehend and change (and thus are more likely to have bugs);
- For another example, for design complexity, it is often argued that the more complex a design of code, the harder it is to change and improve that code (and thus are more likely to have bugs).

On the other hand, the process metrics community [10,50,58,63,78,39,16] explore many process metrics including (a) developer's experience; and (b) how many developers worked on certain file (and, it is argued, many developers working on a single file is much more susceptible to defects); and (c) how long it has been since the last change (and, it is argued, a file which is changed frequently may be an indicator for bugs).

The rest of this section lists prominent results from the Figure 1 survey. From the product metrics community, Zimmermann et al. [91], in their study on Eclipse project using file and package level data, showed complexity based product metrics are much better in predicting defective files. Zhang et al. [87] in their experiments showed that lines of code related metrics are good predictors of software defects using NASA datasets. In another study using product metrics, Zhou et al. [90] analyzed a combination of ten object-oriented software metrics related to complexity to conclude that size metrics were a much better indicator of defects. A similar study by Zhou and Leung et al. [89]

evaluated importance of individual metrics and indicated while CBO, WMC, RFC, and LCOM metrics are useful metrics for fault prediction, but DIT is not useful using NASA datasets. Menzies et al. [43] in their study regarding static code metrics for defect prediction found product metrics are very effective in finding defects. Basili et al. [7] in their work showed object-oriented ck metrics appeared to be useful in predicting class fault-proneness, which was later confirmed by Subramanyam and Krishnan et al. [70]. Nagappan et al. [51] in their study reached similar conclusion as Menzies et al. [43], but concluded "However, there is no single set of complexity metrics that could act as a universally best defect predictor" .

In other studies related to process metrics, Nagappan et al. [52] emphasised the importance of change bursts as a predictor for software defects on Windows Vista dataset. They achieved a precision and recall value at 90% in this study and achieved precision of 74.4% and recall at 88.0% in another study on Windows Server 2003 datasets. In another study by Matsumoto et al. [40] investigated the effect of developer related metrics on defect prediction. They showed improved performance using these metrics and proved module that is revised by more developers tends to contain more faults. Similarly, Schröte et al. [38] in their study showed high correlation number of developers for a file and number of defects in the respective file.

As to the four papers that compare process versus product methods:

– Three of these paper argue that process metrics are best. Rahman et al. [60] found process metrics perform much better than product metrics in both within project and cross project defect prediction setting. Their study also showed product metrics do not evolve much over time and that they are much more static in nature. Hence, they say, product metrics are not good predictors for defects. Similar conclusions (about the superiority of process metrics) are offered by Moser et al. [48] and Graves et al. [24].

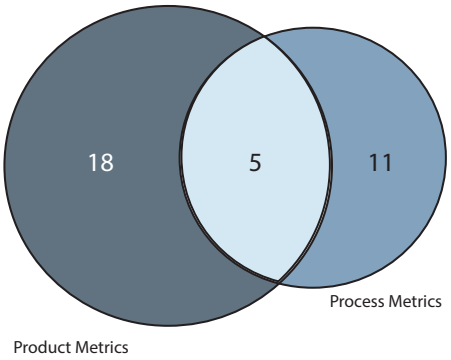

Fig. 1: Number of papers exploring the benefits of process and product metrics for defect prediction. The papers in the intersection are [60, 48, 24, 6] explore both process and product metrics.

– Only one paper argues that product metrics are best. Arisholm et al. [6] found one project where product metrics perform better.
– Only one paper argues that the combination of process and product metrics are better. Kamei et al. [31] found 5 out of 9 versions of 3 project where combination of process and product metrics perform better then just using process metrics and 9 out of 9 cases they are better than just using product metrics.

Of these papers, Moser et al. [48], Arisholm et al. [6], Rahman et al. [60] and Graves et al. [24] based their conclusions on 1,1,12,15 projects (respectively). That is to say, these are all analytics in-the-small studies. The rest of this paper checks their conclusions using analytics in-the-large.

3 Methods

This section describes our methods for comparatively evaluating process-versus-product metrics using analytics in-the-large.

3.1 Data Collection

To collect data, we search Github for Java projects from different software development domains. Although Github stores millions of projects, many of these are trivially very small, not maintained or are not about -software development projects. To filter projects, we used the standard Github "sanity checks" recommended in the literature [29,49]:

– *Collaboration*: refers to the number of pull requests. This is indicative of how many other peripheral developers work on this project. We required all projects to have at least one pull request.
– *Commits*: The project must contain more than 20 commits.
– *Duration*: The project must contain software development activity of at least 50 weeks.
– *Issues*: The project must contain more than 8 issues.
– *Personal Purpose*: The project must not be used and maintained by one person. The project must have at least eight contributors.
– *Software Development*: The project must only be a placeholder for software development source code.
– *Defective Commits*: The project must have at least 10 defective commits with defects on Java files.

These sanity checks returned 700 projects. For each project the data was collected in the following three steps. Note that steps one and three required 2 days (on a single 16 cores machine) and 7 days (or 5 machines with 16 cores), respectively.

1. We collected the process data for each file in each commit by extracting the commit history of the project, then analyzing each commit for our metrics. We used a modified version of Commit_Guru [65] code for this purpose.

Product Metrics					
Metric Name	Median	IQR	Metric Name	Median	IQR
AvgCyclomatic	1	1	CountLine	75.5	150
AvgCyclomaticModified	1	1	CountLineBlank	10.5	20
AvgCyclomaticStrict	1	1	CountLineCode	53	105
AvgEssential	1	0	CountLineCodeDecl	18	32
AvgLine	9	10	CountLineCodeExe	29	66
AvgLineBlank	0	1	CountLineComment	5	18
AvgLineCode	7	8	CountSemicolon	24	52
AvgLineComment	0	1	CountStmt	35	72.3
CountClassBase	1	0	CountStmtDecl	15	28
CountClassCoupled	3	4	CountStmtExe	19	43.8
CountClassCoupledModified	3	4	MaxCyclomatic	3	4
CountClassDerived	0	0	MaxCyclomaticModified	2	4
CountDeclClassMethod	0	0	MaxCyclomaticStrict	3	5
CountDeclClassVariable	0	1	MaxEssential	1	0
CountDeclInstanceMethod	4	7.5	MaxInheritanceTree	2	1
CountDeclInstanceVariable	1	4	MaxNesting	1	2
CountDeclMethod	5	9	%LackOfCohesion	33	71
CountDeclMethodAll	7	12.5	%LackOfCohesionModified	19	62
CountDeclMethodDefault	0	0	RatioCommentToCode	0.1	0.2
CountDeclMethodPrivate	0	1	SumCyclomatic	8	17
CountDeclMethodProtected	0	0	SumCyclomaticModified	8	17
CountDeclMethodPublic	3	6	SumCyclomaticStrict	9	18
SumEssential	6	11			
Process Metrics			Data Statistics		
Metric Name	Median	IQR	Data Property	Median	IQR
la	14	38.9	Defect Ratio	37.60%	20.60%
ld	7.9	12.2	Lines of Code	82K	200K
lt	92	121.8	Number of Files	171	358
age	28.8	35.1	Number of Developers	31	34
ddev	2.4	1.2			
nuc	5.8	2.7			
own	0.9	0.1			
minor	0.2	0.4			
ndev	22.6	22.1			
ncomm	71.1	49.5			
adev	6.1	2.9			
nadev	71.1	49.5			
avg_nddev	2	1.8			
avg_nadev	7	5.2			
avg_ncomm	7	5.2			
ns	1	0			
exp	348.8	172.7			
sexp	145.7	70			
rexp	2.5	3.4			
nd	1	0			
sctr	-0.2	0.1			

Table 4: Statistical median and IQR values for the metrics used in this study.

While going through each commit, we create objects for each new file we encounter and keep track of details (i.e. developer who worked on the file, LOCs added, modified and deleted by each developer, etc.) that we need to calculate. We also keep track of files modified together to calculate co-commit based metrics.

2. Secondly we use Commit_Guru [65] code to identify commits which have bugs in them. This process involves identifying commits which were used to fix some bugs using a keyword based search. Using these commits the

process uses SZZ algorithm [79] to find commits which were responsible for introducing those changes and marking them as buggy commits[6].
3. Thirdly, we used Github release API to collect the release information for each of the projects. We use the release number, release date information supplied from the API to group commits into releases and thus dividing each project the into multiple releases for each of the metrics.
4. Finally we used the Understand from Scitools[7] to mine the product metrics. Understand has a command line interface to analyze project codes and generate metrics from that. We use the data collected from first 2 steps to generate a list of commits and their corresponding files along with class labels for defective and non-defective files. Next we download the project codes from Github, then used the `git commit` information to move the git head to the corresponding commit to match the code for that commit. Understand uses this snapshot of the code to analyze the metrics for each file and stores the data in a temporary storage. We do this for each commit from the last step. To ensure for every analyzed commit we only consider the files which were changed, we only keep files which was either changed as part of that commit or was changed in a defective child commit. Here we also added the class labels to the metrics. In order to only mark files which were defective, we use commit Ids along with file names to add labels.

The data collected in this way is summarized in Table 4.

3.2 Learners

In this section, we briefly explain the four classification methods we have used for this study. We selected the following based on a prominent paper by Ghotra et al.'s [23]. Also, all these learners are widely used in the software engineering community. For all the following models, we use the implementation from Scikit-Learn[8]. Initially, we thought we'd need to apply hyperparameter optimization [72] to tune these learners. However, as shown below, the performance of the default parameters was so promising that we left such optimization for future work.

3.2.1 ***Support Vector Machine***

This is a discriminative classifier, which tries to create a hyper-plane between classes by projecting the data to a higher dimension using kernel tricks [66, 13, 74, 44]. The model learns the separating hyper-plane from the training data and classifies test data based on which side the example resides.

[6] From this point on-wards we will denote the commit which has bugs in them as a "buggy commit"

[7] http://www.scitools.com/

[8] https://scikit-learn.org/stable/index.html

3.2.2 ***Naive Bayes***

This is a probabilistic model, widely used in software engineering community [77,71,45,67,68], that finds patterns in the training dataset and build predictive models. This learner assumes all the variables used for prediction are not correlated, identically distributed. This classifier uses Bayes rules to build the classifier. When predicting for test data, the model uses the distribution learned from training data to calculate the probability of the test example to belong to each class and report the class with maximum probability.

3.2.3 ***Logistic Regression***

This is a statistical predictive analysis method similar to linear regression but uses a logistic function to make prediction. Given 2 classes Y=(0 or 1) and a metric vector $X = x_1, x_2, ..., x_n$, the learner first learns coefficients of each metrics vectors to best match the training data. When predicting for test examples it uses the metrics vectors of the test example and the coefficients learned from training data to make the prediction using logistic function. Logistic regression is widely used in defect prediction [23,85,25,54,56].

3.2.4 ***Random Forest***

This is a type of ensemble learning method, which consists of multiple classification decision trees build on random metrics and bootstrapped samples selected from the training data. Test examples are classified by each decision tree in the Random Forest and then the final decision on classification is decided using a majority voting. Random forest is widely used in software engineering domain [73,86,28,88,27,77,36] and has proven to be effective in defect prediction.

Later in this paper, the following distinction will become very significant. Of the four learners we apply, Random Forests make their conclusion via a majority vote across *multiple models* while all the other learners build and apply a *single model*.

3.3 Experimental Framework

Figure 2 illustrates our experimental rig. For each of our 700 selected Java projects, first, we use the revision history of the project to collect file level change metrics, along with class labels (defective and non-defective commits). Then, using information from the process metrics, we use Understand's command line interface to collect and filter the product metrics. Next, we join the two metrics to create a combined metric set for each project.

Using the evaluation strategy mentioned above, the data is divided into train and test sets. The data is then filtered depending on metrics we are interested in (i.e. process, product or combined) and pre-processed (i.e. data

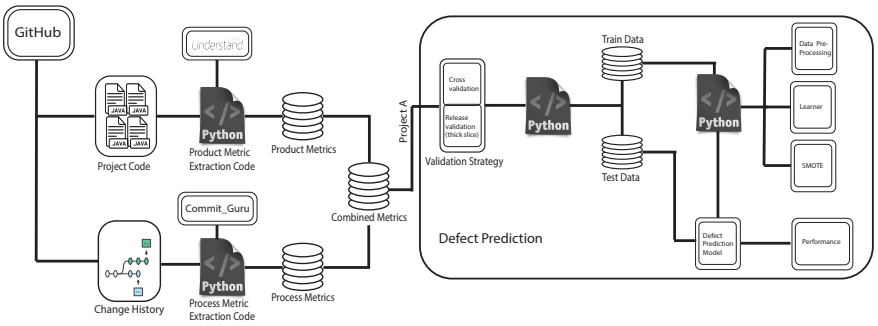

Fig. 2: Framework for this analysis.

normalization, filtering/imputing missing values etc). After pre-processing and metric filtering is completed the data is processed using SMOTE algorithm to handle data imbalance. As described by Chawla et al. [14], SMOTE is useful for re-sampling training data such that a learner can find rare target classes. For more details in SMOTE, see [14,3]. Note one technical detail: when applying SMOTE it is important that it is *not* applied to the test data since data mining models need to be tested on the kinds of data they might actually see in practice.

Finally, we select one learner from four and is applied to the training set to build a model. This is then applied to the test data. As to how we generate our train/test sets, we report results from two methods:

1. *released based* and
2. *cross-validation*

Both, these methods are defined below. We use both methods this since (a) other software analytics papers use *cross-validation* while (b) *released based* is the evaluation procedure of Rahman et al. As we shall see, these two methods offer very similar results so debates about the merits of one approach to the other are something of a moot point. But by reporting on results from both methods, it is more likely that other researchers will be able to compare their results against ours.

In a *cross-validation study*, all the data for each project is sorted randomly M times. Then for each time, the data is divided into N bins. Each bin, in turn, becomes the test set and a model is trained on the remaining four bins. For this study, we used $M = N = 5$.

An alternative to cross-validation is a *release-based approach* such as the one used by Rahman et al. Here, given R releases of the software, we trained on data from release 1 to $R-3$, then tested on release $R-2$,$R-1$ and R. This temporal approach has the advantage that that future data never appears in the training data.

3.4 Evaluation Criteria

In this section, we introduce the following 6 evaluation measures used in this study to evaluate the performance of machine learning models. Based on the results of the defect predictor, humans read the code in order of what the learner says is most defective. During that process, they find true negative, false negative, false positive, and true positive (labeled TN, FN, FP, TP respectively) reports from the learner.

Recall: This is the proportion of inspected defective changes among all the actual defective changes; i.e. TP/(TP+FN). Recall is used in many previous studies [33, 75, 84, 82, 81, 83]. When recall is maximal, we are finding all the target class. Hence we say that *larger* recalls are *better*.

Precision: This is the proportion of inspected defective changes among all the inspected changes; i.e. TP/(TP+FP). When precision is maximal, all the reports of defect modules are actually buggy (so the users waste no time looking at results that do not matter to them). Hence we say that *larger* precisions are *better*.

Pf: This is the proportion of all suggested defective changes which are not actual defective changes divided by everything that is not actually defective; i.e. FP/(FP+TN). A high *pf* suggests developers will be inspecting code that is not buggy. Hence we say that *smaller* false alarms are *better*.

Popt20: A good defect predictor lets programmers find the most bugs after reading the least amount of code[5]. **Popt20** models that criteria. Assuming developers are inspecting the code in the order proposed by the learner, it reports what percent of the bugs are found in the first 20% of the code. We say that *larger* Popt20 values are *better*.

Ifa: Parnin and Orso [57] warn that developers will ignore the suggestions of static code analysis tools if those tools offer too many false alarms before reporting something of interest. Other researchers echo that concern [57, 34, 80]. **Ifa** counts the number of initial false alarms encountered before we find the first defect. We say that *smaller* IFA values are *better*.

AUC_ROC: This is the area under the curve for receiver operating characteristic. This is designated by a curve between true positive rate and false positive rate and created by varying the thresholds for defects between 0 and 1. This creates a curve between (0,0) and (1,1), where a model with random guess will yield a value of 0.5 by connecting (0,0) and (1,1) with a straight line. A model with better performance will yield a higher value with a more convex curve in the upper left part. Hence we say that *larger* AUC values are *better*.

3.5 Statistical Tests

When comparing the results of different models in this study, we used a statistical significance test and an effect size test:

– Significance test is useful for detecting if two populations differ merely by random noise.
– Effect sizes are useful for checking that two populations differ by more than just a trivial amount.

For the significance test, we use the Scott-Knott procedure recommended at TSE'13 [46] and ICSE'15 [23]. This technique recursively bi-clusters a sorted set of numbers. If any two clusters are statistically indistinguishable, Scott-Knott reports them both as belonging to the same "rank".

To generate these ranks, Scott-Knott first looks for a break in the sequence that maximizes the expected values in the difference in the means before and after the break. More specifically, it splits l values into sub-lists m and n in order to maximize the expected value of differences in the observed performances before and after divisions. For e.g., lists l, m and n of size ls, ms and, ns where $l = m \cup n$, Scott-Knott divides the sequence at the break that maximizes:

$$E(\Delta) = \frac{ms}{ls} \times abs(m.\mu - l.\mu)^2 + \frac{ns}{ls} \times abs(n.\mu - l.\mu)^2 \quad (1)$$

Scott-Knott then applies some statistical hypothesis test H to check if m and n are significantly different. If so, Scott-Knott then recurses on each division. For this study, our hypothesis test H was a conjunction of the A12 effect size test (endorsed by [4]) and non-parametric bootstrap sampling [18], i.e., our Scott-Knott divided the data if *both* bootstrapping and an effect size test agreed that the division was statistically significant (90% confidence) and not a "small" effect ($A12 \geq 0.6$).

4 RESULTS

RQ 1: For predicting defects, do methods that work in-the-small, also work in-the-large?

To answer this question, we use Figure 3 and Figure 4 and Figure 5 to compares Recall, Pf, AUC, Popt20, Precision and IFA across four different learners using process, product and combined metrics. In those figures, the metrics are marked as P (process metrics), C (product metrics) and combined (P+C).

For this research question, the key thing to watch in these figure are the vertical colored boxes with a horizontal line running across their middle. This horizontal lines shows the median performance of a learner across 700 Github projects. As we said above in section 3.4, the best learners are those that *maximize* recall, precision, AUC, Popt20 while *minimizing* IFA and false alarms.

Reading the median line in the box plots, we say that compared to the Rahman et al. analytics in-the-small study, this analytics in-the-large study is saying some things are the same and some things that are different. Like Rahman et al., these results show clear evidence of the superiority of process metrics since, with the exception of AUC, across all learners, the median process results from process metrics are always clearly better. That is to say,

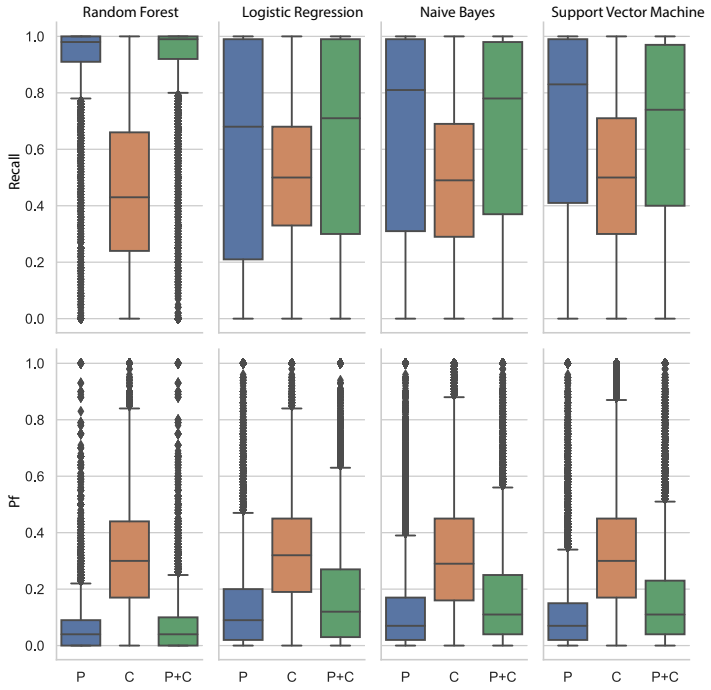

Fig. 3: Cross-validation recall and false alarm results for Process(P), Product(C) and, Combined (P+C) metrics. The vertical box plots in these charts run from min to max while the thick boxes highlight the 25,50,75th percentile.

returning to our introduction, this study strongly endorses the Hersleb hypothesis that how we build software is a major determiner of how many bugs we inject into that software.

As to where we differ from the prior analytics in-the-small study, with the exception of AUC, the median results of Random Forest are clearly better than anything else. That is, unlike the Rahman et al. analytics in-the-small study, we would argue that it is very important which learner is used to for analytics in-the-large. Certain learning in widespread use such as Naive Bayes, Logistic Regression, and Support Vector Machines may not be the best choice for reasoning from hundreds of software projects. Rather, we would recommend the use of Random Forests.

Before going on, we comment on three more aspects of these results. Firstly, we see no evidence of any added value of combining process and product metrics. If we compare the (P+C) results to the (P) results, there is no case in Figure 3 and Figure 4 and Figure 5 where process but product metrics do better than just using process metrics.

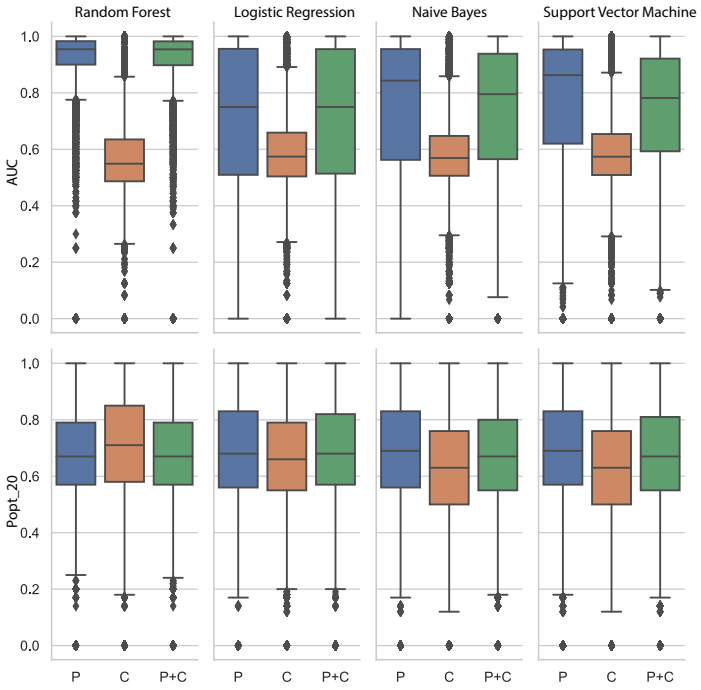

Fig. 4: Cross-validation AUC and Popt20 results for Process(P), Product(C) and Combined (P+C) metrics. Same format as Figure 3.

Secondly, similar to Kamei et al. in case of effort aware evaluation criteria process metrics are superior to product metrics, as can be seen in Figure 5. Note in that figure, many of our learners using process metrics have near zero IFA scores. This is to say that, using process metrics, programmers will *not* be bombarded with numerous false alarms. But unlike Kamei et al. we do not see any significant benefit when accessing the performance in regards to the Popt20, which is another effort aware evaluation criteria used by Kamei et al. and this study.

Thirdly, Figure 6 shows the Random Forest results using release based test sets. As stated in section 3.4 above, there is very little difference in the results between release based test generation and the cross-validation method of Figure 3 and Figure 4 and Figure 5. Specifically, in both our cross-val and thick-slice results, (a) process metrics do best; (b) there is no observed benefit in adding in product metrics and, when using process metrics then random forests have (c) very high precision and recall and AUC, (d) low false alarms; and (e) very low IFA.

RQ 2: Measured in terms of predication variability, do methods that works well in-the-small, also work at at-scale?

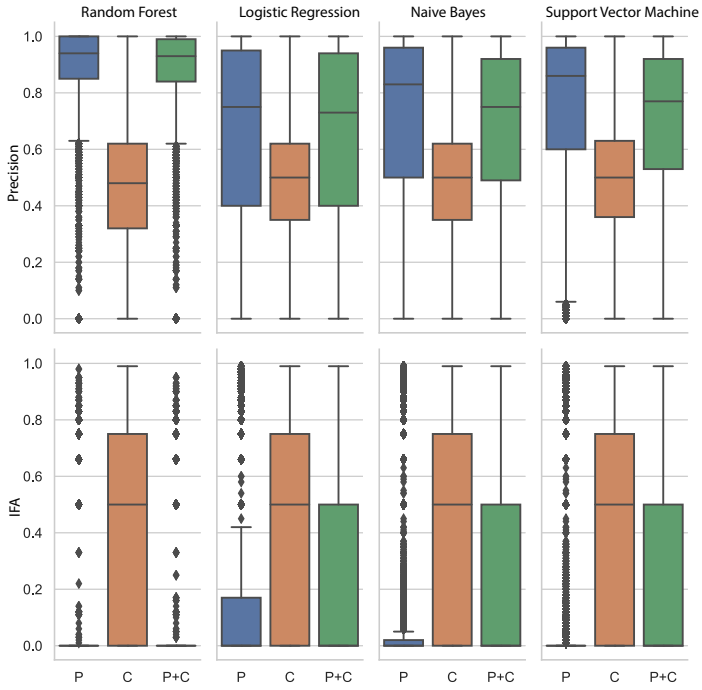

Fig. 5: Cross-validation IFA and precision results for Process(P), Product(C) and Combined (P+C) metrics. Same format as Figure 3.

To answer this research question, we assess our learners not by their median performance, but by there variability.

Rahman et al. commented that many different learners might be used for defect prediction since, for the most part, they often give the same results. While that certainly holds for their analytics in-the-small case study, the situation is very different when reasoning at-scale about 700 projects. Looking at the process metrics results for Figure 3 and Figure 4 and Figure 5, we see that

1. The median performance for random forests is much better than the other learners studied here.
2. With the exception of AUC, the box plots for random forests are much smaller than for other learners. That is, the variance in the predictive perform is much smaller for random forests than for anything else in this study.

The size of both these effects is quite marked. Random forests are usually 20% better (median) than logistic regression. As to the variance, the random forest variance is three to four times *smaller* than the other learners.

Why is Random Forest doing so well? We conjecture that when reasoning about 700 projects that there are many spurious effects. Since Random Forests make their conclusions by reasoning across multiple models, this kind

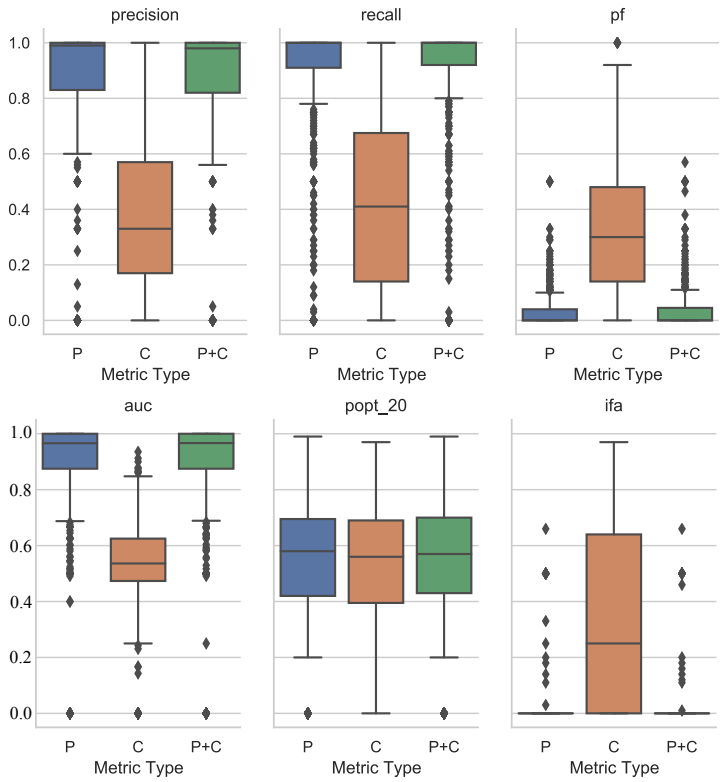

Fig. 6: Release based results for Random Forests. Here the training data was till t-3 th release and rest was test release.

of learner can avoid being confused. Hence, we recommend ensemble methods like random forests for analytics in-the-large.

RQ 3: Measured in terms of granularity, do same granularity that works well in-the-small, also work at at-scale?

In this research question, we try to evaluate if granularity of the metrics matters when predicting for defect when measuring at scale. This is one of the research questions asked in study by Kamei et al. . Here we try to measure if package level prediction is better at identifying defective packages than file level prediction at identifying defective files. There are multiple strategies for create package level metrics such as lifting file level metrics to package level, collecting metrics designed for package level and lifting file level prediction results for package level as explored by Kamei et al. in their study. We explore the first strategy that is to lift the file level metrics to package level. We select this strategy as Kamei et al. in their paper has shown the metrics designed for package level doesn't product good results and both file and result lift ups has similar performance and has been explored by many other researchers. To

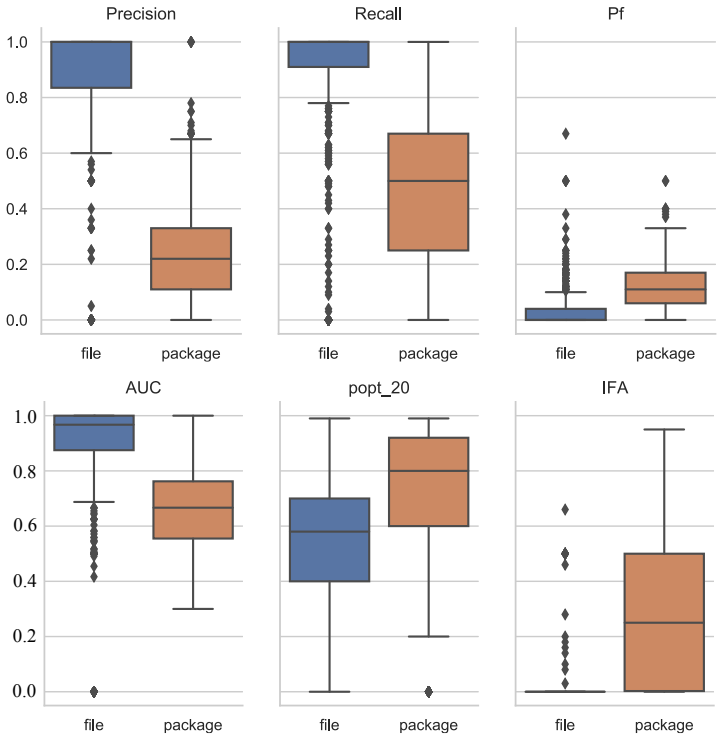

Fig. 7: File vs package level prediction for models built using file level process data and package level process data.

build a defect predictor using package level data, we use the process metrics collected for our tasks. For each commit/release if there is multiple files from same package, we aggregate them to their package level by taking the median values.

Figure 7 show the difference in performance between file level prediction results and package level prediction results. It is evident from the results, that file level prediction shows statistically significant improvement than package level prediction. This result agrees with Kamei et al. and we conclude that the granularity of the metric set does matter and file level level prediction has superior performance than package level prediction.

RQ 4: Measured in terms of stability, are process metrics more/less stable than code metrics, when measured at at-scale?

To answer this research question, we first tag each commit into a release by using the release information from Github. Using this release information, we divide the data into train and test data by using the last 3 releases as test release one by one and other older releases as training data. If a model build using either process and product data significantly differ across last 3 releases

that would imply the model built using that metric will need to be rebuild for each subsequent releases, this in-tern will create instability. To verify the stability of the models built using metrics, we build the models using the training data and then check each of the 3 subsequent releases. We compare both process and product metrics across all 6 criteria mentioned in Section 3.4.

Figure 8 shows the performance of the models. The first row of the figure represents the process metrics, while the second row represents the product metrics. Each column represents the evaluation criteria that we are measuring and inside each plot each box plot represents one of the last 3 releases. We applied scott-knott statistical test on the results to check for each evaluation criteria if any of the releases are statistically significantly different than the others. The results shows no significant difference between 3 releases in all evaluation criteria (all releases for each evaluation criteria in each each metric type) except Popt20. Popt20 is an effort aware criteria as explained in Section 3.4, and we see in both process based and product based models the Popt20 does significantly better in the third release. This result shows none of the models build using process and product metrics degrades over time, thus reducing the instability of the models. We can also say, as over time the performance doesn't degrade and we have already seen in-term of performance process metrics performs much better then product metrics, it is wiser to use process metrics in predicting defects.

RQ 5: Measured in terms of stasis, Are process metrics more/less static than code metrics, when measured at at-scale?

In this research question, we try to find the reason behind difference in performance in models build using process and product data. Most models try to learn how to differentiate between two classes by learning the pattern in the training data and tries to identify similar patterns in the test data to predict. Throughout the life cycle of a project, different parts of the project is updated and changed as part of regular enhancements. This results in introduction of bugs and thus bug fixes for those defective changes. The metrics that we use to create the defect prediction models should be able to reflect those changes, so the model is able to identify the difference between defective and non-defective changes. This basically means if either process or product metrics can capture such differences, then the metric values for a file between release R and $R+1$ should not be highly correlated.

To measure the stasis of the metrics, we used Spearman correlation for every file between two consecutive releases (to check releases based prediction) and two consecutive commits (to check for JIT based predictions). Figure 9 shows the Spearman correlation values for every file between two consecutive releases for all the projects explored as a violin plot for each type of metrics. A wide and short violin plot represents majority value concentrated near a certain value, while a thin and long violin plot represents values being in different range. Figure 9 shows the correlation scores for process and product metrics in both release based and JIT based setting. The process and product metric in release based setting is denoted by P_R and C_R respectively while in

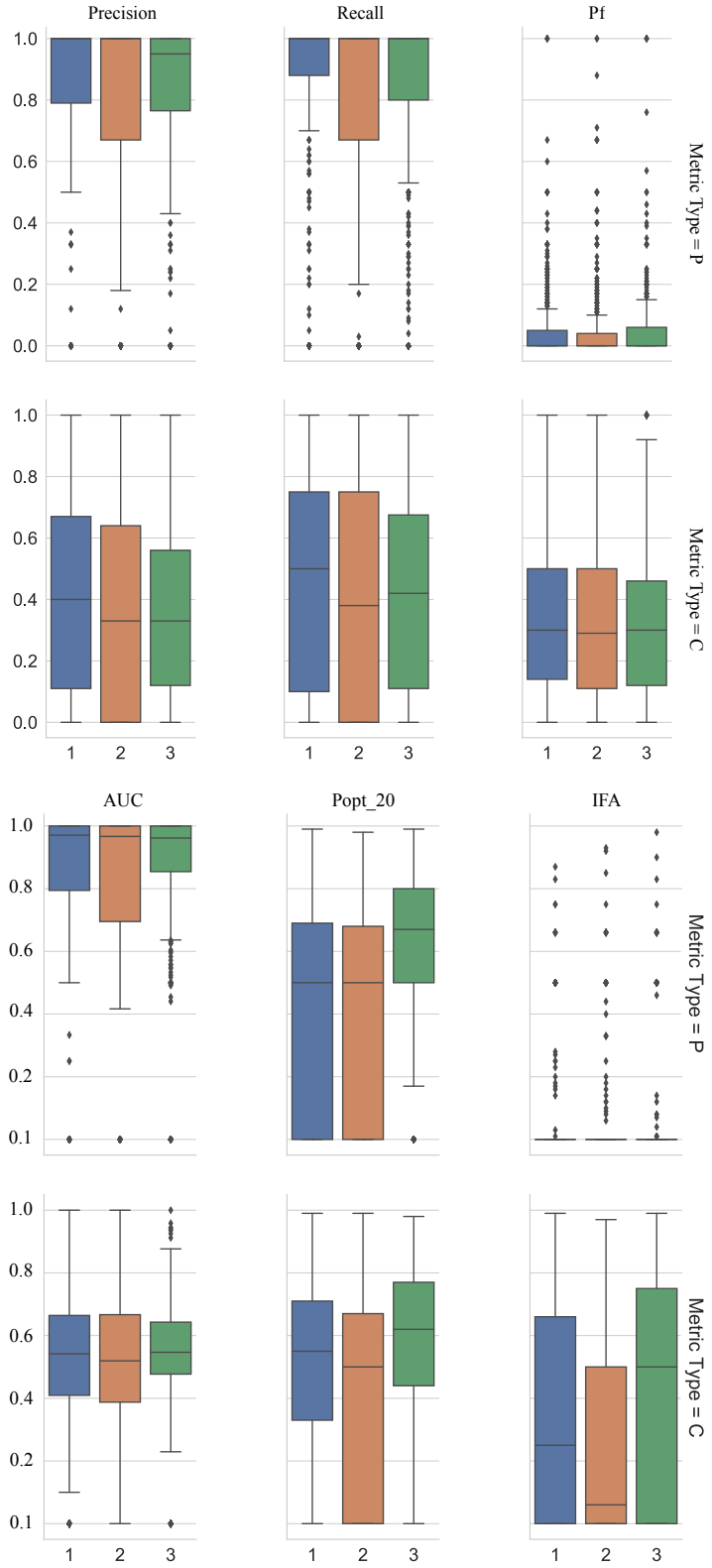

Fig. 8: Stability of the models across last 3 releases built using process (P) and product (C) metrics. Each plot show the one of the six performance criteria used in this study for last 3 releases. First row shows the results for process metric denoted as *Metric Type = P* and the second row show the results for product metric denoted as *Metric Type = C*.

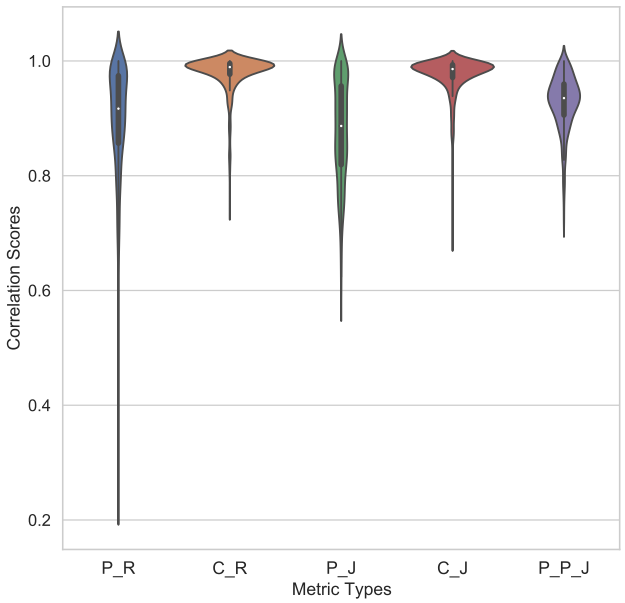

Fig. 9: The plot represents the Spearman correlation of every file between two consecutive check points. Here x-axis label P_R and C_R represents the process and product metrics when the correlation was calculated in release level. While the P_J, C_J and P_P_J represents the process, product and package level process metrics when calculated in JIT based setting.

JIT based setting they are denoted by P_J and C_J respectively. In the figure the P_P_J represents the package level process metric in JIT based setting .We can see from figure 9, the product metrics forms a wide and short violin plot and are very highly correlated. While the process metrics forms a thin and long violin plot ranging between 0.2 to 1 for release based setting and 0.5 to 1 for JIT based setting. If we compare the correlations between release based and JIT based metric sets, we see the correlation value for process metrics increases in JIT based metric sets. The reason behind this increase in correlation value can be explained as in JIT based metrics, we compare between each commits, thus the changes in file are less than the changes when measured between two releases (here each releases contains multiple commits). Similarly we see when process metric has been lifted from file level to package level, the correlation increases.

So why process metrics out perform product metrics? We think the stasis property of the metric set is one of the main reason. As product metric seems

Table 5: Spearman rank correlation between learned and predicted probability on models built using different metrics

Metrics	correlation value	p -value
Process	0.24	<0.001
Product	0.53	<0.001

to be more static, thus changing very little with time and between defective file and non-defective files. When models are created with such static metric sets, it is hard for the model to learn a pattern and differentiate between defective and non-defective changes. While process metrics changes over time, and much less correlated between changes, thus making them a potentially better metric for creating defect prediction models.

RQ 6: Measured in terms of stagnation, Do models built from different sets of metrics stagnate across releases, when measured at at-scale?

In this research question, we try to measure the stagnation property of the models built using the process and product metrics. As suggested by Rahman et al. , we use Spearman rank correlation between the learned probability from the training set and predicted probability from the test set to calculate the correlation between these two. Here if we find a high correlation between the learned and predicted probability, that will indicate the models are probability learning to predict the same set of files defective and finding the same defect percentage in the test set as training set and it is not able to properly differentiate between defective and non-defective files. Table 5 shows the Spearman rank correlation between the learned and predicted probability for models built using process and product metrics. We can see for model built using product data has significantly higher correlation than model built using process data. although this value is significantly higher, both in case of process and product metrics the correlation value is lower than what Rahman et al. reported in their project.

RQ 7: Do stagnant models (based on stagnant metrics) tend to predict recurringly defective entities?

Here we try to verify the stagnation property of the metrics as seen in the previous research question. If a model is stagnant, it will predict the same file as defective regardless of whether the file actually contains defect or not. To evaluate whether or not model build on process and product data is predicting same files as defective, we follow the same approach suggested by Rahman et al. . We divide the test data into 3 parts (a) part 1 only contains files which are defective in both training and test set, we call this recurrent set (b) part 2 consists of files which are defective in training set but not in test set, this is train only set and finally (c) part 3 only contains files which are defective in training and not in test set, we call this test only set. A model if stagnant will have high recall for recurrent set, high pf for train only set and low recall for

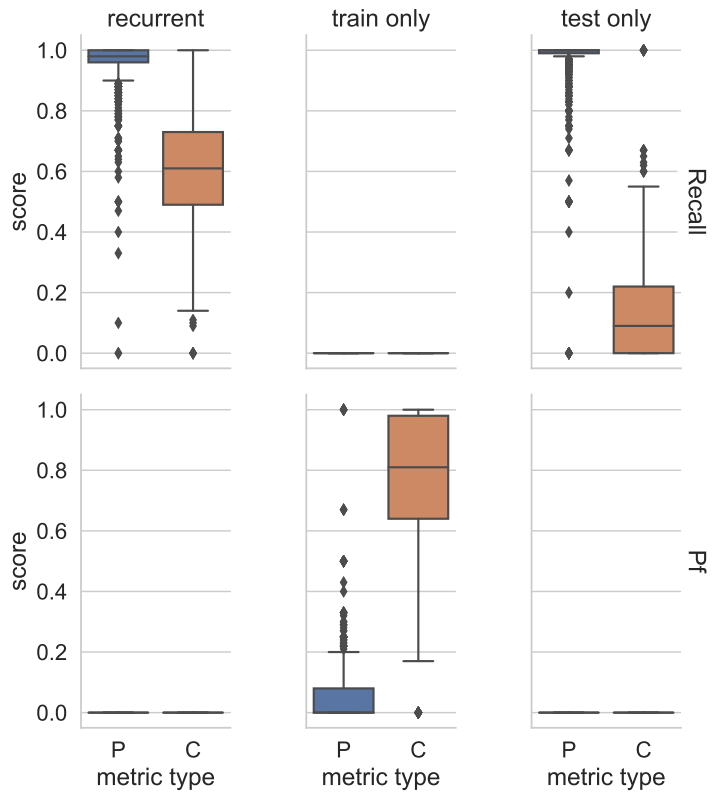

Fig. 10: Performance of the models build using process and product metrics on recurrent, train only and test only test sets.

test only set and that will show the model is actually predicting the same set of files as defective and not able to identify new defective files. Figure 10 shows the recall and pf of the models build using process and product metrics on all 3 types of test set. We can see from the figure in case of recurrent set, models built using both process and product metrics is able to identify recurrently defective files. Although we can see a significant difference between process and product metrics, where process metric is doing much better in recognizing recurrently defective files. In case of train only test set we can see very high pf (median value $\approx$ 0.8) for model build using product data, while for the model built using process data has low pf (median value $\approx$ 0.0). This is a clear indication that model built using product metric is stagnate and identify same set of files as defective regardless whether they are actually defective or not. while the test only set shows a very low recall for model built using product data, while high recall for model built using process data. This indicates model built using product data is unable to identify new defects. Thus this result bolster the claim that process metrics are better at identifying defects than product metrics.

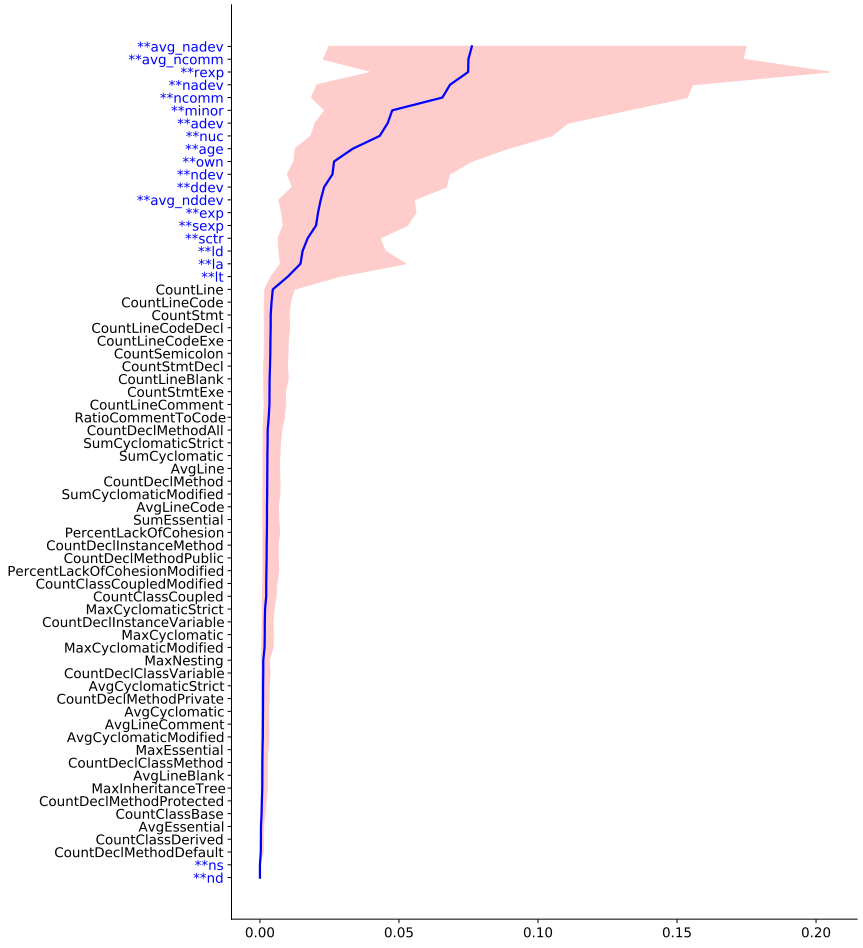

Fig. 11: Metric importance of process+product combined metrics based on Random Forest. Process metrics are marked with two blue asterisks. Blue denotes the median importance in 700 projects while the pink region shows the (75-25)th percentile.**

RQ 8: Measured in terms of metric importance, are metrics that seem important in-the-small, also important when reasoning in-the-large?

To answer this question, we test if what is learned from studying *some* projects is the same as what might be learned from studying *all* 700 projects. That is, we compare the rankings given to process metrics using all the projects (analytics in-the-large) to the rankings that might have been learned from 20 analytics in-the-small projects looking at 5 projects each (where those projects were selected at random).

Figure 11 shows the metric importance of metrics in the combined (process + product) data set. This metric importance is generated according to what metrics are important while building and making prediction in Random Forest. The metric importance returned by Random Forest is calculated using a method implemented in Scikit-Learn. Specifically: how much each metric decreases the weighted impurity in a tree. This impurity reduction is then averaged across the forest and the metrics are ranked. In Figure 11 the metric importance increases from left to right. That is, in terms of defect prediction, the most important metric is the average number of developers in co-committed files (avg_nadev) and the least important metric is the number of directories (nd).

In that figure, the process metrics are marked with two blue asterisks**. Note that nearly all of them appear on the right hand side. That is, in a result consistent with Rahman et al., process metrics are far more important than process metrics.

Figure 12 compares the process metric rankings learned from analytics in-the-large (i.e. from 700 projects) versus a simulation of 20 analytics in-the-small studies that look at five projects each, selected at random. In the figure, the X-axis ranks metrics via analytics in-the-large (using Random Forests applied to 700 projects) and Y-axis ranks process metrics using analytics in-the-small (using Logistic Regression and 20*5 projects). For the Y-axis rankings, the metrics were sorted by the absolute value of their β-coefficients within the learned regression equation.

In an ideal scenario, when the ranks are the same, this would appear in Figure 12 as a straight line at a 45-degree angle, running through the origin. To say the least, this *not* what is observed here. We would summarize Figure 12 as follows: the importance given to metrics by a few analytics in-the-small studies is very different to the importance learned via analytics in-the-large.

5 THREATS TO VALIDITY

As with any large scale empirical study, biases can affect the final results. Therefore, any conclusions made from this work must be considered with the following issues in mind:

(a) *Evaluation Bias*: In all research questions in this study, we have shown the performance of models build with process, product and, process+product metrics and compared them using statistical tests on their performance to make a conclusion about which is a better and more generalizable predictor for defects. While those results are true, that conclusion is scoped by the evaluation metrics we used to write this paper. It is possible that using other measurements, there may be a difference in these different kinds of projects (e.g. G-score, harmonic mean of recall and false-alarm reported in [76]). This is a matter that needs to be explored in future research.

(b) *Construct Validity*: At various places in this report, we made engineering decisions about (e.g.) choice of machine learning models, selecting metric

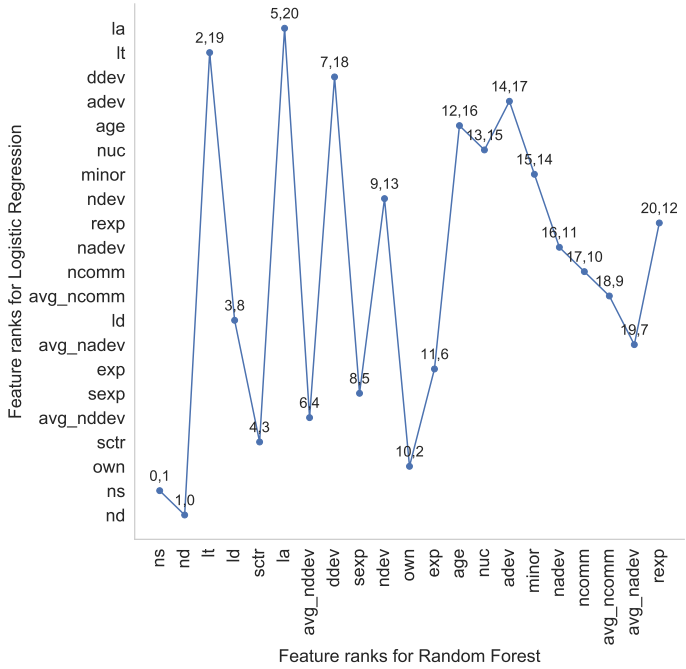

Fig. 12: X-axis ranks metrics via analytics in-the-large (using Random Forests applied to 700 projects). Y-axis ranks process metrics using analytics in-the-small (using Logistic Regression and 20*5 projects).

vectors for each project. While those decisions were made using advice from the literature, we acknowledge that other constructs might lead to different conclusions.

(c) *External Validity*: For this study, we have collected data from 700 Github Java projects. The product metrics collected for each project were done using a commercialized tool called "Understand" and the process metrics were collected using our own code on top of Commit_Guru repository. There is a possibility that calculation of metrics or labeling of defective vs non-defective using other tools or methods may result in different outcomes. That said, the "Understand" is a commercialized tool which has detailed documentation about the metrics calculations and we have shared our scripts and process to convert the metrics to a usable format and has described the approach to label defects.

We have relied on issues marked as a 'bug' or 'enhancement' to count bugs or enhancements, and bug or enhancement resolution times. In Github, a bug or enhancement might not be marked in an issue but in commits. There is also a possibility that the team of that project might be using different tag identifiers for bugs and enhancements. To reduce the impact of this problem, we did take

precautionary steps to (e.g.,) include various tag identifiers from Cabot et al. [12]. We also took precaution to remove any pull merge requests from the commits to remove any extra contributions added to the hero programmer.

(d) *Sampling Bias*: Our conclusions are based on the 700 projects collected from Github. It is possible that different initial projects would have lead to different conclusions. That said, this sample is very large so we have some confidence that this sample represents an interesting range of projects.

6 CONCLUSION

Much prior work in software analytics has focused on in-the-small studies that used a few dozen projects or less. Here we checked what happens when we take specific conclusions, generated from analytics in-the-small, then review those conclusions using analytics in-the-large. While some conclusions remain the same (e.g. process metrics generate better predictors than process metrics for defects), other conclusions change (e.g. learning methods like logistic regression that work well in-the-small perform comparatively much worse when applied in-the-large).

Issues that may not be critical in-the-small become significant problems in-the large. For example, recalling Figure 12, we can say that what seems to be an important metric, in-the-small, can prove to be very unimportant when we start reasoning in-the-large. Further, when reasoning in-the-large, variability in predictions becomes a concern. Finally, certain systems issues seem unimportant in-the-small. But when scaling up to in-the-large, it becomes a critical issue that product metrics are an order of magnitude to harder to manage. We listed above one case study where the systems requirements needed for product metrics meant that, very nearly, we almost did not deliver scientific research in a timely manner.

Based on this experience, we have several specific and one general recommendations. Specifically, for analytics in-the-large, use process metrics and ensemble methods like random forests since they can better handle the kind of large scale spurious singles seen when reasoning from hundreds of projects.

More generally, the SE community now needs to revisit many of the conclusions previously obtained via analytics in-the-small.

7 Acknowledgments

This work was partially funded by NSF Grant #1908762.

References

1. A. Agrawal and T. Menzies. Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In *IST*. ACM, 2018.

2. Amritanshu Agrawal, Wei Fu, and Tim Menzies. What is wrong with topic modeling? and how to fix it using search-based software engineering. *Information and Software Technology*, 98:74–88, 2018.
3. Amritanshu Agrawal and Tim Menzies. "better data" is better than "better data miners" (benefits of tuning SMOTE for defect prediction). *CoRR*, abs/1705.03697, 2017.
4. Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1–10. IEEE, 2011.
5. E. Arisholm and L. C Briand. Predicting fault-prone components in a java legacy system. In *ESEM*. ACM, 2006.
6. Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.
7. Victor R Basili, Lionel C Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996.
8. C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *ISSRE*, 2009.
9. Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In *2009 IEEE 31st International Conference on Software Engineering*, pages 518–528. IEEE, 2009.
10. Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.
11. Lionel C Briand, VR Brasili, and Christopher J Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044, 1993.
12. Jordi Cabot, Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Belén Rolandi. Exploring the use of labels to categorize issues in open-source software projects. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 550–554. IEEE, 2015.
13. Yang Cao, Zhiming Ding, Fei Xue, and Xiaotao Rong. An improved twin support vector machine based on multi-objective cuckoo search for software defect prediction. *International Journal of Bio-Inspired Computation*, 11(4):282–291, 2018.
14. Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.
15. Di Chen, Wei Fu, Rahul Krishna, and Tim Menzies. Applications of psychological science for actionable analytics. *FSE'19*, 2018.
16. Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering*, 67:15–24, 2018.
17. Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.
18. Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. Mono. Stat. Appl. Probab. London, 1994.
19. Norman E Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 357–370, 2000.
20. Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 2016.
21. Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.
22. B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 37th ICSE*, 2015.

23. Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *37th ICSE-Volume 1*, pages 789–800. IEEE Press, 2015.
24. T. L Graves, A. F Karr, J. S Marron, and H. Siy. Predicting fault incidence using software change history. *TSE*, 2000.
25. Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.
26. James Herbsleb. Socio-technical coordination (keynote). In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 1, New York, NY, USA, 2014. Association for Computing Machinery.
27. Dyana Rashid Ibrahim, Rawan Ghnemat, and Amjad Hudaib. Software defect prediction using feature selection and random forest algorithm. In *2017 International Conference on New Trends in Computing Sciences (ICTCS)*, pages 252–257. IEEE, 2017.
28. Shomona Gracia Jacob et al. Improved random forest algorithm for software defect prediction through data mining techniques. *International Journal of Computer Applications*, 117(23), 2015.
29. Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.
30. Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.
31. Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.
32. Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken-ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 196–204. IEEE, 2007.
33. Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.
34. Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2016.
35. Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering*, 25(1):890–939, 2020.
36. Rahul Krishna and Tim Menzies. Bellwethers: A baseline method for transfer learning. *IEEE Transactions on Software Engineering*, 2018.
37. Markus Lumpe, Rajesh Vasa, Tim Menzies, Rebecca Rush, and Burak Turhan. Learning better inspection optimization policies. *International Journal of Software Engineering and Knowledge Engineering*, 22(5):621–644, 8 2012.
38. Lech Madeyski. Is external code quality correlated with programming experience or feelgood factor? In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 65–74. Springer, 2006.
39. Lech Madeyski and Marian Jureczko. Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal*, 23(3):393–422, 2015.
40. S. Matsumoto, Y. Kamei, A. Monden, K. Matsumoto, and M. Nakamura. An analysis of developer metrics for fault prediction. In *6th PROMISE*, 2010.
41. T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *TSE*, 2007.
42. T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *ASE*, 2010.

43. Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2006.
44. Tim Menzies, Suvodeep Majumder, Nikhila Balaji, Katie Brey, and Wei Fu. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 554–563. IEEE, 2018.
45. Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54. ACM, 2008.
46. Nikolaos Mittas and Lefteris Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Transactions on software engineering*, 39(4):537–551, 2013.
47. Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 181–190, New York, NY, USA, 2008. Association for Computing Machinery.
48. Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, pages 181–190. ACM, 2008.
49. Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, Dec 2017.
50. Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 364–373. IEEE, 2007.
51. Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452–461. ACM, 2006.
52. Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 309–318. IEEE, 2010.
53. J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *IEEE TSE*, 2018.
54. Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 382–391. IEEE, 2013.
55. Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2004. ACM.
56. Sinno Jialin Pan, Ivor W Tsang, James T Kwok, and Qiang Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2010.
57. Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.
58. Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500, 2011.
59. Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.
60. Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE, 2013.
61. Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International*

Conference on Software Engineering, ICSE 2014, page 424–434, New York, NY, USA, 2014. Association for Computing Machinery.
62. Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434. ACM, 2014.
63. Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 322–331, 2011.
64. C. Rosen, B. Grawi, and E. Shihab. Commit guru: Analytics and risk prediction of software commits. ESEC/FSE 2015, 2015.
65. Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 966–969. ACM, 2015.
66. Duksan Ryu, Okjoo Choi, and Jongmoon Baik. Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering*, 21(1):43–71, 2016.
67. Chris Seiffert, Taghi M Khoshgoftaar, Jason Van Hulse, and Andres Folleco. An empirical study of the classification performance of learners on imbalanced and noisy software quality data. *Information Sciences*, 259:571–595, 2014.
68. Naeem Seliya, Taghi M Khoshgoftaar, and Jason Van Hulse. Predicting faults in high assurance software. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pages 26–34. IEEE, 2010.
69. Y. Shin and L. Williams. Can traditional fault prediction models be used for vulnerability prediction? *EMSE*, 2013.
70. Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310, 2003.
71. Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1806–1817, 2012.
72. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.
73. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *ICSE 2016*, pages 321–332. ACM, 2016.
74. Divya Tomar and Sonali Agarwal. A comparison on multi-class classification methods based on least squares twin support vector machine. *Knowledge-Based Systems*, 81:131–147, 2015.
75. Huy Tu and Vivek Nair. While tuning is good, no tuner is best. In *FSE SWAN*, 2018.
76. Huy Tu, Zhe Yu, and Tim Menzies. Better data labelling with emblem (and how that impacts defect prediction). *IEEE Transactions on Software Engineering*, 2020.
77. Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.
78. Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.
79. Chadd Williams and Jaime Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.
80. Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. "automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 267–278. IEEE, 2016.
81. Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability*, 65(4):1810–1829, 2016.

82. Xinli Yang, David Lo, Xin Xia, and Jianling Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220, 2017.
83. Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.
84. Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168. ACM, 2016.
85. Feng Zhang, Iman Keivanloo, and Ying Zou. Data transformation in cross-project defect prediction. *Empirical Software Engineering*, 22(6):3186–3218, 2017.
86. Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 309–320. IEEE, 2016.
87. Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance*, pages 274–283. IEEE, 2009.
88. Hongyu Zhang, Xiuzhen Zhang, and Ming Gu. Predicting defective software components from code complexity measures. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pages 93–96. IEEE, 2007.
89. Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on software engineering*, 32(10):771–789, 2006.
90. Yuming Zhou, Baowen Xu, and Hareton Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, 83(4):660–674, 2010.
91. Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 9. IEEE Computer Society, 2007.