Revisiting Process versus Product Metrics: a Large Scale Analysis

Anonymous Author(s)

ABSTRACT

Numerous automated SE methods can build predictive models from software project 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) using 722,471 commits from 770 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). However, 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).

Apart from the above specific conclusions, our more general point is that the SE community now needs to revisit many of the conclusions previously obtained via analytics in-the-small.

1 INTRODUCTION

There exist many automated software engineering techniques for building predictive models from software project data [20]. Such models are cost-effective methods for guiding developers on where to quickly find bugs [38; 50].

Given there are so many techniques, the question naturally arises: which we 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).

To answer that question, we revisited the Rahman et al. ICSE 2013 study *"How, and why, process metrics are better"* [54]. This was an analytics in-the small study that used 12 projects 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;

Their paper is 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 [24]. 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 [8; 34; 54]. 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 [56]. Also, the collected value for these metrics may not translate between projects since those ranges can be highly specific.

Since product versus process metrics is such an important issue, we revisited the Rahman et al. study. To check their conclusions, we ran an analytics in-the-large study that looked at 722,471 commits from 770 Github projects.

Just to be clear: this paper is not an exact replication study of Rahman et al.. When we tried their methodology, we found our data needed another approach (see § 3.3). Instead, (a) we rechecked some of their conclusions but more importantly, (b) we look for what happens when we move from analytics in-the-small to analytics in-the-large. Specifically, we ask four 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?

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 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, [28], [19], [42], [32], [16] 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

[1] For examples of such papers, see Table 3, later in this paper.
[2] 206 citations in Google Scholar, as of May 8, 2020.

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

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.

RQ 4: Measured in terms of system requirements, what works best for analytics in-the-large?

One reason to recommend the use of process variables is that they are an order of magnitude easier to collect. For example, for the 722,471 commits studied in this paper, collecting the product metrics required over 500 days of CPU (using five machines, 16 cores, 7 days). The process metrics, on the other hand, required just 31 days of CPU for data collection. Further, the process metrics needed 2GB of storage while the product metrics needed over 20GB. In the era of cloud computing, such runtimes and, storage requirements may not seem intimidating. But consider: the effort required to collect product metrics have an impact on what science can be achieved, and when. For example, towards the end of the production cycle of this paper, we found a nuanced issue that necessitated the recollection of all our product data. Hence, very nearly, this report did not happen in a timely manner.

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[3].

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

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
entropy	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

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 [18].

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. [57] 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

[3]Note to reviewers: Our data is so large we cannot place in an anonymous Github repo. Zenodo.org will host our data, but only if we use an non-anonymous login. Hence, to maintain double-blind, http://tiny.cc/revisit only contains a sample of our data. If this paper is selected for ASE'20, we will expand that repository to link to data stored at Zenodo.org.

build process metrics. 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 [63].
(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) [39; 40] or process metrics (e.g., recent activity).
(3) Predicting the location of defects so that appropriate resources may be allocated (e.g., [9])
(4) Using predictors to proactively fix defects [4]
(5) Studying defect prediction not only just release-level [14] but also change-level or just-in-time [59].
(6) Exploring "transfer learning" where predictors from one project are applied to another [33; 48].
(7) Assessing different learning methods for building predictors [20]. This has led to the development of hyper-parameter optimization and better data harvesting tools [1; 3].

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 [21; 23; 25; 26; 33; 41; 49; 51; 61; 62; 65; 66; 70; 78; 79; 81] etc. All these methods input project metrics and output a model that can make predictions. Fenton et al. [17] 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. [55] 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 tens 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 [29; 38; 41; 61; 62; 65; 70; 84] 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; 15; 36; 45; 53; 58; 71] 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).

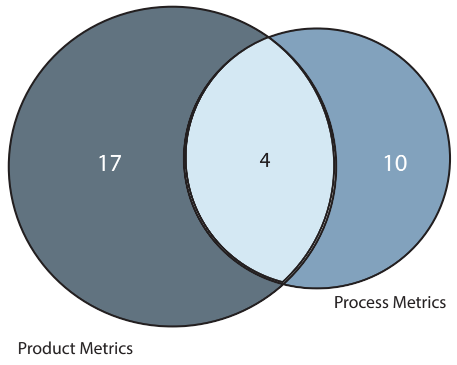

Figure 1: Number of papers exploring the benefits of process and product metrics for defect prediction. The papers in the intersection are [6; 22; 43; 55] explore both process and product metrics.

The rest of this section lists prominent results from the Figure 1 survey. From the product metrics community, Zimmermann et al. [84], 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. [80] 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. [83] 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. [82] 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. [38] 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. [64]. Nagappan et al. [46] in their study reached similar conclusion as Menzies et al. [38], but concluded "However, there is no single set of complexity metrics that could act as a universally best defect predictor" .

	# 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	206
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 papers that experiment with process and/or product metrics.

In other studies related to process metrics, Nagappan et al. [47] 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. [37] 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. [35] 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. [55] 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. [43] and Graves et al. [22].
- Only one paper argues that product metrics are best. Arisholm et al. [6] found one project where product metrics perform better.

Of these papers, Moser et al. [43], Arisholm et al. [6], Rahman et al. [55] and Graves et al. [22] 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 [27; 44]:

- *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.

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

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

- *Defective Commits*: The project must have at least 10 defective commits with defects on Java files.

These sanity checks returned 770 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 [60] code for this purpose. 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 [60] 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 [72] to find commits which were responsible for introducing those changes and marking them as buggy commits[4].

(3) Finally we used the Understand from Scitools[5] 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. Using these information 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. After the metric values for each file in each commit was collected, we filtered out the files which were not in a corresponding commit or in a parent of a defective commit. Here we also added the class labels to the metrics. To filter files and match files with class labels, we need to match commit ids along with file names. Our process data and output from Understand produces the file names in a very different format, in the first case the process data shows a full path, while Understand produces a relative path format (same as Java package information). To overcome this issue, we had to use multiple regular expressions and string processing to match the filenames.

[4]From this point on-wards we will denote the commit which has bugs in them as a "buggy commit"
[5]http://www.scitools.com/

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 [21]. Also, all these learners are widely used in the software engineering community. For all the following models, we use the implementation from Scikit-Learn[6]. Initially, we thought we'd need to apply hyperparameter optimization [67] 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. The model learns the separating hyper-plane from the training data and classifies test data based on which side the example resides.

3.2.2 ***Naive Bayes.*** This is a probabilistic model, widely used in software engineering community [41; 61; 62; 65; 70], 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.

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

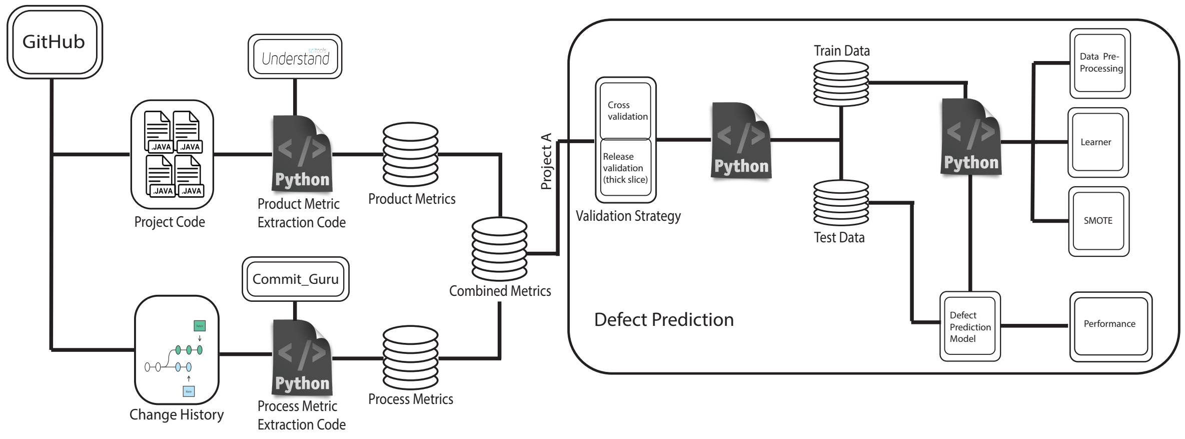

Figure 2: Framework for this analysis.

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 [21; 23; 49; 51; 78].

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 [25; 26; 33; 66; 70; 79; 81] 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 770 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 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. [13], SMOTE is useful for re-sampling training data such that a learner can find rare target classes. For more details in SMOTE, see [2; 13]. 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) *thick slice* 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) *thick slice* is as near as we can come to 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, they trained on data from release 1 to $R - 5$, then tested on release $R-4$,$R-3$, $R-2$,$R-1$, and R. This temporal approach has the advantage that that future data never appears in the training data. On the other hand, this approach has the disadvantage that it did not work for our data:

- When exploring 770 projects, many of them have reached some steady-state where, each month, there is little new activity and zero new reported bugs. Prediction in such "zero cases" is trivially easy (just predict "0") and this can skew performance statistics.
- Also, across 770 projects, releases happening at very different frequencies in different projects. For example, we saw some projects releasing every hour while others released every six months.

Accordingly, to fix both these problems, we used *thick slice release-based* evaluation where we used the first 60% of the project history as training set and then divide the next 40% of the commits in to 5 equal slices for testing.

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 [30; 68; 74–77]. 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 [52] 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 [31; 52; 73]. **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*.

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).

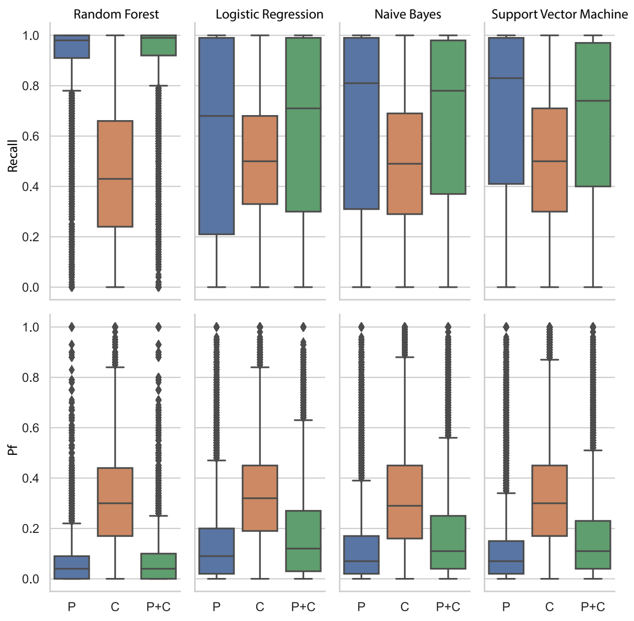

Figure 3: Cross-validation recall and false alarm results for Process(P), Product(C) and, Combined (P+C) metrics. The vertical box plots in these charts run from min to max while the thick boxes highlight the 25,50,75th percentile.

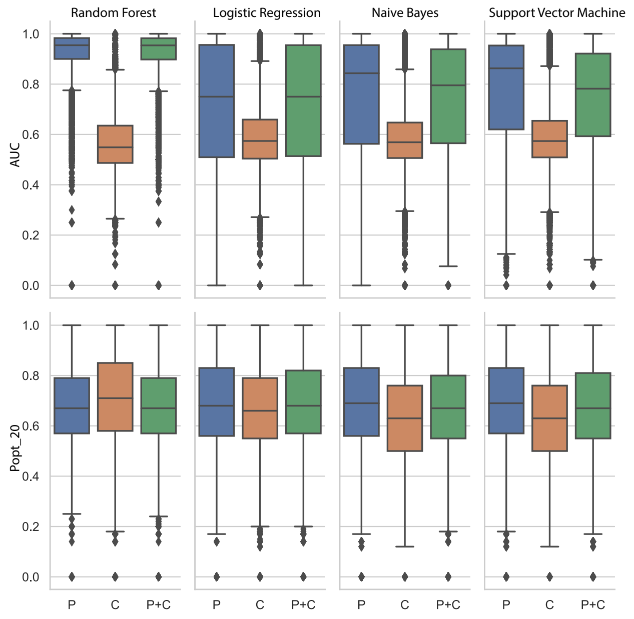

Figure 4: Cross-validation AUC and Popt20 results for Process(P), Product(C) and Combined (P+C) metrics. Same format as Figure 3.

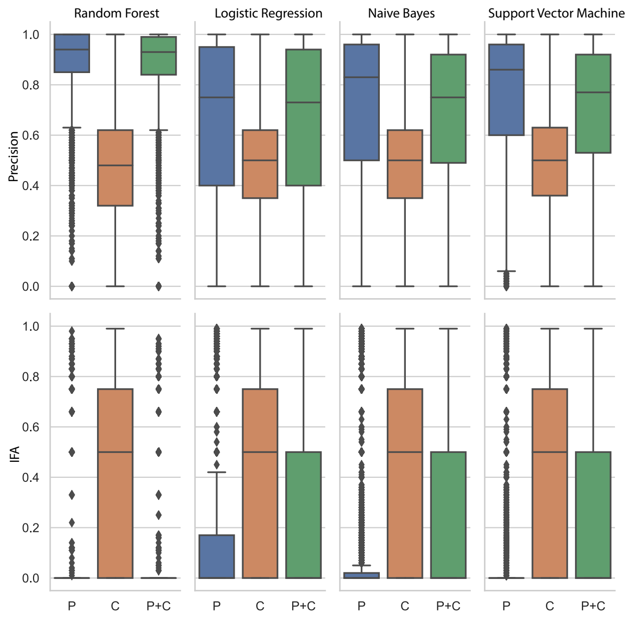

Figure 5: Cross-validation IFA and precision results for Process(P), Product(C) and Combined (P+C) metrics. Same format as Figure 3.

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 770 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, 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.

Secondly, we note that there is one very good result 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.

Thirdly, Figure 6 shows the Random Forest results using thick-slice generation of test sets. As stated in section 3.4 above, there is very little difference in the results between thick-sliced 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?

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 770 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 770 projects that there are many spurious effects. Since Random Forests make their conclusions by reasoning across multiple models, this kind 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 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* 770 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 7 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

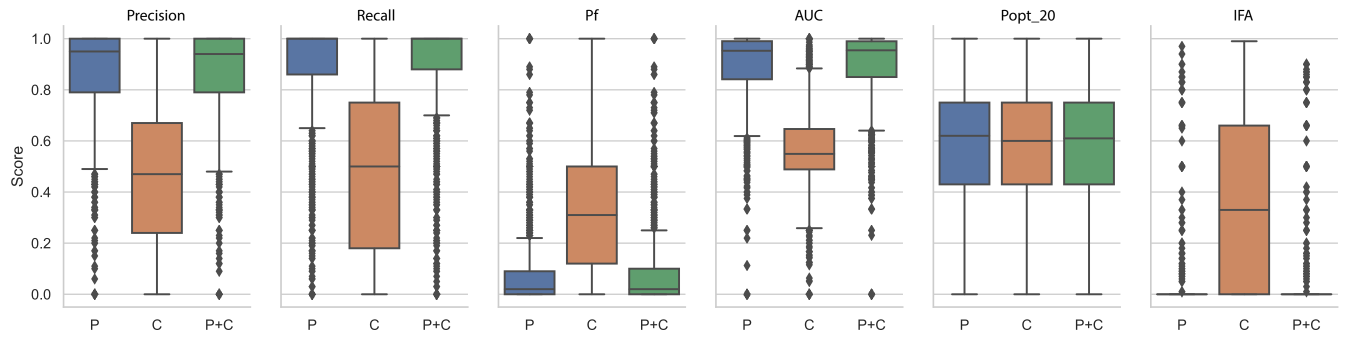

Figure 6: Thick-sliced results for Random Forests.

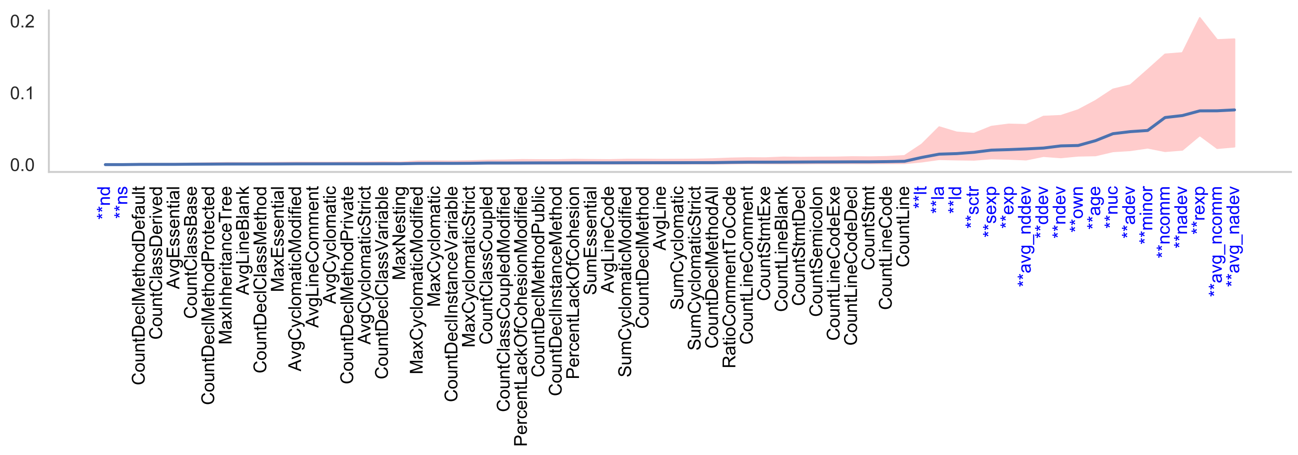

Figure 7: Metric importance of process+product combined metrics based on Random Forest. Process metrics are marked with two blue asterisks. Blue denotes the median importance in 770 projects while the pink region shows the (75-25)th percentile.**

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 7 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 8 compares the process metric rankings learned from analytics in-the-large (i.e. from 770 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 770 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 8 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 8 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.

RQ 4: Measured in terms of system requirements, what works best for analytics in-the-large ?

The data collection times and storage requirements for our product and process metrics have been reported above. Process metrics are far more verbose and hence harder to collect. They required 500 days of CPU (using five machines, 16 cores, 7days) to collect and 23 GB of disk storage. The process metrics, on the other hand, were an order of magnitude faster and smaller to collect and store.

The data collection times and storage requires for our data might seem like a minor issue. But we can report that due to the current Corona virus problem, we lost access to our university cloud

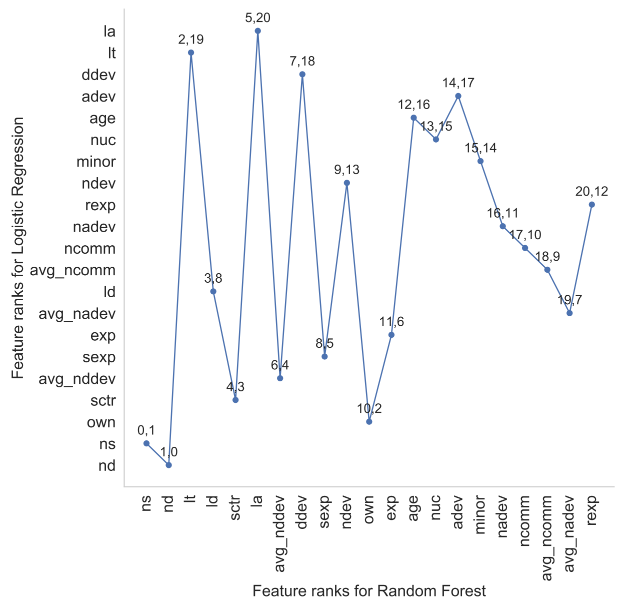

Figure 8: X-axis ranks metrics via analytics in-the-large (using Random Forests applied to 770 projects). Y-axis ranks process metrics using analytics in-the-small (using Logistic Regression and 20*5 projects).

environment for several critical days in the production of this paper. A quirk was discovered in our product data and so we had to recreate it all. While we were successful in doing so, the incident did highlight how systems issues can affect our ability to reason for analytics in-the-large. Hence quite apart from anything said above, we recommend process metrics since they are an order of magnitude easier to manage.

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 **RQ1, RQ2** ,and **RQ3** 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 [69]). 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 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 770 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 770 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 8, 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 blinded for review.

REFERENCES

[1] Agrawal, A., Fu, W., and Menzies, T. What is wrong with topic modeling? and how to fix it using search-based software engineering. *Information and Software Technology 98* (2018), 74–88.

[2] Agrawal, A., and Menzies, T. "better data" is better than "better data miners" (benefits of tuning SMOTE for defect prediction). *CoRR abs/1705.03697* (2017).

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

[4] Arcuri, A., and Briand, L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on* (2011), IEEE, pp. 1–10.

[5] Arisholm, E., and Briand, L. C. Predicting fault-prone components in a java legacy system. In *ESEM* (2006), ACM.

[6] Arisholm, E., Briand, L. C., and Johannessen, E. B. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software 83*, 1 (2010), 2–17.

[7] Basili, V. R., Briand, L. C., and Melo, W. L. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on 22*, 10 (1996), 751–761.

[8] Bird, C., Nagappan, N., Devanbu, P., Gall, H., and Murphy, B. Does distributed development affect software quality? an empirical case study of windows vista. In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 518–528.

[9] Bird, C., Nagappan, N., Gall, H., Murphy, B., and Devanbu, P. Putting it all together: Using socio-technical networks to predict failures. In *ISSRE* (2009).

[10] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. 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* (2011), pp. 4–14.

[11] Briand, L. C., Brasili, V., and Hetmanski, C. J. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering 19*, 11 (1993), 1028–1044.

[12] Cabot, J., Izquierdo, J. L. C., Cosentino, V., and Rolandi, B. 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* (2015), IEEE, pp. 550–554.

[13] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research 16* (2002), 321–357.

[14] Chen, D., Fu, W., Krishna, R., and Menzies, T. Applications of psychological science for actionable analytics. *FSE'19* (2018).

[15] Choudhary, G. R., Kumar, S., Kumar, K., Mishra, A., and Catal, C. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering 67* (2018), 15–24.

[16] D'Ambros, M., Lanza, M., and Robbes, R. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)* (2010), IEEE, pp. 31–41.

[17] Fenton, N. E., and Neil, M. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (2000), pp. 357–370.

[18] Fu, W., Menzies, T., and Shen, X. Tuning for software analytics: Is it really necessary? *Information and Software Technology 76* (2016), 135–146.

[19] Gao, K., Khoshgoftaar, T. M., Wang, H., and Seliya, N. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience 41*, 5 (2011), 579–606.

[20] Ghotra, B., McIntosh, S., and Hassan, A. E. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 37th ICSE*.

[21] Ghotra, B., McIntosh, S., and Hassan, A. E. Revisiting the impact of classification techniques on the performance of defect prediction models. In *37th ICSE-Volume 1* (2015), IEEE Press, pp. 789–800.

[22] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. Predicting fault incidence using software change history. *TSE* (2000).

[23] He, Z., Shu, F., Yang, Y., Li, M., and Wang, Q. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering 19*, 2 (2012), 167–199.

[24] Herbsleb, J. Socio-technical coordination (keynote). In *Companion Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE Companion 2014, Association for Computing Machinery, p. 1.

[25] Ibrahim, D. R., Ghnemat, R., and Hudaib, A. Software defect prediction using feature selection and random forest algorithm. In *2017 International Conference on New Trends in Computing Sciences (ICTCS)* (2017), IEEE, pp. 252–257.

[26] Jacob, S. G., et al. Improved random forest algorithm for software defect prediction through data mining techniques. *International Journal of Computer Applications 117*, 23 (2015).

[27] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (New York, NY, USA, 2014), MSR 2014, ACM, pp. 92–101.

[28] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K., Adams, B., and Hassan, A. E. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance* (2010), pp. 1–10.

[29] Kamei, Y., Monden, A., Matsumoto, S., Kakimoto, T., and Matsumoto, K.-i. The effects of over and under sampling on fault-prone module detection. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)* (2007), IEEE, pp. 196–204.

[30] Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering 39*, 6 (2012), 757–773.

[31] Kochhar, P. S., Xia, X., Lo, D., and Li, S. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), ACM, pp. 165–176.

[32] Kondo, M., German, D. M., Mizuno, O., and Choi, E.-H. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering 25*, 1 (2020), 890–939.

[33] Krishna, R., and Menzies, T. Bellwethers: A baseline method for transfer learning. *IEEE Transactions on Software Engineering* (2018).

[34] Lumpe, M., Vasa, R., Menzies, T., Rush, R., and Turhan, B. Learning better inspection optimization policies. *International Journal of Software Engineering and Knowledge Engineering 22*, 5 (8 2012), 621–644.

[35] Madeyski, L. Is external code quality correlated with programming experience or feelgood factor? In *International Conference on Extreme Programming and Agile Processes in Software Engineering* (2006), Springer, pp. 65–74.

[36] Madeyski, L., and Jureczko, M. Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal 23*, 3 (2015), 393–422.

[37] Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K., and Nakamura, M. An analysis of developer metrics for fault prediction. In *6th PROMISE* (2010).

[38] Menzies, T., Greenwald, J., and Frank, A. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering 33*, 1 (2006), 2–13.

[39] Menzies, T., Greenwald, J., and Frank, A. Data mining static code attributes to learn defect predictors. *TSE* (2007).

[40] Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., and Bener, A. Defect prediction from static code features: Current results, limitations, new approaches. *ASE* (2010).

[41] Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., and Jiang, Y. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering* (2008), ACM, pp. 47–54.

[42] Moser, R., Pedrycz, W., and Succi, G. 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* (New York, NY, USA, 2008), ICSE '08, Association for Computing Machinery, p. 181–190.

[43] Moser, R., Pedrycz, W., and Succi, G. 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* (2008), ACM, pp. 181–190.

[44] Munaiah, N., Kroh, S., Cabrey, C., and Nagappan, M. Curating github for engineered software projects. *Empirical Software Engineering 22*, 6 (Dec 2017), 3219–3253.

[45] Nagappan, N., and Ball, T. 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)* (2007), IEEE, pp. 364–373.

[46] Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering* (2006), ACM, pp. 452–461.

[47] Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. Change bursts as defect predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering* (2010), IEEE, pp. 309–318.

[48] Nam, J., Fu, W., Kim, S., Menzies, T., and Tan, L. Heterogeneous defect prediction. *IEEE TSE* (2018).

[49] Nam, J., Pan, S. J., and Kim, S. Transfer defect learning. In *Software Engineering (ICSE), 2013 35th International Conference on* (2013), IEEE, pp. 382–391.

[50] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. Where the bugs are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2004), ACM, pp. 86–96.

[51] Pan, S. J., Tsang, I. W., Kwok, J. T., and Yang, Q. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks 22*, 2 (2010), 199–210.

[52] Parnin, C., and Orso, A. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis* (2011), ACM, pp. 199–209.

[53] Rahman, F., and Devanbu, P. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), pp. 491–500.

[54] Rahman, F., and Devanbu, P. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 432–441.

[55] Rahman, F., and Devanbu, P. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on* (2013), IEEE, pp. 432–441.

[56] Rahman, F., Khatri, S., Barr, E. T., and Devanbu, P. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, Association for Computing Machinery, p. 424–434.

[57] Rahman, F., Khatri, S., Barr, E. T., and Devanbu, P. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 424–434.

[58] Rahman, F., Posnett, D., Hindle, A., Barr, E., and Devanbu, P. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), pp. 322–331.

[59] Rosen, C., Grawi, B., and Shihab, E. Commit guru: Analytics and risk prediction of software commits. ESEC/FSE 2015.

[60] Rosen, C., Grawi, B., and Shihab, E. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), ACM, pp. 966–969.

[61] Seiffert, C., Khoshgoftaar, T. M., Van Hulse, J., and Folleco, A. An empirical study of the classification performance of learners on imbalanced and noisy software quality data. *Information Sciences 259* (2014), 571–595.

[62] Seliya, N., Khoshgoftaar, T. M., and Van Hulse, J. Predicting faults in high assurance software. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering* (2010), IEEE, pp. 26–34.

[63] Shin, Y., and Williams, L. Can traditional fault prediction models be used for vulnerability prediction? *EMSE* (2013).

[64] Subramanyam, R., and Krishnan, M. S. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering 29*, 4 (2003), 297–310.

[65] Sun, Z., Song, Q., and Zhu, X. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42*, 6 (2012), 1806–1817.

[66] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. Automated parameter optimization of classification techniques for defect prediction models. In *ICSE 2016* (2016), ACM, pp. 321–332.

[67] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* (2018), 1–1.

[68] Tu, H., and Nair, V. While tuning is good, no tuner is best. In *FSE SWAN* (2018).

[69] Tu, H., Yu, Z., and Menzies, T. Better data labelling with emblem (and how that impacts

defect prediction). *IEEE Transactions on Software Engineering* (2020).
[70] Wang, S., and Yao, X. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability 62*, 2 (2013), 434–443.
[71] Weyuker, E. J., Ostrand, T. J., and Bell, R. M. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering 13*, 5 (2008), 539–559.
[72] Williams, C., and Spacco, J. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems* (2008), ACM, pp. 32–36.
[73] Xia, X., Bao, L., Lo, D., and Li, S. "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)* (2016), IEEE, pp. 267–278.
[74] Xia, X., Lo, D., Wang, X., and Yang, X. Collective personalized change classification with multiobjective search. *IEEE Transactions on Reliability 65*, 4 (2016), 1810–1829.
[75] Yang, X., Lo, D., Xia, X., and Sun, J. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology 87* (2017), 206–220.
[76] Yang, X., Lo, D., Xia, X., Zhang, Y., and Sun, J. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security* (2015), IEEE, pp. 17–26.
[77] Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., and Leung, H. 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* (2016), ACM, pp. 157–168.
[78] Zhang, F., Keivanloo, I., and Zou, Y. Data transformation in cross-project defect prediction. *Empirical Software Engineering 22*, 6 (2017), 3186–3218.
[79] Zhang, F., Zheng, Q., Zou, Y., and Hassan, A. E. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), IEEE, pp. 309–320.
[80] Zhang, H. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance* (2009), IEEE, pp. 274–283.
[81] Zhang, H., Zhang, X., and Gu, M. Predicting defective software components from code complexity measures. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)* (2007), IEEE, pp. 93–96.
[82] Zhou, Y., and Leung, H. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on software engineering 32*, 10 (2006), 771–789.
[83] Zhou, Y., Xu, B., and Leung, H. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software 83*, 4 (2010), 660–674.
[84] Zimmermann, T., Premraj, R., and Zeller, A. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (2007), IEEE Computer Society, p. 9.