How, and Why, Process Metrics Are Better

Foyzur Rahman
University of California, Davis, USA
mfrahman@ucdavis.edu

Premkumar Devanbu
University of California, Davis, USA
ptdevanbu@ucdavis.edu

***Abstract*—Defect prediction techniques could potentially help us to focus quality-assurance efforts on the most defect-prone files. Modern statistical tools make it very easy to quickly build and deploy prediction models. Software metrics are at the heart of prediction models; understanding *how* and especially *why* different types of metrics are effective is very important for successful model deployment. In this paper we analyze the applicability and efficacy of *process* and *code* metrics from several different perspectives. We build many prediction models across 85 releases of 12 large open source projects to address the *performance*, *stability*, *portability* and *stasis* of different sets of metrics. Our results suggest that code metrics, despite widespread use in the defect prediction literature, are generally less useful than process metrics for prediction. Second, we find that code metrics have high *stasis*; they don't change very much from release to release. This leads to *stagnation* in the prediction models, leading to the same files being repeatedly predicted as defective; unfortunately, these recurringly defective files turn out to be comparatively less defect-dense.**

I. Introduction

Software-based systems pervade and greatly enhance modern life. As a result, customers demand very high software quality. Finding and fixing defects is expensive; defect prediction models promise greater efficiency by prioritizing quality assurance activities. Since defect distribution is highly skewed [11, 27], such models can usefully finger the most defective bits of code.

Typically defect prediction models rely on *supervised learners*, which use a labeled training dataset to learn the association between measured entity properties (e.g., metrics calculated on files or methods), with the defect proneness of these entities. Careful choice of metrics can improve prediction performance. Researchers have mostly focused on two broad classes of metrics for defect prediction: *code metrics*, which measure properties of the code (*e.g.,* size and complexity), and *process metrics* (*e.g.,* number of changes, number of developers). Researchers have long been interested in which class of metrics (process or code) are better for defect prediction.

Moser *et al.* [18] compared the power of code and process metrics on Eclipse project and found that process metrics outperform code metrics. However, Menzies *et al.* [16] report that code metrics are useful for defect prediction. Arisholm *et al.* found that process and code metrics perform similarly in terms of AUC but code metrics may not be cost-effective [2].

Our work deviates from existing approaches in two important ways. Firstly, (and most importantly) we seek to understand *how* and *why* process metrics are better for defect prediction. Secondly, our methodology is squarely based on a *prediction* setting. Existing studies mostly evaluate different types of metrics using cross-validations on few projects. However, the most attractive use of models is in a *prediction setting*; such models can be used to focus cost-constrained quality control efforts. In a release oriented development process, this means training models on earlier releases to predict defects of latter release. We build and compare prediction models across multiple projects and releases, using different combinations of metrics and learning techniques, using a broad range of settings. Our experiments lead to the following contributions:

- We compare the *performance* of different models in terms of both traditional measures such as AUC and F-score, *and* the newer cost-effectiveness [1] measures.
- We compare the *stability* of prediction performance of the models across time and over multiple releases.
- We compare the *portability* of prediction models: how do they perform when trained and evaluated on completely different projects.
- We study *stasis*, *viz.*., the degree of change (or lack thereof) in the different metrics, and the corresponding models over time. We then relate these changes with their ability to predict defects.
- We investigate whether prediction models tend to favor *recurringly defective files*; we also examine whether such files are relatively more defect-dense, and thus good targets of inspection efforts.

II. Background and Theory

Defect prediction models are mostly built using supervised learning techniques: logistic regression, SVM, decision trees *etc.*. During training, the model systematically learns how to associate various properties of the considered software entities (*e.g.*, methods, files and packages) with the defect proneness of these entities. Researchers have historically hypothesized that properties of the code, measured using *code metrics*, could usefully predict defect-proneness. Code metrics could measure size (larger files may be more defect-prone), or complexity (more complicated files may be more defect-prone). The value of code metrics in defect prediction has been well-explored [12, 16, 29].

However, software development processes, *per se*, can be quite complex. Lately, researchers have been interested in the impact of development process on software quality. Recent studies [4, 19, 21, 23, 26] suggest that an entity's process properties (*e.g.*, developer count, code ownership, developer experience, change frequency) may be important indicators of defect proneness.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

Clearly, the relative efficacy of these metrics in defect prediction is of vital concern to the practitioner. Should she use all types of metrics? Should she mix different types of metrics? Moser *et al.* [18] compared the prediction performance of code and process metrics in three releases of Eclipse and found that process metrics may outperform code metrics in defect prediction. Arisholm *et al.* [2] compared various metrics and prediction techniques on several releases of a legacy Java middleware system named COS and found that code metrics may perform well in terms of traditional performance measures such as AUC while it may not be cost-effective. Menzies *et al.* [16] found code metrics very effective for defect prediction.

Existing studies mostly limit themselves to a cross-validation based model evaluation on a limited set of projects. However, many projects align their development and quality assurance activities with releases. Therefore, a release-based evaluation of model performance may be more appropriate for such settings.

Research Question 1: In *release-based prediction settings*, how do the process and code metrics compare to predict defect locations?

Release-based prediction performance may, however, destabilize after a major shift of activities between releases. For example, a release comprising a burst of new features might be followed-up by a series of incremental quality-improving releases. The process aspects that cause defects may also shift, thus confounding prediction performance. Ekanayake *et al.* [6] found that defect prediction models destabilize, *viz.*., perform poorly over time due to project phase changes. We therefore evaluate how process and product metrics affect the prediction stability of models. Presumably, a more stable model would also adapt better to project phase changes; armed with this information, a practioner can then more effectively select prediction metrics based on specific project dynamics.

Research Question 2: Are process metrics more/less *stable* than code metrics?

Another interesting way of comparing process and code metrics would be the *portability* of the learned prediction models between different projects. Portability would be useful for smaller software companies, without large, available portfolios of software for training prediction models. Even big companies with diverse product portfolios may find it useful to port their prediction models to a new product. Portability (also called "cross-project prediction") recently attracted quite a bit of attention [14, 22, 24, 25, 28]; but we have not found any that compare the portability of different types of metrics.

Research Question 3: Are process metrics more/less *portable* than code metrics?

Besides exploring the practical utility of metrics in defect prediction models, we also seek to understand *why* one class of metrics outperforms another, by exploring the distribution of metrics values in more detail. One important property of metrics is *stasis*. One can reasonably expect that, as a system evolves, the distribution of defects does not remain unchanged. Therefore, we might expect that the values of useful metrics (ones with defect-prediction power) would also tend to vary with release. Metrics whose values remain unchanged would willy-nilly tend to predict the same files as defective, release after release.

Research Question 4: Are process metrics more/less *static* than code metrics?

The above question asks if metrics change (or fail to change) as software evolves; but even a very dynamic metric is a useful predictor only if it co-evolves significantly with defect occurrence. Furthermore, usually, many metrics are used together as a group in a prediction model. Even if a single metric isn't very dynamic, it might synergistically work together with other (also static) metrics to form a good prediction model. It is therefore important to understand whether models stagnate, *viz.*., they tend to repeatedly predict the same files as defective.

Research Question 5: Do models built from different sets of metrics *stagnate* across releases?

Of course, it is possible that even stagnant models work: if the same files are recurringly defective, across multiple releases, then a stagnant model that selects these defect-prone files will be reasonably successful. We hypothesize that stagnant models predict defect distributions close to those they originally learned. Perhaps they flag mostly the defective entities which don't change much, *e.g.*, complicated files which repeatedly become defective. We consider such defective entities (defective in both training release and test release) as *recurringly defective* and the rest as *incidentally defective*. We hypothesize that a stagnant model would predict mostly the recurringly defective entities.

Research Question 6: Do stagnant models (based on stagnant metrics) tend to predict recurringly defective entities?

Arisholm *et al.* [2] report that code metrics perform well (as well as process metrics) when considering cost-*in*sensitive, entity-based measures such as AUC, but not as well when considering cost. While Arisholm *et al.* don't explore the reasons for this result, we believe this arises from prediction bias away from defect *density*. Larger files are more expensive to inspect; if such investment does not pay off in terms of number of defects uncovered, it's not cost-effective. We

TABLE I: Studied Projects and Release Information

Project	Description	Releases	Avg Files	Avg SLOC
CXF	Services Framework	2.1, 2.2, 2.3.0, 2.4.0, 2.5.0, 2.6.0	4038.33	358846.67
Camel	Enterprise Integration Framework	1.4.0, 1.5.0, 1.6.0, 2.0.0, 2.2.0, 2.4.0, 2.7.0, 2.9.1	4600.38	241668.12
Derby	Relational Database	10.2.2.0, 10.3.1.4, 10.4.1.3, 10.5.1.1, 10.6.1.0, 10.7.1.1, 10.8.2.2	2497.29	530633.00
Felix	OSGi R4 Implementation	1.0.3, 1.2.0, 1.4.0, 1.6.0, 2.0.0, 2.0.3, 3.0.0, 3.2.0, 4.0.2	2740.56	249886.22
HBase	Distributed Scalable Data Store	0.16.0, 0.18.0, 0.19.0, 0.20.0, 0.20.6, 0.90.1, 0.90.4, 0.94.0	934.75	187953.38
HadoopC	Common libraries for Hadoop	0.15.0, 0.16.0, 0.17.0, 0.18.0, 0.19.0, 0.20.1	1047.17	142257.33
Hive	Data Warehouse System for Hadoop	0.3.0, 0.4.0, 0.5.0, 0.6.0, 0.7.0, 0.8.0, 0.9.0	966.29	152079.86
Lucene	Text Search Engine Library	2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.4.0, 2.9.0, 3.0.0	990.86	122527.00
OpenEJB	Enterprise Java Beans	3.0, 3.1, 3.1.1, 3.1.2, 3.1.3, 3.1.4, 4.0.0	2895.43	225018.43
OpenJPA	Java Persistence Framework	1.0.0, 1.0.2, 1.1.0, 1.2.1, 1.2.2, 2.0.1, 2.1.0, 2.2.0	3181.50	321033.50
Qpid	Enterprise Messaging system	0.5, 0.6, 0.8, 0.10, 0.12, 0.14, 0.16	1724.00	198311.86
Wicket	Web Application Framework	1.4.0, 1.4.5, 1.4.9, 1.5.0, 6.0.0b2	2295.20	152565.40

TABLE II: Process Metrics

Short Name	Description
COMM	Commit Count
ADEV	Active Dev Count
DDEV	Distinct Dev Count
ADD	Normalized Lines Added
DEL	Normalized Lines Deleted
OWN	Owner's Contributed Lines
MINOR	Minor Contributor Count
SCTR	Changed Code Scattering
NADEV	Neighbor's Active Dev Count
NDDEV	Neighbor's Distinct Dev Count
NCOMM	Neighbor's Commit Count
NSCTR	Neighbor's Change Scattering
OEXP	Owner's Experience
EXP	All Committer's Experience

examine whether code metrics have a *prediction bias* towards larger files (and thus, ones with lower defect density), and whether such prediction bias emanates from the *stagnation* of the models as discussed earlier.

Research Question 7: Do stagnant models have a prediction bias towards larger, less-defect-dense files?

III. Experimental Methodology

A. Projects Studied

Our 12 sample projects are listed in Table I. All are Java-based, and maintained by Apache Software Foundation (ASF); however, they come from a very diverse range of domains. For each project we extracted the commit history from its GIT repository[1]. We also used GIT BLAME on every file at each release to get the detailed contributor information. Our BLAME process uses copy and move detection and ignores whitespace changes, to identify the correct provenance of each line.

All 12 projects use JIRA[2] issue tracking system. From JIRA, we extracted the defect information and the fixing commits for fixed defects. Then, for each fixing commit, we extract the changed lines, author, *etc.* from GIT. Any files modified in these defect-fixing commits are considered as defective.

B. Predicting Defects

Our defect-prediction studies are at file-level. We choose file-level predictive (code and process) metrics that have been widely used in defect prediction literature. We used a wide range of learning techniques: Logistic Regression, J48, SVM, and Naive Bayes, all from WEKA[3]. This reduces the risk of dependence on a particular learning technique. Note, in the default setting, J48 and SVM do not provide a probability of defect proneness, which is important for cost-effectiveness comparison. However, WEKA provides settings for J48 and SVM that do yield probabilities[4]. As did Arisholm *et al.* [2], we find that the prediction performance depends mostly on the types of metrics used, and not on the learning technique. Therefore, for brevity, we typically just report the findings from Logistic Regression, and mention deviations in other techniques, if any.

All of our models are binary classifiers (predicted *defective*|*clean*). However, a file may have zero or more defects fixed during a release. As in prior work, we declare a file as *defective* if it had at least one defect-fix, or *clean* otherwise. Using both code and process metrics, we build models in a release-based prediction setting: we train the model on a given release and evaluate its performance on the next release. So, *e.g.* a model trained on the k-th release, is only tested on the $k+1$-th release. However, as discussed in section IV, for *stability* we evaluate the model on all future releases following a training release. For sensitivity analysis of other research questions, we also evaluated all of our models using all future releases instead of just the immediately succeeding release and found consistent results.

For *portability*, we evaluated models trained on one project on all releases of other projects, ignoring time-ordering.

[1]http://git.apache.org

[2]https://issues.apache.org/jira/

[3]http://www.cs.waikato.ac.nz/ml/weka/

[4] For J48, an option to use unpruned decision tree with Laplace smoothing is available to find the estimate of defect proneness of each file. Similarly, WEKA's SVM implementation allowed us to fit a logistic models to SVM output to find the probability of defect proneness.

TABLE III: Code Metrics

Type	Metrics	Count
File	CountDeclMethodPrivate, AvgLineCode, CountLine, MaxCyclomatic, CountDeclMethodDefault, AvgEssential, CountDeclClassVariable, SumCyclomaticStrict, AvgCyclomatic, AvgLine, CountDeclClassMethod, AvgLineComment, AvgCyclomaticModified, CountDeclFunction, CountLineComment, CountDeclClass, CountDeclMethod, SumCyclomaticModified, CountLineCodeDecl, CountDeclMethodProtected, CountDeclInstanceVariable, MaxCyclomaticStrict, CountDeclMethodPublic, CountLineCodeExe, SumCyclomatic, SumEssential, CountStmtDecl, CountLineCode, CountStmtExe, RatioCommentToCode, CountLineBlank, CountStmt, MaxCyclomaticModified, CountSemicolon, AvgLineBlank, CountDeclInstanceMethod, AvgCyclomaticStrict	37
Class	PercentLackofCohesion, MaxInheritanceTree, CountClassDerived, CountClassCoupled, CountClassBase	5
Method	CountInput, CountOutput, CountPath, MaxNesting	12

C. Process Metrics

Our file-based process metrics are listed in Table II. All process metrics are release-duration. *COMM* measures the number of commits made to a file. *ADEV* is the number of developers who changed the file. *DDEV* is the cumulative number of distinct developers contributed to this file up to this release. *ADD* and *DEL* are the normalized (by the total number of added and deleted lines) added and deleted lines in the file. *OWN* measures the percentage of the lines authored by the highest contributor of a file. *MINOR* measures the number of contributors who authored less than 5% [4] of the code in that file. *OEXP* measures the experience of the highest contributor of that file using the percent of lines he authored in the project at a given point in time. *EXP* measures the geometric mean of the experiences of all the developers. All these metrics are drawn from prior research [2, 4, 17, 20].

We also used a simple line based change entropy metric [8], derived from the location of the changes made: *SCTR* measures the scattering of changes to a file; scattered changes could be more complex to manage, and thus more likely to induce defects. *SCTR* is the standard position deviation of changes from the geographical centre thereof.

Kim *et al.* [10]'s celebrated BugCache is populated using a simple co-commit history. This work suggests a range of different process metrics based on co-commit-neighbors. For a given file $\mathcal{F}$ and release $\mathcal{R}$, these metrics are based on the list of files co-committed with $\mathcal{F}$, weighted by the frequency of co-commits during $\mathcal{R}$. *NADEV*, *NDDEV*, *NCOMM* and *NSCTR* are just the co-commit-neighbor measures of *ADEV*, *DDVEV*, *COMM* and *SCTR*. We measured the usefulness of all the neighbor based metrics, as well as *SCTR* using single variable predictor models, and found them to be highly significant defect predictors with a median AUC of around 0.8. All of our process metrics are cumulated and measured on a per-release basis.

In this paper, we focus on measuring prediction performance, rather than testing hypotheses. Therefore issues such as VIF, goodness of fit, variable significance etc. were not such a concern and following the suggestion of Menzies *et al.* [16], we simply use all the available variables.

D. Code Metrics

We used UNDERSTAND from Scitools[5] to compute code metrics. All 54 code metrics are listed in Table III. Our metrics set includes complexity metrics such as Cyclomatic complexity, essential complexity, number of distinct paths, fan in, fan out etc.; Volume metrics such as lines of code, executable code, comment to code ratio, declarative statement etc.; and Object oriented metrics such as number of base classes, number of children, depth of inheritance tree etc. Space limitations inhibit a detailed description; however, these metrics are well documented at the UNDERSTAND website[6]. Most metrics are file-level; some metrics, (*e.g.* OO metrics) are class level. Since all projects are Java-based, most files contain a single class; so we aggregate class level metrics to file level using `max`. Similarly some metrics are available only at method level (such as fan-in, fan-out) and we aggregate those at file level using `min, max` and `mean`.

E. Evaluation

All of our prediction models output probabilities of defect proneness of files. To classify a file as *defective*, one can use varying minimum thresholds on the probability value. Most models are fallible; thus different choices of threshold will give varying rates of *false* positives/negatives (FP/FN), and *true* classifications (TP/TN)

Accuracy of the model is the proportion of correct predictions, and is defined as $\frac{TP+TN}{TP+FP+TN+FN}$. Ma *et al.* [13] noted, that in a highly class imbalanced data set with very few defective entities, accuracy is not a useful performance measure: even a poor model that declares all files as *clean* will have high accuracy.

Precision measures the percentage of model declared defective entities that are actually defective. PRECISION is defined as $\frac{TP}{TP+FP}$. Low-precision models would waste precious quality-control resources.

Recall identifies the proportion of actually defective entities that the model can successfully identify. Model with low recall

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

[6]http://www.scitools.com/documents/metricsList.php?

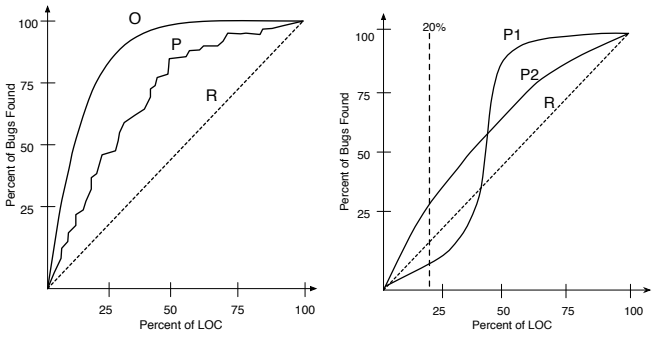


Fig. 1: Cost Effectiveness Curve. On the left, O is the optimal, R is random, and P is a possible, *practical*, predictor model. On the right, we have two different models P1 and P2, with the same overall performance, but P2 is better when inspecting 20% of the lines or less. Figure quoted from our FSE 2011 paper [23]

would be unable to find most of the defects. Recall is identified as $\frac{TP}{TP+FN}$.

A good model yields both high PRECISION and high RECALL. But, it is well known that increasing one often reduces the other; hence the *F-measure*.

F-Measure is the harmonic mean of PRECISION and RECALL.

All of these measures require the use of a minimum probability threshold to declare a file defective. Lessermann *et al.* [12] decry the use of performance measures that require an arbitrary threshold. Mende *et al.* [15] argue that threshold based performance measures make replication difficult. Arisholm *et al.* [2] argues that in the context of SE considering defect proneness as continuous and prioritizing resources in the order of the predicted defect proneness is more suitable. This brings us to measures that are threshold invariant.

ROC Receiver operating characteristic (ROC) is a curve that plots the true positive rates ($TPR = \frac{TP}{TP+FN}$) against False Positive Rates for all possible thresholds between 0 and 1. This threshold invariant measure gives us a 2-D curve, which passes through $(0, 0)$ and $(1, 1)$. The best possible model would have the curve close to $y = 1$, with and area under the curve (AUC) close to 1.0. AUC always yields an area of 0.5 under random-guessing. This enables comparing a given model against random prediction, without worrying about arbitrary thresholds, or the proportion of defective files.

Cost-Effectiveness The above measures ignore cost-effectiveness. Consider a model that accurately predicts defective files, but orders those files in increasing order of defect density. Then we might allocate resources (*e.g.*, code inspectors) to the least defect dense files. *Assuming cost proportional to the size of the file*, this would be undesirable. In contrast, a cost-effective model would not only accurately predict defective files, but also order those files in decreasing order of defect density.

Analogous to ROC, we can have a cost-effectiveness curve (refer to left plot of figure 1), plotting the proportion of defects against proportion of SLOC coming from the ordered (using predicted defect proneness) set of files. Unlike ROC, however, we only consider a small portion of area under the cost-effectiveness curve (AUCEC), tailored for the resource constraints. Thus, during deadlines, we may only consider the cost-effectiveness for at most 10% SLOC, while other times we may consider 20% SLOC. Based on the choice of the maximum proportion of SLOC that a manager may be interested to consider, different models may become more or less competitive. E.g., in the right plot of figure 1, $P2$ is a better cost-effective model up to around 50% SLOC, after that $P1$ performs better.

In this paper we use AUC, AUCEC at 10% (AUCEC_{10}) and 20% (AUCEC_{20}) SLOC to compare the models' performance in a threshold invariant manner. However, to give readers a way to compare with existing literature, we initially report F-Measure at 0.5 threshold (F_{50}), as used by the influential Zimmermann *et al.* paper [29].

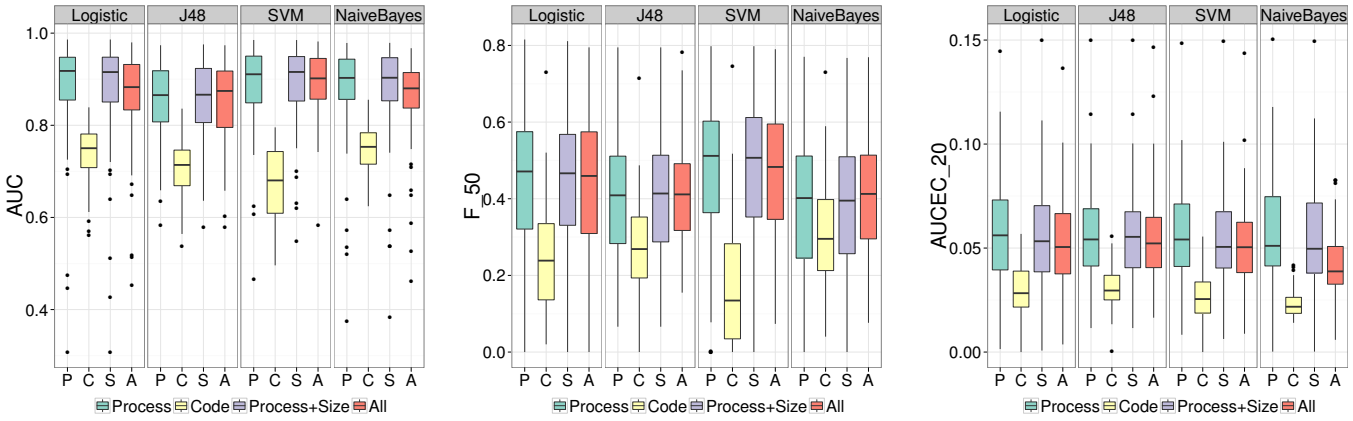
To compare process and code metrics, we evaluate 4 combinations of metrics. First, we build the model with only process metrics. Next we build the model with just code metrics. We then build the model with process metrics and size (*CountLineCode* in table III). Size is a very important metric by itself and there is a considerable body of evidence that size is highly correlated with most product metrics including McCabe and Halstead metrics [7, 9]. This is to ensure that we can separate the influence of the combination of size and process metrics from the entire collection of process and code metrics. Finally, we build the model with the entire collection of process and code metrics. Following Menzies *et al.* [16] and others, we Log transformed all of our process and code metrics. This transformation significantly improves prediction performance.

IV. RESULTS

We begin with comparing the performance of process and code metrics in release based prediction settings using AUC, AUCEC and F_{50}.

> **RQ 1:** In release based prediction settings, how do the process and code metrics compare to predict defect locations?

Figure 2 compares the AUC of different types of metrics for four different classifiers. The metrics are marked in x-axis as "P": Process, "C": Code, "S": Process & Size, and "A": All (process and code metrics combined). We compared the AUC performance for all types of metrics for a given learning technique using Wilcoxon tests and corrected the p-values using Benjamini-Hochberg (BH) correction. We also do the same to compare the performance of different learning techniques for a given set of metrics. We find that process metrics always perform significantly better than code metrics across all learning techniques, with very low p value ($p < 0.001$). However, process metrics and size together may not perform any better than process metrics alone (all p values were insignificant). Combining process and code metrics together also doesn't yield



(a) AUC performance *(b)* F_{50} performance *(c)* $AUCEC_{20}$ performance

Fig. 2: Performance of different classifiers and metrics

any better performance than process metrics alone. Similar to Menzies *et al.* [13], we find that for code metrics, NaiveBayes works best, while J48 and SVM may not perform as well. However, Logistic regression performs well for all types of metrics and is at a statistical dead heat with NaiveBayes for code metrics ($p = 0.664$).

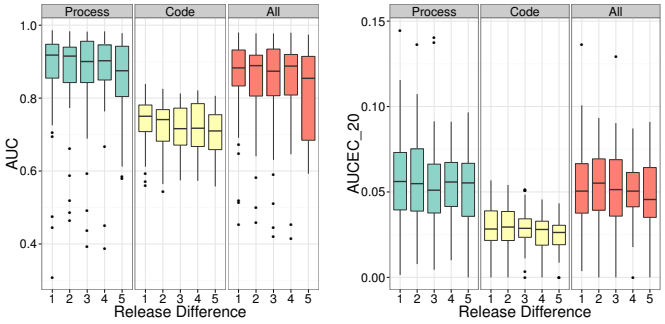
We also present the F_{50} and $AUCEC_{20}$ in figure 2. Again, we compare different classifiers and set of metrics and found that process metrics easily outperform code metrics in terms of both F_{50} and AUCEC ($p < 0.001$ in both cases). Particularly, similar to Arisholm *et al.*'s findings [2], we found code metrics are less effective for cost-effective prediction. Interestingly, while NaiveBayes is the best classifier for code metrics [16] when measured using AUC, but does *worst* for AUCEC ($p < 0.05$ after BH correction against all other classifiers).

In general, our results show models using code metrics provide reasonable AUC, albeit not as good as models using process metrics. For $AUCEC_{20}$, code metrics don't do much better than random: different learning techniques don't help much either. Therefore, for brevity, we only report results only from Logistic regression (LR), which does well for both process and code metrics; indeed LR yields better AUCEC than NaiveBayes for code-metrics based models.

Next we report results on the *stability* of prediction models: as discussed in [5], models with stable prediction performance are more useful in practice, specially so in rapidly evolving projects.

RQ 2: Are process metrics more/less stable than code metrics?

We evaluate stability by using *all available* older releases to predict newer ones. Development activity on a project evolves with time, and older releases may be quite different from newer ones; this approach is a good test of the prediction stability of a set of metrics. Therefore, for this RQ, rather than predicting an immediately subsequent release, we predict *all releases* in a project following a training release.

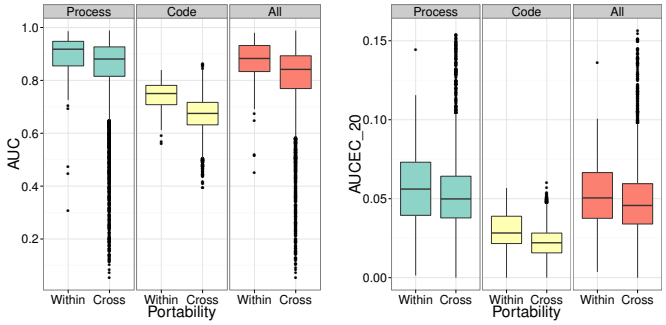


(a) AUC Stability *(b)* $AUCEC_{20}$ Stability

Fig. 3: Stability of different metrics

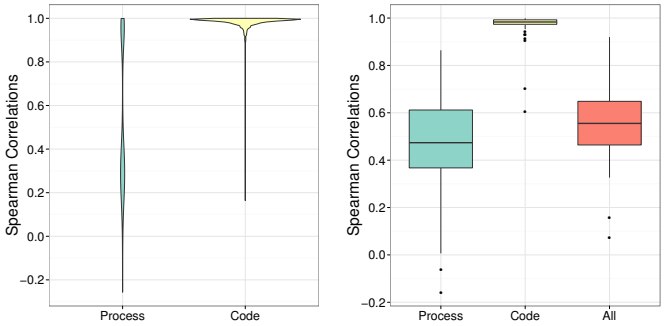
Figure 3a shows the stability of AUC when predicting 1 thru 5 releases in the future. For brevity, only results from LR are presented; other learning models give similar results. We see a perceptible downward trend in performance as we train on older releases to predict newer releases. However, 2-sample Wilcox tests on all pairs of boxplots (*viz.*, for both consecutive and non-consecutive releases) for each types of metrics provide no statistical significance of this trend. We observe similar result in terms of $AUCEC_{10}$ and $AUCEC_{20}$ (figure 3b), for either code or process metrics. Earlier findings of instability were based on a continuous time line [5]. Releases may be capturing more wholesome encapsulations of similar activities, than do equal time-intervals; although some releases may take longer than others, all releases may group activities in the same way. Thus, inherently, release-based models may be less susceptible to concept drift [5].

Portability of prediction models across projects may be quite important for some organizations, specially newer or rapidly changing organizations. Cross-project defect prediction recently attracted quite a bit of attention [14, 24, 25, 28]; however, to our knowledge, none compare the relative merits of different types of metrics in such settings.



(a) AUC Portability *(b)* AUCEC_{20} Portability

Fig. 4: Portability of different metrics



(a) Stasis of Metrics *(b)* Stasis of Models

Fig. 5: Comparing *stasis* of different metrics and models

RQ 3: Are process metrics more/less portable than code metrics?

Figure 4a and 4b compares the portability of models for different sets of metrics in terms of AUC and AUCEC_{20}. Performance degrades in cross-project settings for both process and code metrics; the degradation is statistically significant with low p-values (less than 0.001). It's also clear that code metrics show a larger decline of performance than process metrics and the notches of code metrics based boxplots are nearly non-overlapping. We do see a large number of outliers in process metrics, suggesting less portability in some cases. We observed similar pattern for AUCEC_{10} (p value from Wilcox test is less than 0.001 for code metrics, 0.024 for process metrics, and 0.009 for all metrics) and AUCEC_{20} (p value from Wilcox test is less than 0.001 for code metrics, 0.031 for process metrics, and 0.006 for all metrics), with process metrics showing more portability than code metrics.

Our findings so far indicate that code metrics are *less stable* and *less portable* than process metrics. Why do code metrics show such high "resistance to change"? We guessed that common code metrics are less responsive to development activities, and thus less tied to factors that influence defect-proneness. For example, a defect introduced by replacing a `strncp` call with a `strcp`, wouldn't affect code metrics, but would affect process attributes such as the modification time, number of active developers, ownership, *etc.*. The joint distribution of metrics and defects are estimated and exploited by most learning algorithms; relatively "change-resistant" (more static) metrics might do poorly at tracking changes in defect occurrence.

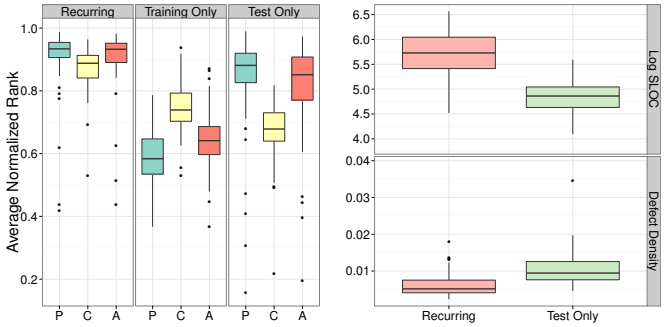
RQ 4: Are process metrics more/less *static* than code metrics?

We use the Spearman correlation of each metric of every file between two successive releases as a measure of stasis. We then combine all the correlations in a violin plot for each group of metrics. This would tell us how similar a group of metrics look in two successive releases. Figure 5a presents the comparison of our Spearman-correlation stasis measure for process and code metrics. As we can see from the figure, code metrics are highly correlated, and therefore changes very little over releases. Process metrics, on the other hand, show quite a range of stasis. While the median value of stasis in process metrics is under 0.5, the top quartile is over 0.9!

To examine this disparity further, we ranked the process metrics using the *median* value of our stasis measure for that metric. The ranking of metrics, in decreasing order of stasis, was OEXP, EXP, OWN, MINOR, DDEV, SCTR, COMM, ADEV, NSCTR, NCOMM, NDDEV, NADEV, DEL, and ADD. Interestingly, the top 5 (OEXP thru DDEV) have a very high median stasis score of *over* 0.93 while the rest 9 have a median stasis score of *under* 0.32. This disparity between mean and top-quartile prompted us to examine in more detail the effects of process-metrics stasis on predictive power. There are 5 metrics with high stasis, and 9 that show low stasis. We hypothesized that stasis is important for predictive power and that low-stasis measures make better predictors.

For comparison purpose, we chose all 5 high-stasis process metrics, and chose different groups of 5 from the low-stasis process metrics. We used a sliding window based approach to select 5 groups of 5 low-stasis metrics: the 5 groups were the metrics ranked $1 \ldots 5$ (by stasis of low to high correlations), and then those ranked $2 \ldots 6$, all the way up to $5 \ldots 9$. We then built prediction models from all groups of 5 and additionally the high stasis metrics group of ranks $10 \ldots 14$. This gave us a total of 6 models, 5 based on low stasis metrics and 1 based on high stasis metrics. We found that the median AUC of models built from metrics with low correlations are over 0.9, while the AUC of models built from highly correlated process metrics are barely around 0.8. A paired Wilcox test of each of the models of low stasis metrics against the model from high stasis metrics suggests that models of low stasis metrics are statistically significantly better predictors than models with high stasis metrics for all performance criteria (AUC, AUCEC_{10}, AUCEC_{20}, and F_{50}) with very low p-values (less than 0.001).

This case-study above suggests that stasis plays an important role in predictive power. We attempted to replicate this study for code metrics, comparing high-stasis and low-stasis code metrics; but unfortunately, code metrics generally have *very high* stasis, around 0.96. This suggests that models based



(a) Ability to rank files according to the change of defect proneness *(b)* SLOC and defect density for change of defect proneness

Fig. 6: Ranking ability, Log SLOC and defect density for change of defect pronenes

on code metrics *would also have high stasis*: these models would repeatedly predict the same files over and over again as defective.

Prediction models learn the joint distribution of metrics and defect proneness from the training release, and use this knowledge to predict defects in the test release. The probability assigned by models to files in the test release reflect this learned defect distribution. In an active project, as different parts of the system become foci of attention and activity, defects gets introduced into different parts. There is thus every reason to expect that the defect distribution in subsequent releases should differ: file f that was buggy in release k isn't necessarily also buggy in $k+1$ and vice versa. We would therefore expect that a good model wouldn't *stagnate*, *viz.*, it would generally indicate a change of defect probability in files from release to release.

RQ 5: Do models built from different sets of metrics *stagnate* across releases?

The *rank correlation* between the predicted probabilities (from test set) and the learned probabilities (from training set) is one measure on the model's adaptation in an active project. A high rank correlation suggests that the model is probably predicting the same set of files as defective. Figure 5b, shows the value range of Spearman correlations between probabilities of defect proneness across all pairs of training-test releases. The difference is stark: the code metrics based models are essentially spitting out the original probabilities it learned from the training data. This clearly indicates that the stasis of code metrics leads to stagnant prediction models, that predict the same files as defective over and over again.

To be fair, even stagnant models might be useful in a project where same set of files become recurringly defective: identifying a critical set of recurringly defective files *unambiguously* might still help focus testing and inspection.

RQ 6: Do stagnant models (based on stagnant metrics) tend to predict recurringly defective entities?

To evaluate the effect of stagnation, we partition the set of files in three sets, based on how much their defect-proneness changes. Included in Set 1 are files which are defective in both training and test; these *recurring* files should be "easy prey" for stagnant models. In contrast (set 2) files which are defective in the training set but not in the test set, are *bait* which might trap a stagnant model into a false positive prediction. Finally files which are defective in the test set, but not in the training set, are *decoys*, which might mislead a stagnant model into a false negative prediction.

For each type of metrics and every *test release*, we rank the files in that release by their predicted defect proneness. We then normalize these ranks using the maximum possible rank (which is the number of files in the associated test release) and partition the normalized ranks in three sets as discussed above. We can then compare the averaged normalized ranks (averaged by the size of the partition) across different partitions for different test releases and metrics types. Normalization allows comparison between releases with varying numbers of files. A better model produces a higher normalized ranks for defective files.

Figure 6a compares average normalized ranks for different types of metrics for different defect occurrences. As we can see from the figure, even for just the recurringly defective files (left plot), process metrics outperform code metrics. Process metrics also (middle plot) avoid ranking "training only" defective files higher, thus doing better at avoiding such false positives. Finally, process metrics can sniff out the files with newly introduced defects ("test only") better. We compared the average normalized rank of the models from each pair of metrics using Wilcoxon test and corrected the p-values using BH correction. The p-values confirm statistical significance of the superiority of process metrics over code metrics. Process metrics were always better (lower normalized rank for "training only", and higher normalized rank for other two cases) than code metrics ($p < 0.001$). Process metrics also outperformed all metrics with lower "training only" ranking ($p < 0.001$). This suggests that in all three defect occurrence scenarios, process metrics are better-suited for prediction models.

All our findings clearly indicate that process metrics are superior to code metrics for building prediction models. Still code metrics are easy to use: multiple tools support easy gathering of code metrics. Their AUC (around 0.8) performance, though inferior to process metrics, is not bad. If process metrics are hard to use, should one use code metrics instead?

Prior work by Arisholm *et al.* [2], in a cross-validation evaluation suggests that in a cost-constrained setting, code metrics are not as useful, and indeed, when measured with AUCEC, don't do much better than random. We supplement Arisholm *et al.* with evaluation in a prediction setting, to understand why code metrics perform so poorly in terms of AUCEC, while giving reasonable AUC. As both AUC and

AUCEC rely on the ordering of defective entities based on the predicted defect probabilities, we conjecture that code metrics based models are clearly prioritizing less defect dense files. Furthermore, our findings suggest that code-metrics models are fairly good at predicting recurringly defective files. These two pieces of evidence suggests that code metrics tend to predict *recurringly defective, but not very defect-dense files*.

Question: *Are recurringly defective files larger and less defect dense, thereby rendering the models, with prediction bias towards such files, less cost- effective?*

Figure 6b compares the Log SLOC and defect density of files that were recurringly defective (defective in both training and test release), with the files that only became defective in the test release. As is evident from the figure, larger files are less defect dense and more likely to stay defective in subsequent releases. We also confirmed the statistical significance of this phenomenon using Wilcoxon test with BH correction. Recurringly defective files are statistically significantly larger ($p < 0.001$) than files that are only defective in the test set. At the same time, files that are defective only in the test set, have statistically significantly higher defect density ($p < 0.001$). Given our observed prediction bias of code metrics based models towards recurringly defective files, such models would be at a disadvantage to predict cost-effectively. Moreover, the inability of code metrics based prediction models to predict newly introduced defects, which may be more defect dense, would only worsen the cost-effectiveness of such models.

V. Threats to Validity

Data Quality We use a large selection of projects from different domains. All projects use a high-fidelity process to link bug-fixing commits to issues in JIRA issue tracking system. Our projects have a median defect linking rate of over 80%, which is much higher than reported (typically under 50%) in the literature [3].

Completeness of Code Metrics We only used code metrics as available from Scitool's popular UNDERSTAND tool. UNDERSTAND does not generate all possible code metrics ever reported or used in literature. However, it does produce a large set of diverse code metrics. Furthermore, the AUC of our code metrics based models are similar or better than that reported by Arisholm *et al.* [2]. Our comparisons of process and code metrics based models have large effect sizes; thus our results appear fairly robust to the choice of code metrics.

Completeness of Process Metrics Our set of process metrics are easily obtained, and based on a single release. We used a diverse set of process metrics widely used in the literature ranging from code ownership and developer experience to file activities. Our neighborhood-based metrics are motivated by highly cited research of Kim *et al.* [10]. Our location of change metric (SCTR) measures simple change entropy [8]. We therefore argue that our set of process metrics are comprehensive.

Stability Analysis We study stability in the context of releases, instead of a continuously shifting time window like Ekanayake *et al.* [5]. We argue that each release is a self contained logically defined epoch, more easily comparable to other releases; furthermore many budgeting/resource decisions are based on a release granularity. Thus we believe this is a suitable granuarity for evaluating prediction models.

Generalizability We use large number of projects consisting of 85 releases. We observed small variances in all of our findings. To avoid ecological fallacy [20], we also compared our findings in a per project setting, and got very similar results. Therefore we believe our result should be generalizable in similar application domains and development dynamics in a release oriented prediction context. However, all of our projects are developed in Java, and are OSS. There is a definite threat to generalizability based on the fact that all are Apache projects; however, we believe this threat is ameliorated by the diversity of the chosen projects. Also, our findings may be less generalizable for commercial projects, which have a completely different governance style, and may demonstrate different influence of process metrics on defect-proneness of files. Therefore we hope other researchers with access to commercial data would replicate our findings.

VI. Conclusion

We studied the efficacy of process and code metrics for defect prediction in a release-oriented setting across a large number of releases from a diverse set of projects. We compared models from different types of metrics using both cost-sensitive and AUC based evaluation across different objectives of performance, stability and portability to understand "when" a set of metrics may be suitable for an organization. Our results strongly suggest the use of process metrics instead of code metrics. We also try to understand "why" a set of metrics may predict a type of defect occurrence by focusing on the *stasis* of metrics. Our findings surprisingly show that code metrics, which is widely used in the literature, may not evolve with the changing distribution of defects, which leads code-metric-based prediction models *stagnating*, and tending to focus on files which are recurringly defective. Finally, we observed that such recurringly defective files are larger and less defect dense, therefore these large files may compromise the cost-effectiveness of the stagnant code-metric-based models with a prediction bias towards such files.

References

[1] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE*, pages 215–224. IEEE Computer Society, 2007.

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

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th FSE*, pages 121–130. ACM, 2009.

[4] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT FSE*, pages 4–14. ACM, 2011.

[5] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein. Tracking concept drift of software projects using defect prediction quality. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 51–60, Washington, DC, USA, 2009. IEEE Computer Society.

[6] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein. Time variance and defect prediction in software projects - towards an exploitation of periods of stability and change as well as a notion of concept drift in software projects. *Empirical Software Engineering*, 17(4-5):348–389, 2012.

[7] K. El Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE TSE*, 27(7):630–650, 2001.

[8] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE*, pages 78–88. IEEE, 2009.

[9] T. Khoshgoftaar and J. Munson. Predicting software development errors using software complexity metrics. *Selected Areas in Communications, IEEE Journal on*, 8(2):253–261, 1990.

[10] S. Kim, T. Zimmermann, E. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th ICSE*, pages 489–498. IEEE Computer Society, 2007.

[11] A. G. Koru and H. Liu. Identifying and characterizing change-prone classes in two large-scale open-source products. *Journal of Systems and Software*, 80(1):63–73, 2007.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE TSE*, 34(4):485–496, July 2008.

[13] Y. Ma and B. Cukic. Adequate and precise evaluation of quality models in software engineering studies. In *Proceedings of the 29th ICSE Workshops*, ICSEW '07, pages 68–, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

[15] T. Mende. Replication of defect prediction studies: problems, pitfalls and recommendations. In T. Menzies and G. Koru, editors, *PROMISE*, page 5. ACM, 2010.

[16] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE TSE*, 33(1):2–13, 2007.

[17] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[18] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 181–190. ACM, 2008.

[19] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM*, pages 364–373. IEEE Computer Society, 2007.

[20] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *ASE'2011*, pages 362–371. IEEE, 2011.

[21] F. Rahman and P. T. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 491–500. ACM, 2011.

[22] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *the 20th ACM SIGSOFT FSE*, pages –. ACM, 2012.

[23] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: hit or miss? In *FSE*, pages 322–331. ACM, 2011.

[24] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw. Engg.*, 14(5):540–578, Oct. 2009.

[25] B. Turhan, A. T. Misirli, and A. B. Bener. Empirical evaluation of mixed-project defect prediction models. In *EUROMICRO-SEAA*, pages 396–403. IEEE, 2011.

[26] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *ESE*, 13(5):539–559, 2008.

[27] H. Zhang. On the distribution of software faults. *IEEE TSE*, 34(2):301–302, March-April 2008.

[28] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 91–100. ACM, 2009.

[29] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.