Less is More: Minimizing Code Reorganization using XTREE

Rahul Krishna[a,*], Tim Menzies[a,*], Lucas Layman[b]

[a]*Department of Computer Science, North Carolina State University, Raleigh, NC, USA*
[b]*Fraunhofer CESE, College Park, USA*

Abstract

Context: Developers use bad code smells to guide code reorganization. Yet developers, textbooks, tools, and researchers disagree on which bad smells are important. How can we offer reliable advice to developers about which bad smells to fix?
Objective: To evaluate the likelihood that a code reorganization to address bad code smells will yield improvement in the defect-proneness of the code.
Method: We introduce XTREE, a framework that analyzes a historical log of defects seen previously in the code and generates a set of useful code changes. Any bad smell that requires changes outside of that set can be deprioritized (since there is no historical evidence that the bad smell causes any problems).
Evaluation: We evaluate XTREE's recommendations for bad smell improvement against recommendations from previous work (Shatnawi, Alves, and Borges) using multiple data sets of code metrics and defect counts.
Results: Code modules that are changed in response to XTREE's recommendations contain significantly fewer defects than recommendations from previous studies. Further, XTREE endorses changes to very few code metrics, so XTREE requires programmers to do less work. Further, XTREE's recommendations are more responsive to the particulars of different data sets. Finally XTREE's recommendations may be generalized to identify the most crucial factors affecting multiple datasets (see the last figure in paper).
Conclusion: Before undertaking a code reorganization based on a bad smell report, use a framework like XTREE to check and ignore any such operations that are useless; i.e. ones which lack evidence in the historical record that it is useful to make that change. Note that this use case applies to both manual code reorganizations proposed by developers as well as those conducted by automatic methods.

Keywords: Bad smells, performance prediction, decision trees.

1. Introduction

According to Fowler [1], bad smells (a.k.a. code smells) are "a surface indication that usually corresponds to a deeper problem". Fowler recommends removing code smells by

> "... applying a series of small behavior-preserving transformations, each of which seem 'too small to be worth doing'. The effect of these refactoring transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors".

While the original concept of bad smells was largely subjective, researchers including Marinescu [2] and Munro [3] provide definitions of "bad smells" in terms of static code attributes such as size, complexity, coupling, and other metrics. Consequently, code smells are captured by popular static analysis tools, like PMD[1], CheckStyle[2], FindBugs[3], and SonarQube[4].

We refer to the process of removing bad smells as *code reorganization*. Code reorganization is an amalgum of perfective and preventive maintenance [4]. In contrast to refactoring, code reorganization is not guaranteed to preserve behavior. Fowler [1] and other influential software practitioners [5, 6] recommend refactoring and code reorganization to remove bad smells. Studies suggest a relationship between code smells and poor maintainability or defect proneness [7, 8, 9], though these findings are not always consistent [10].

The premise of this paper is that not every bad smell needs to be fixed. For example, the origins of this paper was a meeting with a Washington-based software company where managers shared one of their challenges: their releases were delayed by developers spending much time removing bad smells within their code. There is much evidence that this is a common problem. Kim et al. [11] surveyed developers at Microsoft and found that code reorganizations incur significant cost and risks. Researchers are actively attempting to demonstrate the actual costs and benefits of fixing bad smells [12, 13, 14], though more case studies are needed.

In this paper, we focus on the challenge of recommending code reorganizations that result in perceivable benefits (such as reduced defect proneness) and avoid those reorganizations which have no demonstrable benefit and thus waste effort. To this end, this paper evaluates XTREE [15], a framework to evaluate whether a code reorganization is likely to have a perceiv-

*Corresponding author: Tel:+1-919-396-4143(Rahul)
Email addresses: i.m.ralk@gmail.com (Rahul Krishna[a,]), tim.menzies@gmail.com (Tim Menzies[a,]), llayman@cese.fraunhofer.org (Lucas Layman[b])

[1] https://github.com/pmd/pmd
[2] http://checkstyle.sourceforge.net/
[3] http://findbugs.sourceforge.net/
[4] http://www.sonarqube.org/

able benefit in terms of defect-proneness. We focus on bad smells indicated by code metrics such as size and complexity as captured in popular tools such as SonarQube and Klockwork[5]. XTREE examines the historical record of code metrics for a project. If there is no evidence that changing code metric "X" is useful (e.g., lowering "X" reduces defect-proneness), then developers should be discouraged from wasting effort on that change.

Our method uses two oracles: a *primary change oracle* and a *secondary verification oracle*. By combining these two oracles, we can identify and validate useful code reorganizations.

We use the XTREE cluster delta algorithm as the *primary change oracle*. XTREE explores the historical record of a project to find clusters of modules (e.g., files or binaries). It then proposes a "minimal" set of changes Δ that can move a software module M from a defective cluster C_0 to another with fewer defects C_1 (so Δ is some subset of $C_1 - C_0$).

The *secondary verification oracle* checks if the primary oracle is proposing sensible changes. We create the verification oracle using Random Forest [16] augmented with SMOTE (synthetic minority over-sampling technique [17]). In our framework, learning the secondary oracle is a *separate* task from that of learning the primary oracle. This ensures that the verification oracle offers an independent opinion on the value of the proposed changes.

An advantage to the XTREE cluster delta approach is that it avoids the *conjunctive fallacy*. A common heuristic in the bad smell literature [18, 19, 20, 21, 22] is: for all static code measures that exceed some threshold, make changes such that the thresholds are no longer exceeded. That is:

$$\begin{array}{rl} bad & = (a_1 > t_1) \bigvee (a_2 > t_2) \bigvee ... \\ better & = \neg\, bad = (a_1 \leq t_1) \bigwedge (a_2 \leq t_2) \bigwedge ... \end{array} \quad (1)$$

We say that the above definition of "better" is a conjunctive fallacy since it assumes that the best way to improve code is to decreases multiple code attribute measures to below t_i in order to remove the "bad" smells. In reality, the associations between static code measures are more intricate, for example, *decreasing* a_i may necessitate *increasing* a_j. It is easy to see why this is so. For example, Fowler recommends that the Large Class smell be addressed by the Extract Class or Extract Subclass refactoring [1]. If we pull code out of a function since that function has grown too large, that functionality has to go somewhere else. Thus, when *decreasing* lines of code in one module, we may *increase* its coupling to another module. XTREE identifies such associations between metrics, thus avoiding the conjunctive fallacy.

1.1. Research Questions

This paper claims that (a) XTREE is more useful than Equation 1 to identify code reorganizations, and (b) XTREE recommends a small subset of static code measures to change and thus can be used to identify superfluous code reorganizations (i.e., reorganizations based on omitted static code measures).

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

To evaluate these claims, we compared the performance of XTREE with three other methodologies/tools for recommending code reorganizations: (1) VARL based thresholds [20]; (2) Statistical threshold generation [21]; (3) CD, cluster based framework [23] according to the following research questions:

RQ1: Effectiveness: According to the verification oracle, which of the frameworks introduced above is most accurate in recommending code reorganizations that result in reduced numbers of defective modules?

To answer this question, we used data from five OO Java projects (Ivy, Lucene, Ant, Poi, Jedit). It was found that:

Result 1
XTREE is the most accurate oracle on how to change code modules in order to reduce the number of defects.

RQ2: Succinctness: Our goal is to critique and, possibly, ignore irrelevant recommendations for removing bad smells. If the recommended changes are minimal (i.e. affect fewest attributes) then those changes will be easiest to apply and monitor. So, which framework recommends changes to the fewest code attributes?

Result 2
Of all the code change oracles studied, XTREE recommends changes to the fewest number of static code measures.

RQ3: Stopping: Our goal is to discourage code reorganization based on changes that lack historical evidence of being effective. So, how effective is XTREE at identifying what *not* to change?

Result 3
In any project, XTREE's recommended changes to 1–4 of the static code attributes. Any bad smell defined in terms of the remaining 19 to 16 code attributes (i.e. most of them) would hence be deprecated.

RQ4: Stability: Across different projects, how consistent are the changes recommended by our best change oracle?

Result 4
The direction of change recommended by XTREE (e.g., to LOWER lines of code while RAISING coupling) is stable across repeated runs of change oracle.

RQ5: Conjunctive Fallacy: Is it always useful to apply Equation 1; i.e. make code better by reducing the values of multiple code attributes? We find that:

Result 5
XTREE usually recommends reducing lines of code (size of the modules). That said, XTREE often recommends increasing *the values of other static code attributes.*

Note that **RQ3, RQ4, RQ5** confirms the intuitions of the project managers that prompted this investigation. We find evidence that:

- Code reorganizations that decrease multiple measures may not yield improvement. In fact, programmers may need to *decrease* some measures while *increasing* others.
- In the studied projects, XTREE recommends changes to approximately 20% of the code measures, and thus reorganizations based on the remaining 80% are unlikely to provide benefit.

In terms of concrete recommendations for practitioners, we say *look before you leap*:

- Before doing code reorganization based on a bad smell report...
- ... check and discourage any code reorganization for which there is no proof in the historical record that the change improves the code.

Aside: this recommendation applies to both manual code reorganizations proposed by developers as well as the code reorganizations conducted by automatic methods [24]. That is, XTREE could optimize automatic code reorganization by discouraging reorganizations for useless goals.

(Reviewer1a) Beside this introduction, the rest of this paper is formatted as follows. §2 relates the current work to the prior literature on bad smells and code reorganization efforts. In §3, we discuss the choice of our datasets and briefly describe it's structure. §4 details the frequently used techniques on leaning bad smell thresholds to assist reorganization. Specifically, §4.2.2 and §4.2.3 describe our preferred method for performing code reorganizations. Our experimental setup and evaluation strategies are presented in §5. The results and corresponding descriptions are available in §6. The future work and the reliability of our finding are available in §7 and §8 respectively. Finally, the conclusions are presented in §9.

2. Relationship to Prior Work

2.1. Prioritizing Reorganization Efforts

There have been several efforts to prioritize refactoring efforts. These attempts address code smells in particular and have garnered more attention over the past few years. Ouni et al. [25, 26, 27] use search based software engineering to suggest refactoring solutions. They recommend the use of four factors: (1) priority; (2) severity; (3) risk; and (4) importance, all of which are determined by developers. Both the developer's recommendations and the aforementioned factors can and do vary over time especially as classes are modified. Additionally, they also vary with projects.

A similar direction was taken by Vidal et al. [28]. They presented a semi-automated approach for prioritizing code smells. They then recommend a suitable refactoring based on a developer survey. The determination of severity is based primarily on three criteria: (1) the stability of the component in which the smell was found; (2) the subjective assessment that the developer makes of each kind of smell using an ordinal scale; and (3) the related modifiability scenarios.

The standard approach is to develop and evaluate these findings by interviewing human developers. We, however, dissuade practitioners from taking this approach for reasons discussed in depth in §2.3.

In a more recent study, a slightly different approach was proposed by Vidal et al. [29]. In their work, static code metrics are used to detect the presence of code smells. This is followed by a ranking scheme to measure the severity of code smells. This was a fully automated approach. However, the authors fail to report the accuracy of detection of code smells. This issue is further compounded by the use of mean and standard deviation of metrics to detect the presence of code smells which has been criticized by several researchers [20, 21]. Those authors do acknowledge that the prioritization of detection results obtained using their "intensity" measure, at the time of publication, lacked comprehensive experimental validation.

2.2. Preliminary Report on Code Reorganization

There is a distinction between our work and all methods listed above. The code smell prioritization efforts assist developers in choosing *which* refactoring operation to undertake first. Instead of prioritizing refactoring, our work places more focus on assisting developers by recommending useful code changes which in turn helps deprioritize certain code reorganization efforts. Our preferred framework to achieve this is XTREE.

XTREE was first introduced as a four page preliminary report[6] which was presented previously [15]. That short report offered case studies on only two of the five data sets studied here. We greatly expand on this prior work by:

- Evaluating XTREE's recommendations against recommendations from frameworks developed using three other methods proposed by other researchers exploring bad smells.
- Evaluating if XTREE's and other methods' recommended changes were sensible using the *secondary verification oracle*

Note that only sections §4.2.1, §4.2.2, and two-fifths of the results in Figure 6 contain material found in prior papers.

2.3. Why Not Just Ask Developers to Rank Bad Smells?

Why build tools like XTREE to critique proposed developer actions? Much research endorses code smells as a guide for code improvement (e.g., code reorganization or preventative maintenance). A recent literature review by Tufano et al. [30] lists dozens of papers on smell detection and repair tools. Yet other papers cast doubt on the value of bad smells as triggers for code improvement [31, 32, 33].

If the SE literature is contradictory, why not ignore it and use domain experts (software engineers) to decide what bad smells

[6] https://goo.gl/2In3Lr

Fowler'99 [1] and [37]	Lanza'06 [2]	SonarQube [38]	Yamashita'13[32]	Developer Survey 2015
Alt. Classes with Diff. Interfaces				
Combinatorial Explosion [37]				
Comments			11	VL
Conditional Complexity [37]			14	?
Data Class	✓			
Data Clumps				
Divergent Change				
Duplicated Code	✓	✓	1	VH
Feature Envy	✓		8	
Inappropriate Intimacy		✓		L
Indecent Exposure [37]				?
Incomplete Library Class				
Large Class	✓	✓	4	VH
Lazy Class/Freeloader		✓	7	
Long Method	✓	✓	2	VH
Long Parameter List		✓	9	L
Message Chains				H
Middle Man				
Oddball Solution [37]				
Parallel Inheritance Hierarchies				
Primitive Obsession				
Refused Bequest	✓	✓		
Shotgun Surgery	✓			
Solution Sprawl [37]				
Speculative Generality				L
Switch Statements				L
Temporary Field		✓		?

Figure 1: Bad smells from different sources. Check marks (✓) denote a bad smell was mentioned. Numbers or symbolic labels (e.g. "VH") denote a priorization comment (and "?" indicates lack of consensus). Empty cells denote some bad smell listed in column one that was not found relevant in other studies. Note: there are many blank cells.

to fix? We do not recommend this since developer *cognitive biases* can mislead them to assert that some things are important and relevant when they are not. According to Passos et al. [34], developers often assume that the lessons they learn from a few past projects are general to all their future projects. They comment, "past experiences were taken into account without much consideration for their context" [34]. Jørgensen & Gruschke [35] offer a similar warning. They report that the supposed software engineering "gurus" rarely use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects. [35].

Other studies have shown some widely-held views are now questionable given new evidence. Devanbu et al. examined responses from 564 Microsoft software developers from around the world. They comment programmer beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project [36]. If the above remarks hold true for bad smells, then we would expect to see much disagreement on which bad smells are important and relevant to a particular project.

This is indeed the case. The first column of Figure 1 lists commonly mentioned bad smells and comes from Fowler's 1999 text [1] and a subsequent 2005 text by Kerievsky that is widely cited [37]. The other columns show data from other studies on which bad smells matter most. The columns marked as Lanza'06 and Yamashita'13 are from peer reviewed literature. The column marked SonarQube is a popular open source code assessment tool that includes detectors for six of the bad smells in column one. The *developer survey* (in the right-hand-side column) shows the results of an hour-long whiteboard session with a group of 12 developers from a Washington D.C. web tools development company. Participants worked in a round robin manner to rank the bad smells they thought were important (and any disagreements were discussed with the whole group). Amongst the group, there was some consensus on the priority of which bad smells to fix (see the annotations VH=very high, H=high, L=low, VL=very low, and "?"= no consensus).

A blank cell in Figure 1 indicates where other work omits one of the bad smells in column one. Note that most of the cells are blank, and that the studies omit the majority of the Fowler bad smells. SonarQube has no detectors for many of the column one bad smells. Also, nearly half the Yamashita list of bad smells does not appear in column 1. The eight numbers in the Yamashita'13 column show the rankings for the bad smells that overlap with Fowler and Kerievsky; Yamashita also discussed other smells not covered in Fowler'99 [1].

Two of the studies in Figure 1 offers some comments on the relative importance of the different bad smells. Three of the bad smells listed in the top half of the Yamashita'13 rankings also score very high in the developer survey. Those three were *duplicated code, large class*, and *long method*. Note that this agreement also means that the Yamashita'13 study and the developer survey believe that very few code smells are high priority issues requiring code reorganization.

In summary, just because one developer strongly believes in the importance of a bad smell, it does not mean that belief transfers to other developers or projects. Developers can be clever, but their thinking can also be distorted by cognitive biases. Hence, as shown in Figure 1, developers, text books, and tools can disagree on which bad smells are important. Special tools are needed to assess their beliefs, for example, their beliefs in bad smells.

3. Why Use Defect Data?

To assess our planning methods, we opted to use data gathered by Jureczko et al. for object-oriented JAVA systems [39]. The "Jureczko" data records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of nearly two dozen metrics such as number of children (noc), lines of code (loc), etc. For details on the Jureczko data, see Figure 2. The nature of collected data and its relevance to defect prediction is discussed in greater detail by Madeyski & Jureczko [40].

A sample set of values from a data set (ant 1.3) is shown in Figure 3. Each instance notes a class under consideration, 20 static code metric values, and 2 columns counting the defects in the code.

The term "defect" in our work always refers to *defective classes*. The defects themselves are represented in two ways:

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

Figure 2: OO code metrics used for all studies in this paper. Last lines, shown in gray, denote the dependent variables.

1. Raw defect counts (denoted as `nDefects`): This refers total number of defects present in a given class. This is an integer value, such that, $nDefects \in \{0,1,2,\ldots\}$. This representation of defects is used by all the *primary change oracles*.

2. Boolean defects (denoted as `isDefective`): This is a Boolean representation of defects. It is `TRUE` if `nDefects` $>$ 0 otherwise it is `FALSE`. This representation of defects is used by the *secondary verification oracle*.

We use defects to operationalize smell definitions following the findings of several researchers. Li & Shatnawi [41] investigated the relationship between the bad smells and module defect probability. Their study found that, in the context of the post-release system evolution process, bad smells were positively associated with the defect probability in the three error-severity levels. They also reported that Shotgun Surgery, God Class, and God Method smells are associated with higher levels of defects. Olbrich et al. [10] show the impact that God and Brain Class smells have on code directly influences defects in systems.

In a more recent study, Hall et al. [42] further corroborate the claim that smells indicate defect-prone code in several circumstances. Their evaluation of smell detection performance shows that it is difficult to define and quantify smell definitions, either for automatic or manual smell detection. Generally, agreement levels on what code contains a smell are poor between tools, between tools and humans, and even between humans. This in general leads to poor performance of tools that detect code smells. More importantly, they note that arbitrary refactoring is unlikely to significantly reduce fault-proneness and in some cases may increase fault-proneness. Our findings also show that smells have different effects on different systems many of which cannot be quantified with just code smell metrics. These findings lead us to conclusion that using static code metrics and associated defects would best support code reorganization.

		Metrics			Class	
Version	Module Name	wmc	dit	...	nDefects	isDefective
1.3	org.apache.tools.ant.taskdefs.ExecuteOn	11	4	...	0	FALSE
1.3	org.apache.tools.ant.DefaultLogger	14	1	...	2	TRUE
...	...	...	...	...	0	FALSE
1.3	org.apache.tools.ant.taskdefs.Cvs	12	3	...	0	FALSE
1.3	org.apache.tools.ant.taskdefs.Copyfile	6	3	...	1	TRUE

Figure 3: A sample of ant 1.3

4. Learning Bad Smell Thresholds

Having made the case for automated, evidence-based support for assessing bad smells, this section reviews different ways for building those tools (one of those tools, XTREE, will be our recommended *primary change oracle*). Later in this paper, we describe a *secondary verification oracle* that checks the effectiveness of the proposed changes.

The SE literature offers two ways of learning bad smell thresholds. One approach relies on *outlier statistics* [18, 19]. This approach has been used by Shatnawi [20], Alves et al. [21] and Hermans et al. [22]. Another approach is based on *cluster deltas* that we developed for Centroid Deltas [23] and is used here for XTREE. These two approaches are discussed below.

4.1. Outlier Statistics

The outlier approach assumes that unusually large measurements indicate risk-prone code. Hence, they generate one bad smell threshold for any metric with such an "unusually large" measurement. The literature lists several ways to define "unusually large".

4.1.1. Enri & Lewerentz

Given classes described with the code metrics of Figure 2, Enri and Lewerentz [18] found the mean μ and the standard deviation σ of each code metrics. Their definition of problematic outlier was any code metric with a measurement greater than $\mu + \sigma$. Shatnawi and Alves et al. [20, 21] depreciate using $\mu + \sigma$ since it does not consider the fault-proneness of classes when the thresholds are computed. Also, the approach lacks empirical verification.

4.1.2. Shatnawi

Shatnawi [20]'s preferred alternative to $\mu + \sigma$ is to extend the use VARL (Value of Acceptable Risk Level) which was initially proposed by Bender [19] for his epidemiology studies. This approach uses two constants (p_0 and p_1) to compute the thresholds which, following Shatnawi's guidance, we set to $p_0 = p_1 = 0.05$.

VARL encodes the defect count for each class as 0 (no defects known in class) or 1 (defects known in class). Univariate binary logistic regression is applied to learn three coefficients: α is the intercept constant; β is the coefficient for maximizing log-likelihood; and p_0 measures how well this model predicts for defects. A univariate logistic regression was conducted comparing metrics to defect counts. Any code metric with $p > 0.05$ is ignored as being a poor defect predictor. Thresholds are then learned from the surviving metrics M_c using the risk equation proposed by Bender:

$$\textit{bad smell if } M_c > VARL$$

Where,

$$VARL = p^{-1}(p_0) = \frac{1}{\beta}\left(\log\left(\frac{p_1}{1-p_1}\right) - \alpha\right) \qquad (2)$$

4.1.3. Alves et al.

Alves et al. [21] propose another approach that uses the underlying statistical distribution and scale of the metrics. Metric values are weighted according to the source lines of code (SLOC) of the class. All the weighted metrics are then normalized by the sum of all weights for the system. The normalized metric values are ordered in an ascending fashion (this is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale). Alves et al. then select a percentage value (they suggest 70%) which represents the "normal" values for metrics. The metric threshold, then, is the metric value for which 70% of the classes fall below. The intuition is that the worst code has outliers beyond 70% of the normal code measurements i.e., they state that the risk of there existing a defect is moderate to high when the threshold value of 70% is exceeded.

Hermans et al. [22] used this approach in their 2015 paper on exploring bad smells. We explore the correlation between the code metrics and the defect counts with a univariate logistic regression and reject code metrics that are poor predictors of defects (i.e. those with $p > 0.05$).

4.1.4. Discussion of Outlier Methods

The advantage of the outlier-based approaches is that they are simple to implement, but the approaches have two major disadvantages. First, they are *verbose*. A threshold can be calculated for every metric – so, which one should the developers focus on changing? Without a means for prioritizing the thresholds and metrics against one another, developers may have numerous or conflicting recommendations on what to improve. Second, the outlier approaches suffers from *conjunctive fallacy* discussed in the introduction. That is, while they propose thresholds for many code metrics individually, they make no comment on what minimal metrics need to be changed at the same time (or whether or not those changes lead to minimization or maximization of static code measures).

4.2. Cluster Deltas

Cluster deltas are a general framework for learning *conjunctions* of changes that need to be applied at the same time. This approach works as follows:

- Cluster the data.
- Find neighboring clusters C_+, C_- (where C_+ has more examples of defective modules than C_-);
- Compute the delta in code metrics between the clusters using $\Delta = C_- - C_+ = \{\delta | \delta \in C_-, \delta \notin C_+\}$, i.e. *towards* the cluster with lower defects;
- The set Δ are changes needed in defective modules of C_+ to make them more like the less-defective modules of C_-

Note that Δ is a conjunction of recommendations. Since it is computed from neighboring clusters, the examples contain similar distributions and Δ respects the naturally occurring constraints in the data. For example, given a bad smell pertaining to large methods, Δ will not suggest lowering lines of code without also increasing a coupling measure. Cluster deltas are used in CD [23] and XTREE.

4.2.1. CD

Borges and Menzies first proposed CD centroid deltas to generate *conjunctions* of code metrics that need to be changed at the same time in order to reduce defects [23]. CD uses the WHERE clustering algorithm developed by the authors for a prior application [43]. Each cluster was then replaced by its centroid and Δ was calculated directly from the difference between code metric values between one centroid and its nearest neighbor.

One drawback with CD is that it is *verbose* since CD recommended changes to all code metrics with different values in those two centroids. This makes it hard to use CD to critique and prune away bad smells. Further, CD will be shown not to be as effective in proposing changes to reduce defects as XTREE.

4.2.2. XTREE: Mining Project History for Defect-Proneness Attributes

XTREE is a cluster delta algorithm that avoids the problem of verbose Δs. XTREE is our *primary change oracle* that makes recommendations of what changes should be made to code modules. Instead of reasoning over cluster centroids, XTREE utilizes a decision tree learning approach to find the fewest differences between clusters of examples.

XTREE uses a multi-interval discretizer based on an iterative dichotomization scheme, first proposed by Fayyad and Irani [44]. This method converts the values for each code metric into a small number of nominal ranges. It works as follows:

- A code metric is split into r ($r = 2$) ranges, each range is of size n_r and is associated with a set of defect counts x_r with standard deviation σ_r.

On the right-hand-side is a tree generated by iterative dichomization. This tree can be read like a nested if-then-else statement; e.g.

- Lines 3 and 8 show two branches for lines of code (denoted here as *'$loc*) below 698 and above 698.
- Any line with a colon ":" character shows a leaf of this nesting. For example, if some new code module is passed down this tree and falls to the line marked in **orange**, the colon on that line indicates a prediction that this module has a 100% chance of being defective.

Using this tree, XTREE looks for a nearby branch that has a lower chance of being defective. Finding the green desired branch, XTREE reports a bad smell threshold for that module that is the delta between the **orange** current branch and green designed branch. In this case, that threshold relates to:

- Lines of code and comments (*lcom*)
- The cohesion between classes (*cam*) which measures similarity of parameter lists to assess the relatedness amongst class methods.

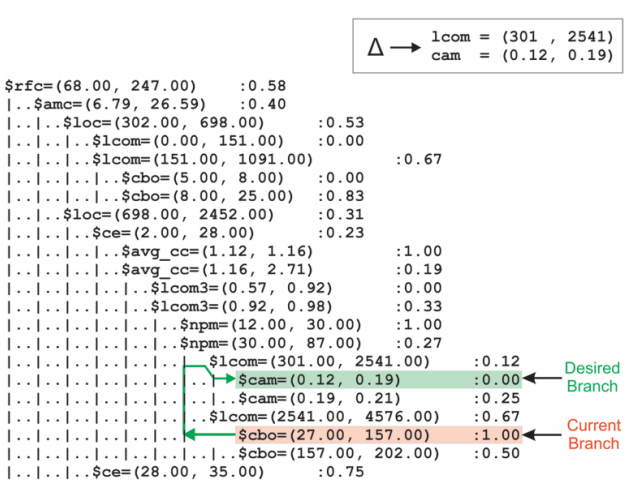

Figure 4: A brief tutorial on XTREE.

- The best split for that range is the one that minimizes the expected value of the defect variance, after the split; i.e. $\sum_r \frac{n_r}{n}\sigma_x$ (where $n = \sum_r n_r$).
- This discretizer then recurses on split to find other splits in a recursive fashion. As suggested by Fayyad and Irani, minimum description length (MDL) is used as a termination criterion for the recursive partitioning.

When discretization finishes, each code metric M has a final expected value M_v for the defect standard deviation across all the discretized ranges of that metric. Iterative dichomization sorts the metrics by M_v to find the code metric that best splits the data i.e., the code metric with smallest M_v.

A decision tree is then constructed on the discretized metrics. The metric that generated the best split forms the root of the tree with its discrete ranges acting as the nodes.

When all the metrics are arranged this way, the process is very similar to a hierarchical clustering algorithm that groups together code modules with similar defect counts and some shared ranges of code metrics. For our purposes, we score each cluster found in this way according to the percent of classes with known defects. For example, the last line of Figure 4 shows a tree leaf with 75% defective modules.

Figure 4 offers a small example of how XTREE builds Δ by comparing branches that lead to leaf clusters with different defect percentages. In this example, assume a project with a table of code metrics data describing its classes in the form of Figure 3. After code inspections and running test cases or operational tests, each such class is augmented with a defect count. Iterative dichomization takes that table of data and, generates the tree of Figure 4.

Once the tree is built, a class with code metric data is passed into the tree and evaluated down the tree to a leaf node (see the **orange** line in Figure 4). XTREE then looks for a nearby leaf node with a lower defect count (see the green line in Figure 4). For that evaluated class, XTREE proposes bad smell thresholds that are the differences between green and **orange**.

4.2.3. XTREE: Recommending Code Reorganizations

Using the training data construct a decision tree as suggested above.

Next, for each test code module, find C_+ as follows: take each test, run it down the decision tree to find a leaf in the decision tree that best matches the test case. After that, find C_- as follows:

- Starting at the C_+ leaf (level N), ascend $lvl \in \{N, N-1, N-2, ...\}$ tree levels;
- Identify *sibling* leaves; i.e. leaf clusters that can be reached from level lvl that are not same as *current* C_+;
- Find the *better* siblings; i.e. those with defect proneness 50% or less than that of C_+ (e.g., if defect-proneness of C_+ is 70%, find the nearest sibling with defect proneness $\leq$ 35%). If none found, then repeat for lvl–= 1. Also, return `nil` if the new lvl is above the root.
- Set C_- to the *closest* better sibling where distance is measured between the mean centroids of that sibling and C_+

Now find $\Delta = C_- - C_+$ by reflecting on the set difference between conditions in the decision tree branching from C_+ to C_-. To find that delta, for discrete attributes, delta is the value of the *desired*; for numerics expressed as ranges, the delta could be any value that lies between $(LOW, HIGH]$ in that range.

Note that XTREE's recommendation does not exhaustively search the tree for the change that reduces defect proneness the most, but rather finds the nearest sibling. This is by design. This design allows XTREE to a) provide recommendations quickly, and b) to recommend changes to a small number of attributes.

5. Setup

The previous section proposed numerous methods for detecting bad smells that need to be resolved. This section offers a way to evaluate them as follows:

- Use each framework discussed in §4 as a *primary change oracle* to recommend how code should be changed in our test data set (Section 5.1).
- Apply those changes. (This is emulated by changing the code metrics in order that all the Δ's are addressed.)[7]
- Run a *secondary verification oracle* to assess the defect-proneness of the changed code.
- Sort the change oracles on how well they reduce defects as judged by the verification oracle.

Using this, we can address the research questions discussed in the introduction.

RQ1: Effectiveness: Which of the methods defined in Section 3 is the best change oracle for identifying what and how code modules should be changed? We will assume that developers update and reorganize their code until the bad smell thresholds are not violated. This code reorganization will start with some *initial* code base that is changed to a *new* code base.

For example, assume that a log history of defects has shown that modules with $loc > 100$ have more defects (per class) than smaller modules and a code module has 500 lines of code. The action is to reduce the size of that module; we reason optimistically that we can change that code metric to 100. Using the secondary verification oracle, we then predict the number of defects in *new*.

We compare d_+, the number of defects in the *initial* code base to d_-, the number of defects in the *new* code base. We evaluate the performance of XTREE, CD, Shatnawi, and Alves change oracles. The best change oracle is the one that maximizes

$$improvement = 100 * \left(1 - \frac{d_-}{d_+}\right) \quad (3)$$

RQ2: Succinctness: Which of the Section 3 methods recommended changes to the fewest code attributes? To answer this question, we will report the frequency at which different attributes are selected in repeated runs of our oracles.

RQ3: Stopping: How effective is XTREE at offering "stopping points" (i.e. clear guidance on what *not* to do)? To answer this point, we will report how often XTREE's recommendations *omit* a code attribute. Note that the *more often* XTREE omits an attribute, the more likely it is *not* to endorse addressing a bad smell based on the omitted attributes.

RQ4: Stability: Across different projects, how variable are the changes recommended by XTREE? To answer this question, we conduct a large scale "what if" study that reports all the possible recommendations XTREE might make. We then count how often attributes are *not* found in the recommendations arising from this "what if" study.

RQ5: Conjunctive Fallacy: Is it always useful to apply Equation 1; i.e. make code better by reducing the values of multiple code attributes? To answer this question, we will look at the *direction of change* seen in the **RQ4** study; i.e. how often XTREE recommends decreasing or increasing a static code attribute.

5.1. Test Data

To explore these research questions, we used data from Jureczko et al.'s collection of object-oriented Java systems [39]. To access that data, go to git.io/vGYxc. The Jureczko data records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of nearly two dozen metrics included in the Chidamber and Kemerer metric suite, such as number of children (noc), lines of code (loc), etc. For details on the Jureczko code metrics, see Figure 2 and corresponding text in §3. For details on the versions of that data that were used for training and testing purposed see the left-hand-side columns of Figure 5.

5.2. Building the Secondary Verification Oracle

As mentioned in the introduction, our proposed framework has two oracles: a primary change oracle (XTREE) and the secondary verification oracle described in this section.

It can be difficult to judge the effects of removing bad smells. Code that is reorganized cannot be assessed just by a rerun of the test suite for three reasons: (1) Test suites may not be designed to identify code smells, they only report pass/fail based on whether or not a specific module runs. It is entirely possible that a test case may pass for code that contains one or more code smells; (2) While smells are certainly symptomatic of design flaws, not all smells cause the system to fail in manner suitable for identification by test cases; and (3) It make take a significant amount of effort to write new test cases that identify bad smells, especially for relatively stable software systems. Additionally, since we can not be sure if reorganizations do/don't change the system behavior, for instance cases where a reorganization effort involves only refactoring, it becomes difficult to assert when a test case will pass or fail.

To resolve this problem, SE researchers such as Cheng et al. [45], O'Keefe et al. [46, 47], Moghadam [48] and Mkaouer et al. [49] use a *secondary verification oracle* that is learned separately from the primary oracle. The verification oracles assesses how defective the code is before and after some code reorganization. For their second oracle, Cheng, O'Keefe, Moghadam and Mkaouer et al. use the QMOOD hierarchical quality model [50]. A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [43]. Hence, for this study, we eschew older quality models like QMOOD. Instead, we use Random Forests [16] to learn defect predictors from OO code metrics. Unlike QMOOD, the predictors are specific to the project.

Random Forests are a decision tree learning method but instead of building one tree, hundreds are built using randomly

[7] Note: This was done by selecting a random number from the $(LOW, HIGH]$ boundaries of the recommended delta(s). The random number indicates what may happen if a developer were to make changes to their code to comply with the recommendations.

	Data set properties					Results from learning								
	training		testing			untuned			tuned			change		
data set	versions	cases	versions	cases	% defective	pd	pf	good?	pd	pf	good?	pd	pf	
jedit	3.2, 4.0, 4.1, 4.2	1257	4.3	492	2	55	29		64	29	y	9	0	⋆
ivy	1.1, 1.4	352	2.0	352	11	65	35	y	65	28	y	0	-7	⋆
camel	1.0, 1.2, 1.4	1819	1.6	965	19	49	31		56	37		5	6	
ant	1.3, 1.4, 1.5, 1.6	947	1.7	745	22	49	13	y	63	16	y	14	3	⋆
synapse	1.0, 1.1	379	1.2	256	34	45	19		47	15		2	-4	
velocity	1.4, 1.5	410	1.6	229	34	78	60		76	60		-2	0	
lucene	2.0, 2.2	442	2.4	340	59	56	25		60	25	y	4	0	
poi	1.5, 2, 2.5	936	3.0	442	64	56	31		60	10	y	4	-21	⋆
xerces	1.0, 1.2, 1.3	1055	1.4	588	74	30	31		40	29		10	-2	×
log4j	1.0, 1.1	244	1.2	205	92	32	6		30	6		-2	0	×
xalan	2.4, 2.5, 2.6	2411	2.7	909	99	38	9		47	9		9	0	×

Figure 5: Training and test *data set properties* for Jureczko data , sorted by % defective examples. On the right-hand-side, we show the *results from learning*. Data is usable if it has a recall of 60% or more and false alarm of 30% or less (and note that, after tuning, there are more usable data sets than before). Results marked with "⋆" show large improvements in performance, after tuning (lower *pf* or higher *pd*). Data in the three bottom rows, marked with "×", are performing poorly– that data so many defective examples that it is hard for our learners to distinguish between classes.

selected subsets of the data. The final predictions come from averaging the predictions over all the trees. Recent studies endorsed the use of Random Forests for defect prediction [51].

Figure 5 shows our results with Random Forests and the Jureczko data. The goal is to build a verification oracle based on Random Forest that accurately distinguishes between defective and non-defective files based on code metrics. Given V released versions, we test on version V and train on the available data from $V-1$ earlier releases.

The three bottom rows are marked with ×: these contain predominately defective classes (two-thirds, or more). It is hard to build a model that distinguishes non-defective files from defective files in these data sets due to the high percentage of defective file examples.

We use Boolean classes in the Jureczko data to identify the presence or absence of defects: `True` if defects >0; `False` if defects = 0. The quality of the predictor is measured using (a) the probability of detection *pd* (i.e., recall): the percent of faulty classes in the test data detected by the *predictor*; and (b) the probability of false alarm *pf* (i.e., false positives): the percent of non-fault classes that are *predicted* to be defective.

5.2.1. Impact of Tuning

The "untuned" columns of Figure 5 show a preliminary study using Random Forest with its "off-the-shelf" tuning of 100 trees per forest. The forests were built from training data and applied to test data not seen during training. In this study, we called a data set "usable" if Random Forest was able to classify the instances with a performance threshold of $pd \geq 60 \wedge pf \leq 30\%$ (determined from standard results in other publications [52]). We found that no data set satisfy this criteria.

To salvage Random Forest we first applied the SMOTE algorithm to improve the performance of the classifier by handling class imbalance in the data sets. Pelayo and Dick [53] report that defect prediction is improved by SMOTE [17]; i.e. an oversampling of minority-class examples and an under-sampling of majority-class examples.

In a recent paper, Fu et al. [54] reported that parameter tuning with differential evolution [55] can quickly explore the tuning options of Random Forest to find better settings for the size of the forest, the termination criteria for tree generation, and other parameters. In a setting similar to theirs, we implemented a multiobjective differential evolution to tune Random Forests for each of the above datasets.

Setting goals for tuning can be a very difficult task. Fu et al. warn that choosing goals must be undertaken with caution. Opting to optimize individual goals such as recall/precision of a learner can have undesirable outcomes. For instance, by tuning for recall we can achieve near 100% recall – but at the cost of a near 100% false alarms. On the other hand, when we tune for false alarms, we can achieve near zero percent false alarm rates by effectively turning off the detector (so the recall falls to nearly zero). Therefore, in our work, we used used a "multiobjective" search to tune for both recall and false alarm at the same time. A multiobjective search attempts to find settings that achieve a *trade-off* between the goals (in this case maximizing recall and minimizing false alarm at the same time).

The effect of tuning and using SMOTE were remarkable.The rows marked with a ⋆ in Figure 5 show data sets whose performance was improved by these techniques. For example, in *poi*, the recall increased by 4% while the false alarm rate dropped by 21%. In *Ivy, Jedit, Lucene,* and *Xalan* there was a significant improvement in one measure (recall or false alarm) with no deterioration in the other measure. Finally, in *Poi* and *Ant* there was a significantly large improvement in one metric with a very small deterioration in the other.

(Reviewer2a) However, as expected, we found that some datasets (xerces, xalan, log4j, ...) were not responsive to our tuning efforts. Since, we could not salvage all the data sets, we eliminated these data sets for which we could not build an adequately performing Random Forest classifier with $pd \geq 60 \wedge pf \leq 30\%$. Thus, our analysis uses the *jedit, ivy, ant, lucene* and *poi* data sets for evaluating recommended changes.

We note that SMOTE-ing and parameter tunings were ap-

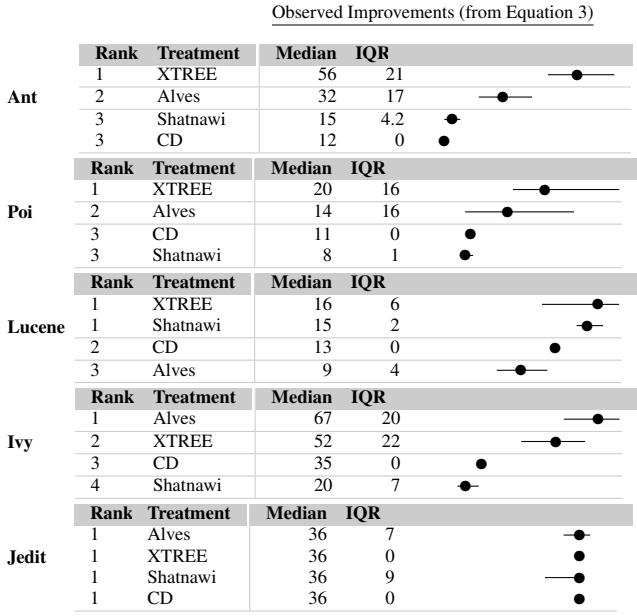

Figure 6: Results for **RQ1** from the Jureczko data sets. Results from 40 repeats. Values come from Equation 3. Values near 0 imply no improvement, *Larger* median values are *better*. Note that XTREE and Alves are usually best and CD and Shatnami are usually worse.

plied to the training data only and not to the test data.

5.3. Statistics

We use 40 repeated runs for each code reorganization recommendation framework for each of the five data sets (we use 40 since that is more than the 30 samples needed to satisfy the central limit theorem). Each code organization framework is trained on versions $1...N-1$ of N available versions in each data set. Each run collects the improvement scores defined in Equation 3: the reduction in the percentage of defective classes as identified by the verification oracle after applying the recommended code metric changes.

We use multiple runs with different random number seeds since two of our methods use some random choices: CD uses the stochastic WHERE clustering algorithm [43] while XTREE non-deterministically picks thresholds randomly from the high and low boundary of a range. Hence, to compare all four methods, we must run the analysis many times.

To rank these 40 numbers collected from CD, XTREE, Shatnawi, and Alves et al., we use the Scott-Knott test recommended by Mittas and Angelis [56]. Scott-Knott is a top-down clustering approach used to rank different treatments. If that clustering finds an interesting division of the data, then some statistical test is applied to the two divisions to check if they are statistically significant different. If so, Scott-Knott recurses into both halves.

To apply Scott-Knott, we sorted a list of $l = 40$ values of Equation 3 values found in $ls = 4$ different methods. Then, we split l into sub-lists m,n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l,m,n of size ls,ms,ns where $l = m \cup n$:

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

We then apply a apply a statistical hypothesis test H to check if m,n are significantly different (in our case, the conjunction of A12 and bootstrapping). If so, Scott-Knott recurses on the splits. In other words, we divide the data if *both* bootstrap sampling and effect size test agree that a division is statistically significant (with a confidence of 99%) and not a small effect ($A12 \geq 0.6$). For a justification of the use of non-parametric bootstrapping, see Efron & Tibshirani [57, p220-223]. For a justification of the use of effect size tests see Shepperd&MacDonell [58]; Kampenes [59]; and Kocaguenli et al. [60]. These researchers warn that even if a hypothesis test declares two populations to be "significantly" different, then that result is misleading if the "effect size" is very small. Hence, to assess the performance differences we first must rule out small effects using A12, a test recently endorsed by Arcuri and Briand at ICSE'11 [61].

The Scott-Knott results are presented in the form of line diagrams like those shown on the right-hand-side of Figure 6. The black dot shows the median Equation 3 values and the horizontal lines stretches from the 25th percentile to the 75th percentile (the inter-quartile range, IQR). As an example of how to read this table, consider the *Ant* results. Those rows are sorted on the median values of each framework. Note that all the methods have Equation 3 > 0%; i.e. all these methods reduced the expected value of the performance score while XTREE achieved the greatest reduction (of 56% from the original value). These results table has a left-hand-side **Rank** column, computed using the Scott-Knott test described above. In the *Ant* results, XTREE is ranked the best, while CD is ranked worst.

6. Results

6.1. RQ1: Effectiveness

Which of the methods defined in Section 3 is the best change oracle for identifying what and how code modules should be changed?

Figure 6 shows the comparison results. Two data sets are very responsive to defect reduction suggestions: Ant and Ivy (both of which show best case improvements over 50%). The expected value of defects is changed less in Jedit. This data set's results are surprisingly uniform; i.e. all methods find the same ways to reduce the expected number of defects.

Figure 8 enables us to explain the uniformity of the results seen with Jedit in Figure 6. Observe how in Figure 8 the only change ever found is a reduction to *rfc*. Clearly, in this data set, there is very little that can be usefully changed.

Two data sets are not very responsive to defect reduction: Poi and Lucene. The reason for this can be see in Figure 5: both these data sets contain more than 50% defective modules. In that space, all our methods lack a large sample of defect-free examples.

Also consider the relative rank of the different approaches, CD and Shatnawi usually perform comparatively worse while XTREE gets top ranked position the most number of times. That said, Alves sometimes beats XTREE (see Ivy) while sometimes it ties (see Jedit).

Features	Ant				Ivy				Lucene				Jedit				Poi			
	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn
wmc	.	92	100	100	18	95	100	100	89	95	100	.	.	63	.	.	.	100	100	.
dit	.	77	100	.	.	87	100	.	.	80	100	.	.	72	100	100	.	46	100	.
noc	.	20	100	.	.	.	100	.	.	26	.	.	.	.	.	.	.	.	.	.
cbo	88	99	100	100	91	100	100	100	60	94	100	100	.	100	100	.	1	74	100	.
rfc	100	100	100	.	8	95	100	.	10	83	100	.	100	100	100	100	100	95	100	.
lcom	.	98	100	100	15	100	100	100	.	94	100	.	.	100	100	.	.	100	100	100
ca	.	93	100	.	7	95	100	.	40	89	100	.	.	63	100	100	.	74	100	.
ce	5	100	100	.	.	97	100	.	.	90	100	.	.	100	100	100	.	64	100	.
npm	.	88	100	.	8	97	100	.	.	93	100	100	.	100	100	.	.	100	100	.
lcom3	.	90	100	.	7	95	100	.	13	79	100	100	.	63	100	100	.	92	100	100
loc	100	99	100	100	97	97	100	100	60	100	100	100	.	100	.	100	100	100	100	100
dam	.	21	100	.	.	22	100	.	.	55	100	.	.	45	100	100	.	73	100	.
moa	.	67	100	.	.	82	100	.	.	60	100	100	.	54	100	100	.	58	100	.
mfa	5	93	100	.	.	90	100	.	5	80	100	.	.	72	100	.	.	72	100	.
cam	.	99	100	100	84	100	100	100	10	94	100	.	.	100	100	.	.	98	100	100
ic	.	52	100	100	.	70	.	.	.	68	100	.	.	36	100	.	.	43	100	100
cbm	.	59	100	.	.	85	.	.	.	71	100	.	.	36	100	100	.	67	100	.
amc	.	99	.	.	.	95	100	.	30	100	.	.	.	100	100	100	.	97	.	.
max cc	.	87	100	100	.	85	100	100	.	71	.	.	.	45	100	.	.	63	100	.
avg cc	12	99	100	100	.	95	100	100	13	98	.	.	.	100	100	100	.	92	100	.

Figure 7: Results for **RQ2**. Percentage counts of how often an approach recommends changing a code metric (in 40 runs). "100" means that this code metric was always recommended. Cells marked with "." indicate 0%. For the Shatnawi and Alves et al. columns, metrics score 0% if they always fail the $p \leq 0.05$ test of §4.1.2. For CD, cells are blank when two centroids have the same value for the same code metrics. For XTREE, cells are blanks when they do not appear in the delta between branches. Note that XTREE mentions specific code metrics far fewer times than other methods.

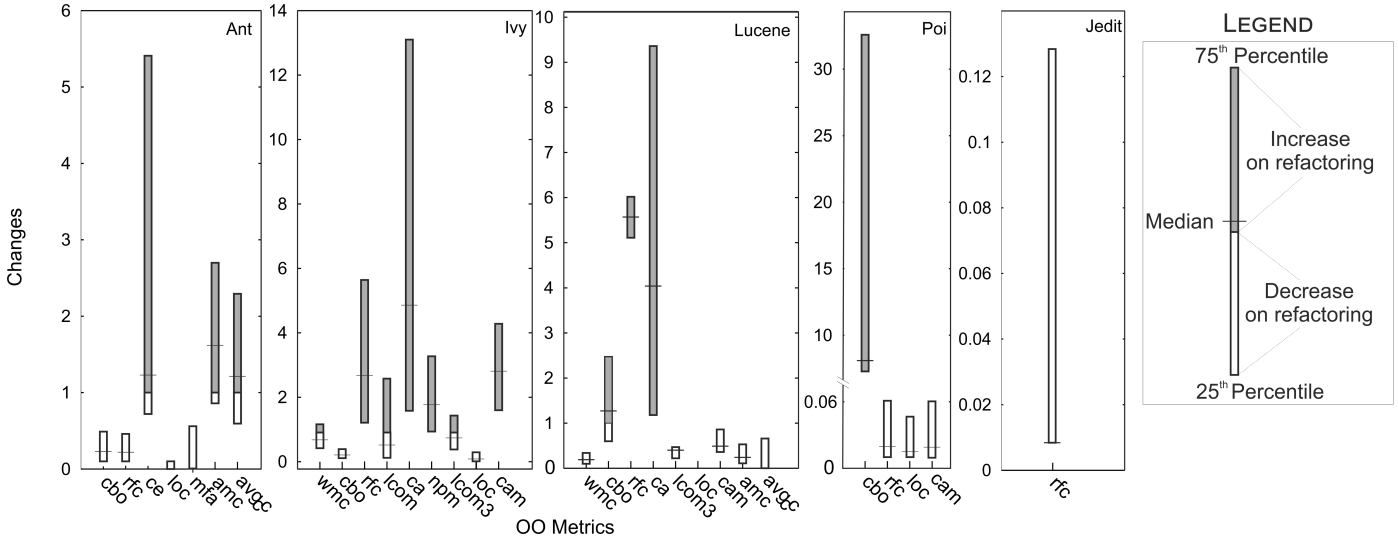

Figure 8: Results from XTREE. While Figure 7 are the *number* of times a code metric was changed, this figure shows the *magnitude* each code metric was changed. Each vertical bar marks the 27,50,75th percentile change seen in 40 repeats. All numbers are ratios of initial to final values. All bar regions marked in gray show *increases*. The interesting feature of these results are that many of the changes proposed by XTREE require *increases* (this puzzling observation is explained in the text).

In summary, our **Result1** is that, of the change oracles studies here, XTREE is the best oracle on how to change code modules in order to reduce defects in our data sets.

6.2. RQ2: Verbosity

Which of the Section 3 methods recommended changes to the fewest code attributes?

Figure 7 shows the frequency with which the methods recommend changes to specific code metrics. Note that XTREE proposes thresholds to few code metrics compared to the other approaches.

Hence, our **Result2** is that, of all the code change oracles studied here, XTREE recommended far fewer changes to static code measures. Note only that, combining Figure 6 with Figure 7, we see that even though XTREE proposes changes to far fewer code metrics, those few code metrics are usually just as effective (or more effective) than the multiple thresholds proposed by CD, Shatnawi or Alves. That is, XTREE proposes *fewer* and better thresholds than the other approaches.

6.3. RQ3: Stopping

How effective is XTREE at offering "stopping points" (i.e. clear guidance on what not to do)?

The **RQ2** results showed that XTREE's recommendations are small in a *relative sense*; i.e. they are relatively smaller than the other methods studied here. Note also that, XTREE's recommendations are small in an *absolute sense*. Consider the frequency of changes in Figure 7. There are very few values for XTREE that are over 33% (in our sense this translates to at least a third of our repeated runs where XTREE mentioned a code attribute). For Ant, Ivy, Lucene, Jedit, and Poi, those frequencies are 3, 3, 3, 4, 1, 2 respectively (out of twenty). This means that, usually, XTREE omits references to 17,17,17,16,19,18 static code attributes (out of 20). Any code reorganization based on a bad smell detector that uses these omitted code attributes could hence be stopped.

	wmc	dit	noc	cbo	rfc	lcom	ca	ce	npm	lcom3	loc	dam	moa	mfa	cam	ic	cbm	amc	max_cc	avg_cc
Ant				−	−			+			−			−				+		+
Ivy	−			−	+	−	+		−	−	−				+					
Poi				+	−						−				−					
Lucene	−			+	+		+			−	−				−			−		−
Jedit					−															

Figure 9: Direct of changes seen in a comparison of statistically significantly different static code attributes measures seen in the clusters found by XTREE. Each dataset contains 20 Static Code Metrics (for a description of each of these metrics, please refer to [43]). The rows contain the datasets, and the columns denote the metrics. A "+" symbol represents a recommendation that requires a significant statistical increase (with a p-value$\leq$0.05), and likewise, a "−" represents a significant statistical decrease.

Hence our **Result3** is that, in a any project, XTREE's recommended changes affect only one to four of the 20 static code attributes. Any bad smell defined in terms of the remaining 19 to 16 code attributes (i.e. most of them) would be deprecated.

6.4. RQ4: Stability

Across different projects, how variable are the changes recommended by our best change oracle?

Figure 7 counted how *often* XTREE's recommendations mentioned a static code attribute. Figure 8, on the other hand, shows the *direction* of XTREE's recommended change:

- Gray bars show an *increase* to a static code measure;
- White bars shows a *decrease* to a static code measure;
- Bars that are all white or all gray indicate that in our 40 repeated runs, XTREE recommended changing an attribute the same way, all the time.
- Bars that are mixtures of white and gray mean that, sometimes, XTREE makes different recommendations about how to change a static code attribute.

Based on Figure 8, we see **Result4** states that the direction of change recommended XTREE is very stable repeated runs of the program (evidence: the bars are mostly the same color).

Figure 8 also comments on the inter-relationships between static code attributes. Note that while some measures in Figure 8 are decreased, many are increased. For example, consider the *Poi* results from Figure 7 that recommends decreasing *loc* but making large increases to *cbo* (coupling between objects). Here, XTREE is advising us to break up large classes class by into services in other classes. Note that such a code reorganization will, by definition, increase the coupling between objects. Note also that such increases to reduce defects would never be proposed by the outlier methods of Shatnawi or Alves since their core assumption is that bad smells arise from unusually large code metrics.

6.5. RQ5: Conjunctive Fallacy

Is it always useful to apply Equation 1; i.e. make code better by reducing the values of multiple code attributes?

In Figure 8, the bars are colored both white and gray; i.e. XTREE recommends *decreasing* and *increasing* static attribute values. That is, always decreasing static code measures (as suggested by Equation 1) is *not* recommended by XTREE. This comments on the interconnectedness of metrics and it is important for a few reasons:

1. When developers follow suggestions offered based on static code metrics, they are often required to reduce several metrics at once. For example, in Figure 7 we notice all methods except XTREE recommend changes to *every* metric. This is not a practical approach and it renders these suggestions useless.
2. When XTREE recommends changes, it respects the interconnectedness of metrics. If we were to reduce LOC, other metrics to do with coupling (cbo, ca, ce, ...) would have to be changed as well. XTREE is aware of this and suggests changes to these metrics in conjunction with LOC. Additionally, it offers a direction of change (increase or decrease).

Now when programmers follow the recommendations of XTREE instead for reorganization, they do so fully aware of the impact that change can have on other metrics. For instance, a programmer who chooses to reduce LOC is now aware that a consequence would be increased coupling. She/he can continue to make this change because similar changes with lower LOC and higher coupling have historically been known to result in fewer defects.

To explore this point further, we conducted the following "what-if" study. Once XTREE builds its trees with its leaf clusters then:

- Gather all clusters C_0 for C_1 such that the percent of defects in C_0 is greater than C_1 ...
- For all attribute measures, identify ones that have a statistically significantly different distribution between each C_0 and C_1...
- Report the direction of change ($+$ indicates an *increase* in values and $-$ indicates a *decrease*)

Note that the changes found in this way are somewhat more general than the results presented in Figure 6. Those results were limited to comments on the test set given to the program given that the tree is constructed using the training set.

Performing a "what-if" analysis using the three steps presented above allow us to reflect on *all possible changes* for any dataset (not limited to only one instance from the test set). This is significant because it comments on some of the commonly

held notions regarding static code attributes. Figure 9 which shows the results of this "what if" analysis can better help understand this. Consider as an example the metric loc, as might be expected, the recommendation is to *always* reduce lines of code (loc). But for the other attributes like afferent and efferent coupling, contrary to popularly recommendations, XTREE in fact suggests to increase those values or leave them unchanged, but *not* decrease. Similarly, coupling between objects (cbo) needs to be decreased in Ant and Ivy but increased in Poi and Lucene. Finally, many metrics such as depth of inheritance tree (dit), are best left unchanged in all cases.

(Reviewer2b) This is quite significant for practitioners attempting to perform code reorganization to similar projects without a historical log. They can use these findings as guide to: (1) deprecate changes the show no demonstrable benefits; and (2) not limit themselves to reducing values all the metrics because an increase can sometimes be more beneficial.

Hence, **Result5** is that while XTREE always recommends reducing loc, it also often recommends increasing the values of other static code attributes.

7. Limitations and Future Work

XTREE offers to assist reorganization efforts in an intelligent fashion, it suffers from two key limitations: (1) They use supervised learning to learn about the domain, therefore they need a sufficiently large dataset so as to be "trained". Figure 5 shows the dataset used in this study all have at least two prior releases; (2) Ability to build a reliable secondary verification oracle. We used the current state of the art in build our verification oracle (Tuned and SMOTE-ed Random Forest).

In this work, both the primary change oracle and the secondary verification oracle used OO metrics in conjunction with the number of associated defects to operationalize code smells. We note this goal can very easily be replaced by other quantifiable objectives. For example, there is much research on *technical debts* in recent years. The notion of technical debt reflects the extra work arising from developers making pragmatic short-term decisions that make it harder, in the long-term, to maintain the software. A study by Ernst et al. [62] showed that practitioners agree on the importance of the issue but existing tools are not currently helpful in managing the debts. A systematic literature survey conducted by Li et al. [63] report that dedicated technical debt management (TDM) tools are needed for managing various types of technical debts. (Reviewer1b) Recently Alves et al. [64] conducted an extensive mapping study on the identification and management of technical debt. They identified several types of debts over the past decade and found that bad smells of the sort discussed in this paper are very well known indicators of design debt. In fact, they happen to be few of most frequently referenced kinds of debts. XTREE could be adapted for this purpose by supporting decisions if and when a technical debt item should be paid off – providing that there existed a working "secondary oracle" (of the kind we define in our introduction) that can recognize quick-and-dirty sections of code.

Our research shows that it is potentially naïve to explore thresholds in static code attributes in isolation to each other. This work clearly demonstrates how changing one necessitates changing other associated metrics. So, for future work, we recommend that researchers and practitioners look for tools that recommend changes to sets of code changes. When exploring candidate technologies, apart from XTREE, researchers may potentially use: (1) Association rule learning; (2) Thresholds generated across synthetic dimensions (eg. PCA); and (3) Techniques that cluster data and look for deltas between them. (Note: we offer XTREE as a possible example of this point).

Lastly, we also plan on extending our work by exploring scalable solutions to achieve similar results in much larger datasets. After this, we shall look at applications beyond that of measuring static code attributes. For example, as an initial attempt, we have been looking at sentiment analysis in stack overflow exchanges to learn dialog pattern that most select for relevant entries.

8. Reliability and Validity of Conclusions

The results of this paper are biased by our choice of code reorganization goal (reducing defects) and our choice of measures collected from software project (OO measures such as depth of inheritance, number of child classes, etc). That said, it should be possible extend the methods of this paper to other kinds of goals (e.g. maintainability, reliability, security, or the knowledge sharing measures) and other kinds of inputs (e.g. the process measures favored by Rahman, Devanbu et al. [65])

8.1. Reliability

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

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

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

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

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

8.2. Validity

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

Based on the case study presented above, as well as the discussion in §2.3, we believe that bad smell indicators (e.g. *loc*$>$ 100) have limited external validity beyond the projects from which they are derived. While specific models are externally valid, there may still be general methods like XTREE for finding the good local models.

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

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

- Either wait till data about those new terms has been collected.
- Or, apply cutting edge transfer learning methods [66, 67, 68] to map data from other projects into the current one.

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

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

9. Conclusions

How to discourage useless code reorganizations? We say:

> *Ignore those changes not supported by the historical log of data from the current project.*

When that data is not available (e.g. early in the project) then developers could use the general list of bad smells shown in Figure 1. However, our results show that bad smell detectors are most effective when they are based on a small handful of code metrics (as done by XTREE). Hence, using all the bad smells of Figure 1 may not be optimal.

For our better guess at how to reduce defects by changing code attributes, see Figure 9. But given the large variance if the change recommendations, we strongly advice teams to use XTREE on their data to find their own best local changes.

XTREE improves on prior methods for generating bad smells:

- As described in §2.3, bad smells generated by humans may not be applicable to the current project. On the other hand, XTREE can automatically learn specific thresholds for bad smells for the current project.
- Prior methods used an old quality predictor (QMOOD) which we replace with defect predictors learned via Random Forests from current project data.
- XTREE's conjunctions proved to be arguably as effective as those of Alves (see Figure 6) but far less verbose (see Figure 7). Since XTREE *approves* of fewer changes it hence *disapproves* of most changes. This makes it a better framework for critiquing and rejecting many of the code reorganizations.

Finally, XTREE does not suffer from the conjunctive fallacy. Older methods, such as those proposed by Shatnawi and Alves assumed that the best way to improve code is to remove outlier values. This may not work since when code is reorganized, *the functionality has to go somewhere*. Hence, reducing the lines of code in one module necessitates *increasing* the coupling that module to other parts of the code. In future, we recommend software team use bad smell detectors that know what attribute measures need *decreasing* as well as *increasing*.

Acknowledgements

The work is partially funded by NSF awards #1506586 and #1302169.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman, Boston, MA, USA, 1999.

[2] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Verlag, 2006.

[3] Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 15–15. IEEE, 2005.

[4] ISO/IEC 14764:2006: Software Engineering – Software Life Cycle Processes – Maintenance. Technical report, ISO/IEC, September 2006.

[5] Steve McConnell. *Code complete*. Pearson Education, 2nd edition, 2004.

[6] Jeff Atwood. Code smells. "https://blog.codinghorror.com/code-smells/", 2006.

[7] Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, 2013.

[8] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 682–691. IEEE Press, 2013.

[9] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 17–23. ACM, 2011.

[10] Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, 2010.

[11] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[12] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 1–8. ACM, 2011.

[13] Nico Zazworka, Carolyn Seaman, and Forrest Shull. Prioritizing design debt investment opportunities. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 39–42. ACM, 2011.

[14] Nico Zazworka, Rodrigo O Spínola, Antonio Vetro, Forrest Shull, and Carolyn Seaman. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 42–47. ACM, 2013.

[15] Rahul Krishna and Tim Menzies. Actionable= cluster+ contrast? In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 14–17. IEEE, 2015.

[16] L Breiman. Random forests. *Machine learning*, pages 5–32, 2001.

[17] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[18] Karin Erni and Claus Lewerentz. Applying design-metrics to object-oriented frameworks. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 64–74. IEEE, 1996.

[19] Ralf Bender. Quantitative risk assessment in epidemiological studies investigating threshold effects. *Biometrical Journal*, 41(3):305–319, 1999.

[20] R. Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering*, 36(2):216–225, March 2010.

[21] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *2010 IEEE Int. Conf. Softw. Maint.*, pages 1–10. IEEE, sep 2010.

[22] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2015.

[23] R Borges and T Menzies. Learning to Change Projects. In *Proceedings of PROMISE'12, Lund, Sweden*, 2012.

[24] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):17, 2015.

[25] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. Search-based refactoring: Towards semantics preservation. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 347–356. IEEE, 2012.

[26] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, 2013.

[27] Ali Ouni, Marouane Kessentini, Slim Bechikh, and Houari Sahraoui. Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal*, 23(2):323–361, 2015.

[28] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, pages 1–32, 2014.

[29] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 16–24, Oct 2015.

[30] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, pages 403–414. IEEE, May 2015.

[31] M.V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 399–408, Sept 2004.

[32] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 242–251, Oct 2013.

[33] D.I.K. Sjoberg, A. Yamashita, B.C.D. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156, Aug 2013.

[34] Carol Passos, Ana Paula Braun, Daniela S. Cruzes, and Manoel Mendonca. Analyzing the impact of beliefs in software project practices. In *ESEM'11*, 2011.

[35] Magne Jørgensen and Tanja M. Gruschke. The impact of lessons-learned sessions on effort estimation and uncertainty assessments. *Software Engineering, IEEE Transactions on*, 35(3):368 –383, May-June 2009.

[36] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 108–119. ACM, 2016.

[37] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesly Professional, 2005.

[38] A. Campbell. SonarQube: Open source quality management, 2015. Website: tiny.cc/2q4z9x.

[39] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1—-9:10. ACM, 2010.

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

[41] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120–1128, 2007.

[42] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–39, 2014.

[43] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 343–351. IEEE, nov 2011.

[44] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous valued attributes for classification learning. In *Thirteenth International Joint Conference on Articial Intelligence*, volume 2, pages 1022–1027. Morgan Kaufmann Publishers, 1993.

[45] Betty Cheng and Adam Jensen. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1341–1348, New York, NY, USA, 2010. ACM.

[46] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring: An empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, September 2008.

[47] Mark Kent O'Keeffe and Mel O. Cinneide. Getting the most from search-based refactoring. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1114–1120, New York, NY, USA, 2007. ACM.

[48] Iman Hemati Moghadam. *Search Based Software Engineering: Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*, chapter Multi-level Automated Refactoring Using Design Exploration, pages 70–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[49] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 331–336, New York, NY, USA, 2014. ACM.

[50] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, January 2002.

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

[52] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw Eng.*,

33(1):2–13, Jan 2007. Available from http://menzies.us/pdf/06learnPredict.pdf.

[53] L. Pelayo and S. Dick. Applying novel resampling strategies to software defect prediction. In *Fuzzy Information Processing Society, 2007. NAFIPS '07. Annual Meeting of the North American*, pages 69–72, June 2007.

[54] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: is it really necessary? *Submitted to Information and Software Technology*, 2016.

[55] Rainer Storn and Kenneth Price. Differential Evolution — A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[56] Nikolaos Mittas and Lefteris Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Trans. Software Eng.*, 39(4):537–551, 2013.

[57] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. Mono. Stat. Appl. Probab. Chapman and Hall, London, 1993.

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

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

[60] Ekrem Kocaguneli, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, and Tim Menzies. Distributed development considered harmful? In *Proceedings - International Conference on Software Engineering*, pages 882–890, 2013.

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

[62] Neil A Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L Nord, and Ian Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 50–60. ACM, 2015.

[63] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[64] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de Mendona, Rodrigo O. Spnola, Forrest Shull, and Carolyn Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100 – 121, 2016.

[65] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings - International Conference on Software Engineering*, pages 432–441, 2013.

[66] Jaechang Nam and Sunghun Kim. Heterogeneous defect prediction. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 508–519, New York, NY, USA, 2015. ACM.

[67] Xiaoyuan Jing, Fei Wu, Xiwei Dong, Fumin Qi, and Baowen Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *Proc. of the 10th Joint Meeting on Foundations of Software Engineering*, pages 496–507, New York, NY, USA, 2015. ACM.

[68] Rahul Krishna, Tim Menzies, and Wei Fu. Too Much Automation? The Bellwether Effect and Its Implications for Transfer Learning. In *ASE'16*, 2016.

RESPONSE TO REVIEWERS

Comments from the editor

The reviewers recognized the improvements at the same time they raised some minor concerns regarding the content of the paper. Please, prepare an evolved version for a next verification. Use the reviewers' suggestions to make all changes needed.…

Thanks again for your comments. We have addressed all the concerns raised by the reviewers.
Reviewer 1 suggested an additional reference that you have the freedom to consider it or not, depending on your interest on it. All other suggestions deserve your attention.

We have taken into account the paper suggested by the reviewer. We found it very relevant to our work and have added it to our review in §7.

Response to Reviewer #1

Initially, Id like to thank the authors for having appropriately addressed the comments from the first review. Now I have just minor comments on the paper:

Thank you for your comments. We have taken your remarks into account and have made appropriate changes to the paper.

I missed a last paragraph on the introduction presenting the structure of the paper (for example, "Besides this introduction, Section 2 presents.… Next, Section 3 discusses").

Thank you highlighting that. We have now added a paragraph toward the end of §1 presenting the structure of the paper. Kindly see content marked Reviewer1a.

On Section 7, when you relate your approach with technical debt, Id like to suggest you to also consider the mapping study cited below …

Thank you for recommending this paper. We have used this citation to expand our §7 accordingly. Please see content marked Reviewer1b in §7.

Response to Reviewer #2

Im happy with the great job the authors have done with the responses.

Thank you very much for your comments. We have now addressed the comments in this revised version.

I believe you missed my point, "Your example is the most favorable example from your point of view. It is one of only two of the eight data sets that improved on both pd and pf."

Thank you for that clarification. We see the issue. We have now added text to section §5.2.1 providing example of failed cases in addition to positive sample. Kindly see content marked Reviewer 2a in §5.2.1.

I still find the last two sentences in the abstract odd: "This recommendation assumes that there is a historical record. If none exists, then the results of this paper could be used as a guide."

Thank you pointing that out. *We have now removed these lines from the abstract* and have added it to relevant section in the body of paper. Briefly, we intended to convey the following with figure 9:

- Some metrics (like loc) are best minimized as much as practically possible;
- Developer need not hesitate leaving some metrics (like certain types of coupling) untouched or even maximize them when that's an option.

Please see content marked Reviewer2b in §6.5.