On Recommending Actionable Changes For Software Projects

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

ABSTRACT

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

1 INTRODUCTION

Over the past decade, advances in AI have enabled a widespread use of data analytics in software engineering. For example, we can now estimate how long it would take to integrate the new code [6], where bugs are most likely to occur [20], or amount of effort it will take to develop a software package [33], etc. Despite these successes, there is a primary operational shortcoming with many software analytic tools lack of insightful analytics.

Business users also lament that most software analytics tools, "Tell us what *is*. But they don't tell us *what to do*". A concern that was also raised by several researchers at a recent workshop on "Actionable Analytics" at 2015 IEEE conference on Automated Software Engineering [13]. For example, most software analytics tools in the area of detecting software defects are mostly *prediction* algorithms such as Support Vector Machines, Naive Bayes, Logistic Regression, Decision Trees, etc [29]. These prediction algorithms report what combinations of software project features predict for the number of defects. But this is different task to *planning*, which answers a more pressing question: what to *change* in order to *reduce* these defects. Accordingly, in this research, we seek tools that offer clear guidance on what to do in a specific project.

The tool assessed in this paper is the XTREE *planning* tool [17]. XTREE employs a $cluster + contrast$ approach to planning where it (a) *Clusters* different parts of the software project based on a quality measure (e.g. the number of defects); (b) Reports the *contrast sets* between neighboring clusters. Each of these contrast sets represent the difference between these clusters and they can be interpreted as plans:

- If a current project falls into cluster C_1,
- Some neighboring cluster C_2 has better quality.
- Then the difference $\Delta = C_2 - C_1$ is a *plan* for changing a project such that it *might* have higher quality.

XTREE uses data from within a software project to these generate plans. Generation of conclusions are an inherent property of XTREE. But, the key question with the first version of XTREE is that some of the conclusions were not really actionable. So, in this work, we explore ways in which XTREE can limit it's conclusions to only actionable ones.

With this in mind, the rest of this paper is designed as follows. The remainder of this section discusses the research questions asked and answered in this paper. In § 2, we discuss the related work in the area of planning in software engineering. In § 3, we discuss the planning algorithms of this paper. The experimental setup is presented in § 4. The answers to the research questions is presented in § 5. We discuss the threats to validity of our experiments is discussed in § 6. Finally, in § 7, we present conclusions and directions to future work.

1.1 Research Questions

RQ1: How many valid changes does baseline XTREE recommend?
Motivation: The first research question seeks to establish a baseline result. Here, we ask how many changes recommended by a simple version of XTREE (labeled $XTREE_{simple}$) are actually implemented by developers.
Approach: We use a set of 3 datasets – train, test, validate. We construct a planner (XTREE) on the train set, and plan to reduce defects in the test set. Then, we look at the validation set, and reflect on the recommended changes. We expect to see a number of changes proposed by XTREE to overlap with those undertaken by developers. This will also offer us a baseline, which can then be improved upon to increase the extent of overlap.
Results: We found that with the existing baseline version of XTREE, around 50% of the changes that were recommended by XTREE were also undertaken by the developers.
RQ2: How many defects baseline XTREE assist in reducing?
Motivation: In this research question we seek to establish the effectiveness of XTREE. That is, for every defective class in the test dataset, we look to see if implementing changes recommended by XTREE leads to the reduction of defects.
Approach: In order to establish this, we look at the defective classes in the test dataset, and the same class in the validation dataset. If the defects have been fixed, we look to see if the recommendations

made by XTREE were heeded during that fix. If so, this will point to the uselessness of XTREE's plans.
Results: We found that XTREE is quite effective in reducing defects. In more than 50% of the cases, the number of defects reduced after using the plans proposed by XTREE.
RQ3: How to extend XTREE to recommend more valid changes?
Motivation: In the first research question, we showed that in 50% of the cases, the plans offered by XTREE were also implemented by developers. In this research question, we ask if there is a way to increase the overlap between the number of plans proposed by XTREE and those undertaken by developers.
Approach: To answer this question, we developed two variants of XTREE — $XTREE_{pruned}$ and $XTREE_{all}$. Both the variants use past historical data to modify the plans generated by XTREE in order to ensure that the plans generated are closely related to the changes usually made by developers. The details of their implementation are discussed elsewhere in the paper.
Results: Our results show that the modified version of XTREE can generate overlaps of upto 80% with what the developers usually do. This a significant improvement to 50% with the traditional XTREE.
RQ4: How many defects does extended XTREE assist in reducing?
Motivation: Having demonstrate that $XTREE_{pruned}$ and $XTREE_{all}$ can generate samples that are very close to those usually made by developers. This research question asks how effective these versions of XTREE are in reducing defects.
Approach: As for the approach, we take similar steps to RQ2. We look at the defective classes in the test dataset, and the same class in the validation dataset. If the defects have been fixed, we look to see if the recommendations made by XTREE were heeded during that fix. If so, this will point to the uselessness of XTREE's plans.
Results: Our results indicate that the modified version of XTREE performs just as well as the traditional XTREE in all the datasets we have studied here.

2 RELATED WORK

Planning has been a subject of much research in artificial intelligence. Here, planning usually refers to generating a sequence of actions that enables an *agent* to achieve a specific *goal* [30]. This can be achieved by classical search-based problem solving approaches or logical planning agents. Such planning tasks now play a significant role in a variety of demanding applications, ranging from controlling space vehicles and robots to playing the game of bridge [10]. Some of the most common planning paradigms include: (a) classical planning [35]; (b) probabilistic planning [1]; and (c) preference-based planning [3].

Existence of a model precludes the use of each of these planning approaches. This is a limitation of all these planning approaches since not every domain has a reliable model. In software engineering, the planning problem translates to proposing changes to software artifacts. Solving this has been undertaken via the use of some search-based software engineering techniques [11]. Examples of algorithms include SWAY, NSGA-II, etc. [7, 24].

These search-based software engineering techniques require access to some trustworthy models that can be used to explore novel solutions. In some software engineering domains there is ready access to such models which can offer assessment of newly generated plans. Examples of such domains within software engineering include automated program repair [18, 34], software product line management [12, 31], etc.

However, not all domains come with ready-to-use models. For example, consider software defect prediction and all the intricate issues that may lead to defects in a product. A model that includes *all* those potential issues would be very large and complex. Further, the empirical data required to validate any/all parts of that model can be hard to find. Also, even when there is an existing model, they can require constant maintenance lest they become out-dated. In such domains, we seek alternate methods for planning that can be automatically updated with new data without a need for comprehensive models. For this, we propose the use of data mining approaches to create a quasi-model of the domain and make of use observable states from this data to generate an estimation of the model. Our preferred tools in this paper (XTREE and BELLTREE) take this approach by constructing decision trees on available data (discussed in § **??**). In § 5, we show that these methodologies have encouraging results.

In summary, for domains with readily accessible models, we recommend the tools used by the search-based software engineering community. For domains, where domain-models are not available, we recommend tools such as ours.

3 PLANNING IN SOFTWARE ANALYTICS

In the previous sections, we introduced the notion of planning for decision making. In this section we offer a description of each of these planning methods:

3.1 $XTREE_{simple}$

XTREE builds a decision tree, then generates plans by contrasting the differences between two branches: (1) the branch where you are; (2) the branch to where you want to be.

XTREE takes a *supervised Cluster + Contrast* approach to planning because we hypothesize that it is useful to reflect on the target class. Thus, XTREE uses a supervised decision tree algorithm of Fig. 1.A. To divide continuous numeric data, we use a Fayyad-Irani discretizer of Fig. 1.B. Next, XTREE builds plans from the branches of the decision trees using the description of Fig. 1.C. In doing so, we ask three questions, the last of which returns the plan:

(1) Which *current* branch does a test case fall in?
(2) Which *desired* branch would the test case want to move to?
(3) What are the *deltas* between current and desired branches?

These *deltas* represent the threshold ranges[1] that represent the plans to reduce the defects.

3.2 $XTREE_{pruned}$

$XTREE_{pruned}$ is structurally similar to $XTREE_{simple}$. It differs in the steps taken after the construction of the planner. While XTREE uses data from the training set to construct a planner, $XTREE_{pruned}$ augments this by pruning the branches of the tree by removing the leaves that represent changes that have never been seen before.

[1]Thresholds are denoted by $[low, high)$ ranges for each metric

Fig. 1.A: Top-down division with Decision Trees

(1) Find a split in the values of independent features (OO metrics) that most reduces the variability of the dependent feature (defect counts). For continuous and discrete values, the *variability* can be measured using standard deviation σ or entropy e respectively. Construct a standard decision tree using these splits.

(2) Discretize all numeric features using the Fayyad-Iranni discretizer [9] (divide numeric columns into bins B_i, each of which select for the fewest cluster ids). Let feature F have bins B_i, each of which contains n_i rows and let each bin B_i have entropy e_i computed from the frequency of clusters seen in that bin. Cull the the features as per [28]; i.e. just use the $\beta = 33\%$ most informative features where the value of feature F is $\sum_i e_i \frac{n_i}{N}$ (N is the number of rows).

Fig. 1.B: A sample XTREE tree.

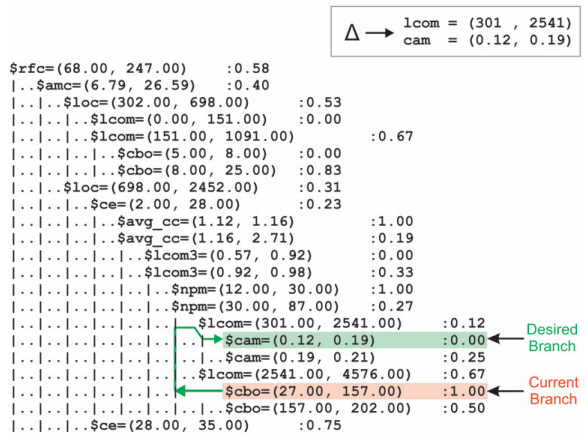

Fig. 1.C: Using XTREE

Using the training data, divide the data using the decision tree algorithm of Fig. 1.A into groups of size $\alpha = \sqrt{N}$. For test item, find the *current* leaf: take each test instance, run it down to a leaf in the decision tree. After that, find the *desired* leaf:

- Starting at *current*, ascend the tree $lvl \in \{0, 1, 2\ldots\}$ levels;
- Identify *sibling* leaves; i.e. leaf clusters that can be reached from level lvl that are not same as *current*
- Using the *score* defined above, find the *better* siblings; i.e. those with a *score* less than $\gamma = 0.5$ times the mean score of *current*. If none found, then repeat for $lvl += 1$. Also, return no plan if the new lvl is above the root.
- Return the *closest* better sibling where distance is measured between the mean centroids of that sibling and *current*

Also, find the *delta*; i.e. the set difference between conditions in the decision tree branch to *desired* and *current*. To find that delta: (1) for discrete attributes, delta is the value from *desired*; (2) for numerics, delta is the numeric difference; (3) for numerics discretized into ranges, delta is a random number selected from the low and high boundaries of the that range.

Finally, return the delta as the plan for improving the test instance.

Figure 1: Generating thresholds using XTREE.

To achieve this, we employ first use the past data to keep track of changes that were done by developers. Next, we construct XTREE using the same strategy as in Fig. 1. Then, we prune the branches of the tree that represent changes that have never been seen before. This pruned tree may then be used to generate plans.

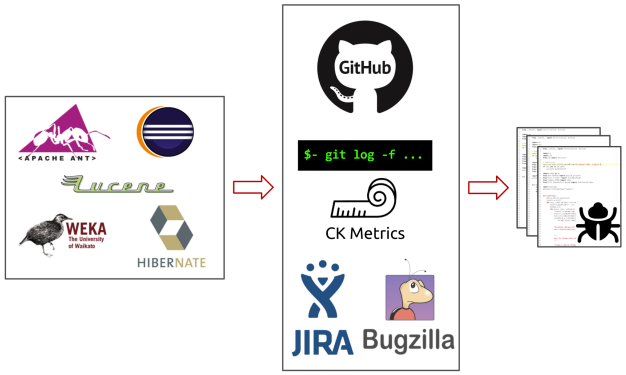

Figure 2: Framework for data generation

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

Figure 3: The figure lists the defect datasets gathered for this project.

This is a very simple approach. We posit that this will enable us to generate plans that not only reduce defects but also increase the likelihood that these plans are similar to what developers usually recommend.

3.3 $XTREE_{all}$

$XTREE_{all}$ is also structurally similar to $XTREE_{simple}$. It add a few additional steps after the construction of the decision tree. We first use data from the training set to construct a planner, then for every defective instance in the test case, we obtain all possible changes from $XTREE_{simple}$.

Next, we use the past data to keep track of changes that were done by developers. Then, we rank the possible changes based on what has previously been seen by developers.

This is also a very simple approach. This ranking process enables us to ensure that the generate plans are most likely to reduce defects and also increase the likelihood that these plans are similar to what developers usually recommend.

4 EXPERIMENTAL SETUP

4.1 Datasets

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

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

The procedure used to mine these datasets is illustrated in Fig. 2. All the datasets are described in terms of the attributes of Fig. 4. In order to gather these datasets, we use the following steps:

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

The above process, when repeated for all the projects, generates the required datasets. In Fig. 3, we summarize all the datasets that were mined for experimentation. All these datasets records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of 20 OO metrics, including CK metrics and McCabes complexity metrics.

4.2 Evaluation Strategy

Our experimental design is shown in Fig. 5. For a project $\mathbb{P}$ with versions $\mathcal{V} = \{1, ..., N\}$, we divide the project data into three parts: we use $\mathcal{V} - 2$ as the *train set*, $\mathcal{V} - 1$ as the *test test*, and $\mathcal{V}$ as the *validation set* in the following fashion:

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

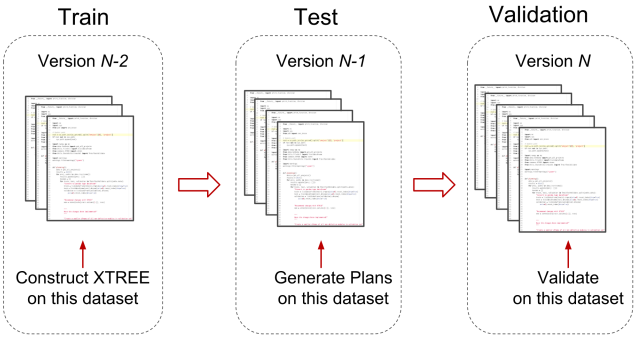

Figure 5: Experimental design

(1) The train test is used to construct the planner. In § **??** we describe the planners and a way to translate these plans into actionable decisions.
(2) Next, the test set used to generate for. That is, for every defective module in the test set, we obtain plans from the train set.
(3) The validation set is used to assess the effect of acting on these plans. This is necessary because it lets us judge the effects of applying the plan. If the defect has been fixed in the validation set as a result of complying with plans from XTREE.

The above steps are repeated for all the N versions of every project.

In software analysis, there is the issue of reliable verification oracles to test effects of planning. To resolve this problem, SE researchers such as Cheng et al. [5], O'Keefe et al. [26, 27], Moghadam [23] and Mkaouer et al. [22] use a *verification oracle* that is learned separately from the primary oracle. The verification oracles assesses how defective the code is before and after some code changes. For their verification oracle, Cheng, O'Keefe, Moghadam and Mkaouer et al. use the QMOOD hierarchical quality model [4]. A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [19].

Hence, for this study, we make use of the validation set in place of verification oracle. These validation sets represent the ground truth and are therefore a true representation of the impact of acting on these plans.

4.3 Performance Measures

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

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

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

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

4.3.2 Overlap. In order to measure the number of changes common to what the developers implemented and what XTREE recommends, we use a measure called overlap. Overlap is given as follows:

$$O = \left(\frac{XTREE_{recommended}}{Changed} + \frac{XTREE_{not-recommended}}{Unchanged} \right) \times 100 \quad (2)$$

Where, $XTREE_{recommended}$ is the changes recommended by XTREE, $Changed$ are the changes implemented by the developers. Similarly, $XTREE_{not-recommended}$ are the changes not-recommended by developers and $Unchanged$ are the change not-performed by the developer. The higher the value of overlap between XTREE's recommendations and the actual changes performed by the developers, the larger the value of O

4.4 Statistics

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

At each position of the moving window, we collect the values of effectiveness R (Eq. 1) and overlap O (Eq. 2). We refrain from performing a cross validation because the process tends to mix the samples from training data (the source) and the test data (other target projects), which defeats the purpose of this study.

To rank the numbers collected above, we use the Scott-Knott test recommended by Mittas and Angelis [21]. Scott-Knott is a top-down clustering approach used to rank different treatments. If that clustering finds an statistically significant splits in data, then some statistical test is applied to the two divisions to check if they are statistically significant different. If so, Scott-Knott recurses into both halves.

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

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

We then apply a statistical hypothesis test H to check if m, n are significantly different. In our case, the conjunction of bootstrapping and A12 test. Both the techniques are non-parametric in nature, i.e., they do not make gaussian assumption about the data. As for hypothesis test, we use a non-parametric bootstrapping test as endorsed by Efron & Tibshirani [8, p220-223]. Even with statistical significance, it is possible that the difference can be so small as to be of no practical value. This is known as a "small effect". To ensure that the statistical significance is not due to "small effect" we use effect-size tests in conjunction with hypothesis tests. A popular effect size test used in SE literature is the A12 test. It has been endorsed by several SE researchers [2, 14, 15, 32**? ?**]. It was first proposed by Vargha and Delany [**?**]. In our context, given the performance measure G, the A12 statistics measures the probability that one treatment yields higher G values than another. If the two algorithms are equivalent, then A12 = 0.5. Likewise if $A12 \geq 0.6$, then 60% of the times, values of one treatment are significantly greater that the other. In such a case, it can be claimed that there is *significant effect* to justify the hypothesis test H. In our case, we divide the data if *both* bootstrap sampling and effect size test agree that a division is statistically significant (with a confidence of 99%) and not a small effect ($A12 \geq 0.6$). Then, we recurse on each of these splits to rank G-scores from best to worst.

5 RESULTS

RQ1: How many valid changes does baseline XTREE recommend?

The purpose of this research question was to establish a baseline result. Here we use XTREE to recommend plans for addressing classes with defects. Then we compare those recommended plans with the changes undertaken by developers to fix those defects. Finally, we compare the overlap between the recommendations offered by XTREE with those undertaken by developers using Eq. 2.

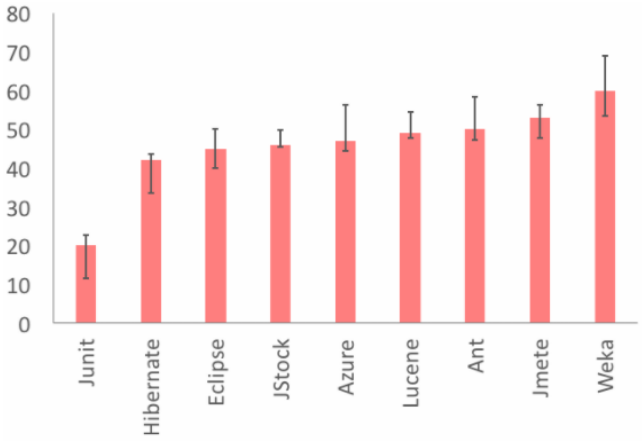

Figure 6: Overlap between changes implemented by developers and those recommended by XTREE.

Our results are shown in Fig. 6. In the figure, we see that plans generated $XTREE_{simple}$ overlap with the developer's fixes by about 50%. That is in 50% of the cases, the recommendations of XTREE is the same as those done by developers. The results are quite encouraging because they offer a number of possibilities for improvement.

RQ2: How many defects baseline XTREE assist in reducing?

RQ3: How to extend XTREE to recommend more valid changes?

RQ4: How many defects does extended XTREE assist in reducing?

6 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. [**?**])

6.1 Reliability

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

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

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

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

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

6.2 Validity

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

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

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

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

- Either wait till data about those new terms has been collected.
- Or, apply cutting edge transfer learning methods [16, 25**?**] 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.

7 CONCLUSIONS AND FUTURE WORK

REFERENCES

[1] Eitan Altman. 1999. *Constrained Markov decision processes.* Vol. 7. CRC Press.

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

[3] Shirin Sohrabi Jorge A Baier and Sheila A McIlraith. 2009. HTN planning with preferences. In *21st Int. Joint Conf. on Artificial Intelligence*. 1790–1797.

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

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

[6] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. 2011. CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice – Experiences from Windows. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. 357 –366.

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T Meyarivan. 2002. A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6 (2002), 182–197.

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

[9] Usama Fayyad and Keki Irani. 1993. Multi-interval discretization of continuous-valued attributes for classification learning. *NASA JPL Archives* (1993).

[10] Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated Planning: theory and practice*. Elsevier.

[11] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2009. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Dept. Comp. Sci, King's College London, Tech. Rep. TR-09-03* (2009).

[12] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon. 2015. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 517–528. https://doi.org/10.1109/ICSE.2015.69

[13] J. Hihn and T. Menzies. 2015. Data Mining Methods and Cost Estimation Models: Why is it So Hard to Infuse New Ideas?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 5–9. https://doi.org/10.1109/ASEW.2015.27

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

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

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

[17] Rahul Krishna, Tim Menzies, and Lucas Layman. 2017. Less is more: Minimizing code reorganization using XTREE. *Information and Software Technology* (mar 2017). https://doi.org/10.1016/j.infsof.2017.03.012 arXiv:1609.03614

[18] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (dec 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

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

[20] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. 2007. Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'". *IEEE Transactions on Software Engineering* 33, 9 (2007). https://doi.org/10.1109/TSE.2007.70721

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

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

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

[24] Vivek Nair, Tim Menzies, and Jianfeng Chen. 2016. An (accidental) exploration of alternatives to evolutionary algorithms for sbse. In *International Symposium on Search Based Software Engineering*. Springer, 96–111.

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

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

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

[28] Vasil Papakroni. 2013. *Data Carving: Identifying and Removing Irrelevancies in the Data*. Master's thesis. Lane Department of Computer Science and Electrical Engineering, West Virginia Unviersity.

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[30] Stuart Russell and Peter Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Egnlewood Cliffs.

[31] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable product line configuration: A straw to break the camel's back. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE.

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

[33] Burak Turhan, Ayşe Tosun, and Ayşe Bener. 2011. Empirical evaluation of mixed-project defect prediction models. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, 396–403.

[34] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings - International Conference on Software Engineering*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[35] Michael Wooldridge and Nicholas R Jennings. 1995. Intelligent agents: Theory and practice. *The knowledge engineering review* 10, 2 (1995), 115–152.