Defect Reduction Planning (using TimeLIME)

Kewen Peng, Tim Menzies, Fellow, IEEE

Abstract—Software comes in releases. An implausible change to software is something that has never been changed in prior releases. When planning how to reduce defects, it is better to use plausible changes, i.e., changes with some precedence in the prior releases. To demonstrate these points, this paper compares several defect reduction planning tools. LIME is a local sensitivity analysis tool that can report the fewest changes needed to alter the classification of some code module (e.g., from "defective" to "non-defective"). TimeLIME is a new tool, introduced in this paper, that improves LIME by restricting its plans to just those attributes which change the most within a project.

In this study, we compared the performance of LIME and TimeLIME and several other defect reduction planning algorithms. The generated plans were assessed via (a) the similarity scores between the proposed code changes and the real code changes made by developers; and (b) the improvement scores seen within projects that followed the plans. For nine project trails, we found that TimeLIME outperformed all other algorithms (in 8 out of 9 trials). Hence, we strongly recommend using past releases as a source of knowledge for computing fixes for new releases (using TimeLIME).

Apart from these specific results about planning defect reductions and TimeLIME, the more general point of this paper is that our community should be more careful about using off-the-shelf AI tools, without first applying SE knowledge. In this case study, it was not difficult to augment a standard AI algorithm with SE knowledge (that past releases are a good source of knowledge for planning defect reductions). As shown here, once that SE knowledge is applied, this can result in dramatically better systems.

Index Terms—Software analytics, Defect Prediction, Defect Reduction, Plausibility Analysis, Interpretable AI

1 INTRODUCTION

“Don’t tell me where I am, tell me where to go.”
– a (very busy) developer

Machine learners generate models. People read models. What kind of learners generate the kind of models that people want to read? If the reader is a busy software developer, then they might not need, or be able to use, complex models. Rather, such a busy developer might instead just want to know the least they need to do to achieve the most benefits. Machine learning for busy developers should not strive for elaborate models or increasing the expressive power of the language of the models. Rather, a better goal might be to find the smallest model with the most impact.

For example, suppose some AI model has classified a module as “defective”. If a developer then asks “what can I do to fix that?” then, ideally, we should, be able to reflect on the model to learn a defect reduction plan; i.e., a small set of actions that reduces the odds of that module being defective. But for many machine learning algorithms, it can be (very) difficult to learn a succinct reduction plan by reflecting on the arcane internal structure of, say, a neural network classifier.

To better support busy developers, this paper proposes a new machine learning algorithm called TimeLIME that generates defect reduction plans by reflecting over black box AI defect prediction models. Internally, TimeLIME uses a widely-cited sensitivity analysis tool called LIME [1] (first presented at KDD’16). LIME finds changes that alter a classification (e.g., from “defective” to “non-defective”) by exploring neighborhoods of similar examples. But classic LIME has a problem - it generates surprising and unprecedented plans that had never been seen before in the history of the project.

When we first observed this, our initial response was quite favorable. Perhaps, we thought, LIME would be offering novel and powerful suggestions that would lead to greater defect reductions than ever seen before. However, as shown in this paper, classic LIME’s plans are sub-optimal.

TimeLIME is an experiment with using SE knowledge to improve AI tools. To standard LIME, this new algorithm adds the following knowledge:

- Software comes in releases.
- An implausible change to software is something that has never been changed in prior releases.
- It is better to use plausible changes, i.e., changes with some precedence in the prior releases.

When TimeLIME generates plans, it restricts those plans to using just the attributes which have changed the most across the history a project. To test if TimeLIME is better than some other planner (e.g. LIME), we ran simulations over the historical record of eight software projects. Given project information divided into oldest, newer, and most recent data, we:
1) Used the oldest data to determine what attributes were often changed in a project,
2) Use the newer data to build plans using LIME, TimeLIME, and five other planning algorithms;
3) Divided the most recent data into:
   - Those projects that followed the plans;
   - And those that did not.

As shown in this paper, we show that the projects that followed TimeLIME’s plans had the fewest defects.

This paper is structured around three research questions:

- RQ1: Does TimeLIME provide succinct plans? Classic LIME, proposes changes to dozens of attributes. TimeLIME, on the other hand, restricts itself to just the most changed attributes. Hence, its proposed changes are far smaller than those found by other planners.

- K. Peng and T. Menzies are with the Department of Computer Science, North Carolina State University, Raleigh, USA.
E-mail: kpeng@ncsu.edu, timm@ieee.org
• RQ2: Could developers apply the changes proposed by TimeLIME? To answer this question, we look at project activity in the period after TimeLIME makes its recommendations. We find a very large overlap (median=80%) between TimeLIME’s recommendations and the actual actions taken by developers.

• RQ3: Is TimeLIME better at defect reduction? As shown below, the changes proposed by TimeLIME are associated with a much larger reduction in defects than classic LIME and other benchmark algorithms.

Based on the above, we will conclude that:

• When generating defect reduction plans, it is most beneficial to focus on on plausible changes; i.e., the changes with some precedence in the prior releases.

• The SE community should be more careful about using off-the-shelf AI tools, without first tuning them with SE knowledge. As shown here, it may not be a complex matter to apply that knowledge. Further, once that SE knowledge is applied, this can result in dramatically better systems.

The rest of this paper is structured as follows. §2 discusses defect prediction, code refactoring, and challenges of using human opinions in SE. §3 introduces some prior works in the field of defect reduction and their methodologies. §4 presents the basic framework of LIME as well as TimeLIME, the new method proposed in this paper. §5 shows our method for ranking different planning methods. §6 describes experiment and the datasets, predictive model, and planners evaluated in this work. §7 and §8 report and discuss the results respectively. The credibility and reliability of our conclusions is discussed by §9. Recent related works are shown in §10, which also declares the major difference that distinguishes the contribution of this paper. Future work and directions are illustrated in §11. Finally, we conclude this work in §12.

1.1 Reproduction Package

In order to support reproducibility and open science in software engineering, we have made available on-line at http://github.com/ai-se/TimeLIME all the scripts and data needed to repeat the analysis of this paper.

2 Background

2.1 Challenges with Using Human Opinions

This paper is an algorithmic analysis of historical SE data where we ran simulations over the historical record of eight software projects. An alternate approach to this algorithmic analysis of historical SE data is to use qualitative methods. Qualitative methods rely on surveys of human subject matter experts (e.g., programmers). Much has been learned from such studies of subject matter experts [2]. Nevertheless, in the particular case of large scale defect prediction, we prefer our algorithmic approach, for two reasons:

• Scalability: It is hard to scale qualitative investigations of human beliefs to a large number of projects. We mention this since while this paper studies just eight projects, our long-term goal is to develop software analysis methods that applies to hundreds to thousands of projects. While some progress has been seen recently with scaling qualitative methods [3], at the time of this writing, we assert that it is far easier to scale an algorithmic analysis of historical SE data.

• Lack of consensus: multiple studies report that human beliefs in software quality may often be inconsistent and even incorrect. Devanbu et al. have conducted a case study among 564 Microsoft software developers to show that human beliefs on software quality can be quite varied and may not be necessarily correspond with actual evidence within current projects [4], [5]. Similarly assertions are also made in Passos’ paper, where the author reports that conflicting beliefs can be held by different stakeholders of the software development team. There also exist cases that a belief is correct for past projects but not the current work [6]. A more recent study by Shrikanth et al. also reports such much variability of human beliefs about defect prediction [7]. Shrikanth studies 10 beliefs held by software developers about defect prediction, which were initially summarized by Wan et al in 2018 [8]. By measuring the actual support of these beliefs within the project, Shrikanth found that:
  • Among over 300,000 changes seen in different open-source projects, only 24% of the projects support all 10 beliefs.
  • What is believed the most by developers does not necessarily have the strongest support within projects. For example, a belief that is acknowledged by 35% of the developers has the most support whereas a belief held by 76% of the developers is only ranked 7th out of 10 beliefs.
  • As a project grows to mature, the beliefs actually tend to be weakened rather than strengthened.

Not only do practitioners have conflicting beliefs about what causes defects, but we also can see that researchers who have studied many projects also disagree on what factors matter the most to defect reduction. For example, as discussed later in the paper, Alves [9], Shatnawi [10], and Oliveira [11] all offer different models about what matters most for software quality.

In summary, many studies report a significant disconnect between human beliefs and patterns supported by data. Hence, we are nervous about using the opinion of experts’ opinions as the “ground truth” to evaluate (e.g., ) defect reduction plans. Accordingly, we use an algorithm analysis since that can use historical SE data to generate the ground truth needed to evaluate a method.

2.2 Defect Prediction

The case study of this paper comes from defect prediction and planning. This section discussed the value of that kind of analysis.

During software development, testing often has some resource limitations. For example, the effort associated with coordinated human effort across a large code base can grow exponentially with the scale of the project [12]. Hence, to effectively manage resources, it is common to match the quality assurance (QA) effort to the perceived criticality and bugginess of the code. Since every decision is associated with a human and resource cost to the developer team, it is impractical and inefficient to distribute equal effort to every component in a software system [13]. Learning defect
prediction (using data miners) from static code attributes (like those shown in Table 1) is one very cheap way to “peek” at the code and decide where to spend more QA effort.

Recent results show that software defect predictors are also competitive widely-used automatic methods. Rahman et al. [14] compared (a) static code analysis tools FindBugs, Jlint, and PMD with (b) defect predictors (which they called “statistical defect prediction”) built using logistic regression. No significant differences in cost-effectiveness were observed. Given this equivalence, it is significant to note that defect prediction can be quickly adapted to new languages by building lightweight parsers to extract code metrics. The same is not true for static code analyzers - these need extensive modification before they can be used in new languages. Because of this ease of use, and its applicability to many programming languages, defect prediction has been extended many ways including:

1) Application of defect prediction methods to locating code with security vulnerabilities [15].
2) Understanding the factors that lead to a greater likelihood of defects such as defect prone software components using code metrics (e.g., ratio comment to code, cyclomatic complexity) [16], [17] or process metrics (e.g., recent activity).
3) Predicting the location of defects so that appropriate resources may be allocated (e.g., [18]).
4) Using predictors to proactively fix defects [19].
5) Studying defect prediction not only just release-level [20] but also change-level or just-in-time [21].
6) Exploring “transfer learning” where predictors from one project are applied to another [22], [23].
7) Assessing different learning methods for building predictors [24]. This has led to the development of hyper-parameter optimization and better data harvesting tools [25], [26].

This paper extends defect prediction and planning in yet another way: exploring the trade-offs between explanation and planning and the performance of defect prediction models. But beyond the specific scope of this paper, there is nothing in theory stopping the application of this paper to all of the seven areas listed above (and this would be a fruitful area for future research).

### 2.3 Code refactoring

Code refactoring is an important part of software maintenance. The process is meant to improve the internal quality of software by better structuring the existing code, without changing the external behavior [27], [28]. Such restructuring is assumed to positively affect the software quality by reducing complexity, enhancing maintainability, etc. [29], [30]. Much research has studied the relation between code refactoring process and software quality metrics [31]–[34]. Frequently, it is assumed that (a) metrics like ca, cbm, cbo etc. from Table 1 are good indicators for software complexity and maintainability; (b) good refactoring should cause nontrivial changes to such metrics. In this spirit, many early studies proposed quantitative code refactoring methods [30], [35]–[37]. The essential principle behind the quantitative methods is that the change in certain internal code metrics could significantly improve the corresponding external quality attributes. By knowing which internal code metrics to change, one can classify the corresponding refactoring method needed for the specific purpose. Table 2 illustrates some sample methods taken from Strogglylos and Spinellis, Du Bois, and Kataoka et al. and how these methods might affect different code metrics [30], [36], [37].

Apart from the quantitative evaluation of code refactoring methods, other related works chose to use qualitative attributes such as maintainability, modifiability, and understandability [38]–[40]. Other studies have also shown a correlation between external quality attributes and internal quality attributes (such as the OO metrics used in this paper) [41], [42].

Although defect reduction is not one of the primary goals of code refactoring (since refactoring should not change the external behavior), good refactoring could be helpful to minimize the probability of introducing bugs in

| Metric | Name | Description |
|--------|------|-------------|
| amc    | average method complexity | 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. |
| 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     | efficient couplings | How many other classes is used by the specific class. |
| dam    | data access | Ratio of private (protected) attributes to total attributes |
| dit    | depth of inheritance tree | It’s defined as the maximum length from the node to the root of the tree |
| ic     | inheritance coupling | Number of parent classes to which a given class is coupled (includes counts of methods and variables inherited) |
| lcm    | lack of cohesion in methods | Number of pairs of methods that do not share a reference to an instance variable. |
| locm3  | another lack of cohesion measure | If \( n \) is the number of methods, attributes, in a class and \( \mu(a) \) is the number of methods accessing an attribute, then \( \text{locm3} = (\frac{1}{2} \sum_{i=1}^{n} \mu(a_i)) / (1 - \mu(a)) \) |
| roc    | lines of code | Total lines of code in this file or package. |
| 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 |
| mna    | aggregation | Count of the number of data declarations (class fields) whose types are user defined classes |
| noc    | number of children | Number of direct descendants (subclasses) for each class |
| npm    | number of public methods | Npm metric simply counts all the methods in a class that are declared as public. |
| rfc    | response for a class | Number of methods invoked in response to a message to the object. |
| wmc    | weighted methods per class | A class with more member functions than its peers is considered to be more complex and more error prone. |
| defect | defect | Number of bugs which can be transformed into boolean values for classification. |

**Table 1:** The C-K OO metrics used in defect prediction. The last variable “defect” is the dependent variable.
the future. Therefore, defect reduction could be imported as an extension of the practice. By studying the relation between code metrics and defect proneness using a defect prediction model (or merely the distribution of code metrics themselves), further knowledge could be extracted to generate plans in order to avoid bugs in future releases.

In this paper, various defect reduction algorithms are presented along with our proposed approach, TimeLIME. All the methods attempt the approach of quantitative code refactoring, which is to recommend plans on internal quality metrics, in this case, the C-K OO metrics. The form of the proposed plans by each method is an min-max scaled interval. Compared to the methods previously mentioned in Table 2, which only present a tendency using "+" and "-" signs, plans of this format are more specific and thus more controllable by software developers. On the other hand, if developers are seeking for simpler and more frugal plans, the interval can be easily transformed into a general tendency, which may guide developers back into the existing code refactoring methods.

### 3 Prior Work in Planning Defect Reduction

Over the years, several researchers have proposed various ways to identify appropriate changes on code metrics. This section will illustrate 4 methods that rely on either outlier statistics or cluster deltas.

**Outlier statistics:** The general principle underlying outlier statistics methods is that in the distribution of values for each code metric, there are some extremely large/small values that are associated with greater defect proneness. Therefore, by changing those metrics to not have such outlier values, the code base may be found fewer bugs. This paper presents 3 outlier statistics methods and the major distinction among them is their different ways to identify the threshold for outlier values. In the following text, the methods of Alves et al., Oliveira et al, Shatnawi are based on outlier statistics.

**Cluster deltas** is a framework for learning conjunctions of rules that need to be applied to the code metrics simultaneously. Unlike outlier statistics, which merely studies the statistical distribution of code metrics, cluster deltas is a supervised learner that take account of whether the code base is defective. In the following text, Krishna’s XTREE method uses cluster deltas to learn association rules concerning about when and where to apply a code change.

#### 3.1 Alves, 2010

Alves et al. [9] offers an unsupervised approach that learns from the statistical distribution and scale of OO metrics. At the beginning, Alves’ method will weight each metric value according to the lines of code (LOC in Table 1) of its code class. The weighted metric values will then be normalized by the total sum of weights and sorted in an ascending order. Note that the sorted result is just equivalent to a cumulative probability function where x-axis stands for the weight percentage from 0 to 100% and y-axis the metric scale.

After that, a threshold percentage will be customized (Alves et al. recommends 70%) to identify normal metric values against abnormal metric values. For example, a threshold of 70% will identify the value for each metric where 70% of the classes fall below. The intuition behind this is straightforward: they believe that a code class with outlier metric values that exceed 70% of its peers is more likely to be found bugs.

When we implemented the Alves’ method in our experiment, we augmented the original implementation by also studying the correlation between the code metrics and the defect state of the class. By fitting each dependent variable and the independent variable with a univariate logistic regression classifier:

- we were able to reject metrics that are poor indicators of defects (here we define “poor” as a logistic regression with p-value > 0.05).
- For those metrics that survived from the rejection, the planner will identify the normal range according to the threshold, i.e., [0, 70%] for each metric.
- Finally, during the planning process, any “survived” metric exceeding the threshold value will be proposed to reduce its value to the normal range.

#### 3.2 Shatnawi, 2010

Shatnawi [10] in 2010 provided an alternative to Alves’ method by using VARL (Value of Acceptable Risk Level) to compute the outlier threshold. Initially proposed by Bender [43] in 1999 in his epidemiological studies, the VARL function is a supervised learner that uses the interpretation of the univariate logistic regression model to derive the threshold for an acceptable risk level given by a probability $p_0$ (i.e., $p_0 = 0.05$). That is to say, the VARL believes that the probability $p_0$ of an event is less than 0.05 of the value of the dependent variable is smaller than VARL. The VARL function is as follow:

$$VARL = \frac{1}{\alpha}(\log\left(\frac{p_0}{1-p_0}\right) - \alpha)$$

Here, $\alpha$ is the intercept of the logistic regression, $\beta$ is the coefficient of the logistic regression, and $p_0$ is the acceptable risk probability as described above.

Similar to our procedure of implementing Alves’ method, we ruled out metrics with p-value > 0.05, and computed the VARL for the remaining metrics. We define the proposed plan for each metric as $[0, VARL]$, which
means a metric value exceeding VARL will be recommended a reduction by the planner.

3.3 Oliveira, 2014

Oliveira et al. [11] approach an totally different threshold definition than the previous 2 methods. Instead of deriving an absolute threshold like Alves et al. and Shatnawi did, Oliveira et al. choose to use the relative threshold, which indicates the percentage of classes the the upper bound (threshold) shall be applied to. The general format of their defect reduction rules is as follow:

\[ p\% \text{ of the classes must have } M \leq K \]

Here, \( M \) is the code metric; \( K \) is the threshold value for the corresponding metric; \( p\% \) is the minimum percentage of code classes that are required to follow the restriction specified above.

In order to compute the pair of values \((p, K)\) for each metric \( M \), Oliveira defines 3 functions: \( \text{Compliance}(p, k) \), \( \text{Penalty1}(p, k) \), and \( \text{Penalty2}(p, k) \). The \( \text{Compliance} \) method reports the percentage of classes that follow the rule defined by each pair of values \((p, K)\). The \( \text{Penalty1} \) penalizes the model if the compliance rate is lower than a constant percentage (i.e., 90%). \( \text{Penalty2} \) computes the distance between \( k \) and the median of the preset Tail-th percentile for each metric (Oliveira et al. suggest 90-th percentile). Summing up the 2 penalty values to obtain the total penalty, the method chooses the pair of values \((p, K)\) with the lowest total penalty where a tie will be broken by choosing the highest \( p \) and the lowest \( K \).

3.4 XTREE, 2020

Earlier in 2020, Krishna [44] proposed XTREE, a novel defect reduction planning method that does not rely on outlier statistics. The XTREE planner consists of 3 major parts: (1) Frequent pattern mining; (2) Decision tree construction; and (3) Random walk traversal.

For the first step, XTREE attempts to find what code metrics usually change together by applying an association rule learner on historical data. Since metrics in Table 1 are continuous, XTREE will first discretize the values into intervals using Fayyad-Irani. Then a FP-growth algorithm [45] is used to mine frequent itemsets (in our experimentation XTREE uses \( \text{minSupport} = 5% \times \text{total}_\text{size} \)).

For the second part, the returned maximal frequent itemsets will be used to construct a decision tree. After that, in the third part, the plans will be generated by traversing the decision tree to seek for the closest branch with highest improvement in the probability of the non-defective label. An example of the traversal procedure is illustrated in the Figure 1. Once the current branch is found, the plan will be the \( \Delta \) from the current branch to a nearby desired branch with lower probability of defects.

4 New Methods for Planning Defect Reduction (LIME and TimeLIME)

4.1 LIME

One of the starting points of this research was the realization that the LIME algorithm, first published at KDD’16 [1] could be applied to defect reduction planning. The internal framework of LIME is depicted in Table 3. In summary, given an instance \( I \) of class \( X \), LIME conducts a sensitivity analysis in the neighborhood around \( I \) to determine what could change the class from \( X \) to \( Y \). Using the synthetic data generated around \( I \), LIME can get the classification/regression result from any black-box learner, which will then be used to fit a linear model describes the local region. The parameters of the fitted linear model are then reported as a way to understand how changes in values can adjust the classification; e.g., see Figure 2.

![Fig. 2: An example of output generated by Table 3 when applied to the data sets of the form of Table 1. The y-axis shows the feature name and the confidence interval during which the explanation stays effective. The x-axis indicates the importance weight of each attribute. The prediction label of this instance is 1 (defective), and the weights show how each feature contributes to the prediction. A positive weight means the feature encourages the classifier to predict the instance as a positive label (defective), and vice versa for the negative weight. Larger weights indicate greater feature importance in terms of the prediction value based on that feature weighted by a similarity kernel.](image-url)
IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 6

• LIME is designed to be an add-on to other AI systems (e.g., neural network, support vector machine, and so on). Hence, it treats those AI tools as a “black box” that is queried within its processing.

• Within LIME, some sample generator is used to generate synthetic data which later gets passed to the black box and a similarity kernel, along with the original training data.

• The similarity kernel is an instrument used to weight the prediction results of training data returned by the black box by how similar they are to the instance T.

• The K-Lasso is the procedure that learns the importance weights from the K features selected with Lasso using a class of linear models.

TABLE 3: Inside LIME. From [1]. The feature importance weights are passed to Algorithm 1 and 2, as later elaborated in §6.3. For a sample of the output feature importance weights, see Figure 2.

This paper utilizes LIME and its capability in interpretation to generate defect reduction plans. If a black-box model can predict defects accurately, then it might be "knowledgeable" enough to provide more informative plans than a subject matter experts can provide. The key question is, therefore, how could we access the knowledge owned by a black-box model. In this paper, we imported LIME as the core component of our defect reduction algorithm as we also leverage other software domain knowledge to help LIME restrict the proposed plans in an effective fashion.

Sometimes, we are asked why we are basing our on LIME and not other other tools that explain how to change attributes in order the change the classification of an instance. To say the least, there are very many alternate algorithms. A recent survey by Mueller et al. summarized various kinds of change-explanation generation tools. [46]. Mentioned in their study, Mueller et.al report that this literature is truly vast. Consequently, there are many alternatives to LIME including the abductive framework of Menzies et al. [47] or ANCHORS [48] (which is another change-explanation algorithm generated by the same team that created LIME).

We based our work on LIME, for several reasons. Firstly, LIME scales to large problems. Much recent work has results in methods to scale data mining to very large data sets. Since LIME is based on data mining, then LIME can use those scalability results in order to generate explanations for very large problems.

Secondly, and this is more of a low-level systems reason, alternatives to LIME such as ANCHORS assume discrete classes. Our data has continuous classes which could be binarized into two discrete classes– but only at the cost of losing the information about local gradients. Hence, at least for now, we explore LIME (and will explore ANCHORS in future work).

Lastly, LIME is a widely-cited algorithm. At the time of this writing, LIME has received over 3,000 citations since it was published in 2016. Hence, methods used to improve LIME could also be useful for a wide range of other research tasks. This paper proposes precedence plausibility as a way to improve LIME.

![Fig. 3: TimeLIME: overview of the algorithm, plus the K-test evaluation rig. Note that for evaluating other benchmark algorithms, the area bounded by the dotted line will be replaced by the corresponding algorithm. For further details on TimeLIME, see Algorithm 2.](image)
4.2 TimeLIME

TimeLIME extends LIME by restricting the generated recommendations to the attributes which were seen to be frequently modified within the history of a software projects. Figure 3 offers a graphical overview of this system.

TimeLIME evolved out of comments we heard at workshop on “Actionable Analytics” at ASE’15 [49]. There, business users complained about analytic models saying that rather applying a black-box data mining algorithm, they preferred an approach with a seemingly intuitive appeal. Since software engineers are the target audience of analytics in SE, it is crucial to ensure the proposed recommendations are valued by them. Chen et al. say the term “actionable” can be defined as a combination of “comprehensible” and “operational” [20]. But how to assess “operational”?

In this paper we make the following assumption about “operational”: a proposed change to the code is plausible if it has occurred before. That is, in this work, we claim a plan is the most operational when it has the most precedence in the history log of the project.

Using this assumption, we can generate operational analytics by:

- Looking at two releases of a project and report the attributes that have changed between them;
- Next, when generating explanations, we only used those attributes that have the most changes.

After conducting a survey on 92 controlled experiments published in 12 major software engineering journals, Kampenes et al. [50] argues that in SE, size change can be measured via Hedge’s g value [51]:

$$ g = (M_1 - M_2) / (S_{pooled}) $$

(1)

Here, $M_1$ and $M_2$ are the means of an attribute in two consecutive releases and $S_{pooled}$ comes from 2. This expression is the pooled and weighted standard deviation ($n$ and $s$ denote the sample size and the standard deviation respectively).

$$ S_{pooled} = \sqrt{\left((n_1-1)s_1^2 + (n_2-1)s_2^2\right)/(n_1 + n_2 - 2)} $$

(2)

For the details on how Equation 2 was applied, see §6.3.

5 Assessing Planners: the K-test

This paper claims that plans from TimeLIME planner (that focus on attributes with a history of most change) outperform those generated from LIME, XTREE, Alves, Shatnaw, and Oliveira. To defend that claim, we need some way to assess different planning methods.

There’s an expression in Latin, post hoc ergo propter hoc, which means “after this, then because of this”. This expression refers to the logical fallacy that “if event B follows event A, then event A must be the cause of event B”. The assertion is obviously flawed since other events could be the true trigger of event B. This is why, in this study, we need to carefully evaluate the effectiveness of plans to see if knowledge learned from past code change records actually helps make plans on future code changes.

To address this concern, we use Krishna’s K-test [44]. The K-test uses historical data from multiple software releases to compare the effectiveness of different plans $P_1, P_2, \ldots$. The test is a kind of simulation study that assumes developers were told about a plan at some prior time. Given project information divided into oldest, newer, and most recent, we will use the oldest data to determine what attributes where often changed in a project. Then, using the newer data, we will build plans using LIME, TimeLIME, XTREE, Alves, Shatnaw, and Oliveira. Finally, we will divide the changes between the newer and the most recent into the changes that overlap with the plans, and those that do not.

More precisely, we use consecutive releases $x, y, z$ of some software system. These releases are required to contain named regions of code $C_1, C_2, \ldots$ that can be found in releases $x, y, z$. For example, $C_1$ could be an object-oriented class or a function or a file that is found in all releases. The K-test then assumes that there exists a quality measure $Q$ that reports the value of the regions of named code in different releases. In this study, we will use NDPV (Number of Defects in Previous Version) as the quality measure, which is described later in §6.4. Some method is then applied that uses $Q$ to reflect on the releases $x, y$ in order to infer a plan $P_1$ for improving release $z$.

Given above, the K-test collects following quantities to address our claims made in introduction:

- **RQ1: Smaller**: To measure the succinctness of plans, we collect the number of changes within the plan proposed for each code $C_i$ in release $y$.
- **RQ2: Ready to apply**: To measure how likely a plan can be realized by developers, we compute $J_{y,z} = \Delta_{y,z} \cap P_i$: the overlap between the proposed plan and the code changes.
- **RQ3: Better**: To measure which planner is better at reducing defects, we collect $Q_z - Q_y$: i.e., the change in the number of bugs of the named code $C_i$ between releases $y, z$. We the weight the change $Q_z - Q_y$ by $J_{y,z}$. The intuition is that the planner cannot get credit in a bug-reducing code file if its plan shares little or none similarity with the actual actions done by developers.

The K-test defines better plans as follows:

**DEFINITION**: Plan $P_i$ is “better” that plan $P_j$ if, in release $z$, $P_i$ is associated with most quality improvements.

That is, increasing the size of the overlap of the proposed plan is associated with increasing quality in release $z$; i.e.,

$$ (Q_z - Q_y) \propto |J_{y,z}| $$

For our purposes, the K-test procedure in this paper consists of three steps:

- **Train some defect reduction planner on version $x$.**
- **Use trained planner to generate plans with the aim of fixing bugs reported in version $y$.** In this step, classical LIME planner and TimeLIME planner will utilize the explanations from the explainer and TimeLIME, in addition, will use the historical data analysis to generate plans.
- **On the same set of files that are reported buggy in version $y$, we measure $J_{y,z}$, the overlap score of each plan and the changes in the version $z$, using the Jaccard similarity function.** We also record $Q_z - Q_y$, the change in the number of bugs between the version $y$ and version $z$.

1. Note the connection here to temporal validation in machine learning [52]. In the K-test, no knowledge of the final release $z$ is used to generate the plans.
TABLE 4: A contrived example of how to compute the overlap score using Jaccard similarity function in Eq. (3). Plans P that match the developer actions are marked gray.

For each instance, we compare the extent of overlap between the recommended plan \( P_i \) generated by the planner and the actual developer action in the next release as \( \Delta_{y,z} \) using the Jaccard similarity coefficient.

\[
J_{y,z}(P_i, \Delta_{y,z}) = \frac{(P_i \cap \Delta_{y,z})}{(P_i \cup \Delta_{y,z})}
\]

(3)

Then we convert the coefficient into percentage as our overlap score. As an example shown in Table 4, the overlap score is

\[
\frac{3}{4} \times 100\% = 75\%
\]

Formally speaking, the \( K \)-test is not a deterministic statement that some plan will necessarily improve quality is some future release of a project. Such deterministic causality is a precisely defined concept with the property that a single counterexample can refute the causal claim [53]. The \( K \)-test does not make such statements.

Instead, the \( K \)-test is a statement of historical observation. Plans that are “better” (as defined above) are those which, in the historical log, have been associated with increased values on some quality measure. Hence, they have some likelihood (but no certainty) that they will do so for future projects.

6 EXPERIMENTAL METHODS

The experiment reports the performance of TimeLIME and other state-of-the-art works by comparing the quality of plans recommended by each method. Firstly, we use an over-sampling tool called SMOTE [54] to transform the imbalanced datasets in which defective instances may only take a small ratio of the population. This was needed since, in many of the prior papers that explored our data, researchers warn that small target classes made it harder to build predictors [55].

Secondly, as discussed above, we train the predictor \( P \) and explainer \( E \) on data of version \( x \). Then in version \( y \) we use the explainer to generate explanations only on those data that are reported as buggy. We also use the predictor \( P \) to determine whether we should provide recommendation plans to the instance.

Then we measure the overlap score of our recommended plan and the actual change on the same file in version \( z \). To do this, only select instances that are defective and whose file name has appeared in all releases of data to be instances in need of plans.

The above steps are applied for each benchmark method as well as the TimeLIME planner proposed by this paper. The visualization of the experimental rig is shown in Figure 3. In the classical LIME planner, we use the simple strategy which is to change as many features as it can in order to reduce the number of bugs. On the other hand, for TimeLIME, we first input historical data from the older release to compute the variance of each feature. Then we selected the top-\( M \) features with the largest variance as precedent features, meaning any recommendation on other features will be rebutted. After getting recommended plans from both planners, we assess the performance of two planners using the overlap score as described in §6.4.

Note that the parameter \( M \) can be user-specified and the features may vary with respect to different projects and the releases used as historical data. Here we set the default value of \( M \) to be 5, which means only 25% of all twenty features can be mutated. Our results from experiments suggest that \( M = 5 \) is a useful default setting. Future work shall explore and compare other values of \( M \).

6.1 Data

To empirically evaluate classical LIME vs TimeLIME, we use the standard datasets and measures widely used in defect prediction. In this paper, we selected 8 datasets from the publicly available SEACRAFT project [56] collected by Jureczko et al. for open-source JAVA systems (http://tiny.cc/defects). These datasets keep the logs of past defects as shown in Table 5 and summarize software components using the CK code metrics as shown in Table 1. Note that all the metrics are numerical and can be automatically collected for different systems [57]. The definition and nature of each attribute in the metrics is elaborated by prior researchers Jureczko and Madeyski [58], [59]. Another reason this paper selects these 8 datasets is that they all contain at least 3 consecutive releases, which is required by the evaluation measure described in §5. Since Camel dataset contains 4 consecutive releases, the experiment has 9 trials in total.

6.2 Learner

While other benchmark algorithms don’t need the predictive learn within their model, LIME do require the user to pass in the customized learner, which can be used to generate explanations. Since the goal of this paper is to examine the performance of the defect reduction tools rather than the predictive model, this paper takes one classifier to apply the explanation algorithm on.

Our choice of classifier is guided by the Ghotra et al. [60] study that explored 30 classification techniques for defect prediction. They found that all the classifiers they explored fell into four groups and that Random Forest classifiers were to be found in their top-ranked group.

A Random Forest classifier is an ensemble learner that fits a number of decision tree classifiers on different sub-
samples of the dataset and generates predictions via average voting from all the classifiers [61]. It is impossible to visualize a fitted Random Forest classifier as a finite set of rules and conditions due to the voting process. Therefore, Random Forest classifier is considered a non-interpretable model. Hence, it is a suitable choice for this study.

6.3 Planners

This section discusses the internals of our planners, including a RandomWalk planner (which we use to compare our results against a baseline random guesser).

Using LIME, we generate plans to change each defective classification by the learner model. We use the default parameter setting of LIME, which is 5000 samples around the instance neighborhood, and the entropy-based discretizer. The explanation object return by a LIME explainer is a tuple in which each element contains the feature name and the corresponding feature importance. It also provides a discretized interval indicating the range of values during which the feature will maintain the same effect to the prediction result. As described in Algorithm 1, the simple planner based on the classical LIME will recommend changes on all features that contribute to the defective prediction. Algorithm 2 shows the TimeLIME planner. Each planner uses feature ranges in the form of an interval, generated by flipping the discretized interval relative to the midpoint of the feature value range [0, 1].

Apart from the planners discussed previously, we also use a planner named RandomWalk as a “straw-man” baseline algorithm. This planner, as shown in Algorithm 3, assigns random recommendations to each variable stochastically. In our experiment setting, we set the probability to 0.5 meaning that all features have 50% chance to be recommended a change.

Figure 3 shows the overall procedure of the experimentation with TimeLIME embedded as the evaluated planner. The region bounded by dashes could be replaced by any other algorithms described in §3 as well as the RandomWalk planner.

6.4 Performance Criteria

The two performance criteria in this experiment, as described in the §5, are the overlap score of individual plans and the number of bugs reduced/added in the next release of the project. The function used for computing the overlap score is the Jaccard similarity function in Eq. 3, and the other criterion is measured by the metric NDPV (Number of Defects in Previous Version), which returns the number of bugs fixed (or added) in a given file during the development of the previous release. The nature of NDPV and similar metrics have been evaluated by plentiful researchers [62]–[65].

7 Results

7.1 RQ1: Does TimeLIME provide succinct plans?

Figure 4 reports the mean size of plans across all instances in release z. We note that:
- RandomWalk method’s plans are so large since this planner does not use information from the domain to constrain its results.
- TimeLIME generates much smaller plans compared to many other planners including classical LIME.
- The only planner that consistently produces smaller plans in the Shatnawi method but, as seen in the RQ2 results (below), the Shatnawi obtains performance that is far worse than TimeLIME.
Note that since TimeLIME in the experiment restricts plans to the top 5 features with highest Hedge’s $q$ scores, the size of an TimeLIME plan will never be more than 5. However, as shown in the figure, the average size of TimeLIME plans is always smaller than 5. This implies that the original code refactoring plans, proposed by the classical LIME planner, do contain unprecedented changes which then get rejected by the TimePlanner. In summary, for RQ1, we say:

**Answer 1:** Several planners, including LIME, generate plans that are far larger than those found by TimeLIME. And the only planner that always generates smaller plans has much worse performance.

Fig. 4: RQ1 results: The mean size of TimeLIME’s plans (across all instances in release $z$) is often much smaller than LIME, and RandomWalk. Y-axis shows the number of features changed by recommended plans.

7.2 RQ2: Could developers apply the changes proposed by TimeLIME?

To answer this question, we look at project activity in the period after TimeLIME makes its recommendations.

A recommendation/plan can be proven useful if there is evidence indicating developers could actually apply those kinds of changes. Table 6 and Table 7 comments on how often developers are willing to perform the plans suggested by different planners. Both tables are generated using the $K$-test procedure described above. Each cell in Table 6 shows the median value of the $J_{y,z}$ overlap score measured from Eq. 3 in §5 for all instances within the projects.

Table 7 shows the interquartile range (IQR) of all overlap scores quantile among all plans generated by planners. With similar median scores, a smaller IQR means the planner is more stable and robust. It is noteworthy that the Random Planner always obtains very small IQRs in all project. This is because plans generated by Algorithm 3 are equivalently bad as indicated from the median scores. On contrary, TimeLIME has similarly small IQRs while maintaining the highest median scores in all project, which means it prevails other planners in terms of providing plans that better resemble developers’ choices.

To summarize the results from different planners:

- Unsurprisingly, Random Planner has the lowest similarity scores among all projects.
- The 4 prior works (XTREE, Alves, Shatnawi, and Oliveira) are equivalently good.
- Different projects have different baselines for the similarity evaluation. For example, with Xalan, every planner expect Random obtains a relatively high scores whereas they perform equally poorly in the Ant project.
- TimeLIME obtains the highest score in every project with a relatively IQR scores. This means the performance of TimeLIME, in terms of similarity to actual actions, is not only good but also stable.
- The classical LIME planner has a volatile performance: It is either performs best or worst. In other words, compared to TimeLIME, it is not recommended.

In summary, we answer RQ2 as follows:

**Answer 2:** We find a very large overlap (median=80%) between TimeLIME’s recommendations and the actual actions taken by developers.

Hence we say developers would be able to apply TimeLIME’s recommendations.

7.3 RQ3: Is TimeLIME better at defect reduction?

As discussed earlier, better plans in defect reduction field are believed to be those that are (a) easier to apply while (b) maintaining the effectiveness in reducing bugs. The first criterion has already been met. As seen there, the plans made by TimeLIME are much smaller, hence easier to apply, than the other methods studied here. Also, as seen above, the plans from TimeLIME correspond well to the known actions of developers.

To measure the second criterion, we chose to use a weighted sum function to compute the net gain of each planner. The weighted sum function in Eq. (4) weights the NDPV by the overlap score of the plan.

In the experiment, each plan $p_i$ from the all $N$ plans returns an overlap score $s_i$ and a NDPV number $n_i$ (positive number indicates bugs reduced, negative number indicates bugs added). Then we weight the NDPV $n_i$ by the planner by $s_i$ to compute the aggregate score $S$.

$$S = \sum s_i \ast n_i$$

Note that the larger the overlap the greater the change in the number of defects introduced.

Additionally, given that the total number of bugs varies from each project as shown in Table 5, a project with more bugs reduced in the validation dataset will expect the planner to score more than the planner whose validation dataset has fewer bugs reduced so that their performance can be considered proportionally similar. For example, project A has $NDPV = 100$ in release $y$ and another project B has $NDPV = 10$ in its next release $y$. If one would like to see similar performance of a planner on these 2 projects, the weighted score in project $A$ $S_A$ is expected to be 10 times higher than $S_B$ since there are potentially more bugs that can be reduced by a planner in project $A$ than in project $B$.
this paper, the similarity score and weighted net gain are standard measurement in evaluating a plan. As applied in
ners, we have faced the difficulty of defining what is the
LIME against those generated by different benchmark plan-

The lesson we have learned from this work is that when
automated defect reduction tools are looking into the future
to make plans, it is important that they also look back
into the history in order to find precedence which supports
their decisions. This work has proven that this approach is
admissible and could dramatically improve the quality of
plans.

and it won’t make any sense if a planner gains the same
score in both projects. From this perspective, we scale the
final score \( S \) in Eq. 4 by the sum of NDPV within the project
to get the scaled score \( S_{\text{scaled}} \):

\[
S_{\text{scaled}} = \frac{\sum_{i=1}^{N} s_i \cdot n_i}{\sum_{i=1}^{N} n_i} \tag{5}
\]

The visualized result in Table 8 shows that the TimePlanner
obtains highest average \( S_{\text{scaled}} \) scores in most of the projects
(8 out of 9).

The overall result is very clear:

Answer 3: The changes proposed by TimeLIME are associated
with a much larger reduction in defects than classic LIME and other benchmark algorithms.

8 DISCUSSION

The lesson we have learned from this work is that when
automated defect reduction tools are looking into the future
to make plans, it is important that they also look back
into the history in order to find precedence which supports
their decisions. This work has proven that this approach is
admissible and could dramatically improve the quality of
plans.

On the other hand, while comparing plans from Time-
LIME against those generated by different benchmark plan-
ners, we have faced the difficulty of defining what is the
standard measurement in evaluating a plan. As applied in
this paper, the similarity score and weighted net gain are
selected as the measurements to determine what is a good
plan. However, more aspects need to be taken account of.
The size of the plan, for example, is one of them.

Since each plan consists of intervals, each of which
represents the desired change in the corresponding code
metric, the size of an interval could be a crucial factor
when evaluating the feasibility and precision of a plan. If
the interval is too small, like \([0, 0.001]\), the plan can be
hard to achieve and thus loses feasibility. If the interval is
too large, like \([0, 0.99]\), the plan will probably overlap with
most of the actual actions taken by developers, no matter
the actual action reduces or adds bugs in the upcoming
release. This kind of plans will obtain very high similarity
scores but they are meaningless as they loses precision.
Such trade-off between feasibility and precision could be
a tricky burden when someone wants to define what is a
good plan. In this paper, we attempts to bypass such
doubts by utilizing the weighted net gain scores. In this
measurement, a defect reduction planner will be awarded
once its plan is similar to a defect-reducing action, and
will be penalized if the plan is similar to a defect-adding
action. That is to say, an ideal planner with the highest
score should provide plans that are very similar to defect-
reducing actions while very dissimilar to those introduce
bugs. In such case, if a planner keeps offering plans with no
precision (like \([0, 0.99]\)), it will be strongly penalized since
the plan also has a very high similarity score with defect-
adding actions. As reported in Figure 5, the average size of
a single change proposed by TimeLIME planner is around
0.3. Some benchmark planners such as the classical LIME,
Oliveira, and so on also have similar mean sizes. However,
we do see a relatively intensive fluctuation on XTREE and
Shatnawi planners: in some trials they propose changes of
large range, and in other trial the size of the interval is
tiny (like Shatnawi in Camel1 and Synapse). Our current study does not show a correlation between the size and the effectiveness of such changes, but as discussed above, we do recommend planners to avoid making extremely large or small changes.

Turning now to another matter, the above results show that the Shatnawi planner performed poorly. When comparing the result shown in Table 6, 7 and the result in Table 8, it is obvious that although Shatnawi achieves similarity scores close to other benchmark methods, the weight net gain is actually quite poor, which means the plans proposed by Shatnawi are just as good at reducing bugs as at adding bugs. One possible cause of the poor performance could be that Shatnawi’s method failed to find metrics with strong correlation to the dependent variable, or in some cases, the changes proposed by it are too small (as seen in Figure 5) to be effective. This finding indicates the practical usefulness of our measurements and simulates us to further explore various aspects that should be considered during the performance evaluation.

9 Threats to Validity

Due to the complexity of the experiment designed in this case study, there are many factors that can threaten the validity of these results.

9.1 Learner Bias

This paper selects Random Forest classifier as the black-box classifier because prior research has shown that Random Forest classifier is ranked as one of the top models among all 32 classifiers used in defect prediction. However, the preeminent predictive power of Random Forest classifier does not ensure that explanations derived from it are preeminent code refactoring plans as well. Other methods from the top rank may be more suitable in the problem of explanation generation while we haven’t explored more.

9.2 Instrument Bias

Explainable AI is experiencing its resurgence and various approaches are proposed to generate explanations. Although LIME is one of the widely cited and well-known tools, it is possible other tools are more suitable in solving SE problems, which can make solutions from LIME sub-optimal. Hence, to verify if adding in SE knowledge can always improve AI tools, we need to make a comprehensive exploration that includes more explanation generation methods.

9.3 Hyper-parameter Tuning

Past researches have shown how hyper-parameter optimization can boost the performance of a classifier used in defect prediction. Since in this paper we concentrate on the modification of the explainer instead of the learner, we used a simple grid search to find the optimal parameter setting. It can be possible that the current setting is sub-optimal and by using the actually optimal settings we might receive different experiment result.

10 Related work

Much research urges that interpretability should become an important factor in assessing analytical models in software engineering because software developers expect the model to provide understandable suggestions that can be actually achieved in real-world practice [66]–[68].

Recently at TSE’20, Jiarpakdee et al. modified LIME using hyper parameter optimization techniques, and assessed its performance in defect prediction via output stability [69]. The result has shown that explanations generated from their method are not only more stable among re-generations, but also understandable to software developers. The major difference is that:

- Jiarpakdee et al. assess the viability of applying model-agnostic techniques (such as LIME) in defect prediction whereas this paper assesses the practical effectiveness of LIME in re-organizing a project
- Jiarpakdee et al. explore possible means to improve the explanation generation procedure where as this paper explores methods to refine LIME’s results into more actionable and effective plans for defect reduction.

11 Future work

For future work, we need to take action to retire the above threats to validity.

11.1 More Learners

More black-box learners should be used in the experiment to construct a more comprehensive comparison. Although the limited sample amount of defect prediction datasets has ruled out many deep learning models such as Neural Network due to the overhead, there are still many other models, including but not limited to Random Subspace Sampling and Sequential Minimal Optimization, applicable for this experiment.
LIME’s plans: We find that, by respecting temporal precedence, TimeLIME does not respect temporal precedence as the classical LIME planner does. This paper has compared TimeLIME, a defect reduction planner built upon the restricted LIME explanations, with other planners including the classical LIME planner which have appeared in the historical record of that project. When dealing with temporal data (e.g., successive software releases), it is useful to restrict any conclusions to actions that have appeared in the historical record of that project. This paper has compared TimeLIME, a defect reduction planner built upon the restricted LIME explanations, with other planners including the classical LIME planner which does not respect temporal precedence as TimeLIME does. We find that, by respecting temporal precedence, TimeLIME’s plans:

- Are succinct: In terms of the average size of recommended plans. The TimeLIME generally generates smaller plans than the classical LIME and RandomWalk in every project. The plans are also equivalently succinct compared to other benchmark methods in this paper. Smaller plans are preferred to larger plan since the latter can be faster to apply.
- Better resemble developers’ actions: In terms of the overlap between the proposed plans and the developer actions in the upcoming release, plans proposed by TimeLIME better match what developers actually do.
- Are better at reducing defects: In terms of the scaled weighted scores $S_{\text{scaled}}$ that indicate the overall net gain received per project. TimeLIME gets the highest score among all planners in 8 out of 9 trials. (while the classical LIME wins in only 1 project).

In conclusion, we assert two things. Firstly, the above results clearly show that a planner with precedence-based reasoning embedded can generate better plans in defect reduction, in terms of how achievable, precise, and effective these plans are.

Secondly, and more generally, our community should be more careful about using off-the-shelf AI tools without first adapting them using SE knowledge. We think it is rash and ill-advised just to throw standard AI tools at SE problems. Those AI methods can be greatly enhanced via SE knowledge. As shown here, adding that knowledge is not a complex thing to do. Further, once that knowledge is applied, this can result in dramatically better systems.

ACKNOWLEDGEMENTS

This work was partially funded by a research grant from the National Science Foundation (CCF #1703487) and the Laboratory for Analytical Sciences, North Carolina State University.

REFERENCES

[1] M. T. Ribeiro, S. Singh, and C. Guestrin, “‘why should i trust you?’ explaining the predictions of any classifier,” in Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, 2016, pp. 1135–1144.
[2] A. Begel, J. Bosch, and M.-A. Storey, “Social networking meets software development: Perspectives from github, mood, stack exchange, and topcoder,” IEEE Software, vol. 30, no. 1, pp. 52–66, 2013.
[3] D. Chen, K. Stolte, and T. Menzies, “Replication can improve prior results: a github study of pull request acceptance,” in 2019 IEEE/ACM 27th International Conference on Program Companion (ICPC). IEEE, 2019, pp. 170–190.
[4] P. Devanbu, T. Zimmermann, and C. Bird, “Belief & evidence in empirical software engineering,” in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016, pp. 108–119.
[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, “Don’t touch my code! examining the effects of ownership on software quality,” in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 4–14.
[6] C. Passos, A. P. Braun, D. S. Cruzes, and M. Mendonca, “Analyzing the impact of beliefs in software project practices,” in 2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, 2011, pp. 444–452.
[7] N. Shriran and T. Menzies, “Assessing practitioner beliefs about software defect prediction,” arXiv, pp. arXiv–1912, 2019.
[8] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, “Perceptions, expectations, and challenges in defect prediction,” IEEE Transactions on Software Engineering, 2018.
[9] T. L. Alves, C. Ypma, and J. Visser, “Deriving metric thresholds from benchmark data,” in 2010 IEEE International Conference on Software Maintenance. IEEE, 2010, pp. 1–10.
[10] R. Shatnawi, “A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems,” IEEE TSE, 2010.
[11] P. Oliveira, M. T. Valente, and F. P. Lima, “Extracting relative thresholds for source code metrics,” in 2014 Software Evolution Work-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). IEEE, 2014, pp. 254–263.
[12] W. Fu, T. Menzies, and X. Shen, “Tuning for software analytics: Is it really necessary?” IST, 2016.
[13] L. C. Briand, V. Brusil, and C. J. Hetmanski, “Developing interpretable models with optimized set reduction for identifying high-risk software components,” IEEE Transactions on Software Engineering, vol. 19, no. 11, pp. 1028–1044, 1993.
Kewen Peng is a first year Ph.D. student in Computer Science at North Carolina State University. His research interests include using and refining artificial intelligence methods to solve problems in software engineering.

Tim Menzies (IEEE Fellow) is a Professor in CS at North Carolina State University. His research interests include software engineering (SE), data mining, artificial intelligence, search-based SE, and open access science. http://menzies.us