Noname manuscript No.
(will be inserted by the editor)

Learning Actionable Analytics from Multiple Software Projects

Rahul Krishna · Tim Menzies

Received: date / Accepted: date

Abstract The current generation of software analytics tools are mostly prediction algorithms (e.g. support vector machines, naive bayes, logistic regression, etc). While prediction is useful, after prediction comes *planning* about what actions to take in order to improve quality. This research seeks methods that generate demonstrably useful guidance on "what to do" within the context of a specific software project. Specifically, we propose XTREE (for within-project planning) and BELLTREE (for cross-project planning) to generating plans that can improve software quality. Each such plan has the property that, if followed, it reduces the expected number of future defect reports. To find this expected number, planning was first applied to data from release x. Next, we looked for change in release $x + 1$ that conformed to our plans. This procedure was applied using a range of planners from the literature, as well as XTREE. In 10 open-source JAVA systems, several hundreds of defects were reduced in sections of the code that conformed to XTREE's plans. Further, when compared to other planners, XTREE's plans were found to be easier to implement (since they were shorter) and more effective at reducing the expected number of defects.

Keywords Data Mining, Actionable Analytics, Planning, bellwethers, defect prediction.

Rahul Krishna
Computer Science
NC State University
E-mail: i.m.ralk@gmail.com

Tim Menzies
Computer Science
NC State University
E-mail: timm@ieee.org

1 Introduction

Data mining tools have been succesfully applied to many applications in software engineering; e.g. (Czerwonka et al., 2011; Ostrand et al., 2004; Menzies et al., 2007a; Turhan et al., 2011; Kocaguneli et al., 2012; Begel and Zimmermann, 2014; Theisen et al., 2015). Despite these successes, current software analytic tools have certain drawbacks. At a workshop on "Actionable Analytics" at the 2015 IEEE conference on Automated Software Engineering, business users were vocal in their complaints about analytics (Hihn and Menzies, 2015). "Those tools tell us *what is*, " said one business user, "But they don't tell us *what to do*". Hence we seek new tools that offer guidance on "what to do" within a specific project.

We seek such new tools since current analytics tools are mostly *prediction* algorithms such as support vector machines (Cortes and Vapnik, 1995), naive Bayes classifiers (Lessmann et al., 2008), logistic regression (Lessmann et al., 2008). For example, defect prediction tools report what combinations of software project features predict for some dependent variable (such as the number of defects). Note that this is a different task to *planning*, which answers the question: what to *change* in order to *improve* quality.

More specifically, we seek plans that propose *least* changes while most *improving* software *quality* where:

- *Quality* = defects reported by the development team;
- *Improvement* = lowered likelihood of future defects.

This paper advocates the use of the *bellwether effect* (Krishna et al., 2016, 2017a; Mensah et al., 2018) to generate plans. This effect states that:

> *" . . . When a community of programmers work on a set of projects, then within that community there exists one exemplary project, called the bellwether*[1]*, which can best define quality predictors for the other projects . . . "*

Utilizing the bellwether effect, we propose a cross-project variant of our XTREE contrast set learner called BELLTREE where

$$BELLTREE = Bellwether{+}XTREE$$

BELLTREE searches for an exemplar project, or *bellwether* (Krishna et al., 2017a), to construct plans from other projects. As shown by the experiments of this paper, these plans can be remarkably effective. In 10 open-source JAVA systems, hundreds of defects could potentially be reduced in sections of the code that followed the plans generated by our planners. Further, we show that planning is possible across projects, which is particularly useful when there are no historical logs available for a particular project to generate plans from.

The structure of this paper is as follows: the rest of this section highlights the key contributions of this work(§ 1.1), and relationships between this work and our prior work (§ 1.3). In § 2, we introduce the research questions asked in this paper and briefly discuss our findings. In § 3 we discuss the background which include some of related work in the area. There, in § 4.1, the notion of planning and the different kinds of planners studied here. § 6 contains the research methods, datasets, and evaluation strategy. In § 7 we answer the research questions. In § 8 we discuss the implications of our findings. Finally, § 9 and § 10 present threats to validity and conclusions respectively.

[1] According to the Oxford English Dictionary, the bellwether is the leading sheep of a flock, with a bell on its neck.

1.1 Contributions

The key contributions of this work are:

1. New kinds of software analytics techniques: This work combines planning (Krishna et al., 2017a) with cross-project learning using bellwethers (Krishna et al., 2016). Note that our previous work (Krishna et al., 2016; Krishna and Menzies, 2018) explored prediction and not the planning as described here. Also, previously, our planners (Krishna et al., 2017a) only explored within-project problems (but not cross-project).

2. Compelling results about planning: Our results show that planning is successful in producing actions that can reduce the number of defects; Further, we see that plans learned on one project can be translated to other projects.

3. More evidence of generality of bellwethers: Bellwethers were originally used in the context of prediction (Krishna et al., 2016) and have been shown to work for (i) defect prediction, (ii) effort estimation, (iii) issues close time, and (iv) detecting code smells (Krishna and Menzies, 2018). This paper extends those results to show that bellwethers can also be used from cross-project planning. This is an important result of much significance since, it suggests that general conclusions about SE can be easily found (with bellwethers).

4. An open source reproduction package containing all our scripts and data. For readers interested in replicating this work, kindly see `https://git.io/fNcYY`.

1.2 Post Hoc Ergo Propter Hoc?

The Latin expression *post hoc ergo propter hoc* translates to "after this, therefore because of this". This Latin expression is the name given to the logical fallacy that "since event Y followed event X, event Y must have been caused by event X". This can be a fallacy since another event Z may have influenced Y.

This concern was very present in our minds as we developed this paper. Prior to this paper, it was an open issue if XTREE/BELLTREE's plans work on future data. Accordingly we carefully evaluated if knowledge of past changes were useful for planning future changes. The details of that evaluation criteria are offered later in this paper (see "The $\mathbb{K}$-test" of §6.1.1). For now, all we need say is that:

- We sorted our data via its associated timestamps into *older*, *newer*, and *latest* (later in this paper we will call these *train, test, validate*, respectively). We say that the *older* plans are those learned from the *older data*.
- If developers of *newer* code knew about the older plans, then they would apply them either (a) *very little*, (b) *some*, (c) *more*; or (d) *mostly*.
- We also note that it is possible to automatically identify each of those four kinds developers as those whose changes between *newer* and *latest* overlap with the older plans (a) *very little*, (b) *some*, (c) *more*; or (d) *mostly*.

The experiments of this paper show that, when we explored real world data from from the *newer* and *latest* versions, then:

- If projects changes overlap *very little* with older plans, then defects are not reduced.
- But if projects changes *mostly* overlap with older plans, then defect are much lower.

To be clear, XTREE/BELLTREE *does not* generate causal models for software defects. However, our results suggest that it can be very useful to follow our plans.

1.3 Relationship to Prior Work

As for the connections to prior research, as shown in Fig. 1, originally in 2007 we explored software quality prediction in the context of training and testing within the same software project (Menzies et al., 2007c). After that we found ways in 2009 to train these predictors on some projects, then test them on others (Turhan et al., 2009). Subsequent work in 2016 found that bellwethers were a simpler and effective way to implement transfer learning (Krishna et al., 2016), which worked well for a wide range of software analytics tasks (Krishna and Menzies, 2018).

In the area of planning, we introduced the possibility of using XTREE for planning as a short report at a workshop on "Actionable Analytics" in ASE '15 (Krishna and Menzies, 2015), we followed this up a slightly more detailed report in the IST journal (Krishna et al., 2017b). These initial findings on XTREE were also presented at the IEEE ASE'17 Doctoral Symposium (Krishna, 2017). The panel highlighted the following limitations:

– *Inadequate Validation.* Our initial report uses *defect predictors* to assess plan effectiveness. However, the performance of those defect prediction schemes were limited to at most 65% (as shown in Figure 5 of (Krishna et al., 2017b)).
– *Smaller Datasets.* Due to the limited predictive performance of the defect predictors used in the previous studies, the results were reported on only five projects.
– *Metric interdepencies ignored.* The previous variant of XTREE also did not take into consideration the interaction between individual metrics.

Accordingly, in this paper we present a updated variant of XTREE, including new experiments on more projects.

Further, this current article addresses a much harder question: can plans be generated from one project and applied to the another? In answering this, we have endeavored to avoid our mistakes from the past, e.g., the use of overly complex methodologies to achieve a relatively simpler goal. Accordingly, this work experiments with bellwethers to see if this simple method works for planning as with prediction.

One assumption across much of our work is the *homogeneity* of the learning, i.e., although the training and testing data may belong to different projects, they share the same attributes (Krishna et al., 2016, 2017a; Krishna and Menzies, 2018; Menzies et al., 2007c; Turhan et al., 2009). Since that is not always the case, we have recently been exploring heterogeneous learning where attribute names may change between the training and test sets (Nam et al., 2017). Heterogeneous planning is primary focus of our future work.

2 Research Questions

The work in this paper is gudeied by the following research questions.

	Data source= Within	Data source = Cross	
Prediction	TSE '07 (Menzies et al., 2007c)	EMSE '09 (Turhan et al., 2009) ASE '16 (Krishna et al., 2016) TSE '18 (Krishna and Menzies, 2018)	TSE '17 (Nam et al., 2017)
Planning	IST '17 (Krishna et al., 2017a)	This work	Future work
	Homogeneous		Heterogeneous

Fig. 1: Relationship of this paper to our prior research.

RQ1: How well do planners' recommendations match developer actions?

Motivation: There is no point offering plans that no one will follow. Accordingly, on this research question, we ask how many of a planner's recommendations match with the actions taken by developers to fix defects in their files.
Approach: We measure the *overlap* between the planners' recommendations developers' actions. Then, plot the aggregate number files for overlap values ranging from 0% to 100% in bins of size 25% (for ranges of $0-25\%$, $26-50\%$, $51-75\%$, and $76-100\%$). Planners that have the larger aggregate number files for higher overlap ranges are considered better.
Evaluation: We compare XTREE with three other outlier statistics based planners from current literature namely, those of Alves et al. (2010), Shatnawi (2010), and Oliveira et al. (2014).

Result: XTREE significantly outperforms all other outlier statistics based planners. Further, in all the projects studied here, most of the developers actions to fix defects in a file has a $76-100\%$ overlap with the recommendations offered by XTREE.

RQ2: Do planners' recommendations lead to reduction in defects?

Motivation: The previous research question measured the extent to which a planner's recommendations matched the actions taken by developers to fix defects in their files. But, a high overlap in most files does not necessarily mean that the defects are actually reduced. Likewise, it is also possible that defects are added due to other actions the developer took during the development. Thus, here we ask how many defects are reduced, and how many are added, in response to larger overlap with the planners' recommendations.
Approach: Like before, we measure the *overlap* between the planners' recommendations developers' actions. Then, we plot the aggregate number defects reduced in file with overlap values ranging from 0% to 100% in bins of size 25% (for ranges of $0-25\%$, $26-50\%$, $51-75\%$, and $76-100\%$). Planners that have a large number defects reduced for higher overlap ranges are considered better.
Evaluation: Similar to RQ1, we compare XTREE with three other outlier statistics based planners of Alves et al., Shatnawi, and Oliveira, for the overall number of defects reduced and number of defects added.

Result: Plans generated by XTREE are superior to other outlier statistics based planners in all 10 projects. Planning with XTREE leads to the far larger number of defects reduced as opposed to defects added in 9 out of 10 projects studied here.

RQ3: Are cross-project plans generated by BELLTREE as effective as within-project plans of XTREE?

Motivation: The previous research questions we assume that there exists historical data to construct the planning algorithms. However, given the pace of software change, for new projects, it is quite possible that there is insufficient historical data to perform planning. Thus, this research question asks if it is possible to use data from other software projects to construct planners to generate recommendations.

Approach: We use a cross-project planner that discovers the *bellwether* dataset. Using this bellwether project, we construct XTREE as generate plans as usual. We refer to this combination of using Bellwethers with XTREE as BELLTREE.
Evaluation: Here we compare BELLTREE with a conventional XTREE and with one other outlier statistics based planner (Shatnawi) to measure the number of defects reduced and number of defects added.
Result: The effectiveness of BELLTREE is comparable to the effectiveness of XTREE. In 8 out of 17 BELLTREE outperformed XTREE and 9 out of 17 cases, XTREE outperformed BELLTREE. BELLTREE and XTREE outperformed other planners in all cases.

3 Motivation

3.1 Defect Prediction

As projects evolve with additional functionalities, they also add defects, as a result the software may crash (perhaps at the most inopportune time) or deliver incorrect or incomplete functionalities. Consequently, programs are tested before deployment. However, an exhaustive testing is expensive and software assessment budgets are finite (Lowry et al., 1998). Exponential costs quickly exhaust finite resources, so standard practice is to apply the best available methods only on code sections that seem most critical.

One approach is to use defect predictors learned from static code metrics. Given software described in terms of the metrics of Table 1, data miners can learn where the probability of software defects is the highest. These static code metrics can be automatically collected, even for very large systems (Nagappan and Ball, 2005). Further, these static code metrics based defect predictors can be quickly adapted to new languages by building lightweight parsers to computes metrics similar to that of Table 1. Over the past decade, defect predictors have granered a significant amount of interest. They are frequently reported to be capable of finding the locations of over 70% (or more) of the defects in code (Menzies et al., 2007d; Nam et al., 2013a; Fu et al., 2016; Ghotra et al., 2015; Lessmann et al., 2008; Nam et al., 2017; Krishna and Menzies, 2018). Further, these defect predictors seem to have some level of generality Nam et al. (2013a); Nam and Kim (2015a); Krishna et al. (2016); Krishna and Menzies (2018). The success of these methods in finding bugs is markedly higher than other currently-used industrial methods such as manual code reviews (Shull et al., 2002). Although other methods like manual code reviews are much more accurate in identifying defects, they take much higher effort to find a defect and also are relatively slower. For example, depending on the review methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six people (Menzies et al., 2002). For these reasons, researchers and industrial practitioners use static code metrics to guide software quality predictions. Defect prediction has been favored by organizations such as Google Lewis et al. (2013) and Microsoft (Zimmermann et al., 2009).

Although the ability to predict defects in software systems is viewed favorably by researchers and industrial practitioners, the current generation of defect prediction is subject to several criticisms. There is are open debates on the efficacy of static code metrics and the existence of causal links between these metrics and the defect counts.

While a number of studies favor static code metrics, there are some that prefer other type of metrics. We explore these in greater detail in § 3.2.

Another major criticism of software defect prediction is that they lack actionable guidance, i.e., while these techniques enable developers to target defect-prone areas faster, but do not guide developers toward a particular action that leads to a fix. Without a such guidance, developers are often tasked with making a majority of the decisions. However, this could be problematic since researchers have cautioned that developers' cognitive biases often leads to misleading assertions on how best to make a change. For instance, Passos et al. (Passos et al., 2011) remarks that developers often assume that the lessons they learn from a few past projects are general to all their future projects. They comment, "past experiences were taken into account without much consideration for their context" (Passos et al., 2011). Such warnings are also echoed by Jørgensen & Gruschke (Jørgensen and Gruschke, 2009). They report that the supposed software engineering experts seldom use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects. Other studies have shown that some widely-held views are now questionable given new evidence. Devanbu et al. observes that, on examination of responses from 564 Microsoft software developers from around the world, the programmer beliefs can vary significantly with each project, but that these beliefs do not necessarily correspond with actual evidence in that project (Devanbu et al., 2016).

For the above reasons, in this paper, we seek newer analytics tools that go beyond traditional defect prediction to offer "plans". Instead of just pointing to the likelihood of defects, these "plans" offer a set of changes that can be implemented to reduce the likelihood of future defects. We explore the notion of planning in greater detail in the following section (see § 4).

3.2 Choice of Software Metrics

The data used in our studies use *static code metrics* to quantify the aspects of software design. These metrics have been measured in conjunction with faults that are recorded at a number of stages of software development such as during requirements, design, development, in various testing phases of the software project, or with a post-release bug tracking systems. Over the past several decades, a number of *metrics* have been proposed by researchers for the use in software defect prediction. These metrics can be classified into two categories: (a) Product Metrics, and (b) Process Metrics.

Product metrics are a syntactic measure of source code in a specific snapshot of a software project. The metrics consist of McCabe and Halstead complexity metrics, LOC (Lines of Code), and Chidamber and Kemerer Object-Oriented (CK OO) metrics as shown in as shown in Table 1. McCabe (1976) and Halstead (1977) metrics are a set of static code metrics that provide a quantitative measure of the code complexity based on the decision structure of a program. The idea behind these metrics is that the more structurally complex a code gets, the more difficult it becomes to test and maintain the code and hence the likelihood of defects increases. McCabe and Halstead metrics are well suited for traditional software engineering and are inadequate in and of themselves. To measure aspects of object oriented (OO) design such as classes, inheritance, encapsulation, message passing, and other unique aspects of OO approach, (Chidamber and Kemerer, 1994) developed as set of OO metrics.

Table 1: Sample static code attributes.

Metric	Description
wmc	weighted methods per class
dit	depth of inheritance tree
noc	number of children
cbo	increased when the methods of one class access services of another.
rfc	number of methods invoked in response to a message to the object.
lcom	number of pairs of methods that do not share a reference to an instance variable.
ca	how many other classes use the specific class.
ce	how many other classes is used by the specific class.
npm	number of public methods
locm3	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 $lcom3 = ((\frac{1}{a}\sum_j^a \mu(a_j)) - m)/(1-m)$.
loc	lines of code
dam	ratio of private (protected) attributes to total attributes
moa	count of the number of data declarations (class fields) whose types are user defined classes
mfa	number of methods inherited by a class plus number of methods accessible by member methods of the class
cam	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.
ic	number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)
cbm	total number of new/redefined methods to which all the inherited methods are coupled
a mc	average methods oer class
max_cc	maximum McCabe's cyclomatic complexity seen in class
avg_cc	average McCabe's cyclomatic complexity seen in class
defect	Defects found in post-release bug-tracking systems.

When used in conjunction with McCabe and Halstead metrics, these measures lend themselves to a more comprehensive analysis.

Process metrics differ from product metrics in that they are computed using the data obtained from change and defect history of the program. Process metrics measure such aspects as the number of commits made to a file, the number of developers who changed the file, the number of contributors who authored less than 5% of the code in that file, the experience of the highest contributor. All these metrics attempt to comment on the software development practice rather than the source code itself.

The choice of metrics from the perspective of defect prediction as has been a matter of much debate. In recent years, a number of researchers and industrial practitioners (at companies such as Microsoft) have demonstrated the effectiveness of static code metrics to build predictive analytics. A commonly reported effect by a number of researchers like (Al Dallal and Briand, 2010; Shatnawi and Li, 2008; Madeyski and Jureczko, 2015; Chidamber et al., 1998; Menzies et al., 2007c; Alves et al., 2010; Bener et al., 2015; Shatnawi, 2010; Oliveira et al., 2014) is that OO metrics show a strong correlation with fault proneness. A comprehensive list of research on the correlation between product metrics and fault proneness can be found in Table 1 of the survey by (Rathore and Kumar, 2019).

Some researchers have criticized the use of static code metrics to learn defect predictors. For instance, (Graves et al., 2000) critiqued their effectiveness due to the fact that many metrics are highly correlated with each other, while (Rahman and Devanbu, 2013) claim that static code metrics may not evolve with the changing distribution of defects, which leads code-metric-based prediction models becoming stagnated. However, on close inspection of both these studies, we noted that some of the most informative static code metrics have not been accounted for. For example, in

Dataset	Versions	N	Bugs (%)	Description
Lucene	2.2 – 2.4	782	438 (56.01)	Information retrieval software library
Ant	1.3 – 1.7	1692	350 (20.69)	A software tool for automating software build processes
Ivy	1.1, 1.4,2.0	704	119 (16.90)	A transitive package manager
Jedit	4.0 – 4.3	1749	303 (17.32)	A free software text editor
Poi	1.5, 2, 2.5, 3.0	1378	707 (51.31)	Java libraries for manipulating files in MS Office format.
Camel	1.0, 1.2, 1.4,1.6	2784	562 (20.19)	A framework for message-oriented middleware.
Log4j	1.0, 1.1,1.2	449	260 (57.91)	A Java-based logging utility.
Velocity	1.4, 1.5,1.6	639	367 (57.43)	A template engine to reference objects in Java.
Xalan	2.4, 2.5, 2.6,2.7	3320	1806 (54.40)	A Java implementation of XLST, XML, and XPath.
Xerces	1.0, 1.2, 1.3,1.4	1643	654 (39.81)	Software libraries for manipulating XML.

Fig. 2: The figure lists defect datasets used in this paper.

the case of (Graves et al., 2000), they only inspect the McCabe and Halstead metrics and not object oriented metrics. In the case of (Rahman and Devanbu, 2013), (a) 37 out of 54 static code metrics (over $\frac{2}{3}$) are file-level metrics, most of which are not related to OO design, and (b) many of the metrics are repeated variants of the same measure (e.g., $CountLineCode$, $RatioCommentToCode$, $CountLineBlank$, etc are all measure of lines of code in various forms).

Given this evidence that static code metrics relate to defects, we use these metrics for our study. The defect dataset used in the rest of this this paper comprises a total of 38 datasets from 10 different projects taken from previous transfer learning studies. This group of data was gathered by Jureczko et al. (Jureczko and Madeyski, 2010). They recorded 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 extended CK metrics, McCabes and complexity metrics, see Table 1 for description. We obtained the dataset from the SEACRAFT repository[2] (formerly the PROMISE repository (Menzies et al., 2016)).

4 What is Planning?

We distinguish planning from prediction for software quality as follows: Quality prediction points to the likelihood of defects. Predictors take the form:

$$out = f(in)$$

where *in* contains many independent features (such as OO metrics) and *out* contains some measure of how many defects are present. For software analytics, the function f is learned via mining static code attributes.

On the other hand, quality planning seeks precautionary measures to significantly reduce the likelihood of future defects.

For a formal definition of plans, consider a defective test example Z, a planner proposes a plan "Δ" to adjust attribute Z_j as follows:

$$\forall \delta_j \in \Delta : Z_j = \begin{cases} Z_j \pm \delta_j & \text{if } Z_j \text{ is numeric} \\ \delta_j & \text{otherwise} \end{cases}$$

[2] https://zenodo.org/communities/seacraft/

DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
·	·	·	+	·	+	+	+	+

(a) Recommendations from some planner. The terms highlighted in the first row come from Figure 1. In the second row, a '+' represents an *increase*; a '−' represents an *decrease*; and a '·' represents *no-change*.

Action	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Class			+	−	+	−	−	−	−
Extract Method				+		+	+	+	+
Hide Method									
Inline Method				−		−	−	−	−
Inline Temp								−	
Remove Setting Method				−		−	−	−	−
Replace Assignment								−	
Replace Magic Number								+	
Consolidate Conditional				+		+	+	−	+
Reverse Conditional									
Encapsulate Field						+	+	+	+
Inline Class			−	+	−	+	+	+	+

(b) A sample of possible actions developers can take. Here a '+' represents an *increase*, a '−' represents a *decrease*, and an empty cell represents *no-change*. Taken from Stroggylos and Spinellis (2007); Du Bois (2006); Kataoka et al. (2002); Bryton and e Abreu (2009); Elish and Alshayeb (2011, 2012). The action highlighted in gray shows an action matching XTREE's recommendation from Figure 3.A.

```
class StoreManagement{
  // ... Some Opertaions
  private static void showMenu() { ... }
  public static void showActoions() {
    int choice;
    do {
      showMenu();
      switch choice {
      case 1: displayAllInventoryItems();
      case 2: findCompanyName();
      case 3: modifyPrice();
      case 4: saveToDisk();
              update();
              System.out.println();
              System.exit(0);
      }
    } while(choice != 4)
  }
  // ... Other Methods
}
```

(c) Before 'extract method'

```
class StoreManagement{
  // ... Some Opertaions
  private static void showMenu() { ... }
  public static void showActoions() {
    int choice;
    do {
      showMenu();
      switch choice {
      case 1: displayAllInventoryItems();
      case 2: findCompanyName();
      case 3: modifyPrice();
      case 4: exit();
      }
    } while(choice <= 4)
  }
  public static void exit() {
          saveToDisk();
          update();
          System.out.println();
          System.exit(0);
  }
  // ... Other Methods
}
```

(d) After 'extract method'

Fig. 3: An example of how developers might use XTREE to reduce software defects.

The above plans are described in terms of a range of numeric values. In this case, they represent an increase (or decrease) in some of the static code metrics of Table 1. However, these numeric ranges in and of themselves may not very informative. It would be beneficial to offer a more detailed report on how to go about implementing these plans. For example, to (say) simplify a large bug-prone method, it may be useful to suggest to a developer to reduce its size (e.g., by splitting it across two simpler functions).

In order to operationalize such plans, developers need some guidance on what to change in order to achieve the desired effect. There are two places to look for that guidance:
1. In other projects;
2. In the current project.

As to the first approach (*using other projects*), several recent papers have discussed how code changes adjust static code metrics (Stroggylos and Spinellis, 2007; Du Bois, 2006; Kataoka et al., 2002; Bryton and e Abreu, 2009; Elish and Alshayeb, 2011, 2012). For example, Fig. 3(b) shows a summary of that research. We could apply those results as follows:
- Suppose a planner has recommended the changes shown in Fig. 3(a).
- Then, we use 3(b) to look-up possible actions developers may take. Here, we see that performing an "extract method" operation may help alleviate certain defects (this is highlighted in gray).
- In 3(c) we show a simple example of a class where the above operation may be performed.
- In 3(d), we demonstrate how a developer may perform the "extract method".

While using other projects may be useful, that approach has a problem. Specifically: what happens if the proposed change has not been studied before in the literature? For this reason, we prefer to use the second approach (i.e. *use the current project*). In that approach, we look through the developer's own history to find old examples where they have made the kinds of changes recommended by the plan. Other researchers also adopt this approach (see (Nayrolles and Hamou-Lhadj, 2018) at MSR 2018). In the following:
- Using frequent itemset mining, we summarize prior changes in the current project (for details on this kind of learning, see Fig. 4.C).
- Next, when we learn plans, we reject any that are not known prior changes.

In this way, we can ensure that if a developer asks "how do I implement this plan?", we can reply with a relevant example of prior changes to the current project.

4.1 Planning in Software Engineering

We say that Fig. 3 is an example of *code-based planning* where the goal is to change a code base in order to improve that code in some way. The rest of this section first discusses other kinds of planning before discussing *code based planning* in greater detail.

Planning is extensively explored in artificial intelligence research. There, it usually refers to generating a sequence of actions that enables an *agent* to achieve a specific *goal* (Russell and Norvig, 1995). 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 (Ghallab et al., 2004). Some of the most common planning paradigms include: (a) classical planning (Wooldridge and Jennings, 1995); (b) probabilistic planning (Bellman, 1957; Altman, 1999; Guo and Hernández-Lerma, 2009); and (c) preference-based planning (Son and Pontelli, 2006; Baier and McIlraith, 2009). 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.

We know of at least two two kinds of planning research in software engineering. Each kind is distinguishable by *what* is being changed.

- In *test-based planning*, some optimization is applied to reduce the number of tests required to achieve to a certain goal or the time taken before tests yield interesting results (Tallam and Gupta, 2006; Yoo and Harman, 2012; Blue et al., 2013).
- In *process-based planning* some search-based optimizer is applied to a software process model to infer high-level business plans about software projects. Examples of that kind of work include our own prior studies sarching over COCOMO models Menzies et al. (2007b, 2009) or Ruhe et al.'s work on next release planning in requirements engineering (Ruhe and Greer, 2003; Ruhe, 2010).

In software engineering, the planning problem translates to proposing changes to software artifacts. These are usually a hybrid task combining probabilistic planning and preference-based planning using search-based software engineering techniques (Harman et al., 2009, 2011). These search-based techniques are evolutionary algorithms that propose actions guided by a fitness function derived from a well established domain model. Examples of algorithms used here include GALE, NSGA-II, NSGA-III, SPEA2, IBEA, MOEA/D, etc. (Krall et al., 2015; Deb et al., 2002; Zitzler et al., 2002; Zitzler and Künzli, 2004; Deb and Jain, 2014; Cui et al., 2005; Zhang and Li, 2007). As with traditional planning, these planning tools all require access to some trustworthy models that can be used to explore some highly novel examples. 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 (Weimer et al., 2009; Le Goues et al., 2012, 2015), software product line management (Sayyad et al., 2013; Metzger and Pohl, 2014; Henard et al., 2015), automated test generation (Andrews et al., 2007, 2010), etc.

However, not all domains come with ready-to-use models. For example, consider 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. Worse yet, our experience has been that accessing and/or commissioning a model can be a labor-intensive process. For example, in previous work (Menzies et al., 2007b) we used models developed by Boehm's group at the University of Southern California.Those models took as inputs project descriptors to output predictions of development effort, project risk, and defects. Some of those models took decades to develop and mature (from 1981 (Boehm, 1981) to 2000 (Boehm et al., 2000)). Lastly, even when there is an existing model, they can require constant maintenance lest they become outdated. Elsewhere, we have described our extensions to the USC models to enable reasoning about agile software developments. It took many months to implement and certify those extensions (Ii et al., 2009; Lemon et al., 2009). The problem of model maintenance is another motivation to look for alternate methods that can be quickly and automatically updated whenever new data becomes available.

In summary, for domains with readily accessible models, we recommend the kinds of tools that are widely used in the search-based software engineering community such as GALE, NSGA-II, NSGA-III, SPEA2, IBEA, particle swarm optimization, MOEA/D, etc. In other cases where this is not an option, we propose the use of data mining approaches to create a quasi-model of the domain and make use of observable states from this data to generate an estimation of the model. Examples of such a data mining approaches are described below. These include five methods described in the rest of this paper:

– Our approaches: XTREE, BELLTREE, and
– Three other approaches: Alves et al. (Alves et al., 2010), Shatnawi (Shatnawi, 2010), and Oliveira et al. (Oliveira et al., 2014)

4.2 Code based Planning

Looking through the SE literature, we can see that researchers have proposed three methods that rely on *outlier statistics* to identify suitable changes to source code metrics. The general principle underlying each of these methods is that any metric has an *unusually* large (or small) value needs to be change so as not to have such large (or small) values. The key distinction between the methods is how they determine what the threshold for this unusually large (or small) value ought to be. These methods, proposed by Alves et al. (Alves et al., 2010), Shatnawi (Shatnawi, 2010), and Oliveira et al. (Oliveira et al., 2014), are described in detail below.

4.2.1 Alves

Alves et al. (Alves et al., 2010) proposed an unsupervised approach that uses the underlying statistical distribution and scale of the OO metrics. It works by first weighting each metric value according to the source lines of code (SLOC) of the class it belongs to. All the weighted metrics are then normalized by the sum of all weights for the system. The normalized metric values are ordered in an ascending fashion (this is equivalent a density function, where the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale).

Alves et al. then select a percentage value (they suggest 70%) which represents the "normal" values for metrics. The metric threshold, then, is the metric value for which 70% of the classes fall below. The intuition is that the worst code has outliers beyond 70% of the normal code measurements i.e., they state that the risk of there existing a defect is moderate to high when the threshold value of 70% is exceeded.

Here, we explore the correlation between the code metrics and the defect counts with a univariate logistic regression and reject code metrics that are poor predictors of defects (i.e. those with $p > 0.05$). For the remaining metrics, we obtain the threshold ranges which are denoted by $[0, 70\%)$ ranges for each metric. The plans would then involve reducing these metric range to lie within the thresholds discovered above.

4.2.2 Shatnawi

Shatnawi (Shatnawi, 2010) offers a different alternative Alves et al by using VARL (Value of Acceptable Risk Level). This method was initially proposed by Bender (Bender, 1999) for his epidemiology studies. This approach uses two constants (p_0 and p_1) to compute the thresholds, which Shatnawi recommends to be set to $p_0 = p_1 = 0.05$. Then using a univariate binary logistic regression three coefficients are learned: α the intercept constant; β the coefficient for maximizing log-likelihood; and p_0 to measure

how well this model predicts for defects. (Note: the univariate logistic regression was conducted comparing metrics to defect counts. Any code metric with $p > 0.05$ is ignored as being a poor defect predictor.)

Thresholds are learned from the surviving metrics using the risk equation proposed by Bender:

$$\textit{Defective if Metric} > \textit{VARL}$$

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

In a similar fashion to Alves et al., we deduce the threshold ranges as $[0, VARL)$ for each selected metric. The plans would again involve reducing these metric range to lie within the thresholds discovered above.

4.2.3 Oliveira

Oliveira et al. in their 2014 paper offer yet another alternative to absolute threshold methods discussed above (Oliveira et al., 2014). Their method is still unsupervised, but they propose complementing the threshold by a second piece of information called the *relative threshold*. This measure denotes the percentage of entities the upper limit should be applied to. These have the following format:

$$p\% \textit{ of the entities must have } M \leq k$$

Here, M is an OO metric, k is the upper limit of the metric value, and p (expressed as %) is the minimum percentage of entities are required to follow this upper limit. As an example Oliveira et al. state, "85% of the methods should have $CC \leq 14$. Essentially, this threshold expresses that high-risk methods may impact the quality of a system when they represent more than 15% of the whole population"

The procedure attempts derive these values of (p, k) for each metric M. They define a function `ComplianceRate(p, k)` that returns the percentage of system that follows the rule defined by the relative threshold pair (p, k). They then define two penalty functions: (1) `penalty1(p, k)` that penalizes if the compliance rate is less than a constant $Min\%$, and (2) `penalty2(k)` to define the distance between k and the median of preset $Tail$-th percentile. (Note: according to Oliveira et al., median of the tail is an idealized upper value for the metric, i.e., a value representing classes that, although present in most systems, have very high values of M). They then compute the total penalty as `penalty` = `penalty1(p, k)` + `penalty2(k)`. Finally, the relative threshold is identified as the pair of values (p, k) that has the lowest total `penalty`. After obtaining the (p, k) for each OO metric. As in the above two methods, the plan would involve ensuring the for every metric M $p\%$ of the entities have a value that lies between $(0, k]$.

5 Supervised Planning with XTREE and BELLTREE

The rest of this paper comparatively evaluates:

- The value of the changes proposed by the above methods (from Alves, Shatnawi,Oliviera et al.);
- Against the changes proposed by the XTREE/BELLTREE method described below.

Fig. 4.A: To determine which of metrics are usually changed together, we use frequent itemset mining. Our dataset is continuous in nature (see (a)) so we first discretize using Fayyad-Irani (Fayyad and Irani, 1993); this gives us a representation shown in (b). Next, we convert these into "transactions" where each file contains a list of discretized OO-metrics (see (c)). Then we use the *FP-growth* algorithm to mine frequent itemsets. We return the *maximal frequent itemset* (as in (d)). Note: in (d) the row in green is the maximal frequent itemset.

	rfc	loc	dit	cbo	Bugs
1.java	0.6	100	1	4	0
2.java	0.9	223	4	5	1
3.java	1.1	290	5	7	1
4.java	2.1	700	10	12	3
5.java	2.3	800	11	15	3

(a)

	rfc	loc	dit	cbo
1.java	A	A	A	A
2.java	A	A	B	A
3.java	A	A	B	A
4.java	B	B	C	B
5.java	B	B	C	B

(b)

	Items
1.java	rfc_A, loc_A, dit_A, cbo_A
2.java	rfc_A, loc_A, dit_B, cbo_A
3.java	rfc_A, loc_A, dit_B, cbo_A
4.java	rfc_B, loc_B, dit_C, cbo_B
5.java	rfc_B, loc_B, dit_C, cbo_B

(c)

Items (`min_sup=60`)	Support
rfc_A	60
loc_A	60
~~dit_A~~	~~40~~
$\{rfc_A, loc_A\}$, $\{loc_A, cbo_A\}$, ...	60
$\{rfc_A, loc_A, cbo_A\}$	60
~~$\{rfc_A, loc_A, cbo_A, dit_{B,C}\}$~~	~~40~~

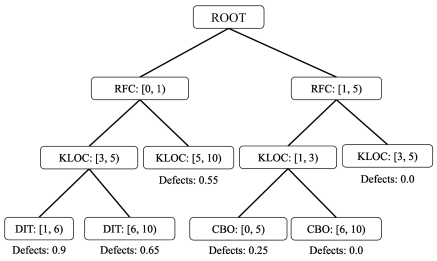
(d)

Fig. 4.B: To build the decision tree, we find the most informative feature,i.e., the feature which has the lowest mean entropy of splits and construct a decision tree recursively in a top-down fashion as show below.

Algorithm 1 N-ary Decision Tree

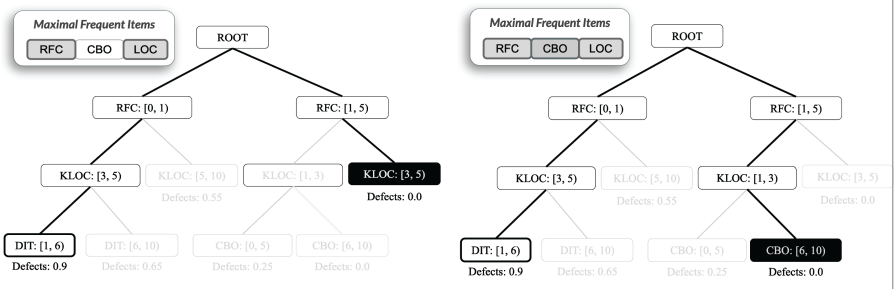
procedure NARY_DTREE(train)
  features = train[train.columns[:-1]]
  for $f \in features$ **do**
    Split using Fayyad-Irani method
    Compute entropy of splits
  end for
  f_{best} ← Feature with least entropy
  Tree ← Tree.`add_node`(f_best)
  D_v ← Induced sub-datasets from train based on f_{best}
  for $d \in D_v$ **do**
    Tree_v ← NARY_DTREE(d)
    Tree ← Tree_v
  end for
return Tree
end procedure

(a) Decision Tree Algorithm



(b) Example decision tree

Fig. 4.C: For ever test instance, we pass it down the decision tree constructed in Fig. 4.B. The node it lands is called the "start". Next we find all the "end" nodes in the tree, i.e., those which have the lowest likelihood of defects (labeled in black below). Finally, perform a random-walk to get from "start" to "end". We use the mined itemsets from Fig. 4.A to guide the walk. When presented with multiple paths, we pick the one which has the largest overlap with the frequent items. e.g., in the below example, we would pick path (b) over path (a).



(a) (b)

Fig. 4: XTREE Framework

5.1 Within-Project Planning With XTREE

Planning with XTREE is comprised of three steps namely, (a) Frequent pattern mining; (b) Decision tree construction; and (c) Planning with random walk traversal.
Step-1: Frequent pattern mining. The first step in XTREE is to determine which metrics are most often changed together. The OO metrics tabulated in Table 1 are not independent of each other. In other words, changing one metric (say LOC) would lead to a corresponding change in other metrics (such as CBO). We refrain from using correlation to determine which metrics change together because correlation measures the existence of a monotonic relationships between two metrics. We cannot assume that the metrics are monotonically related; moreover, it is possible that more than two metrics are related to each other. Therefore, we use frequent pattern mining (Han et al., 2007), which represents a more generalized relationship between metrics, to detect which of the metrics change together.

Our instrumentation is shown in Fig. 4.A. We use the FP-Growth algorithm (Han et al., 2007) to identify the *maximal frequent itemset* (highlighted in green in Fig. 4.A-(d)). This represents the longest set of metrics that change together atleast *support*% (in our case 60%) of the time. The following steps use the *maximal frequent itemset* to guide the generation of plans.
Step-2: Decision tree construction. Having discovered which metrics change together, we next establish what range of values for each metrics point to a high likelihood of defects. For this we use a decision tree algorithm (see Fig. 4.B). Specifically, we do the following:
1. Each of the OO metrics (from Table 1) are discretized into a range of values with the help of Fayyad-Irani discretizer (Fayyad and Irani, 1993).
2. We sort the OO metrics from the most discriminative to the least discriminative.
3. We begin by constructing a tree with the most discriminative OO metric, e.g., in Fig. 4.B (b) this would be *rfc*.
4. Then, we repeat the above to steps on the remaing OO metrics.
5. When we reach a predetermined termination criteria of having *less than* $\sqrt{N}$ *samples in subsequent splits*, we do not recurse futher. Here, N is the number of OO metrics, i.e., $N = 20$.
6. Finally, we return the constructed decision tree.

The leaf nodes of the decision tree contain instances of the training data that are most alike. The mean defect count of these instances represents the defect probability. In the case of Fig. 4.B (b), if $rfc = [0, 1)$, $KLOC = [3, 5)$, and $DIT = [1, 6)$ then the probability of defect is 0.9.
Step-3: Random Walk Traversal. With the last two steps, we now know (1) which metrics change together and (2) what ranges of metrics indicate a high likelihood of defects. with this information, XTREE builds plans from the branches of the tree as follows. Given a "defective" test instance, we ask:
1. Which *current* node does the test instance fall into?
2. What are all the *desired* nodes the test case would want to emulate? These would be nodes with the *lowest* defect probabilities.

Finally, we implement a random-walk (Ying et al., 2018; Sharma et al., 2016) model to find paths that lead from the *current* node the *desired* node. Of all the paths that lead from the *current* node to the *desired* node, we select the path that has the highest overlap with the *maximal frequent itemset*. As an example, consider Fig. 4.C.

Here, of the two possible paths Fig. 4.C(a) and Fig. 4.C(b), we choose that latter because it traverses through all the metrics in the maximal frequent itemset.

How are plans generated?

The path taken by the random-walk is used to generate a plan. For example, in the case of Fig. 4.C, it works as follows:

1. The test case finds itself on the far left, i.e., the "current node" has: $RFC : [0, 1)$, $KLOC : [3, 5)$ and $DIT : [1, 6)$
2. After implementing the random walk, we find that "desired" node is on the far right (highlighed in black)
3. The path taken to get from the "current node" to the "desired node" would require that the following changes be made.
   - $RFC : [0, 1) \longrightarrow [1, 5)$;
   - $KLOC : [0, 1) \longrightarrow [1, 3)$; and
   - $CBO : [6, 10)$

   The plan would then be these ranges of values.

5.2 Cross-project Planning with BELLTREE

Many methods have been proposed for transferring data or lessons learned from one project to another, for examples see (Nam et al., 2013b; Nam and Kim, 2015b; Jing et al., 2015; Kocaguneli and Menzies, 2011; Kocaguneli et al., 2015; Turhan et al., 2009; Peters et al., 2015). Of all these, the bellwether method described here is one of the simplest. Transfer learning with bellwethers is just a matter of calling existing learners inside a for-loop. For all the training data from different projects $\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{S}...$, a bellwether learner conducts a round-robin experiment where a model is learned from project, then applied to all others. The *bellwether* is that project which generates the best performing model. The *bellwether effect*, states that models learned from this bellwether performs as well as, or better than, other transfer learning algorithms.

For the purposes of prediction, we have shown previously that bellwethers are remarkably effective for many different kinds of SE tasks such as (i) defect prediction, (ii) effort estimation, and (iii) detecting code smells (Krishna and Menzies, 2018). This paper is the first to check the value of bellwethers for the purposes of planning. Note also that this paper's use of bellwethers enables us to generate plans from different data sets from across different projects. This represents a novel and significant extension to our previous work (Krishna et al., 2017a) which was limited to the use of datasets from within a few projects.

BELLTREE extends the three bellwether operators defined in our previous work (Krishna and Menzies, 2018) on bellwethers: DISCOVER, PLAN, VALIDATE. That is:

1. DISCOVER: *Check if a community has bellwether.* This step is similar to our previous technique used to discover bellwethers (Krishna et al., 2016). We see if standard data miners can predict for the number of defects, given the static code attributes. This is done as follows:
   - For a community C obtain all pairs of data from projects $\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{S}...$ such that $x, y \in C$;
   - Predict for defects in y using a quality predictor learned from data taken from x;

- Report a bellwether if one x generates consistently high predictions in a majority of $y \in C$.

Note, since the above steps perform an all-pairs comparison, the theoritical complexity of the DISCOVER phase will be be $O(N^2)$ where N is the number of projects.

2. PLAN: *Using the bellwether, we generate plans that can improve a new project.* That is, having learned the bellwether on past data, we now construct a decision tree similar to within-project XTREE. We then use the same methodology to generate the plans.
3. VALIDATE: *Go back to step 1* if the performance statistics seen during PLAN fail to generate useful actions.

6 Methods

The following experiment compare XTREE and BELLTREE against Alves, Shatnawi, Oliveira et al.

6.1 A Strategy for Evaluating Planners

It can be somewhat difficult to judge the effects of applying plans to software projects. These plans cannot be assessed just by a rerun of the test suite for three reasons: (1) The defects were recorded by a post release bug tracking system. It is entirely possible it escaped detection by the existing test suite; (2) Rewriting test cases to enable coverage of all possible scenarios presents a significant challenge; and (3) It may take a significant amount of effort to write new test cases that identify these changes as they are made.

To resolve this problem, SE researchers such as Cheng et al. (Cheng and Jensen, 2010), O'Keefe et al. (O'Keeffe and Cinnéide, 2008; O'Keeffe and Cinneide, 2007), Moghadam (Moghadam, 2011) and Mkaouer et al. (Mkaouer et al., 2014) use a *verification oracle* learned separately from the primary oracle. This oracles assesses how defective the code is before and after some code changes. For their oracle, Cheng, O'Keefe, Moghadam and Mkaouer et al. use the QMOOD quality model (Bansiya and Davis, 2002). A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects (Menzies et al., 2013). As a results, we eschew using these methods in favor of evaluation strategies discussed in the rest of this section.

6.1.1 The $\mathbb{K}$-test

This section offers details on the evaluation method introduced at the end of § 1.2.

In order to measure the extent to which the recommendations made by planning tools matches those undertaken by the developers, we assess the impact making those changes would have on an upcoming release of a project. For this purpose, we propose the $\mathbb{K}$-test.

We say that a project $\mathcal{P}$ is released in versions $\mathcal{V} \in \{\mathcal{V}_i, \mathcal{V}_j, \mathcal{V}_k\}$. Here, in terms of release dates, $\mathcal{V}_i$ precedes $\mathcal{V}_j$, which in turn precedes $\mathcal{V}_k$. We will use these three sets for *train*, *test*, and *validation*, respectively[3]. These three sets are used as follows:

[3] And recall in § 1.2 these versions were given less formal names, specifically *older, newer, latest*.

	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Version $\mathcal{V}_k$	3	4	4	2	5	2.5	3	400	6
$\mathcal{P} \rightarrow \mathcal{V}_{k+1}$	·	·	·	[4, 7]	·	[3, 6]	[4, 7]	[1000, 2000]	[1, 4]
$\mathcal{D} \rightarrow \mathcal{V}_{k+1}$	3	4	3	5	3	5	4	1500	2

$$Overlap = \frac{|\mathcal{D} \cap \mathcal{P}|}{|\mathcal{D} \cup \mathcal{P}|} \times 100 = \frac{7}{9} \times 100 = 77.77\%$$

Fig. 5: A simple example of computing overlap. Here a '·' represents *no-change*. Columns shaded in gray indicate a match between developer's changes and planner's recommendations.

1. First, train the planner on version $\mathcal{V}_i$. Note: this could either be data that is either from a previous release, or it could be data from the bellwether project.
2. Next, use the planner to generate plans to reduce defects for files that were reported to be buggy in version $\mathcal{V}_j$.
3. Finally, on version $\mathcal{V}_k$, for *only* the files that were reported to be buggy in the previous release, we measure the OO metrics.

Having obtained the changes at version $\mathcal{V}_k$ we can now (a) measure the *overlap* between plans recommended by the planner and the developer's actions, and (b) count the number of defects reduced (or possibly increased) when compared to the previous release. Using these two measures, we can assess the impact of implementing these plans. Details on measuring each of these are discussed in the subsequent parts of this section.

To compute that overlap, we proceeded as follows. Consider two sets of changes:

1. $\mathcal{D}$: The changes that developers made, perhaps in response to the issues raised in a post-release issue tracking system;
2. $\mathcal{P}$: The plans recommended by an automated planning tool, *overlap* attempts to compute the extent to which a developer's action matches that of the actions recommended by planners.

To measure this *overlap*, we use Jaccard similarity:

$$Overlap = \frac{|\mathcal{D} \cap \mathcal{P}|}{|\mathcal{D} \cup \mathcal{P}|} \times 100 \quad (1)$$

In other words, we measure the ratio of the size of the intersection between the developers plans and the size of all possible *changes*. Note that the *larger* the intersection between the changes made by the developers to the changes recommended by the planner, then the *greater* the overlap.

An simple example of how overlap is computed is illustrated in Fig. 5. Here, we have 9 metrics and let's say a defective file version $\mathcal{V}_k$ has metric values corresponding to row labeled Version $\mathcal{V}_k$. The row labeled $\mathcal{P} \rightarrow \mathcal{V}_{k+1}$ contains set of treatments recommended by a planner $\mathcal{P}$ for version $\mathcal{V}_{k+1}$ (note that the recommendations are ranges of values rather than actual numbers). Finally, the row labeled $\mathcal{D} \rightarrow \mathcal{V}_{k+1}$ are the result of a developer taking certain steps to possibly reduce the defects in the file for version $\mathcal{V}_{k+1}$. We see that in two cases (CBO and FOUT) the developers actions led to changes in metrics that were not prescribed by the planner. But in 7 cases, the developers actions matched the changes prescribed by the planner. Computing overlap as per Equation 1, produces an overlap value of 77%.

6.2 Presentation of Results

Using the $\mathbb{K}$-test and overlap counts defined above, we can measure the overlap between the planners' recommendations and developers actions. With this, plot three kinds of charts to discuss our results:

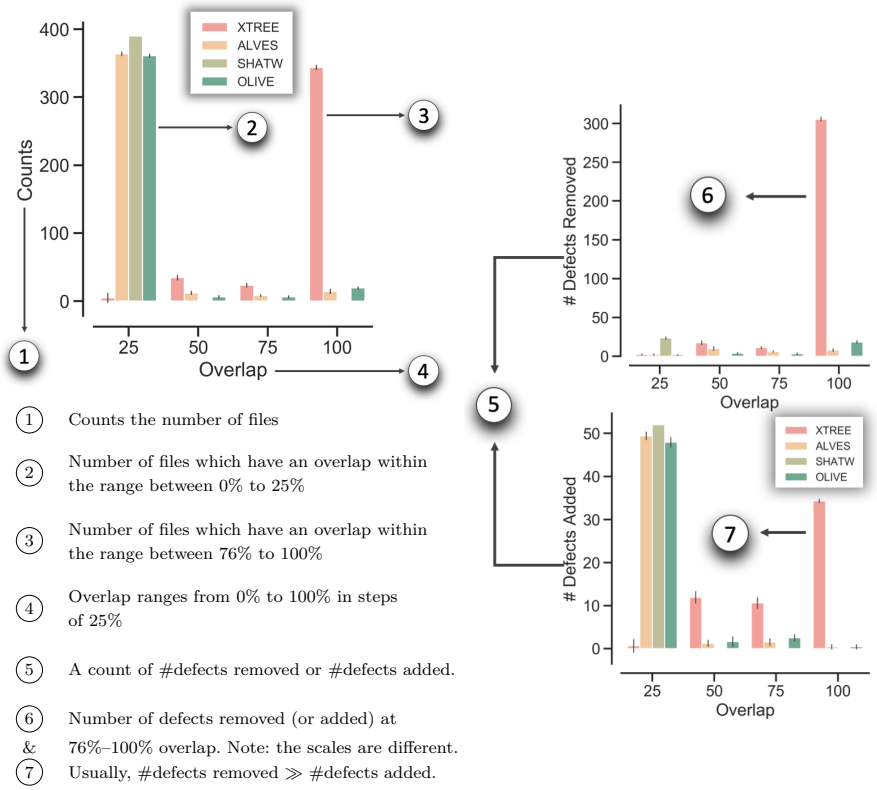


Fig. 6: Sample charts to illustrate the format used to present the results.

1. *Overlap vs. Counts*: A plot of overlap ranges (x-axis) versus the count of files that have that specific overlap range (on the y-axis). This is illustrated in Fig. 6. Here the overlap counts (x-axis) have 4 ticks: 0 (labeled 100). We see that, in the case of XTREE, the number of files that have between $76\% - 100\%$ overlap is significantly larger than any other overlap range. This implies that most of the changes recommended by *XTREE* are exactly what the developers would have actually done. On the other hand, for the other three planners (Alves, Shatnawi, and Oliveira) the number of files that have between $0\% - 25\%$ overlap is significantly larger than any other overlap range. This means that those planners' recommendation are seldom what developers actually do.
2. *Overlap vs. Defects reduced*: Just because there is an overlap, it does not necessarily mean that the defects were actually reduced. To measure what impact overlaps between planners' recommendations and developers actions have on reduction of defects, we plot a chart of overlap (x-axis) against the actual number of defects reduced. This is illustrated in Fig. 6. The key distinction between this chart and the previous chart is the y-axis, here the y-axis represents the number of defects reduced. Larger y-axis values for larger overlaps are desirable because this means that more the developers follow a planners' actions, higher the number of defects reduced.

3. Overlap vs. Defects increased: It is also possible that defects are increased as a result of overlap. To measure what impact overlaps between planners' recommendations and developers actions have on *increasing* defectiveness, we plot a chart of overlap (x-axis) against the actual number of defects increased. This is illustrated in Fig. 6. The key distinction between this chart and the previous two charts is the y-axis, here the y-axis represents the number of defects *increased*. Lower y-axis values for larger overlaps are desirable because this means that more the developers follow a planners' actions, lower the number of defects increased.

7 Experimental Results

All our experiments were conducted on a 6 core, 3.7 GHz, Intel i7-8700K running an Ubuntu 18.04 operating system.

RQ1: How well do planners recommendations match developer actions?

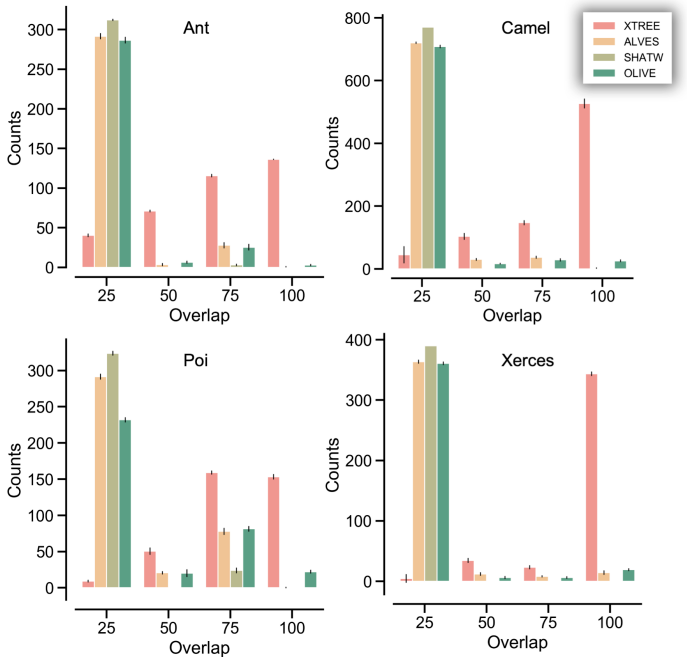


Fig. 7: A count of number of test instances where the developer changes overlaps a planner recommendation. The overlaps (in the x-axis) are categorized into four ranges for every dataset (these are $0 \leq\ Overlap \leq 25$, $26 \leq\ Overlap \leq 50$, $51 \leq\ Overlap \leq 75$, and $76 \leq\ Overlap \leq 100$). For each of the overlap ranges, we count the the number of instances in the validation set where overlap between the planner's recommendation and the developers changes fell in that range. Note: *Higher counts* for larger overlap is *better*, e.g., $Count([75, 100]) > Count([0, 25))$ is considered better.

To answer this question, we measure the *overlap* between the planners' recommendations and the developer's actions. To measure this, we split the available data into training, testing, and validation sets. That is, given versions $\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3$...., we,
1. *train* the planners on version $\mathcal{V}_1$; then

2. *generate plans* using the planners for version $\mathcal{V}_2$;
3. then *validate* the effectiveness of those plans on $\mathcal{V}_2$ using the $\mathbb{K}$-test.

Then, we repeat the process by training on $\mathcal{V}_2$, testing on $\mathcal{V}_3$, and validating on version $\mathcal{V}_4$, and so on. For each of these $\{train, test, validation\}$ sets, we measure the *overlap* and categorize them into 4 ranges:
- very little, i.e. $0 - 25\%$;
- some, i.e. $26\% - 50\%$;
- more, i.e. $51\% - 75\%$;
- mostly, i.e. $76\% - 100\%$.

Fig. 7 shows the results of planning with several planners: XTREE, Alves, Shatnawi, and Oliveira. Note, for the sake of brevity, we illustrate results for 4 projects– Ant, Camel, Poi, and Xerces. A full set set results for all projects are available at `https://git.io/fjkNM`.

We observe a clear dichotomy in our results.
- All outlier statistics based planners (i.e., those of Alves, Shatnawi, and Oliveira) have overlaps only in the range of 0% to 25%. This means that *most of the developers actions did not match the recommendations proposed by these planners.*
- In the case of XTREE, the largest number of files had an overlap of 76% to 100% and second largest was between 51% to 75%. This means that, in a majority of cases developers actions are 76% to 100% similar to XTREE's recommendations. At the very least, there was an 51% similarity between XTREE's recommendations and developers actions.

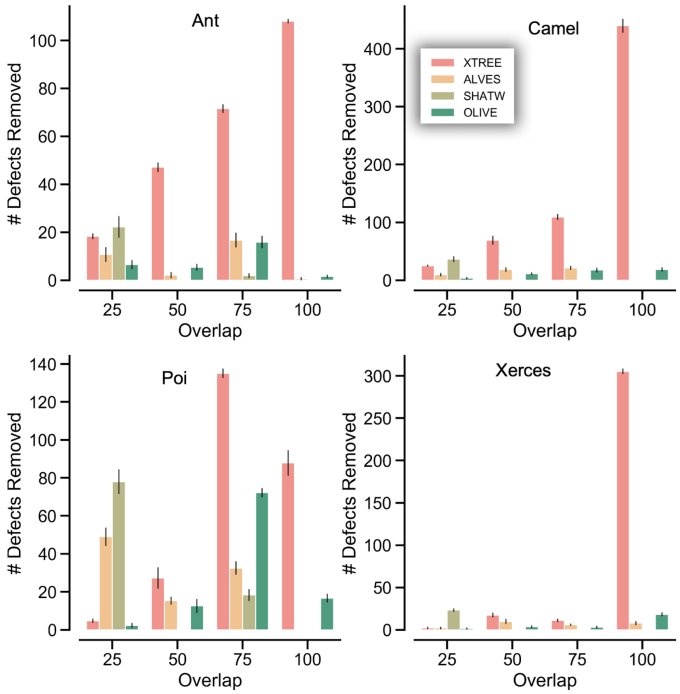
We observe this trend in all 18 datasets– XTREE significantly outperformed other threshold based planners in terms of the overlap between the plans and the actual actions undertaken by the developers. Note that reason the results are very negative about the methods of Alves, Shatnawi, Oliveira, et al. is because their recommendations would be very hard to operationalize (since those recommendations were seldom seen in the prior history of a project). Thus, our response to this research question can be summarized as follows:

Result: XTREE significantly outperforms all the other outlier statistics based planners. Further, in all the projects studied here, most of the developer actions to fix defects in a file has as 76%–100% overlap with the recommendations offered by XTREE.

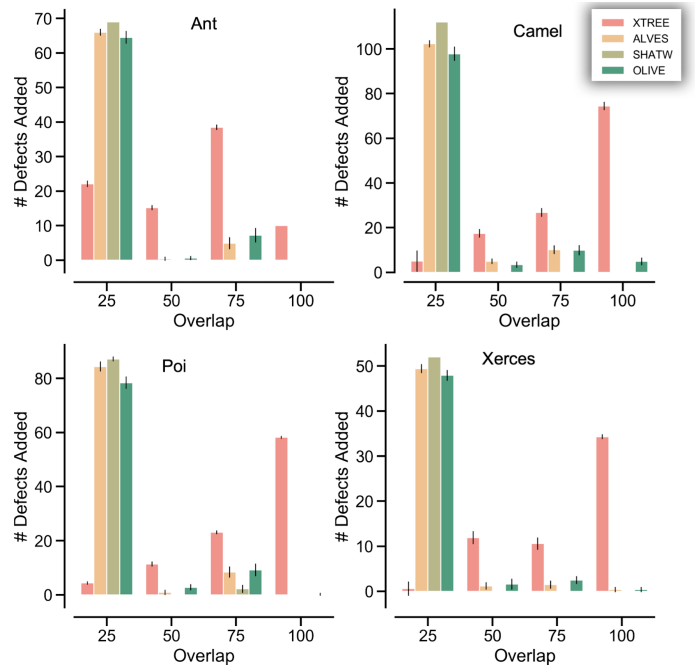
RQ2: Do planners' recommendation lead to reduction in defects?

In the previous research question measured the extent to which a planner's recommendations matched the actions taken by developers to fix defects in their files. But, the existence of a high overlap in most files does not necessarily mean that the defects are actually reduced. Likewise, it is also conceivable that that defects are added due to other actions the developer took during their development. Thus, it is important to ask how many defects are reduced, and how many are added, in response to larger overlap with the planners' recommendations.

Our experimental methodology to answer this research question is as follows:
- Like before, we measure the *overlap* between the planners' recommendations developers' actions.
- Next, we plot the aggregate number defects reduced and in file with overlap values ranging from 0% to 100% in bins of size 25% (for ranges of $0 - 25\%$, $26 - 50\%$, $51 - 75\%$, and $76 - 100\%$).



(a) Defects Reduced



(b) Defects Increased

Fig. 8: A count of total number *defects reduced* and *defects increased* as a result each planners' recommendations. The overlaps are again categorized into four ranges for every dataset (denoted by $min \leq Overlap < max$). For each of the overlap ranges, we count the total number of *defects reduced* and *defects increased* in the validation set for the classes that were defective in the test set as a result of overlap between the planner's recommendation and the developers changes that fell in the given range

Note that, we refrain from computing the correlation between *overlap* and defects increased/decreased because we were interested only in the cases with large overlaps, i.e., cases where overlap $> 75\%$. In these cases, we measure what impact the changes have on bug count. Correlation, we disovered, was ill-suited for this purpose because it does not distinguish between low-/high-overlaps it only measures the linearity of the relationship between overlap and defect count. For example, in an ideal case where every plan offered by XTREE is followed, the overlaps at 0% — 99% would be zero and so would the value of correlation, but would be most misleading.

Similar to RQ1, we compare XTREE with three other outlier statistics based planners of Alves et al., Shatnawi, and Oliveira, for the overall number of defects reduced and number of defects added. We prefer planners that have a large number defects reduced for higher overlap ranges are considered better.

Fig. 8 shows the results of planning with several planners: XTREE, Alves, Shatnawi, and Oliveira. Note that, similar to the previous research question, we only illustrate results for 4 projects– Ant, Camel, Poi, and Xerces. A full set of results for RQ2 for all projects are available at `https://git.io/fjIvG`.

We make the following observations from in our results:

1. *Defects Decreased*: Fig. 8(a) plots the number of defects *removed* in files with various overlap ranges. It is desirable to see larger defects removed with larger overlap. We note that:
   - When compared to other planners, the number of defects removed as a result of recommendations obtained by XTREE is significantly larger. This trend was noted in all the projects we studied here.
   - In the cases of Ant, Camel, and Xerces there are large number of defect reduced when the overlap lies between 76% and 100%. Poi is an exception– here, we note that the largest number of defects are removed when the overlap is between 51% and 75%.
2. *Defects Increased*: Fig. 8(b) plots the number of defects *added* in files with various overlap ranges. It is desirable to see lower number of defects added with larger overlap. We note that:
   - When compared to other planners, the number of defects added as a result of recommendations obtained by XTREE is comparatively larger. This trend was noted in all the projects we studied here. This is to be expected since, developers actions seldom match the recommendations of these other planners.
   - In all the cases the number of defects removed was significantly larger than the number of defects added. For example, in the case of Camel, 420+ defects were removed at 76% – 100% overlap and about 70 defects were added (i.e., 6× more defects were removed than added). Likewise, in the case of Xerces, over 300 defects were removed and only about 30 defects were added (i.e., 10× more defects were removed than added).

The ratio of defects removed to the number of defects added is very important to asses. Fig. 9 plots this ratio at 76% – 100% overlap (it applied equally for the other overlap ranges as they have far fewer defects removed and added). From this chart, we note that out of 18 datasets, in 14 cases XTREE lead to a significant reduction in defects. For example, in the case of Ivy and Log4j, there were no defects added at all.

However, in 4 cases, there were more defects added than there were removed. Given the idiosyncrasies of real world projects, we do not presume that developers will always take actions as suggested by a planner. This may lead to defects being increased,

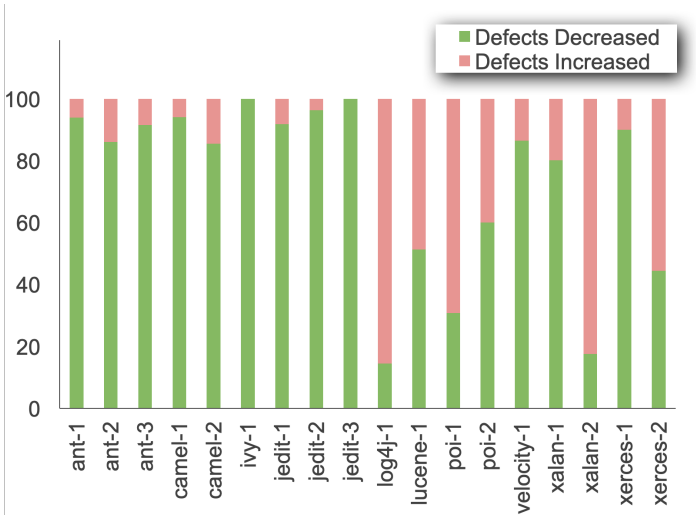


Fig. 9: A count of total number *defects reduced* and *defects increased* as a result each planners' recommendations. The overlaps are again categorized into four ranges for every dataset (denoted by $min \leq Overlap < max$). For each of the overlap ranges, we count the total number of *defects reduced* and *defects increased* in the validation set for the classes that were defective in the test set as a result of overlap between the planner's recommendation and the developers changes that fell in the given range

however, based on our results we notice that this is not a common occurrence. In summary, our response to this research question is as follows:

Result: Plans generated by XTREE are superior to other outlier statistics based planners in all 10 projects. Planning with XTREE leads to the far larger number of defects reduced as opposed to defects added in 9 out of 10 projects studied here.

RQ3: Are cross-project plans generated by BELLTREE as effective as within-project plans of XTREE?

In the previous two research questions, we made an assumption that there are past releases that can be used to construct the planners. However, this may not always be the case. For new project, it is quite possible that there are not any historical data to construct the planners. In such cases, SE literature proposes the use of *transfer learning*. In this paper, we leverage the so-called *bellwether* effect to identify a bellwether project. Having done so, we construct a planner quite similar to XTREE with the exception that the training data comes from the bellwether project. This variant of our planner that uses the bellwether project is called the BELLTREE (see § 5.2 for more details).

To answer this research question, we train XTREE on within-project data and generate plans for reducing the number of defects. We then compare this with plans derived from the bellwether data and BELLTREE. We hypothesized that since bellwethers have been demonstrated to be efficient in prediction tasks, learning from the bellwethers for a specific community of projects would produce performance scores comparable to within-project data. Our experimental methodology to answer this research question is as follows:

	[0, 25)			[25, 50)			[50, 75)			[75, 100]		
	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE
ant-1	13	13	12	3	33	19	0	27	10	0	62	54
ant-2	0	6	13	0	42	33	0	27	27	0	124	61
ant-3	22	18	6	1	71	42	0	47	27	0	108	124
camel-1	76	29	10	0	90	30	0	52	20	0	226	98
camel-2	36	25	30	0	109	100	0	69	68	0	439	277
ivy-1	1	4	12	0	10	42	0	5	13	0	12	25
jedit-1	13	9	11	8	35	44	0	39	50	0	136	108
jedit-2	28	24	10	1	77	34	0	36	39	0	107	135
jedit-3	18	30	28	1	67	75	0	28	35	0	70	106
log4j-1	5	1	0	0	7	14	0	3	8	0	8	50
poi-1	1	0	7	5	0	80	0	2	19	0	81	90
poi-2	78	4	0	18	135	0	0	27	2	0	87	83
velocity-1	51	2	6	0	25	15	0	39	32	0	90	48
xalan-1	22	6	2	105	43	51	13	60	66	0	409	230
xalan-2	110	0	6	0	38	49	0	102	54	0	83	408
xerces-1	23	2	11	0	11	13	0	17	24	0	305	49
xerces-2	7	0	2	0	3	11	0	6	18	0	117	305

(a) Defects Reduced. *Higher defect reduction* for larger Overlap is considered *better*.

	[0, 25)			[25, 50)			[50, 75)			[75, 100]		
	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE
ant-1	15	1	3	0	10	11	0	2	1	0	4	2
ant-2	63	9	1	0	33	10	0	11	2	0	20	4
ant-3	69	22	9	0	38	33	0	15	11	0	10	20
camel-1	36	10	5	0	11	25	0	6	14	0	14	31
camel-2	112	5	2	0	26	15	0	17	9	0	74	15
ivy-1	6	1	0	0	3	2	0	2	1	0	0	0
jedit-1	37	3	2	2	20	10	0	11	6	0	12	6
jedit-2	15	2	5	0	8	19	0	2	11	0	4	12
jedit-3	3	1	1	0	1	0	0	1	1	0	0	0
log4j-1	73	1	2	1	14	7	0	13	2	0	47	7
poi-1	190	1	1	6	7	1	0	5	0	0	182	6
poi-2	87	4	0	2	23	7	0	11	5	0	58	184
velocity-1	21	1	4	4	3	14	0	3	17	0	14	10
xalan-1	152	2	3	21	46	29	6	33	31	0	101	217
xalan-2	506	27	3	0	25	48	0	87	32	0	388	101
xerces-1	52	0	0	0	10	1	0	11	1	0	34	1
xerces-2	169	4	0	0	14	11	0	9	12	0	146	34

(b) Defects Increased. In comparison to defects reduced in Fig. 8(a) above, we would like to have as little defects increased as possible.

Fig. 10: A count of total number *defects reduced* and *defects increased* as a result each planners' recommendations. The overlaps are again categorized into four ranges for every dataset (denoted by $min \leq Overlap < max$). For each of the Overlap ranges, we count the total number of *defects reduced* and *defects increased* in the validation set for the classes that were defective in the test set as a result of Overlap between the planner's recommendation and the developers changes that fell in the given range

1. Like before, we measure the *overlap* between the planners' recommendations developers' actions.
2. Next, we tabulate the aggregate number defects reduced (Fig. 10(a)) and the number of defects increased (Fig. 10(b)) in files with overlap values ranging from 0% to 100% in bins of size 25% (for ranges of $0-25\%$, $26-50\%$, $51-75\%$, and $76-100\%$).

Similar to previous research questions, we compare XTREE with BELLTREE and a random oracle (RAND). We prefer planners that have a large number defects reduced for higher overlap ranges and planner that have lower number of defects added are are considered better.

We make the following observations from in our results:

1. *Defects Decreased*: Fig. 8(a) plots the number of defects *removed* in files with various overlap ranges. It is desirable to see larger defects removed with larger overlap. We note that:
   - When compared to other planners, the number of defects removed as a result of recommendations obtained by XTREE is significantly larger. This trend was noted in all the projects we studied here.
   - In the cases of Ant, Camel, and Xerces there are large number of defect reduced when the overlap lies between 76% and 100%. Poi is an exception– here, we note that the largest number of defects are removed when the overlap is between 51% and 75%.
2. *Defects Increased*: Fig. 8(b) plots the number of defects *added* in files with various overlap ranges. It is desirable to see lower number of defects added with larger overlap. We note that:
   - When compared to other planners, the number of defects added as a result of recommendations obtained by XTREE is comparatively larger. This trend was noted in all the projects we studied here. This is to be expected since, developers actions seldom match the recommendations of these other planners.
   - In all the cases the number of defects removed was significantly larger than the number of defects added. For example, in the case of Camel, 420+ defects were removed at 76% – 100% overlap and about 70 defects were added (i.e., 6× more defects were removed than added). Likewise, in the case of Xerces, over 300 defects were removed and only about 30 defects were added (i.e., 10× more defects were removed than added).

The ratio of defects removed to the number of defects added is very important to asses. Fig. 9 plots this ratio at 76% – 100% overlap (it applied equally for the other overlap ranges as they have far fewer defects removed and added). From this chart, we note that out of 18 datasets, in 14 cases XTREE lead to a significant reduction in defects. For example, in the case of Ivy and Log4j, there were no defects added at all.

However, in 4 cases, there were more defects added than there were removed. Given the idiosyncrasies of real world projects, we do not presume that developers will always take actions as suggested by a planner. This may lead to defects being increased, however, based on our results we notice that this is not a common occurrence.

In summary, our response to this research question is as follows:

Result: The effectiveness of BELLTREE and XTREE are similar. If within-project data is available, we recommend using XTREE. If not, BELLTREE is a viable alternative.

8 Discussion

When discussing these results with colleagues, we are often asked the following questions.

1. Why use automatic methods to find quality plans? Why not just use domain knowledge; e.g. human expert intuition? Recent research has documented the wide variety of conflicting opinions among software developers, even those working within the same project. According to Passos et al. (Passos et al., 2011), developers often assume that the lessons they learn from a few past projects are general to all their future projects. They comment, "past experiences were taken into account without much consideration for their context". Jorgensen and Gruschke (Jørgensen and Gruschke, 2009) offer a similar warning. They report that the supposed software engineering "gurus" rarely use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects (Jørgensen and Gruschke, 2009). Other studies have shown some widely-held views are now questionable given new evidence. Devanbu et al. examined responses from 564 Microsoft software developers from around the world. They comment programmer beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project (Devanbu et al., 2016). Given the diversity of opinions seen among humans, it seems wise to explore automatic oracles for planning.

2. Does using BELLTREE guarantee that software managers will never have to change their plans? No. Software managers should evolve their policies when the evolving circumstances require such an update. But how to know when to retain current policies or when to switch to new ones? Bellwether method can answer this question.

Specifically, we advocate continually retesting the bellwether's status against other data sets within the community. If a new bellwether is found, then it is time for the community to accept very different policies. Otherwise, it is valid for managers to ignore most the new data arriving into that community.

9 Threats to Validity

Sampling Bias: Sampling bias threatens any classification experiment; what matters in one case may or may not hold in another case. For example, data sets in this study come from several sources, but they were all supplied by individuals. Thus, we have documented our selection procedure for data and suggest that researchers try a broader range of data.

Evaluation Bias: This paper uses one measure for the quality of the planners and other quality measures may be used to quantify the effectiveness of planner. A comprehensive analysis using these measures may be performed with our replication package. Additionally, other measures can easily be added to extend this replication package.

Order Bias: Theoretically, with prediction tasks involving learners such as random forests, there is invariably some degree of randomness that is introduced by the algorithm. To mitigate these biases, researchers, including ourselves in our other work, report the central tendency and variations over those runs with some statistical test. However, in this case, all our approaches are *deterministic*. Hence, there is no need to repeat the experiments or run statistical tests. Thus, we conclude that while order

bias is theoretically a problem, it is not a major problem in the particular case of this study.

10 Conclusions and Future Work

Most software analytic tools that are currently in use today are mostly prediction algorithms. These algorithms are limited to making predictions. We extend this by offering "planning": a novel technology for prescriptive software analytics. Our planner offers users a guidance on what action to take in order to improve the quality of a software project. Our preferred planning tool is BELLTREE, which performs cross-project planning with encouraging results. With our BELLTREE planner, we show that it is possible to reduce several hundred defects in software projects.

It is also worth noting that BELLTREE is a novel extension of our prior work on (1) the bellwether effect, and (2) within-project planning with XTREE. In this work, we show that it is possible to use bellwether effect and within-project planning (with XTREE) to perform cross-project planning using BELLTREE, without the need for more complex transfer learners. Our results from Fig. 7 show that BELLTREE is just as good as XTREE, and both XTREE/BELLTREE are much better than other planners.

Hence our overall conclusion is to endorse the use of planners like XTREE (if local data is available) or BELLTREE (otherwise).

Acknowledgements

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

References

Al Dallal J, Briand LC (2010) An object-oriented high-level design-based class cohesion metric. Information and software technology 52(12):1346–1361

Altman E (1999) Constrained Markov decision processes, vol 7. CRC Press

Alves TL, Ypma C, Visser J (2010) Deriving metric thresholds from benchmark data. In: 2010 IEEE Int. Conf. Softw. Maint., IEEE, pp 1–10, DOI 10.1109/ICSM.2010.5609747

Andrews JH, Li FCH, Menzies T (2007) Nighthawk: A Two-Level Genetic-Random Unit Test Data Generator. In: IEEE ASE'07

Andrews JH, Menzies T, Li FCH (2010) Genetic Algorithms for Randomized Unit Testing. IEEE Transactions on Software Engineering

Baier SSJA, McIlraith SA (2009) Htn planning with preferences. In: 21st Int. Joint Conf. on Artificial Intelligence, pp 1790–1797

Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Trans Softw Eng 28(1):4–17, DOI 10.1109/32.979986

Begel A, Zimmermann T (2014) Analyze this! 145 questions for data scientists in software engineering. In: Proc. 36th Intl. Conf. Software Engineering (ICSE 2014), ACM

Bellman R (1957) A markovian decision process. Indiana Univ Math J 6:679–684

Bender R (1999) Quantitative risk assessment in epidemiological studies investigating threshold effects. Biometrical Journal 41(3):305–319

Bener A, Misirli AT, Caglayan B, Kocaguneli E, Calikli G (2015) Lessons learned from software analytics in practice. In: The Art and Science of Analyzing Software

Data, Elsevier, pp 453–489
Blue D, Segall I, Tzoref-Brill R, Zlotnick A (2013) Interaction-based test-suite minimization. In: Proc. the 2013 Intl. Conf. Software Engineering, IEEE Press, pp 182–191
Boehm B (1981) Software Engineering Economics. Prentice Hall
Boehm B, Horowitz E, Madachy R, Reifer D, Clark BK, Steece B, Brown AW, Chulani S, Abts C (2000) Software Cost Estimation with Cocomo II. Prentice Hall
Bryton S, e Abreu FB (2009) Strengthening refactoring: towards software evolution with quantitative and experimental grounds. In: Software Engineering Advances, 2009. ICSEA'09. Fourth Intl. Conf., IEEE, pp 570–575
Cheng B, Jensen A (2010) On the use of genetic programming for automated refactoring and the introduction of design patterns. In: Proc. 12th Annual Conf. Genetic and Evolutionary Computation, ACM, New York, NY, USA, GECCO '10, pp 1341–1348, DOI 10.1145/1830483.1830731
Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Transactions on software engineering 20(6):476–493
Chidamber SR, Darcy DP, Kemerer CF (1998) Managerial use of metrics for object-oriented software: An exploratory analysis. IEEE Transactions on software Engineering 24(8):629–639
Cortes C, Vapnik V (1995) Support-vector networks. Machine learning 20(3):273–297
Cui X, Potok T, Palathingal P (2005) Document clustering using particle swarm optimization. . . . Intelligence Symposium, 2005 . . .
Czerwonka J, Das R, Nagappan N, Tarvo A, Teterev A (2011) Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth Intl. Conference on, pp 357 –366
Deb K, Jain H (2014) An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. Evolutionary Computation, IEEE Transactions on 18(4):577–601, DOI 10.1109/TEVC.2013.2281535
Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6:182–197
Devanbu P, Zimmermann T, Bird C (2016) Belief & evidence in empirical software engineering. In: Proc. 38th Intl. Conf. Software Engineering, ACM, pp 108–119
Du Bois B (2006) A study of quality improvements by refactoring
Elish K, Alshayeb M (2011) A classification of refactoring methods based on software quality attributes. Arabian Journal for Science and Engineering 36(7):1253–1267
Elish K, Alshayeb M (2012) Using software quality attributes to classify refactoring to patterns. JSW 7(2):408–419
Fayyad U, Irani K (1993) Multi-interval discretization of continuous-valued attributes for classification learning. NASA JPL Archives
Fu W, Menzies T, Shen X (2016) Tuning for software analytics: is it really necessary? Information and Software Technology (submitted)
Ghallab M, Nau D, Traverso P (2004) Automated Planning: theory and practice. Elsevier
Ghotra B, McIntosh S, Hassan AE (2015) Revisiting the impact of classification techniques on the performance of defect prediction models. In: 37th ICSE-Volume 1, IEEE Press, pp 789–800

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Transactions on software engineering 26(7):653–661

Guo X, Hernández-Lerma O (2009) Continuous-time markov decision processes. Continuous-Time Markov Decision Processes pp 9–18

Halstead MH (1977) Elements of software science, vol 7. Elsevier New York

Han J, Cheng H, Xin D, Yan X (2007) Frequent pattern mining: current status and future directions. Data mining and knowledge discovery 15(1):55–86

Harman M, Mansouri SA, Zhang Y (2009) Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Department of Computer Science, KingâĂŹs College London, Tech Rep TR-09-03

Harman M, McMinn P, De Souza J, Yoo S (2011) Search based software engineering: Techniques, taxonomy, tutorial. Search 2012:1–59, DOI 10.1007/978-3-642-25231-0_1

Henard C, Papadakis M, Harman M, Traon YL (2015) Combining multi-objective search and constraint solving for configuring large software product lines. In: 2015 IEEE/ACM 37th IEEE Intl. Conf. Software Engineering, vol 1, pp 517–528, DOI 10.1109/ICSE.2015.69

Hihn J, Menzies T (2015) Data mining methods and cost estimation models: Why is it so hard to infuse new ideas? In: 2015 30th IEEE/ACM Intl. Conf. Automated Software Engineering Workshop (ASEW), pp 5–9, DOI 10.1109/ASEW.2015.27

Ii PG, Menzies T, Williams S, El-Rawas O (2009) Understanding the value of software engineering technologies. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, pp 52–61

Jing X, Wu G, Dong X, Qi F, Xu B (2015) Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In: FSE'15

Jørgensen M, Gruschke TM (2009) The impact of lessons-learned sessions on effort estimation and uncertainty assessments. Software Engineering, IEEE Transactions on 35(3):368 –383

Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proc. 6th Int. Conf. Predict. Model. Softw. Eng. - PROMISE '10, ACM Press, New York, New York, USA, p 1, DOI 10.1145/1868328.1868342

Kataoka Y, Imai T, Andou H, Fukaya T (2002) A quantitative evaluation of maintainability enhancement by refactoring. In: Software Maintenance, 2002. Proceedings. Intl. Conf., IEEE, pp 576–585

Kocaguneli E, Menzies T (2011) How to find relevant data for effort estimation? In: Empirical Software Engineering and Measurement (ESEM), 2011 Intl. Symposium on, IEEE, pp 255–264

Kocaguneli E, Menzies T, Bener A, Keung J (2012) Exploiting the essential assumptions of analogy-based effort estimation. IEEE Transactions on Software Engineering 28:425–438

Kocaguneli E, Menzies T, Mendes E (2015) Transfer learning in effort estimation. Empirical Software Engineering 20(3):813–843, DOI 10.1007/s10664-014-9300-5

Krall J, Menzies T, Davies M (2015) Gale: Geometric active learning for search-based software engineering. IEEE Transactions on Software Engineering 41(10):1001–1018

Krishna R (2017) Learning effective changes for software projects. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, pp 1002–1005

Krishna R, Menzies T (2015) Actionable = cluster + contrast? In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), pp 14–17, DOI 10.1109/ASEW.2015.23

Krishna R, Menzies T (2018) Bellwethers: A baseline method for transfer learning. IEEE Transactions on Software Engineering pp 1–1, DOI 10.1109/TSE.2018.2821670

Krishna R, Menzies T, Fu W (2016) Too much automation? the bellwether effect and its implications for transfer learning. In: Proc. 31st IEEE/ACM Intl. Conf. Automated Software Engineering - ASE 2016, ACM Press, New York, New York, USA, pp 122–131, DOI 10.1145/2970276.2970339

Krishna R, Menzies T, Layman L (2017a) Less is more: Minimizing code reorganization using XTREE. Information and Software Technology DOI 10.1016/j.infsof.2017.03.012, 1609.03614

Krishna R, Menzies T, Layman L (2017b) Less is more: Minimizing code reorganization using xtree. Information and Software Technology 88:53–66

Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: 2012 34th Intl. Conf. Software Engineering (ICSE), IEEE, pp 3–13, DOI 10.1109/ICSE.2012.6227211

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. IEEE Transactions on Software Engineering 41(12):1236–1256, DOI 10.1109/TSE.2015.2454513

Lemon B, Riesbeck A, Menzies T, Price J, D'Alessandro J, Carlsson R, Prifiti T, Peters F, Lu H, Port D (2009) Applications of Simulation and AI Search: Assessing the Relative Merits of Agile vs Traditional Software Development. In: IEEE ASE'09

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. IEEE Trans Softw Eng 34(4):485–496, DOI 10.1109/TSE.2008.35

Lewis C, Lin Z, Sadowski C, Zhu X, Ou R, Whitehead Jr EJ (2013) Does bug prediction support human developers? findings from a google case study. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 372–381, URL http://dl.acm.org/citation.cfm?id=2486788.2486838

Lowry M, Boyd M, Kulkami D (1998) Towards a theory for integration of mathematical verification and empirical testing. In: Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on, IEEE, pp 322–331

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

McCabe TJ (1976) A complexity measure. IEEE Transactions on software Engineering (4):308–320

Mensah S, Keung J, MacDonell SG, Bosu MF, Bennin KE (2018) Investigating the significance of the bellwether effect to improve software effort prediction: Further empirical study. IEEE Transactions on Reliability (99):1–23

Menzies T, Raffo D, on Setamanit S, Hu Y, Tootoonian S (2002) Model-based tests of truisms. In: Proceedings of IEEE ASE 2002, available from http://menzies.us/pdf/02truisms.pdf

Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007a) Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn

Defect Predictors'". IEEE Transactions on Software Engineering 33(9):637–640, DOI 10.1109/TSE.2007.70721

Menzies T, Elrawas O, Hihn J, Feather M, Madachy R, Boehm B (2007b) The business case for automated software engineering. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, pp 303–312

Menzies T, Greenwald J, Frank A (2007c) Data mining static code attributes to learn defect predictors. IEEE transactions on software engineering 33(1):2–13

Menzies T, Greenwald J, Frank A (2007d) Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering Available from `http://menzies.us/pdf/06learnPredict.pdf`

Menzies T, Williams S, Boehm B, Hihn J (2009) How to avoid drastic software process change (using stochastic stability). In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 540–550, DOI 10.1109/ICSE.2009.5070552, URL `http://dx.doi.org/10.1109/ICSE.2009.5070552`

Menzies T, Butcher A, Cok D, Marcus A, Layman L, Shull F, Turhan B, Zimmermann T (2013) Local versus Global Lessons for Defect Prediction and Effort Estimation. IEEE Transactions on Software Engineering 39(6):822–834, DOI 10.1109/TSE.2012.83

Menzies T, Krishna R, Pryor D (2016) The promise repository of empirical software engineering data. north carolina state university, department of computer science

Metzger A, Pohl K (2014) Software product line engineering and variability management: achievements and challenges. In: Proc. on Future of Software Engineering, ACM, pp 70–84

Mkaouer MW, Kessentini M, Bechikh S, Deb K, Ó Cinnéide M (2014) Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: Proc. 29th ACM/IEEE Intl. Conf. Automated Software Engineering, ACM, New York, NY, USA, ASE '14, pp 331–336, DOI 10.1145/2642937.2642965

Moghadam IH (2011) Search Based Software Engineering: Third Intl. Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, chap Multi-level Automated Refactoring Using Design Exploration, pp 70–75. DOI 10.1007/978-3-642-23716-4_9

Nagappan N, Ball T (2005) Static analysis tools as early indicators of pre-release defect density. In: Proc. 27th Intl. Conf. Software engineering, ACM, pp 580–586

Nam J, Kim S (2015a) Heterogeneous defect prediction. In: Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015, ACM Press, New York, New York, USA, pp 508–519, DOI 10.1145/2786805.2786814

Nam J, Kim S (2015b) Heterogeneous defect prediction. In: Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015, ACM Press, New York, New York, USA, pp 508–519, DOI 10.1145/2786805.2786814

Nam J, Pan SJ, Kim S (2013a) Transfer defect learning. In: Proc. Intl. Conf. on Software Engineering, pp 382–391, DOI 10.1109/ICSE.2013.6606584

Nam J, Pan SJ, Kim S (2013b) Transfer defect learning. In: Proc. Intl. Conf. Software Engineering, pp 382–391, DOI 10.1109/ICSE.2013.6606584

Nam J, Fu W, Kim S, Menzies T, Tan L (2017) Heterogeneous defect prediction. IEEE Transactions on Software Engineering PP(99):1–1, DOI 10.1109/TSE.2017.2720603

Nayrolles M, Hamou-Lhadj A (2018) Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects.

In: Mining Software Repositories
O'Keeffe M, Cinnéide MO (2008) Search-based refactoring: An empirical study. J Softw Maint Evol 20(5):345–364, DOI 10.1002/smr.v20:5
O'Keeffe MK, Cinneide MO (2007) Getting the most from search-based refactoring. In: Proc. 9th Annual Conf. Genetic and Evolutionary Computation, ACM, New York, NY, USA, GECCO '07, pp 1114–1120, DOI 10.1145/1276958.1277177
Oliveira P, Valente MT, Lima FP (2014) Extracting relative thresholds for source code metrics. In: 2014 Software Evolution Week - IEEE Conf. Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, pp 254–263, DOI 10.1109/CSMR-WCRE.2014.6747177
Ostrand TJ, Weyuker EJ, Bell RM (2004) Where the bugs are. In: ISSTA '04: Proc. the 2004 ACM SIGSOFT Intl. symposium on Software testing and analysis, ACM, New York, NY, USA, pp 86–96
Passos C, Braun AP, Cruzes DS, Mendonca M (2011) Analyzing the impact of beliefs in software project practices. In: ESEM'11
Peters F, Menzies T, Layman L (2015) LACE2: Better privacy-preserving data sharing for cross project defect prediction. In: Proc. Intl. Conf. Software Engineering, vol 1, pp 801–811, DOI 10.1109/ICSE.2015.92
Rahman F, Devanbu P (2013) How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 432–441
Rathore SS, Kumar S (2019) A study on software fault prediction techniques. Artificial Intelligence Review 51(2):255–327, DOI 10.1007/s10462-017-9563-5, URL `https://doi.org/10.1007/s10462-017-9563-5`
Ruhe G (2010) Product release planning: methods, tools and applications. CRC Press
Ruhe G, Greer D (2003) Quantitative studies in software release planning under risk and resource constraints. In: Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 Intl. Symposium on, IEEE, pp 262–270
Russell S, Norvig P (1995) Artificial Intelligence: A Modern Approach. Prentice-Hall, Egnlewood Cliffs
Sayyad AS, Ingram J, Menzies T, Ammar H (2013) Scalable product line configuration: A straw to break the camel's back. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th Intl. Conf., IEEE, pp 465–474
Sharma A, Jiang J, Bommannavar P, Larson B, Lin J (2016) Graphjet: real-time content recommendations at twitter. Proceedings of the VLDB Endowment 9(13):1281–1292
Shatnawi R (2010) A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. IEEE Transactions on Software Engineering 36(2):216–225, DOI 10.1109/TSE.2010.9
Shatnawi R, Li W (2008) The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. Journal of systems and software 81(11):1868–1882
Shull F, ad B Boehm VB, Brown A, Costa P, Lindvall M, Port D, Rus I, Tesoriero R, Zelkowitz M (2002) What we have learned about fighting defects. In: Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada, pp 249–258
Son TC, Pontelli E (2006) Planning with preferences using logic programming. Theory and Practice of Logic Programming 6(5):559–607
Stroggylos K, Spinellis D (2007) Refactoring–does it improve software quality? In: Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth Intl. Workshop on, IEEE, pp 10–10

Tallam S, Gupta N (2006) A concept analysis inspired greedy algorithm for test suite minimization. ACM SIGSOFT Software Engineering Notes 31(1):35–42

Theisen C, Herzig K, Morrison P, Murphy B, Williams L (2015) Approximating attack surfaces with stack traces. In: ICSE'15

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14(5):540–578

Turhan B, Tosun A, Bener A (2011) Empirical evaluation of mixed-project defect prediction models. In: Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conf., IEEE, pp 396–403

Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: Proc. Intl. Conf. Software Engineering, IEEE, pp 364–374, DOI 10.1109/ICSE.2009.5070536

Wooldridge M, Jennings NR (1995) Intelligent agents: Theory and practice. The knowledge engineering review 10(2):115–152

Ying R, He R, Chen K, Eksombatchai P, Hamilton WL, Leskovec J (2018) Graph convolutional neural networks for web-scale recommender systems. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, pp 974–983

Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2):67–120

Zhang Q, Li H (2007) MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. Evolutionary Computation, IEEE Transactions on 11(6):712–731, DOI 10.1109/TEVC.2007.892759

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction. Proc 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E p 91, DOI 10.1145/1595696.1595713

Zitzler E, Künzli S (2004) Indicator-Based Selection in Multiobjective Search. In: Parallel Problem Solving from Nature - PPSN VIII, Lecture Notes in Computer Science, vol 3242, Springer Berlin Heidelberg, pp 832–842, DOI 10.1007/978-3-540-30217-9_84

Zitzler E, Laumanns M, Thiele L (2002) SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In: Evolutionary Methods for Design, Optimisation, and Control, CIMNE, Barcelona, Spain, pp 95–100