From Prediction to Planning: Improving Software Quality with BELLTREE

Journal:	*Transactions on Software Engineering*
Manuscript ID	TSE-2018-07-0276
Manuscript Type:	Regular
Keywords:	H.2.8.d Data mining < H.2.8 Database Applications < H.2 Database Management < H Information Technology and Systems, Actionable Analytics, D.2.19.b Planning for SQA and V&V < D.2.19 Software Quality/SQA < D.2 Software Engineering < D Software/Software Engineering, Bellwethers, defect prediction

From Prediction to Planning: Improving Software Quality with BELLTREE

Rahul Krishna, Tim Menzies
Comptuer Science, North Carolina State University, USA
i.m.ralk@gmail.com, timm@ieee.org

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 probability of future defect reports. When compared to other planning algorithms from the SE literature, we find that this new approach is most effective at learning plans from one project, then applying those plans to another. In 10 open-source JAVA systems, several hundreds of defects were 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.

Index Terms—Data mining, actionable analytics, planning, bellwethers, defect prediction.

1 Introduction

Data mining tools have been applied to many applications in SE (e.g. [1], [2], [3], [4], [5], [6], [7]). 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 [8]. "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 [9], naive Bayes classifiers [10], logistic regression [10]. 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 offer *least* changes in software but most *improve* the *quality*; here:

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

This paper advocates the use of the *bellwether effect* [11], [12], [13] 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 (a portmanteau, *BELLTREE = Bellwether+XTREE*).

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

BELLTREE searches for an exemplar project, or *bellwether* [12], 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 introduces the four research questions asked in this paper, our contributions (§ 1.1), and relationships between this and our prior work (§ 1.2). In § 2 we discuss the notion of planning. There, in § 2.1, we discuss the planners studied here. § 3 contains the research methods, datasets, and evaluation strategy. In § 4 we answer the research questions. In § 5 we discuss the implications of our findings. Finally, § 6 and § 7 present threats to validity and conclusions respectively.

RQ1: Is within-project planning with XTREE comparatively more effective?

In this research question, we explore what happens when XTREE is trained on past data from *within* a project. XTREE uses historical logs from past releases of a project to recommend changes that might reduce defects in the next version of the software. Since such effects might not actually be casual, our first research question compares the effectiveness of XTREE's recommendations against alternative planning methods. Recent work by Shatnawi [14], Alves et al. [15], and Oliveira et al. [16] assume that unusually large measurements in source code metrics point to larger likelihood of defects. Those planners recommend changing all such unusual code since they assume that, otherwise, this may lead to defect-prone code. When XTREE is compared to the methods of Shatnawi, Alves, Oliveira et al., we find that:

Result: Planning with XTREE leads to the largest number of defects being reduced. This was true in 9 out of 10 projects studied here. Also, plans generated by XTREE are superior to other methods in all 10 projects.

Here, by "superior" we mean "larger defect reduction".

RQ2: Is cross-project planning with BELLTREE effective?

This research question addresses cases where projects lack local data (perhaps due to the project being relatively new). We ask if we can move data *across* from other projects to generate actionable plans.

For prediction tasks, in the case of domains like defect prediction, effort estimation, code smell detection, etc., it has been shown that the *bellwether effect* [12] can be used to make cross-project prediction when a project lacks sufficient within-project data. We ask if a similar approach is possible for planning across projects. To assert this, we first discover a *bellwether* dataset from the available projects, and then we construct the BELLTREE planner.

As with RQ1, we note that in a cross-project setting, plans generated using BELLTREE were effective in generating actionable plans. Also, when compared plans from Shatnawi, Alves, Oliveira et al., we find BELLTREE to be a significantly superior approach to cross-project planning.

Result: For cross-project planning, we see that recommendations from BELLTREE produces large reductions in the number of defects. This was true in 8 out of 9 projects studied here. Further, plans generated by BELLTREE were significantly superior to other planners.

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

In this research question, we compare the effectiveness of plans obtained with BELLTREE (cross-project) to plans obtained with XTREE (within-project).

This is an important result—when local data is missing, projects can use lessons learned from other projects.

Result: The effectiveness of BELLTREE is comparable to the effectiveness of XTREE.

RQ4: How many changes do the planners propose?

In the final research question, we ask how many attributes are recommended to be changed by different planners.

This result has a lot of practical significance since developers have a hard time following those plans that recommend too many changes.

Result: XTREE/BELLTREE recommends the least number of changes compared to other planners, while also producing the best overall performance (measured in terms of defect reduction).

1.1 Contributions

1. New kinds of software analytics techniques: This work combines planning [12] with cross-project learning using bellwethers [11]. This is a unique approach since our previous work on bellwethers [11], [17] explored prediction and not planning as described here. Also, our previous work on planning [12] explored with-project problems but not cross-project problems as explored here.

2. Compelling results about planning: Our results show that planning is quite 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 just used in the context of prediction [11] and have been shown to work for (i) defect prediction, (ii) effort estimation, (iii) issues close time, and (iv) detecting code smells [17]. This paper extends those results with a new finding 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 Relationship to Prior Work

As for the connections to prior research, this article significantly extends those results. 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 [18]. After that we found ways in 2009 to train these predictors on some projects, then test them on others [19]. Subsequent work in 2016 found that bellwethers were a simpler and effective way to implement transfer learning [11], which worked well for a wide range of software analytics tasks [17]. Meanwhile, in the area of planning, we conducted some limited within-project planning in 2017 on recommending what to change in software [12]. This current article now addresses a much harder question: can plans be generated from one project and applied to the another? While answering this question, 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 [11], [12], [17], [18], [19]. 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 [20]. Heterogeneous planning is primary focus of our future work.

This paper extends a short abstract presented at the IEEE ASE'17 Doctoral Symposium [21]. Most of this paper, including all experiments, did not appear in that abstract.

		Data source		
		Within	Cross	
Task	Prediction	TSE '07 [18]	EMSE '09 [19] ASE '16 [11] TSE '18 [17]	TSE '17 [20]
	Planning	IST '17 [12]	This work	Future work
		Homogeneous		Heterogeneous

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

amc	average method complexity
avg cc	average McCabe
ca	afferent couplings
cam	cohesion amongst classes
cbm	coupling between methods
cbo	coupling between objects
ce	efferent couplings
dam	data access
dit	depth of inheritance tree
ic	inheritance coupling
lcom (lcom3)	2 measures of lack of cohesion in methods
loc	lines of code
max cc	maximum McCabe
mfa	functional abstraction
moa	aggregation
noc	number of children
npm	number of public methods
rfc	response for a class
wmc	weighted methods per class
#defects	raw defect counts

Fig. 2: OO code metrics used for all studies in this paper. Last line, shown in gray, denotes the dependent variable. For more details, see [12].

2 About 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 generates a concrete set of actions that can be taken (as 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:

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

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 Fig. 2.

In order to operationalize such plans, developers need some guidance on what to change in order to achieve the desired effect. There are two ways to generate that guidance. One way is to is to use a technique used by Nayrolles et al. [28] at MSR 2018. 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.

When that data is not accessible, another way to operationalize plans is via guidance from the literature. Many papers have discussed how code changes adjust static code metrics [22], [23], [24], [25], [26], [27]. Fig. 3(b) shows a summary of that research. For example, in Fig. 3, say a planner has recommended the changes shown in Fig. 3(a). Then, we use 3(b) to look-up possible actions developers may take, there 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".

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

(a) Recommendations from some planner. Here a '+' represents an *increase*, a '−' 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 [22], [23], [24], [25], [26], [27]. The action highlighted in gray shows an action matching XTREE's recommendation.

```
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.

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 discusses other kinds of planning.

2.1 Other Kinds of Planning

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* [29]. 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 [30]. Some of the most common planning paradigms include: (a) classical planning [31]; (b) probabilistic planning [32], [33], [34]; and (c) preference-based planning [35], [36]. 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.

Apart from *code-based planning*, we know of at least two other kinds of planning research in SE. 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 [37], [38], [39].
- 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 combining simulated annealing with the COCOMO models or Ruhe et al.'s work on next release planning in requirements engineering [40], [41].

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 [42], [43]. 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. [44], [45], [46], [47], [48], [49], [50]. 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 [51], [52], [53], software product line management [54], [55], [56], automated test generation [57], [58], 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 [59] 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 [60] to 2000 [61]). Lastly, even when there is an existing model, they can require constant maintenance lest they become out-dated. 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 [62], [63]. 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 of use 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 section: XTREE, BELLTREE and the approaches of Alves et al. [15], Shatnawi [14], and Oliveira et al. [16]

2.1.1 Within-Project Planning With XTREE

XTREE builds a decision tree, then generates plans by contrasting the differences between two branches: (1) the current branch; (2) the desired branch. XTREE uses a supervised regression tree algorithm that is constructed on discretized values OO metrics (we use Fayyad-Irani discretizer [64]). Next, XTREE builds plans from the branches of the tree as follows, for every test instance, we ask:

1) Which *current* branch matches the test instance?
2) Which *desired* branch would the test want to emulate?
3) What are the *deltas* between current and desired?

See Fig. 4 for an example of how plans are generated using these three questions. See Fig. 3 for details on how these plans translate back to changes in the code.

Fig. 4.A: Using XTREE
Using the training data, construct a decision tree. For each test item, find the *current* leaf: take each test instance, run it down to a leaf in the decision tree. After that, find the *desired* leaf:

- Starting at *current*, ascend the tree levels;
- Identify *sibling* leaves; i.e. leaf clusters that can be reached from level lvl that are not same as *current*
- Find the *better* siblings; i.e. those with a *score* (#defects) less than $\gamma = 0.5$ times the score of *current* branch. If none found, then repeat for $lvl+ = 1$. Also, return no plan if the new lvl is above the root.
- Return the *closest* better sibling to the *current*.

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

Fig. 4.B: A sample decision tree.

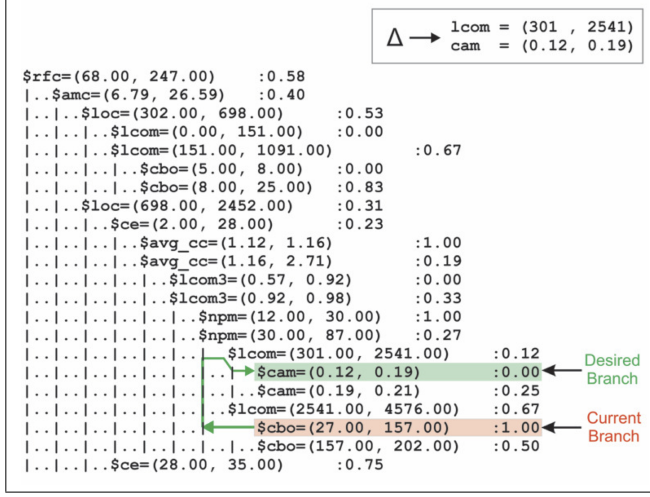


Fig. 4: Generating thresholds using XTREE.

2.1.2 Cross-project Planning with BELLTREE

Many methods have been proposed for transferring data or lessons learned from one project to another, for examples see [65], [66], [67], [68], [69], [19], [70]. 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 [17]. 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 [12] which was limited to the use of datasets from within a few projects.

BELLTREE extends the three bellwether operators defined in our previous work [17] 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 [11]. 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 $y \in C$.
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.

2.1.3 Alves

Looking through the SE literature, we can see researchers have proposed three other methods analogous to XTREE planning. Those other methods are proposed by Alves et al. [15] (described in this section) plus those of Shatnawi [14] and Oliveira et al. [16] described below.

Alves et al. [15] 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.

2.1.4 Shatnawi

Shatnawi [14] offers a different alternative Alves et al by using VARL (Value of Acceptable Risk Level). This method was initially proposed by Bender [71] 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:

$$Defective\ if\ Metric > VARL$$

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

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.

2.1.5 Oliveira

Oliveira et al. in their 2014 paper offer yet another alternative to absolute threshold methods discussed above [16]. 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:

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

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 $(0, k]$.

3 Research Methods

The rest of this paper compares XTREE and BELLTREE against Alves, Shatnawi, Oliveira et al.

3.1 Datasets

The defect dataset used in this study 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. [72]. 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 CK metrics and McCabes complexity metrics, see Fig. 2 for description. Since we attempt to learn plans from within the project and across projects, we explore *homogeneous* transfer of plans. Homogeneity requires that the attributes (static code metrics) are the same for all the datasets and all the projects. We obtained the dataset from the SEACRAFT repository[2] (formerly the PROMISE repository [73]). For more information see [11].

3.2 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. [74], O'Keefe et al. [75], [76], Moghadam [77] and Mkaouer et al. [78] 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 [79].

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

Dataset	Versions	# samples	Bugs (%)
Lucene	2.0, 2.2,2.4	782	438 (56.01)
Ant	1.3, 1.4, 1.5, 1.6, 1.7	1692	350 (20.69)
Ivy	1.1, 1.4,2.0	704	119 (16.90)
Jedit	3.2, 4.0, 4.1, 4.2,4.3	1749	303 (17.32)
Poi	1.5, 2, 2.5, 3.0	1378	707 (51.31)
Camel	1.0, 1.2, 1.4,1.6	2784	562 (20.19)
Log4j	1.0, 1.1,1.2	449	260 (57.91)
Velocity	1.4, 1.5,1.6	639	367 (57.43)
Xalan	2.4, 2.5, 2.6,2.7	3320	1806 (54.40)
Xerces	1.0, 1.2, 1.3,1.4	1643	654 (39.81)

Fig. 5: The figure lists defect datasets used in this paper. The bellwether dataset is highlighted in light gray.

A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [80]. Hence, we eschew older quality models like QMOOD and propose a verification oracle based on the *overlap*. between two sets: (1) The changes that developers made, perhaps in response to the issues raised in a post-release issue tracking system; and (2) Plans recommended by an automated planning tool such as XTREE/BELLTREE. Using these two sources of changes, it is possible to compute the extent to which a developer's action matches that of the actions recommended by planners. This is measured using *overlap*:

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

That is, we measure overlap using the size of the intersections divided by the size of the union of the *changes*. Here $\mathcal{D}$ represents the changes made by the *developers* and $\mathcal{P}$ represents the changes recommended by the *planner*. Accordingly, the larger the intersection between the changes made by the developers to the changes recommended by the planner, the greater the overlap.

As an example, consider Fig. 6; there we have 2 sets of changes: (1) Changes made by developers ($\mathcal{D}$), and (2) Changes recommended by the planner ($\mathcal{P}$). In each case we have 3 possible actions for every metric: (1) Make no change ('$\cdot$'), (2) Increase ('$+$'), and (3) Decrease ('$-$'). The intersection of the changes represents the number of times the actions taken by the developers is the same as the actions recommended by the planner. This the above example, the intersection, $\mathcal{D} \cap \mathcal{P} = 7$, out of a total of $\mathcal{D} \cup \mathcal{P} = 9$ possible actions. This leads to $Overlap = \frac{7}{9} \times 100 = 77.77\%$.

	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Planner ($\mathcal{P}$)	·	·	·	+	·	+	+	+	+
Developer ($\mathcal{D}$)	·	·	–	+	–	+	+	+	+

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

Fig. 6: A simple example of computing overlap. Here a '$+$' represents an *increase*, a '$-$' represents a *decrease*, and a '$\cdot$' represents *no-change*. Columns shaded in gray indicate a match between developer's change and the recommendation made by a planner.

3.2.1 The $\mathbb{K}$-test

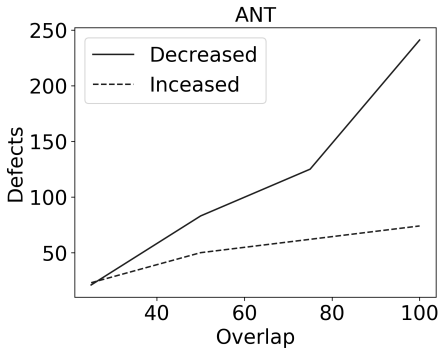
Measuring overlap as described in the previous section only measures the extent to which the recommendations made by planning tools match those undertaken by the developers. Note that overlap does not measure the *quality* of the recommendation. Specifically, it does not tell us what the impact of making those changes would be to future releases of a project. Therefore, it is necessary to augment the overlap with a measure of how many defects are reduced as a result of making the changes.

For this purpose, we propose the $\mathbb{K}$-test. Given a project $\mathcal{P}$ with versions $v \in \{\mathcal{P}_i, \mathcal{P}_j, \mathcal{P}_k\}$ ordered chronologically (that is, version $i < j < k$ in terms of release dates), we divide project data into three sets *train*, *test*; and *validation* which are used as follows:

1) First, train the planner on version $\mathcal{P}_i$. Note: this could either be data that is either a previous release (version i), or it could be data from the bellwether dataset.
2) Next, use the planner to generate plans to reduce defects for version $\mathcal{P}_j$.
3) Finally, on version $\mathcal{P}_k$, we measure the OO metrics for each class in $\mathcal{P}_j$, then we (a) measure the overlap between plans recommended by the planner and the developer's actions; (b) count the number of defects reduced/increased when compared to the previous release as a result of implementing these plans.

As the outcome of the $\mathbb{K}$-test we obtain the number of defects (increased or decreased) and the extent of overlap (from 0% to 100%). These two measures enable us to plot the operating characteristic curve for the planners (referred to henceforth as planner effectiveness curve). The operating characteristic (OC) curve depicts the effectiveness of a planner with respect to its ability to reduce defects. The OC curve plots the overlap of developer changes with the planner's recommendations versus the number of defects reduced. A sample curve for one of our datasets is shown in Fig. 7.

For each of the datasets with versions i, j, k we (1) train the planner on version i; (2) deploy the planner to recommend plans for version j; and (3) validate plans for version j. Following this, we plot the "planner effectiveness curve". Finally, as in Fig. 7, we compute the area under the planner effectiveness curve (AUPEC) using simpsons rule [81].



$$AUPEC = \int_0^1 f(x)\,dx$$

$$\approx \tfrac{\Delta x}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + \cdots + 4f(x_{n-1}) + f(x_n)\right)$$

Here, the variable x represents the overlap between plans and developer changes and $f(x)$ represents the number of defects reduced as result of the overlap.
We should interpret AUPEC as follows:

1) *Defects Reduced:* AUPEC is always greater than zero and *larger values* of AUPEC point to *more defects reduced* with increasing overlap.
2) *Defects Increased:* AUPEC is still always greater than zero and *smaller values* of AUPEC point to *less defects increased* with increasing overlap.

In the above plot, we show an example of XTREE on Ant. We see that the more developers used our plans (and moved right across the x-axis), then subsequent changes to the code removed far more defects than it added.
Note: Since the actual number of defects vary from one project to another, we report the AUPEC score as a percentage of theoretical best. The theoretical best for AUPEC for defects reduced will be 100% and 0% for defects increased.

Fig. 7: AUPEC = Area Under Planner Effectiveness Curve.

4 Experimental Results

RQ1: Is within-project planning with XTREE comparatively more effective?

We answer this question in two parts: (a) First, we assess the effectiveness of XTREE (using Area Under Planner Effectiveness Curve); (b) Next, we compare XTREE with other threshold based planners. In each case, we split the available data into training, testing, and validation. That is, given versions $v_1, v_2, ..., v_K$, we, *train* the planners on version v_1; then *generate plans* using the planners for version v_2; then *validate* the effectiveness of those plans on v_3 using the $\mathbb{K}$-test. Then, we repeat the process by training on v_2, testing on v_3, and validating on version v_4, and so on.

For each of these $\{train, test, validation\}$ sets, we generate performance statistics as per Fig. 7; i.e. plot the planner effectiveness curve to measure the number of defects reduced (and increased) as a function of extent of overlap. Then, we measure the Area-Under the Planner Effectiveness Curve (AUPEC).

Fig. 8(a) shows the results of planning with XTREE (see column labeled XTREE). The columns constitutes of 2 parts (labeled $\bigtriangledown$ and $\bigtriangleup$) where:

(a) $\bigtriangledown$ represents AUPEC for the number of defects *reduced*, and *larger* values are *better*;
(b) $\bigtriangleup$ indicates the number of defects *increased* in response to overlap with XTREE's plans, and *smaller* values are better.

We observe that, in 14 out of 18 cases, the *AUPEC of defect reduced in much larger* than AUPEC of defects increased. This indicates that within-project XTREE is very effective in generating plans that reduce the number of defects. Further, we note that in terms of the number of defects reduced, in all 18 datasets, XTREE significantly outperforms other threshold based planners.

It is worth noting that, in the case of AUPEC for defects increased, XTREE's plans do seem to result in larger increase in the number of defects increased when compared to other threshold based learners. But, the number of defects reduced as result of XTREE's plans are much larger than the occasional increase in defects. Thus, in summary,

		XTREE		Alves		Shatnawi		Oliveira	
		▽	△	▽	△	▽	△	▽	△
Ant	ant-1	61†‡	9	41	9	40	10	19	0
	ant-2	42†	21	35	16	33	15	7	5
	ant-3	53†	24	33	15	33	16	21	9
Camel	camel-1	45†	4	24	1	22	3	19	3
	camel-2	37†	7	27	4	26	1	8	3
Ivy	ivy-1	29†	9	24	8	25	8	5	1
Jedit	jedit-1	42†	13	26	7	26	7	16	6
	jedit-2	59†	5	35	3	35	3	24	2
	jedit-3	73†	1	51	1	49	1	22	0
Log4j	log4j-1	16†‡	41	2	22	3	21	11	17
Lucene	lucene-1	57†	41	8	17	8	18	46	24
Poi	poi-1	22†	52	18	45	17	45	4	8
	poi-2	63†‡	14	4	6	5	5	58	8
Velocity	velocity-1	43†	4	7	1	6	2	33	83
Xalan	xalan-1	31†‡	15	10	8	9	8	19	6
	xalan-2	54†	56	1	36	2	34	43	20
Xerces	xerces-1	23†‡	6	17	4	5	6	6	2
	xerces-2	24†	42	18	31	19	30	6	11

(a) Within Project

		BELLTREE		Alves		Shatnawi		Oliveira	
		▽	△	▽	△	▽	△	▽	△
Ant	ant-1	59†	9	22	0	28	0	19	0
	ant-2	42†	21	8	5	9	6	7	5
	ant-3	61†‡	27	25	11	29	10	23	10
Camel	camel-1	48†‡	4	27	3	37	4	21	3
	camel-2	37†	6	11	3	15	4	8	2
Ivy	ivy-1	29†	9	5	1	2	1	5	1
Jedit	jedit-1	44†‡	13	21	7	24	8	17	6
	jedit-2	66†‡	5	29	2	31	2	26	2
	jedit-3	73†	1	23	0	21	0	22	0
Log4j	log4j-1	13†	39	14	21	18	25	11	17
Lucene	lucene-1	57†	41	8	17	8	18	46	24
Poi	poi-1	22†	52	10	14	20	20	14	8
	poi-2	47†	32	15	18	18	22	13	17
Velocity	velocity-1	48†‡	1	53	4	81	5	39	3
Xalan	xalan-1	34†‡	15	20	9	29	11	20	6
	xalan-2	54†	73	1	39	2	47	43	25
Xerces	xerces-1	22†	5	7	2	14	2	5	1
	xerces-2	54†‡	41	8	13	14	15	5	10

(b) Cross-Project

Fig. 8: Area Under Planner Effectiveness Curve (AUPEC) obtained with the $\mathbb{K}$-test for all planners. ▽ indicates AUPEC for defects reduced and △ indicates AUPEC for defects increased. *Larger* values for ▽ are preferable and *smaller* values for △ are preferable. For each row, cells with the largest AUPEC values shaded in gray. Note that in 14 out of 18 cases, XTREE/BELLTREE reduces far more defects than it increases. Cells labeled with † indicates the best planner for *reducing* defects. Note that in all cases, XTREE/BELLTREE outperform other planners. To compare XTREE with BELLTREE, cells are labeled with ‡. In 5 cases XTREE is better than BELLTREE, in 7 cases BELLTREE is better than XTREE, and in 6 cases they are comparable.

Result: In 9 out of 10 projects (14 out of 18 datasets), planning with XTREE leads to the largest number of defects reduced. Also, plans generated by XTREE are superior to other methods in all 10 projects.

RQ2: Is cross-project planning with BELLTREE effective?

In the previous research question, we construct XTREE using historical logs of previous releases of a project. However, when such logs are not available, we may seek to generate plans using data from across software projects. To do this we offer BELLTREE, a planner that makes use of the *Bellwether Effect* in conjunction with XTREE to perform cross-project planning. In this research question, we assess the effectiveness of BELLTREE. For details of construction of BELLTREE, see § 2.1.2.

Our experimental methodology for answering this research question is as follows:

1) We first discover the bellwether data from the available projects. For the projects studied here, we discovered that *Lucene* was the bellwether (in accordance with our previous findings [11], [17]).
2) Next, we construct XTREE, but we do this using the bellwether dataset. We call this variant BELLTREE.
3) For each of the other projects, we use BELLTREE constructed above to recommend plans.
4) Then, we use the subsequent releases of the above projects to validate the effectiveness of those plans.

Finally, we generate performance statistics as per Fig. 7; i.e. plot the planner effectiveness curve to measure the number of defects reduced (and increased) as a function of extent of overlap. Then, we measure the Area-Under the Planner Effectiveness Curve (AUPEC). Figure 8(b) shows the AUPEC scores that were the outcome cross-project planning with BELLTREE (see column labeled BELLTREE). Similar to our findings in RQ1, we note that, in 15 out of 18 cases, AUPEC of defect reduced is much larger that AUPEC of defects increased. This indicates that cross-project planning BELLTREE is also very effective in generating plans that reduce the number of defects. Further, when we train each of the other planners with the bellwether dataset and compare them with XTREE, we note that, as with RQ1, BELLTREE outperforms other threshold based planners for cross-project planning.

Result: BELLTREE helps reduce a large number of defects in 15 out of 18 datasets (9 out of 10 projects). Plans generated by BELLTREE were also significantly superior to other planners in all 10 projects.

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

The third research question compares within-project planning with XTREE to cross-project planning with BELLTREE.

Metrics	Ant				Ivy				Camel				Xerces				Velocity			
	XTREE	Alves	Shatnawi	Oliveira	XTREE	Alves	Shatnawi	Oliveira	XTREE	Alves	Shatnawi	Oliveira	XTREE	Alves	Shatnawi	Oliveira	XTREE	Alves	Shatnawi	Oliveira
wmc		89	100	94		94	100	94		94	100	92		87	100	94		84	100	92
dit		47	100	66		100	28	49		40		62	16	90	87	52		67	100	44
noc			100				100			2					100				65	
cbo		1	12	92	100	1	1	96		3	39	93	88	1		86		7	100	89
rfc		19	21	97		23		100		6	100	98		16	24	96		79	64	99
lcom		76	79	87		23	91	92		57	100	87		46	36	91		38	100	85
ca	74	11	29	75		3	75	83	97	14	100	77		18	2	68	83	10	100	84
ce		3	21	90		1	33	82		4	40	92		7	87	79		37	65	92
npm		99	100	92		28	100	90		91	100	89		97	100	94	96	86	59	88
lcom3			9	92			100	89	52		39	81	100		75	57			64	78
loc		100	99	100		28	100	100		100	100	100	35	100	26	100		99	88	100
dam			37	47			28	44			39	37		9		37		52		55
moa		56	100	35		51	100	39		45	100	39		29	88	18		39	100	27
mfa	67	52	100	82		30	100	53		50	100	70		50	24	59	100	45	64	64
cam			100	96			71	97			65	96		5	11	97		4	100	96
ic		9	21	47			71	31		2	39	31			100	23		6	35	25
cbm	100		100	52				32			100	36	31		24	28			64	29
amc		49	100	98		26	32	96		72	39	98		55	24	88		88	100	97
max_cc		63	100	78		19	100	64	100	55	75	60		44	100	51		100	100	59
avg_cc				96				92				91				87				89

Metrics	Xalan				Poi				Log4j				Jedit			
	XTREE	Alves	Shatnawi	Oliveira	XTREE	Alves	Shatnawi	Oliveira	XTREE	Alves	Shatnawi	Oliveira	XTREE	Alves	Shatnawi	Oliveira
wmc		72	100	90		98	91	94		80	100	93		87	69	94
dit		59	77	67		50	100	50		16	100	46		69	100	66
noc			22		70	1	50								100	
cbo	100	1		92	100	2	49	90	100	1	17	89	36	6		95
rfc	69		22	96		2	49	97		15	16	97		42	72	98
lcom		31	30	83	9	78	100	94		65	84	91	37	58	60	93
ca		8	77	78		7	7	80		12	84	75	12	23	17	87
ce		11	22	88		7		88		6	1	85		21		93
npm	78	96	100	89	13	98	100	93		97	100	87		97	100	90
lcom3	65		22	84	41		56	97				91	22		59	92
loc		100	100	100		100	100	100		100	100	100	1	100	98	100
dam			100	51			6	80			16	49		15		60
moa	35	35	47	35		30	56	30		46	100	54		67	100	54
mfa		64	75	81	4	64	100	84		38	16	52		51	62	75
cam			77	95			49	97			83	97		2	77	98
ic	77			45			43	43		2	100	28		7	22	47
cbm	44		24	52	99		100	62			100	31	11		100	51
amc		70	32	99	13	66	62	99		82	17	99	36	84	64	99
max_cc		81	76	76	14	81	57	74		53	83	82	100	70	50	87
avg_cc				94				97				96				98

Fig. 9: The number of changes recommended by each of the planners. The values on each row represents the percentage score indicating the number of times the metric has recommended for change. Note that XTREE (highlighted in gray) recommends changes to far fewer metrics than the other methods.

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. We found that this was indeed the case; i.e. in terms of AUPEC, both XTREE and BELLTREE are similar to each other.

Fig. 8 tabulates the AUPEC scores for the comparison between the use of within-project XTREE (see Fig. 8(a)) and cross-project BELLTREE(see Fig. 8(b)) for reducing the number of defects. We note that, out of 18 datasets from 10 projects, the AUPEC scores are quite comparable. In 5 cases XTREE performs better than BELLTREE, in 7 cases BELLTREE outperforms XTREE, and in 6 cases the performance is the same. In summary, we make the following observations:

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.

RQ4: How many changes do the planners propose?

This question naturally follows the findings of the previous research questions. Here, we ask how many changes each of the planners recommend. This is important because having plans recommend far too many changes would make it challenging for practical use.

Our findings for XTREE tabulated in Fig. 9[3] show that XTREE (BELLTREE) proposes far fewer changes compared to other planners. This is because, both XTREE and BELLTREE operate based on supervised learning incorporating

3. Space limitations prohibit showing results of BELLTREE. We notice a very similar trend to XTREE. Interested readers can use our replication package (https://git.io/fNcYY) to further evaluate these results.

two stages of data filtering and reasoning: (1) Discretization of attributes based on information gain, and (2) Plan generation based on contrast sets between adjacent branches. This is different to the other approaches. The operating principle of the other approaches is that attribute values larger than a certain threshold must always be reduced. Hence, they usually propose plans that use all attributes in an unsupervised manner, without first filtering out the less important attributes based on how they impact the quality of software. This leads to those planners being far more verbose and, possibly, harder to operationalize.

Result: Our planning methods (XTREE/BELLTREE) recommend far fewer changes than other methods.

5 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. [82], 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 [83] 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 [83]. 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 [84]. 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.

6 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: AUPEC (see Fig. 7). 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.

7 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. 8 show that BELLTREE is just as good as XTREE, and both XTREE/BELLTREE are much better than other planners.

Further, we can see from Fig. 9 that both BELLTREE and XTREE recommend changes to very few metric, while other unsupervised planners such as Shatnawi, Alves, and Olivera, recommend changing most of the metrics. This is not practical in many real world scenarios.

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

[1] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "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*, march 2011, pp. 357 –366.

[2] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA '04: Proc. the 2004 ACM SIGSOFT Intl. symposium on Software testing and analysis*. New York, NY, USA: ACM, 2004, pp. 86–96.

[3] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'",," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 637–640, sep 2007.

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

[5] E. Kocaguneli, T. Menzies, A. Bener, and J. Keung, "Exploiting the essential assumptions of analogy-based effort estimation," *IEEE Transactions on Software Engineering*, vol. 28, pp. 425–438, 2012.

[6] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proc. 36th Intl. Conf. Software Engineering (ICSE 2014)*. ACM, June 2014.

[7] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating attack surfaces with stack traces," in *ICSE'15*, 2015.

[8] J. Hihn and T. Menzies, "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)*, Nov 2015, pp. 5–9.

[9] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[10] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, jul 2008.

[11] R. Krishna, T. Menzies, and W. Fu, "Too much automation? the bellwether effect and its implications for transfer learning," in *Proc. 31st IEEE/ACM Intl. Conf. Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016, pp. 122–131.

[12] R. Krishna, T. Menzies, and L. Layman, "Less is more: Minimizing code reorganization using XTREE," *Information and Software Technology*, mar 2017.

[13] S. Mensah, J. Keung, S. G. MacDonell, M. F. Bosu, and K. E. Bennin, "Investigating the significance of the bellwether effect to improve software effort prediction: Further empirical study," *IEEE Transactions on Reliability*, no. 99, pp. 1–23, 2018.

[14] R. Shatnawi, "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 216–225, March 2010.

[15] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *2010 IEEE Int. Conf. Softw. Maint.* IEEE, sep 2010, pp. 1–10.

[16] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *2014 Software Evolution Week - IEEE Conf. Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, feb 2014, pp. 254–263.

[17] R. Krishna and T. Menzies, "Bellwethers: A baseline method for transfer learning," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[18] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[19] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[20] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[21] R. Krishna, "Learning effective changes for software projects," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 1002–1005.

[22] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth Intl. Workshop on*. IEEE, 2007, pp. 10–10.

[23] B. Du Bois, "A study of quality improvements by refactoring," 2006.

[24] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Software Maintenance, 2002. Proceedings. Intl. Conf.* IEEE, 2002, pp. 576–585.

[25] S. Bryton and F. B. e Abreu, "Strengthening refactoring: towards software evolution with quantitative and experimental grounds," in *Software Engineering Advances, 2009. ICSEA'09. Fourth Intl. Conf.* IEEE, 2009, pp. 570–575.

[26] K. Elish and M. Alshayeb, "A classification of refactoring methods based on software quality attributes," *Arabian Journal for Science and Engineering*, vol. 36, no. 7, pp. 1253–1267, 2011.

[27] ——, "Using software quality attributes to classify refactoring to patterns." *JSW*, vol. 7, no. 2, pp. 408–419, 2012.

[28] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *Mining Software Repositories*, 2018.

[29] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Egnlewood Cliffs, 1995.

[30] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[31] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.

[32] R. Bellman, "A markovian decision process," *Indiana Univ. Math. J.*, vol. 6, pp. 679–684, 1957.

[33] E. Altman, *Constrained Markov decision processes*. CRC Press, 1999, vol. 7.

[34] X. Guo and O. Hernández-Lerma, "Continuous-time markov decision processes," *Continuous-Time Markov Decision Processes*, pp. 9–18, 2009.

[35] T. C. Son and E. Pontelli, "Planning with preferences using logic programming," *Theory and Practice of Logic Programming*, vol. 6, no. 5, pp. 559–607, 2006.

[36] S. S. J. A. Baier and S. A. McIlraith, "Htn planning with preferences," in *21st Int. Joint Conf. on Artificial Intelligence*, 2009, pp. 1790–1797.

[37] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 35–42, 2006.

[38] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[39] D. Blue, I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Interaction-based test-suite minimization," in *Proc. the 2013 Intl. Conf. Software Engineering*. IEEE Press, 2013, pp. 182–191.

[40] G. Ruhe and D. Greer, "Quantitative studies in software release planning under risk and resource constraints," in *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 Intl. Symposium on*. IEEE, 2003, pp. 262–270.

[41] G. Ruhe, *Product release planning: methods, tools and applications*. CRC Press, 2010.

[42] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," *Department of Computer Science, Kings College London, Tech. Rep. TR-09-03*, 2009.

[43] M. Harman, P. McMinn, J. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," *Search*, vol. 2012, pp. 1–59, 2011.

[44] J. Krall, T. Menzies, and M. Davies, "Gale: Geometric active learning for search-based software engineering," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 1001–1018, 2015.

[45] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2002.

[46] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization," in *Evolutionary Methods for Design, Optimisation, and Control*. CIMNE, Barcelona, Spain, 2002, pp. 95–100.

[47] E. Zitzler and S. Künzli, "Indicator-Based Selection in Multiobjective Search," in *Parallel Problem Solving from Nature - PPSN VIII*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, vol. 3242, pp. 832–842.

[48] K. Deb and H. Jain, "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*, vol. 18, no. 4, pp. 577–601, Aug. 2014.

[49] X. Cui, T. Potok, and P. Palathingal, "Document clustering using particle swarm optimization," *... Intelligence Symposium, 2005. ...*, 2005.

[50] Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 6, pp. 712–731, Dec. 2007.

[51] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. Intl. Conf. Software Engineering*. IEEE, 2009, pp. 364–374.
[52] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "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, jun 2012, pp. 3–13.
[53] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, dec 2015.
[54] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th Intl. Conf*. IEEE, 2013, pp. 465–474.
[55] A. Metzger and K. Pohl, "Software product line engineering and variability management: achievements and challenges," in *Proc. on Future of Software Engineering*. ACM, 2014, pp. 70–84.
[56] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon, "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, May 2015, pp. 517–528.
[57] J. H. Andrews, F. C. H. Li, and T. Menzies, "Nighthawk: A Two-Level Genetic-Random Unit Test Data Generator," in *IEEE ASE'07*, 2007.
[58] J. H. Andrews, T. Menzies, and F. C. H. Li, "Genetic Algorithms for Randomized Unit Testing," *IEEE Transactions on Software Engineering*, Mar. 2010.
[59] T. Menzies, O. Elrawas, J. Hihn, M. Feather, R. Madachy, and B. Boehm, "The business case for automated software engineering," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 303–312.
[60] B. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.
[61] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.
[62] P. G. Ii, T. Menzies, S. Williams, and O. El-Rawas, "Understanding the value of software engineering technologies," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 52–61.
[63] B. Lemon, A. Riesbeck, T. Menzies, J. Price, J. D'Alessandro, R. Carlsson, T. Prifiti, F. Peters, H. Lu, and D. Port, "Applications of Simulation and AI Search: Assessing the Relative Merits of Agile vs Traditional Software Development," in *IEEE ASE'09*, 2009.
[64] U. Fayyad and K. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," *NASA JPL Archives*, 1993.
[65] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. Intl. Conf. Software Engineering*, 2013, pp. 382–391.
[66] J. Nam and S. Kim, "Heterogeneous defect prediction," in *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*. New York, New York, USA: ACM Press, 2015, pp. 508–519.
[67] X. Jing, G. Wu, X. Dong, F. Qi, and B. Xu, "Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning," in *FSE'15*, 2015.
[68] E. Kocaguneli and T. Menzies, "How to find relevant data for effort estimation?" in *Empirical Software Engineering and Measurement (ESEM), 2011 Intl. Symposium on*. IEEE, 2011, pp. 255–264.
[69] E. Kocaguneli, T. Menzies, and E. Mendes, "Transfer learning in effort estimation," *Empirical Software Engineering*, vol. 20, no. 3, pp. 813–843, jun 2015.
[70] F. Peters, T. Menzies, and L. Layman, "LACE2: Better privacy-preserving data sharing for cross project defect prediction," in *Proc. Intl. Conf. Software Engineering*, vol. 1, 2015, pp. 801–811.
[71] R. Bender, "Quantitative risk assessment in epidemiological studies investigating threshold effects," *Biometrical Journal*, vol. 41, no. 3, pp. 305–319, 1999.
[72] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predict. Model. Softw. Eng. - PROMISE '10*. New York, New York, USA: ACM Press, 2010, p. 1.
[73] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of empirical software engineering data. north carolina state university, department of computer science," 2016.
[74] B. Cheng and A. Jensen, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Proc. 12th Annual Conf. Genetic and Evolutionary Computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 1341–1348.
[75] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring: An empirical study," *J. Softw. Maint. Evol.*, vol. 20, no. 5, pp. 345–364, Sep. 2008.
[76] M. K. O'Keeffe and M. O. Cinneide, "Getting the most from search-based refactoring," in *Proc. 9th Annual Conf. Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1114–1120.
[77] I. H. Moghadam, *Search Based Software Engineering: Third Intl. Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. Multi-level Automated Refactoring Using Design Exploration, pp. 70–75.
[78] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proc. 29th ACM/IEEE Intl. Conf. Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 331–336.
[79] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.
[80] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus Global Lessons for Defect Prediction and Effort Estimation," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 822–834, jun 2013.
[81] R. L. Burden and J. D. Faires, *Numerical Analysis: 4th Ed*. Boston, MA, USA: PWS Publishing Co., 1989.
[82] C. Passos, A. P. Braun, D. S. Cruzes, and M. Mendonca, "Analyzing the impact of beliefs in software project practices," in *ESEM'11*, 2011.
[83] M. Jørgensen and T. M. Gruschke, "The impact of lessons-learned sessions on effort estimation and uncertainty assessments," *Software Engineering, IEEE Transactions on*, vol. 35, no. 3, pp. 368 –383, May-June 2009.
[84] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *Proc. 38th Intl. Conf. Software Engineering*. ACM, 2016, pp. 108–119.

Rahul Krishna is a Ph.D. student in the department of Computer Science at NC State University. He received his master's degree in Electrical Engineering also at NC State University. His interest lies in exploring ways in which artificial intelligence can be used to generate actionable analytics for software engineering. He currently works on developing machine learning algorithms that go beyond prediction to assist in decision making. For more information, visit http://rkrsn.us.

Tim Menzies (Ph.D., UNSW, 1995) is a full Professor in CS at NC State University, where he explores SE, data mining, AI, search-based SE, and open access science. He is the author of over 250 referred publications and co-founder of the PROMISE conference series devoted to reproducible experiments in SE (http://tiny.cc/seacraft). Dr. Menzies also serves as associated editor of many journals: IEEE TSE, ACM TOSEM, Empirical Software Engineering, ASE Journal, Information Software Technology, IEEE Software, and the Software Quality Journal. For more, see http://menzies.us.

Don't Tell Me What Is, Tell Me What Ought To Be! Learning Effective Changes for Software Projects

Rahul Krishna
Comptuer Science, North Carolina State University, USA
i.m.ralk@gmail.com

***Abstract*—The primary motivation of much of software analytics is decision making. How to make these decisions? Should one make decisions based on lessons that arise from within a particular project? Or should one generate these decisions from across multiple projects? This work is an attempt to answer these questions. Our work was motivated by a realization that much of the current generation software analytics tools focus primarily on prediction. Indeed prediction is a useful task, but it is usually followed by "planning" about what actions need to be taken. This research seeks to address the planning task by seeking methods that support actionable analytics that offer clear guidance on *what to do*. Specifically, we propose XTREE and BELLTREE algorithms for generating a set of actionable plans within and across projects. Each of these plans, if followed will improve the quality of the software project.**

***Keywords*—Data mining, actionable analytics, bellwethers, defect prediction.**

I. Introduction

Over the past decade, advances in AI have enabled a widespread use of data analytics in software engineering. For example, we can now estimate how long it would take to integrate the new code [1], where bugs are most likely to occur [2], or amount of effort it will take to develop a software package [3], etc. Despite these successes, there are two primary operational shortcomings with many software analytic tools: (a) conclusion instability as a result of constant influx of new data; and (b) lack of insightful analytics.

In several applications where local data is scarce, researchers use transfer learning. They report that the use of data from other projects can yield comparable predictors to just using local data [4]. However, new projects are constantly being created. Rahman et al. [5] caution that if quality predictors are always being updated based on the specifics of new data, then those new predictors may suffer from over-fitting. Such over-fitted models are "brittle" in the sense that they can undergo constant changes whenever new data arrives and lead to unstable conclusions. Conclusion instability is unsettling for software project managers struggling to find general policies. We require methods to support managers, who seek stability in their conclusions, while also allowing new projects to take full benefit from data arriving from all the other projects. Our research [6] has offered strong evidence that organizations can declare some prior project as the *"bellwether"*[1] that can then offer predictions that generalize across N other projects.

In addition to unstable conclusions, business users also lament that most software analytics tools, "Tell us what *is*. But they don't tell us *what to do*". A concern that was also raised by several researchers at a recent workshop on "Actionable Analytics" at 2015 IEEE conference on Automated Software Engineering [7]. For example, most software analytics tools in the area of detecting software defects are mostly *prediction* algorithms such as Support Vector Machines, Naive Bayes, Logistic Regression, Decision Trees, etc [8]. These prediction algorithms report what combinations of software project features predict for the number of defects. But this is different task to *planning*, which answers a more pressing question: what to *change* in order to *reduce* these defects. Accordingly, in this research, we seek tools that offer clear guidance on what to do in a specific project.

The tool assessed in this paper is the XTREE *planning* tool [9]. XTREE employs a $cluster + contrast$ approach to planning where it (a) *Clusters* different parts of the software project based on a quality measure (e.g. the number of defects); (b) Reports the *contrast sets* between neighboring clusters. Each of these contrast sets represent the difference between these clusters and they can be interpreted as plans, i.e.,

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

XTREE uses data from within a software project to generate plans. But, in several cases local data may not readily available. To overcome this limitation, we incorporate our findings from bellwethers to extend XTREE to use the bellwether projects. We call this tool BELLTREE and we show that it can be used to generate *stable* plans for cross-company planning.

II. Contributions of this work

1. New kinds of software analytics techniques: This research introduces the notion of planning in software engineering. In addition to showing that planning in effective in a within-project setting [9], we also show that with bellwethers [6], plans can be generated for cross-project problems with encouraging results. This is a unique approach that combines our efforts to address the problems highlighted in §I.

2. Compelling results of planning: Our results have established that planning is quite successful in producing actions that can reduce the number of defects. In Figure 2, we show that planning can reduce defects by more than 40% in 3 out of the 4 datasets studied here (>80% in the certain cases).

3. Evidence of generality of bellwethers: The more the bellwether effect is explored, the more we learn about its broad applicability. Originally, we explored this just in the context of defect prediction [6], it has now been shown to work also in effort estimation, predicting when issues will close, and detecting code smells [10]. Our preliminary results reported in this work show that bellwethers can also be used for cross-project planning with the use of BELLTREE. This is an important result of much significance since, where bellwethers occur, reasoning about multiple software projects becomes a

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

simple matter of discovering bellwethers (see [6]).

4. Replication Package: For readers this work who wish to replicate our findings, we have made available a replication package at https://git.io/v7c9k.

III. Related Work

Planning has been a subject of much research in artificial intelligence. Here, planning usually refers to generating a sequence of actions that enables an *agent* to achieve a specific *goal* [11]. 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 [12]. Some of the most common planning paradigms include: (a) classical planning [13]; (b) probabilistic planning [14], [15], [16]; and (c) preference-based planning [17], [18].

Existence of a model precludes the use of each of these planning approaches. This is a limitation of all these planning approaches since not every domain has a reliable model. In software engineering, the planning problem translates to proposing changes to software artifacts. Solving this has been undertaken via the use of some search-based software engineering techniques [19]. Examples of algorithms include SWAY, NSGA-II, MOEA/D, etc. [20], [21], [22].

These search-based software engineering techniques require access to some trustworthy models that can be used to explore novel solutions. In some software engineering domains there is ready access to such models which can offer assessment of newly generated plans. Examples of such domains within software engineering include automated program repair [23], [24], software product line management [25], [26], etc.

However, not all domains come with ready-to-use models. For example, consider software defect prediction and all the intricate issues that may lead to defects in a product. A model that includes *all* those potential issues would be very large and complex. Further, the empirical data required to validate any/all parts of that model can be hard to find. Also, even when there is an existing model, they can require constant maintenance lest they become out-dated. These problems are the key motivations for us to look for alternate methods for planning that can be automatically updated with new data without a need for comprehensive models.

In summary, for domains with readily accessible models, we recommend the tools widely used in the search-based software engineering community such as SWAY, NSGA-II, MOEA/D, etc. In 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 of use observable states from this data to generate an estimation of the model. Our preferred tools in this paper XTREE and BELLTREE take this approach and as presented elsewhere in this paper, these methodologies have very encouraging results.

IV. Planning in Software Engineering

A. 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:

$$out = f(in)$$

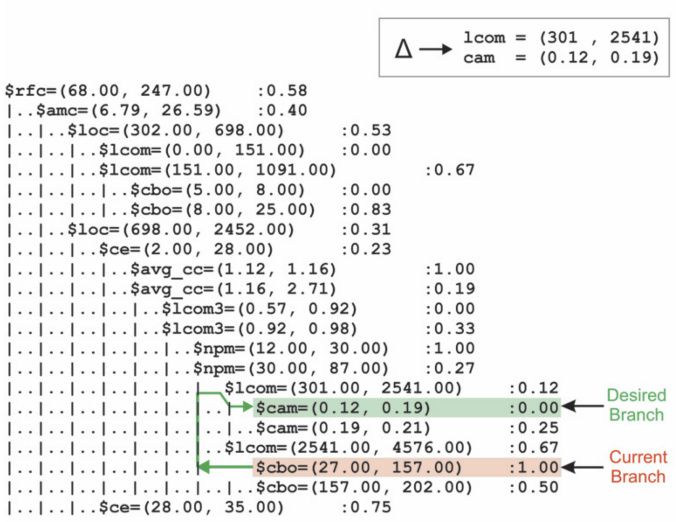


Fig. 1: Generating thresholds using XTREE.

where *in* contains many independent features and out contains some measure of how many defects are present. For software analytics, the function f is learned via data mining (with static code attributes for instance). Contrary to this, quality planning generates a concrete set of actions that can be taken (as precautionary measures) to significantly reduce the likelihood of defects occurring in the future. For a formal definition of plans, consider a test example Z, planners proposes a plan D to adjust attribute Z_j as follows:

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

With this, to (say) simplify a large bug-prone method, our planners might suggest to a developer to reduce its size (i.e. refactor that code by splitting it simpler functions).

B. XTREE

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

The specifics of the algorithm used to divide the data and construct the decision tree were presented in greater detail in our previous work [9]. Next, XTREE builds plans from the branches of the decision tree by asking the following three questions for each test case (the last of which returns the plan):

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

As a motivating example, consider Figure 1 with XTREE constructed with training data consisting of OO code metrics [27] and associated defect counts. A defective test case with the same code metrics is passed into the tree and evaluated down the tree to a leaf node with a defect probability of 1.0 (see the **orange** line in Figure 1). XTREE then looks for a nearby leaf node with a lower defect probability (see the green line in Figure 1). XTREE then evaluates the differences (of *deltas*) between green and **orange**. These *deltas* represent the threshold ranges[2] that represent the plans to reduce the defects.

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

C. BELLTREE

BELLTREE is structurally similar to XTREE. It differs in the source of data used for analytics. While XTREE uses data from within the project, BELLTREE first starts by looking for the bellwether dataset. To do this, we employ the strategy discussed in our previous work [6]. This helps in identifying a bellwether dataset. Once the bellwethers are discovered, we construct a supervised decision tree similar to XTREE. Plans are generated by using the same procedure as §IV-B. Note that the use of bellwethers enables BELLTREE to leverage data from across different projects. This presents a novel extension to XTREE.

V. Research Questions

RQ1. How prevalent are bellwethers? It is important to establish the prevalence of bellwethers first as this determines if it is possible to learn plans from the bellwether data. If bellwethers occur infrequently, we cannot rely on them for planning. We have initially shown that bellwethers are prevalent in defect prediction [6]. Further evidence was seen in [10], where we explored three additional sub-domains within software engineering namely, defect prediction, effort estimation, issue lifetime estimation, and detection of code smells. In a result consistent with bellwethers being *very* prevalent, we found that all these domains have a bellwether dataset.

RQ2. Does within-project planning with XTREE offer significant improvements in reducing defects? This research question seeks to establish if our preferred planning tool (XTREE) is effective in generating actionable plans in a within-project setting. Our initial findings showed that XTREE was indeed an effective planner that can generate plans that are also succinct and stable. Further, these plans are not subject to conjunctive fallacy [9].

RQ3. Does cross-project planning with BELLTREE offer significant improvements in reducing defects? Having established the prevalence of bellwether datasets and the efficacy of planning with XTREE, here we ask if it is possible for us to transfer plans across projects using the bellwether data and XTREE (referred to as BELLTREE). Our preliminary results are very encouraging. We show that BELLTREE can be a very effective cross-project planner.

RQ4. Are cross-project plans any better than within project plans? This research question assesses the quality of plans obtained using XTREE and BELLTREE. This is important because within-project data is not always available (especially if a project is in it's early stage of development) and it may be useful to look to other similar projects for planning. Our preliminary results have suggested that the effectiveness of plans generated from within project data and XTREE is statistically comparable to plans derived with cross-project data and BELLTREE. Thus, when project specific data is not available, one may use cross-project data to derive plans.

VI. Evaluating Plans

To evaluate plans, we propose the use of a *verification oracle* [9]. Oracles have been commonly used by several SE researchers such as Cheng et al [28], O'keefe et al. [29], Mkaouer et al. [30]. They use an oracle that is learned separately from the planner. The verification oracle 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 [31]. A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [32].

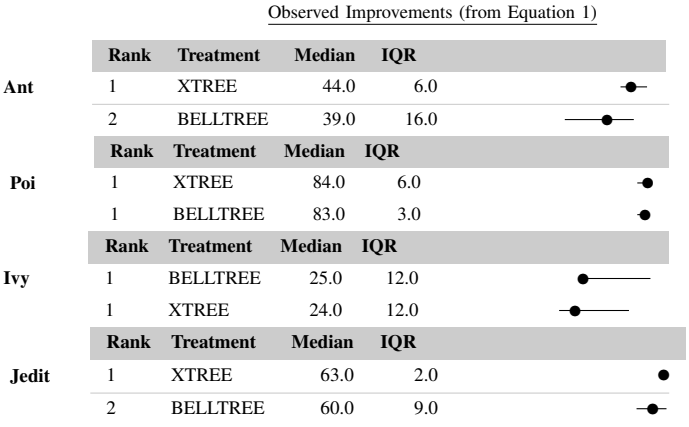


Fig. 2: Results comparing XTREE trained on local datasets and BELLTREE. Results from 30 repeats. Values come from Eq. 1. Values near 0 imply no improvement, *Larger* median values are *better*.

Hence, for this study, we eschew older quality models like QMOOD. Instead, we use Random Forests [33] to learn defect predictors from OO code metrics. Unlike QMOOD, the predictors are specific to the project. Additionally, classifiers such as Random Forest have shown to be very efficient in detecting bugs [34].

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

> *The verification oracle should be built with completely different data to the planner.*

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

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

The value of the measure R has the following properties:

- If $R = 0\%$, this means "no change from baseline";
- If $R > 0\%$, this indicates "improvement over the baseline";
- If $R < 0\%$, this indicates "optimization failure".

Ideally, an effective planner should have an improvement of $R > 0$, where larger values indicate better performance.

VII. Current State and Future Work

As mentioned earlier in the paper, this work represents our efforts to address to key issues in modern software analytics: (a) conclusion instability; and (b) generating insightful analytics. To this end, we undertook two concurrent research efforts to address each of these issues.

While attempting to stabilize the pace of conclusion change, we discovered the bellwether effect [6]. Our results provided evidence that it is possible to *slow* the pace of conclusion change in software analytics (for defect prediction models) using bellwethers. Further exploration demonstrated that the so called *bellwether effect* is quite prevalent in several sub-domains of software engineering such as code-smell detection, effort estimation, and estimation of issue lifetimes [10].

In order to generate actionable analytics for software engineering, we developed the XTREE planner [9]. Initial motivation for XTREE was to address the varied opinions in literature on how best to undertake code reorganization so as to reduce bad smells. We showed that by leveraging historical logs of data, planners such as XTREE can offer actionable recommendations on how to undertake code reorganization in order to reduce defects in code. Further, we showed that in addition to generating effective plans, XTREE recommends of far fewer changes. Thus making it a better framework for critiquing and rejecting many of the code reorganizations.

The initial version of XTREE was limited to using data from within a project to generate plans. This paper represents our initial attempts to transfer plans from across other projects to a test project. For this purpose, we developed BELLTREE. It uses the same framework as XTREE but uses bellwethers as the source of data for planning. Our results comparing BELLTREE with XTREE on a set of open source java projects is shown in Figure 2. In two of the four datasets, we note that BELLTREE performed just as well as XTREE and two other cases XTREE outperformed BELLTREE (but not by a significant amount). Our initial finding is that if within-project data from previous releases are available, we may use XTREE. If not, using bellwethers would be a reasonable alternative.

Our initial results of using BELLTREE are encouraging and deserves much further exploration. Starting early this summer, we have deployed an enhanced version of XTREE on-site in conjunction with our industrial partners with the following goals: (1) Qualitatively validate the usefulness of the plans; (2) Establish the receptiveness of developers actively using our tool; and (3) Solicit developers' feedback on usefulness of plans generated by XTREE.

REFERENCES

[1] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, march 2011, pp. 357 –366.

[2] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, 2007.

[3] B. Turhan, A. Tosun, and A. Bener, "Empirical evaluation of mixed-project defect prediction models," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, 2011, pp. 396–403.

[4] F. Peters, T. Menzies, and L. Layman, "LACE2: Better privacy-preserving data sharing for cross project defect prediction," in *Proceedings - International Conference on Software Engineering*, vol. 1, 2015, pp. 801–811.

[5] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 61:1–61:11.

[6] R. Krishna, T. Menzies, and W. Fu, "Too much automation? the bellwether effect and its implications for transfer learning," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016, pp. 122–131.

[7] J. Hihn and T. Menzies, "Data mining methods and cost estimation models: Why is it so hard to infuse new ideas?" in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, Nov 2015, pp. 5–9.

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

[9] R. Krishna, T. Menzies, and L. Layman, "Less is more: Minimizing code reorganization using XTREE," *Information and Software Technology*, mar 2017.

[10] R. Krishna and T. Menzies, "Simpler Transfer Learning (Using "Bellwethers")," *TSE (under review)*, pp. 1–18, mar 2017. [Online]. Available: http://arxiv.org/abs/1703.06218

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Egnlewood Cliffs, 1995.

[12] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[13] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.

[14] R. Bellman, "A markovian decision process," *Indiana Univ. Math. J.*, vol. 6, pp. 679–684, 1957.

[15] E. Altman, *Constrained Markov decision processes*. CRC Press, 1999, vol. 7.

[16] X. Guo and O. Hernández-Lerma, "Continuous-time markov decision processes," *Continuous-Time Markov Decision Processes*, pp. 9–18, 2009.

[17] T. C. Son and E. Pontelli, "Planning with preferences using logic programming," *Theory and Practice of Logic Programming*, vol. 6, no. 5, pp. 559–607, 2006.

[18] S. S. J. A. Baier and S. A. McIlraith, "Htn planning with preferences," in *21st Int. Joint Conf. on Artificial Intelligence*, 2009, pp. 1790–1797.

[19] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," *Dept. Comp. Sci, Kings College London, Tech. Rep. TR-09-03*, 2009.

[20] V. Nair, T. Menzies, and J. Chen, "An (accidental) exploration of alternatives to evolutionary algorithms for sbse," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 96–111.

[21] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2002.

[22] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization," in *Evolutionary Methods for Design, Optimisation, and Control*. CIMNE, Barcelona, Spain, 2002.

[23] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings - International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[24] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, dec 2015.

[25] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013.

[26] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 517–528.

[27] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[28] B. Cheng and A. Jensen, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 1341–1348.

[29] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring: An empirical study," *J. Softw. Maint. Evol.*, vol. 20, no. 5, pp. 345–364, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1002/smr.v20:5

[30] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 331–336. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642965

[31] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.979986

[32] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus Global Lessons for Defect Prediction and Effort Estimation," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 822–834, jun 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6363444/

[33] L. Breiman, "Random forests," *Machine learning*, pp. 5–32, 2001.

[34] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: is it really necessary?" *Information and Software Technology*, 2016.

Summary of changes:

The initial version was a 4-page abstract presented at ASE 2015 doctoral symposium highlighting the possibility of cross-project planning.

After soliciting advice from the panel at the doctoral symposium (esp. regarding our evaluation strategy), we undertook this work to demonstrate the cross-project planning was possible and that it is quite effective.