How to Learn Useful Changes to Software Projects (to Reduce Runtimes and Software Defects)

Rahul Krishna
Tim Menzies,
Xipeng Shen
CS, NC State, USA
{i.m.ralk, tim.menzies, xipengshen}@gmail.com

Andrian Marcus
CS, UT Dallas
Texas, USA
amarcus@utdallas.edu

Naveen Lekkalapudi
Bellhops
Tennessee, USA
navek91@gmail.com

Lucas Layman
Fraunhofer CESE
College Park, USA
llayman@fc-md.umd.edu

ABSTRACT

Business users now demand more insightful analytics; specifically, tools that generate "plans"– specific suggestions on what to change in order to improve the predicted values.

This paper proposes XTREE, a planner for software projects. XTREE receives tables of data with independent features and a corresponding weighted class which indicates the quality ("bad" or "better") of each row in the table. Plans are edits to the rows which ensures the changed row is more likely to be of a "better" quality. XTREE learns those plans by building a decision tree across the data, then reporting the differences in the branches from some current branch to another desired branch. Using data from 11 software projects, XTREE can find better plans compared to three alternate methods. Those plans have lead to improvements with a median size of (56%, 28%) and largest size of (60%, 77%) in (defect counts, runtimes), respectively.

Keywords

Defect prediction; Configuration; Planning; Decision trees.

1. INTRODUCTION

Business users are demanding tools that support business-level interpretations of their data. At a panel on software analytics at ICSE'12, industrial practitioners lamented the state of the art in software analytics [1]. Panelists commented "prediction is all well and good, but what about decision making?". Note that these panelists were more interested in the interpretations and follow-up that occurs after the mining, rather than just the mining itself. So:

- Instead of just accepting *predictions* on how many software defects to expect, business users might now demand a *plan* to reduce the likelihood of those defects.
- Instead of just accepting *predictions* on the runtime time of their software, business users might now demand a *plan* to reduce that runtime.

In response to this business-level demands for planners, we propose a novel *planning* method called XTREE for learning changes to a software system such that its performance "improves", according to some measure. This paper uses XTREE to reduce the expected value of the defects in Jureczko et al.'s JAVA systems [2]; and the runtimes in software configured by Siegmund et al. [3].

The contributions of this paper are (1) the new XTREE algorithm and (2) an evaluation strategy that shows XTREE performing significantly "better" than planners proposed in our prior work [4, 5], where "better" means *effective* (plans change the expected values of the class); *succinct* (it is inconvenient if plans always require changing everything); and *surprising* (a planner should sometimes tell us things we do not expect since, otherwise, there is no value added from using the planner).

The rest of this paper describes our data, our planners, and the experiment that ranks XTREE against alternate approaches. This is followed by notes on related work and validity. To enable reproducibility, all scripts and data used in this study are available online at `http://git.io/vG3DG`.

ISCE '16 Austin, Texas

2. PRELIMINARIES

2.1 What is a "Plan"?

Our planners use tables of data with independent features and a dependent class feature. Classes have weights that indicate what rows are "bad" or "better". Plans change a row such that it is more likely to be "better". Specifically, for every test example Z, planners proposes a plan Δ to adjust feature Z_j:

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

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

Note that we make no assumption that a plan mentions every feature (so plan1 can be more succinct than plan2 when plan1 mentions fewer features than plan2).

2.2 From Prediction to Planning

This paper is about the next step *after* prediction. Suppose a business user is presented with a prediction and they do not like what they see; e.g. the runtimes are too long or the number of defects is too high. This user may then ask a *planning* question; i.e. "what can we change to do better than that?".

Before exploring automatic methods to answer the planning question, we first comment on two manual methods.

One way to propose changes to a project would be to ask some smart experienced person for their opinion on how to (e.g.) reduce defects and/or decrease runtimes. Sometimes such advice is an ef-

fective strategy and sometimes it is not. According to Passos et al. [6], developers may 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" [6]. Jørgensen & Gruschke [7] 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. [7]. Accordingly, we propose a "trust, but verify" approach. After a software guru offers some sage wisdom, it is wise to ask some other oracle if there are any better options (just as a sanity check). The rest of this paper discusses some methods to build automatic oracles to implement that sanity check.

Another way to find changes to a project might be to rely on the peer review processes used by the SE research community. This approach would propose changes to software projects that concur with internationally accept best practices. There are two problems with that approach. Firstly, given the rapid pace of change in software engineering, we may be asking questions for which there is no current widely accepted concept of "best practice".

Secondly, given the diversity of SE products and practices and personnel, it may well be that the current project being discussed is substantively different to prior work. Numerous recent *local learning* results compare (1) single models learned from all available data to (2) multiple models learned from clusters within the data [8–14]. A repeated result in those studies is that the local models generated better effort and defect predictions (better median results, lower variance in the predictions). §5.2.3 of this paper offers yet another locality result:

- One standard rule in the literature is that it is useful to implement modules such that they are internally cohesive (use much of their own local methods) while being loosely coupled with other classes [15].
- While that may be true in general, for particular classes other changes may be more important (later in this paper, we show one set of results where that is indeed the case).

In summary, it is useful to have automatic methods to recommend changes. Such methods can fill in for human expertise (if such experts are absent) or to offer a second opinion. Also, prior to making automatic recommendations, it is wise to first stratify the data (clump it into related examples) then generate advice specific to each clump. Accordingly, the rest of this paper defines and evaluates automatic methods to find plans from N examples divided into many clumps.

2.3 Trusting the Changes

XTREE is evaluated by comparing predicted performance scores before and after a planner makes changes to the feature values of an example: After making those changes, we may have a new example that has never been seen before. Therefore, it must be asked *"can we trust the predictions made on such new examples?"*

To answer this question, we note that data miners explore two "clouds" of data: (1) the cloud of training examples and (2) the cloud of test examples (for a visualization of these clouds, see Figure 2). We should mistrust the predictions made by a model if it is being applied to examples that are too far away from the training cloud. To test for "too far", we can run a data mining experiment that tests how well a model learned from the training data applies to the test data. Such experiments return some performance value.

Note that predictions about changes that fall within the space of the training+test data, will be at least as accurate as the predictions

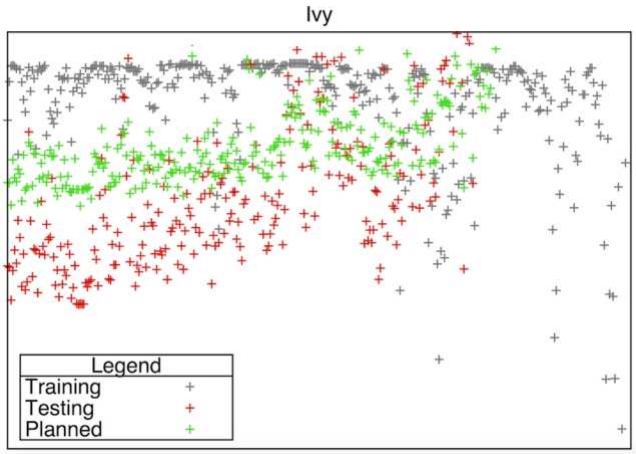

Figure 2: Gray, red, green show (1) training examples, (2) test examples and (3) tests that have been altered by planners. This figure uses axes generated from the first two components of a PCA analysis of all points.

of the original test data. With this, we assert that the predictions for changes that move examples towards/away from the training data can be trusted more/less (respectively).

Accordingly, we need *trust-increasing* planners to generate new examples *closer* to the training examples. To see how this works, Figure 2 is from the *ivy* data set (one of the Jureczko data sets used in this paper). It shows: (1) the training examples in gray, (2) the test examples in red, and (3) the changed examples displaced after applying a plan (in green). Note that the the changed examples cases (shown in green) fall closer to the training cases (shown in gray) than the test cases (shown in red).

In that green region of changed examples, our belief in the value of predictions will be just as much as, if not more than, our belief in the value of the predictions in the red region (that contains the original test data). This pattern of Figure 2 (where the new examples are found closer to the training cases than the test cases) has been observed in all the other data sets studied in this paper. Hence, we can assert that predictors learned from these training examples have some authority in the regions containing the changes examples.

That said, the above comes with some important caveats:

- The quality of the prediction depends on the nature of the training data. Thus, we strongly recommend that both the data set and the predictor be assessed prior to planning. This ensures that the predictor's performance is adequate for a data set. We tackle this issue in detail in §3.
- Planners should be designed to be *trust increasing*. We list four such planning methods in §4.
- Where possible, planners should be assessed via some external oracle that can accurately assess new examples. For an example of that kind of analysis, see §6.3.

3. TEST DATA

To assess our planning methods, our data (see git.io/vGYxc) comes from Jureczko et al.'s object-oriented JAVA systems [2] and software system configuration data from by Siegmund et al. [3].

The Jureczko data records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of nearly two dozen metrics such as number of

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

Figure 1: OO measures used in our defect data sets. Last lines, shown in denote the dependent variables.

data set	cases	% defective
ant	947	22
camel	1819	19
jedit	1257	2
ivy	352	11
log4j	244	92
lucebe	442	59
poi	936	64
synapse	379	34
velocity	410	34
xalan	2411	99
xerces	1055	74

Figure 3: Jureczko data: columns in the format of Figure 1.

Project	Domain	Lang.	LOC	Features	Config
BDBC: Berkeley DB	Database	C	219,811	18	2560
BDBJ: Berkeley DB	Database	Java	42,596	32	400
Apache	Web Server	C	230,277	9	192
SQLite	Database	C	312,625	39	3,932,160
LLVM	Compiler	C++	47,549	11	1024
x264	Video Enc.	C	45,743	16	1152

Figure 4: Siegmund data.

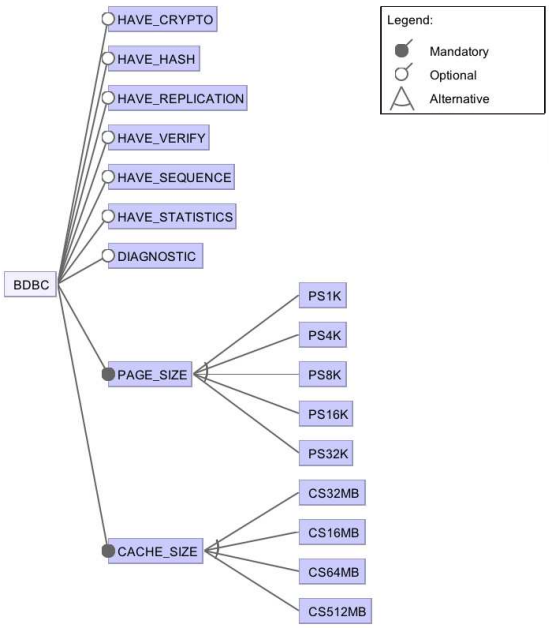

Figure 5: Berkeley database feature model ("C" version).

children (noc), lines of code (loc), etc. For details on the Jureczko data, see Figure 1 and Figure 3. All the planning methods of this paper reflect on the numeric value of the raw defect counts. The predictor considers defects to be a Boolean class data set, where defects are `TRUE` if the numeric defect count is greater than zero and `FALSE` otherwise.

The Siegmund data, described in Figure 4, records the runtimes of compiled systems. To obtain the data, Siegmund et al. perturbed the configuration parameters in the Makefiles of six systems: Apache, SQLite, LLVM, x264 and two versions of the Berkeley database (one written in "C" and one in Java). Then, the performance was measured using standard benchmarking tools (delivered by ORACLE for Berkley data sets and other popular tools such as AUTOBENCH and HTTPREF for the rest) [3]. It's worth noting that in each of the above data sets, several features are interdependent, this is expressed using a *feature model*. Figure 5 shows an example of such a feature model defining valid combinations of settings in the Berkley database ("C" version). These feature models were used by Siegmund et al. to ensure all their perturbations are valid (we will use the same models to cull invalid plans).

Our evaluation strategy (discussed below) divides this data into a training and a testing set. From the training set we apply a data miner (to learn a quality predictor) and various planning methods (to learn different plans). Next, we try applying each of those plans to the test set and ask the quality predictor to assess the changed examples. Finally, we say that the "best" planner is the one that most reduces the predicted values in the changed examples.

As mentioned in the last section, this approach depends on having effective predictors for assessing the results. For the Siegmund data, this criteria was relatively easy to achieve. The data in those data sets have a continuous class (runtime of the compiled system) so the performance of a quality predictor can be measured in terms

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

Figure 6: Training and test *data set properties* for Jureczko data , sorted by % defective examples. On the right-hand-side, we show the *results from learning*. Data is usable if it has a recall of 60% or more and false alarm of 30% or less (and note that, after tuning, there are more usable data sets than before). Results marked with "⋆" show large improvements in performance, after tuning (lower *pf* or higher *pd*). Data in the three bottom rows, marked with "×", are performing poorly– that data so few non-defective examples that it is hard for our learners to distinguish between classes.

of difference between the predicted runtime p of test case items and their actual runtimes a using $s = (1 - \frac{abs(a-p)}{a}) \times 100\%$ (*higher* values are *better*). This paper explores six Siegmund configuration data sets: Berkeley DB (Java and C versions), Apache, SQLite, LLVM, and x264. As a preliminary study, we split that data into equal sized train:test groups and trained a Random Forest Regressor (from the SciKit learn kit [16]) on one half, then applied to the other. After repeating this process 40 times, we achieved nearly perfect accuracy of s ={98.45%, 99.62%, 94.86%, 98.89%, 98.60%, 99.87%}; i.e. we are confident that the predictors from the Siegmund data can assess our plans. (Aside: if the reader doubts that such high scores are achievable, we note that these scores are consistent with those achieved by predictors built by Siegmund et al. [3].)

It proved to be more complicated to commission the Jureczko data sets for this study. For that data, we found that the quality predictors built from this data are far from perfect; However, for some data sets, the predictors could be salvaged using the techniques discussed in this section.

Figure 6 shows our preliminary studies with the Jureczko data. Given access to V released versions, we test on version V and train on the available data from $V - 1$ earlier releases (as shown in Figure 6, this means that we are training on hundreds to thousands of classes and testing on smaller test suites). Note the three bottom rows marked with ×: these contain predominately defective classes (two-thirds, or more). In such data sets, it is hard to distinguish good from bad (since there are so many bad examples).

In order to identify the presence (or absence) of defects, we can consider using Boolean classes in the Jureczko data (`True` if defects $>$ 0; `False` if defects = 0). For such data, quality of the predictor can be measured using (1) the probability of detection (a.k.a. "pd" or recall): the percent of faulty classes in the test data detected by the *predictor*; and (2) the probability of false alarm (a.k.a. "pf"): the percent of non-fault classes that are *predicted* to be defective.

As a preliminary study, we split the Jureczko data into train and test groups. Random Forest (again, from the SciKit learn kit [16]) was built from the training data, then applied to the test data. The "untuned" columns of Figure 6 shows those results. We called a data set "usable" if Random Forest was able to classify majority of the instances correctly. For this purpose, we set a threshold of $pd \geq 60 \wedge pf \leq 30\%$ to select suitable data sets. With this threshold, however, none of our data sets were be suitable for this study.

Fortunately, the "tuned" columns of Figure 6 show that we can salvage some of the data sets. Pelayo and Dick [17] report that defect prediction is improved by SMOTE [18]; i.e. an over-sampling of minority-class examples and an under-sampling of majority-class examples. Also, Fu et al. [19] report that parameter tuning with differential evolution [20] can quickly explore the tuning options of Random Forest to find better settings for the (e.g.) size of the forest, the termination criteria for tree generation, etc. The rows marked with a ⋆ in Figure 6 show data sets whose performance was improved remarkably by these techniques. For example, in *poi*, the recall increased by 4% while the false alarm rate dropped by 21%. However, as might have been expected, we could not salvage the data sets in the three bottom rows.

In summary, while we cannot trust predictors from some of our Jureczko data sets, we can plan ways to reduce defects in *jedit, ivy, ant, lucene* and *poi*. Accordingly, when this study explores the Jureczko data, we will use these five data sets.

(Aside: One important detail to be stressed here is that, when we applied SMOTE-ing and parameter tunings, those techniques were applied to the training data and *not* the test data; i.e. we took care that no clues from the test set were ever used in this tuning process.)

4. FOUR PLANNING METHODS

This section describes XTREE (which we call Method4) and three alternate methods for learning plans. XTREE, a novel planner introduced in this work, uses the decision tree learner of Figure 7.D. The other methods use the top-down bi-clustering method described in Figure 7.C which recursively divides the data in two using a dimension that captures the greatest variability in the data. We proposed Method 1 and 2 in 2012 [4] while Methods 3 comes from research conducted earlier this year [5]. All methods have the properties proposed in §2: (1) they are *local learners* that find plans particular to each test case; and (2) they are *trust-increasing*; i.e. the changed examples are moved *closer* to the training data.

4.1 Methods

Our description of the methods adopts the following convention. All variables set via our engineering judgement with Greek letters;

e.g. $\alpha, \beta, \gamma, \omega$. In this paper, we show our current settings to these variables produces useful results. Elsewhere [19, 21], we are exploring tuning methods to find better settings but we have nothing definitive yet to report on auto-tuning planners.

4.1.1 Method1= CD= Centroid Deltas

Summary1: Method1 computes a plan from the difference between where you are (which we will call C_i) and where you want to be (which we will call C_j).

Assumption1: Large data sets can be adequately represented by a few dozen (or so) centroids.

Details1: Method1 clusters project data by reflecting on the independent variables, then reports the delta between the cluster centroids. After clustering training data using the WHERE algorithm of Figure 7.C, Method1 replaces all clusters with a centroid C_i computed from the median value of each continuous/discrete feature. Then, it finds the closest centroid C_j that has a better performance score. For Jureczko data, "better" means fewer defective examples while for the Seigmund data, "better" means lower median runtimes for the examples in that cluster. Method1 then caches the delta between the independent features between C_i and C_j. For continuous features, this delta is $C_j - C_i$. For discrete values, this delta is the value of that feature in C_j. Finally, for every test case, Method1 use the distance measure d shown in Figure 7.B to find the nearest centroid C_i. It then proposes a plan for improving that test case that is the conjunction of all the deltas between C_i and C_j.

4.1.2 Method2=CD+FS=Method1+Feature Selection

Summary2: Method2 works like Method1 but now the plans only mention the $\beta = 33\%$ most informative features. Hence, Method2's plans are simpler.

Assumption2: When reasoning about centroids, we can just use features that best distinguish centroids; i.e. whose values appear in just a few centroids.

Details2: A common result is that the signal in a table of data is mostly contained in a handful of features [22, 23]. Papakroni [24] has tested for this effect in the Jureczko data sets. Papakroni found no loss of efficacy in defect prediction after sorting all features by their information content, then making predictions using (a) all features or (b) just using 33% most informative features.

Based on the above, it might be possible to simplify the plans found by Method1 by pruning back the features in those plans. Following on from Papakroni, our Method2 returns plans containing just the top $\beta = 33\%$ most informative features. Here, "informative" means that the values of a feature are good for selecting a small set of clusters (ideally, just one). This can be estimated using the Fayyad-Iranni INFOGAIN algorithm [25] of Figure 7.E.

4.1.3 Method3= Best In Cluster (BIC)= Method1 + Gradient

Summary3: Method3 is like Method1, but it uses more knowledge about the training data.

Assumption3: There exists "gradients" between clusters which, if used, will better guide us to finding beneficial plans.

Details3: Method3 generates clusters from the training data just like Method1. Following this, it summarizes the clusters C_X using "Best-In-Cluster" B_X, which is (1) the centroid of the cluster for the Jureczko data; (2) the cluster's member with the fastest runtime for the Seigmund data. Following this, Method3 connects each cluster C_i to a nearest neighbor C_j by a *gradient*. Each gradient has a (bottom,top) end labelled (C_i, C_j) containing the (worst,best) performance scores, respectively.

For each test instance, Method3 finds the nearest gradient, runs

---

Figure 7.A: Measuring Variability
For continuous and discrete values, the *variability* can be measured using standard deviation σ or entropy e. Note that $\sigma = \sqrt{\sum_{i=1}^{n} \left(\frac{(x_i - \bar{x})^2}{n-1} \right)}$ where $\bar{x}$ is the mean of numeric features $x_1, x_2, ..x_n$. Also $e = -\sum_{i=1}^{n} p_i log_2(p_i)$ for n discrete values at frequency $f_1, f_2, ..f_n$ for $N = \sum_{i=1}^{n} f_i$ and $p_i = f_i/N$.

Figure 7.B: Measuring distance
We use Aha et al.'s standard Euclidean distance measure [26]. For F independent features, the measure returns $d(X, Y) = \sqrt{\sum_i^F w_i \Delta(X_i, Y_i)^2}$. Here, w_i is a weight term for each feature (usually set to 1). Within Δ, if X_i, Y_i are both missing values, then return 1. Otherwise, replace any missing items with values that maximizes the following. For numerics, Δ normalizes X_i, Y_i (to the range 0,1 for min,max) then returns $X_i - Y_i$. For discrete variables, Δ returns 0,1 if X_i, Y_i are the same,different (respectively).

Figure 7.C: Top-down Clustering with WHERE
WHERE divides data into groups of size $\alpha = \sqrt{N}$. Using this measure, WHERE runs as follows:

1. Find two distance cases, X, Y by picking any case W at random, then setting X to its most distant case, then setting Y to the case most distant from X (which requires only $O(2N)$ comparisons).
2. Project each case Z to position x on a lines running from X to Y: if a, b are distances Z to X, Y then $x = (a^2 + c^2 - b^2)/(2ac)$.
3. Split the data at the median x value of all cases.
4. For splits larger than $\alpha = \sqrt{N}$, recurse from step1.

In terms of related work, the above is similar in approach to Boley's PDDP algorithm [27], but PDDP requires an $O(N^2)$ calculation at each recursive level to find the PCA principle component. Our method, on the other hand, performs the same task with only $O(2N)$ distance calculations using the FASTMAP heuristic [28] shown in step1. Platt [29] notes that FASTMAP is a Nyström approximation to the first component of PCA.

Figure 7.D: Top-down division with Decision Trees
Find a split in the values of independent features that most reduces the variability of the dependent feature (measured using Figure 7.A). Construct a standard decision tree using these splits.

Figure 7.E: Finding the most informative rows
Discretize all numeric features using the Fayyad-Iranni discretizer [25] (divide numeric columns into bins B_i, each of which select for the fewest cluster ids). Let feature F have bins B_i, each of which contains n_i rows and let each bin B_i have entropy e_i computed from the frequency of clusters seen in that bin (computed from Figure 7.A). Cull the the features as per [24]; i.e. just use the $\beta = 33\%$ most informative features where the value of feature F is $\sum_i e_i \frac{n_i}{N}$ (N is the number of rows).

Figure 7: Some algorithms used in this paper.

Using the training data, divide the data using the decision tree algorithm of Figure 7.D into groups of size $\alpha = \sqrt{N}$. For test item, find the *current* leaf: take each test instance, run it down to a leaf in the decision tree. After that, find the *desired* leaf:

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

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

Figure 8: XTREE

$$\text{project data} \begin{cases} \textit{train} \begin{cases} \text{learn a predictor (e.g. via Random Forest)} \\ \text{learn a planner (e.g. via XTREE)} \end{cases} \\ \textit{test} \begin{cases} \textit{before} = \text{Performance scores in } \textit{test} \\ \textbf{if } \textit{before} > 0 \\ \textbf{then} \begin{cases} \textit{test}' = \text{planner}(\textit{test}) \\ \textit{after} = \text{predictor}(\textit{test}') \\ \textbf{return } R = (1 - \frac{\textit{after}}{\textit{before}}) \times 100\% \end{cases} \end{cases} \end{cases}$$

Figure 9: Experimental design .

up to the top (best) end C_j, then extracts B_j (which is the best-in-cluster associated with C_j). The returned plan is then computed as a fraction ($\omega = 0.5$) of the distance between the test case and it's corresponding B_j.

4.1.4 Method4=XTREE= Deltas in Decision Branches

Summary4: Method4 builds a decision tree, then generates plans from the difference between two branches: (1) the branch to where you are and (2) the branch to where you want to be.

Assumption4: One potential problem with Methods 1,2 and 3 is the *unsupervised* nature of the clustering algorithm (WHERE) that executes without knowledge of the target class. *Supervised* methods assume that it is useful to also reflect on the target class.

Details4: XTREE uses a supervised decision tree algorithm of Figure 7.D to divide the data. Next, XTREE builds plans from the branches of the decision trees using the code of Figure 8. That code asks three questions, the last of which returns the plan:

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

5. EXPERIMENTS

This section describes an experimental design (and results) for evaluating the above four methods.

5.1 Experimental Design

5.1.1 A Strategy for Evaluating Planners

Our experimental design is shown in Figure 9. We divide the project data into two disjoint sets *train* and *test* (so *train* $\cap$*test* $= \emptyset$). Next, from the train set, we build both a *planner* and a *predictor*.

Our general framework does not commit to any particular choice of planner or predictor but, for the purposes of this paper:

- Our *planner* will be one of Methods 1,2,3,4;
- Our *predictor* will be the Random Forest Classifier [30] (for discrete classes) and Random Forest Regressor (for continuous classes) taken from SciKit Learn [16]. We use these data miners since extensive studies have shown these to be amongst the better alternatives for mining software data [31].

As for the *test* data, this is passed to the predictor to measure performance statistics related to effectiveness.

If our predictors fail to perform effectively on the test data, then we cannot trust them to comment on our plans. Accordingly, if that performance is unsatisfactory, we abort. Recall from §3 that this step indicated we should not use some of the Jureczko data.

Else, we (1) apply the planner to alter the *test* data; then (2) apply the predictor to the altered data $test'$; then (3) return data on the *before, after* predictions expressed as percent improvement, denoted by $R = (1 - \frac{after}{before}) \times 100\%$, with the following 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".

5.1.2 Statistical Methods

Our methods use some stochastic algorithms; e.g. WHERE's selection of "what example to explore first" (see Figure 7.C) and XTREE' occasional use of a random guess when deciding what part of a discretized range to include in the plan (see Figure 8). Hence, we report the R values seen in 40 repeated runs, with different random number seeds (we use 40 since that is more than the 30 samples needed to satisfy the central limit theorem).

To rank our methods using the results from these 40 repeats, we use the Scott-Knott test recommended by Mittas and Angelis [32].

In accordance to that test, using the median values of each method, we sort a list of $l = 40$ values of R values found in $ls = 4$ different methods. Then, we split l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l, m, n of size ls, ms, ns where $l = m \cup n$:

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

We then apply a apply a statistical hypothesis test H to check if m, n are significantly different (in our case, the conjunction of A12 and bootstrapping). If so, Scott-Knott recurses on the splits. In other words, we divide the data if *both* bootstrap sampling and effect size test agree that a division is statistically significant (with a confidence of 99%) and not a small effect ($A12 \geq 0.6$).

For a justification of the use of non-parametric bootstrapping, see Efron & Tibshirani [33, p220-223]. For a justification of the use of effect size tests see Shepperd&MacDonell [34]; Kampenes [35]; and Kocaguenli et al. [36]. These researchers warn that even if a hypothesis test declares two populations to be "significantly" different, then that result is misleading if the "effect size" is very small. Hence, to assess the performance differences we first must rule out small effects using A12, a test recently endorsed by Arcuri and Briand at ICSE'11 [37].

5.1.3 Report Format

Our results are presented in the form of line diagrams like those shown on the right-hand-side of the following example table. The black dot shows the median R value and the horizontal likes stretch from the 25th percentile to the 75th percentile (a region called the inter-quartile range, or IQR).

	Rank	Treatment	Median	IQR
Example	1	XTREE	62	6
	2	BIC	50	12
	3	CD	44	18
	3	CD+FS	43	13

In this example table, the rows are sorted on the median values of each method. Note that all the methods have $R > 0\%$; i.e. all these methods reduced the expected value of the performance score in that experiment while XTREE achieved the greatest reduction (of 62% from the original value).

The above example table has a left-hand-side **Rank** column, computed using the Scott-Knott test described above. In this example table, XTREE is ranked the best, while CD and CD+FS together are ranked the worst.

5.1.4 Other Details

Figure 10 and Figure 11 show the effectiveness of our methods seen in 40 repeats with each data set. In these experiments, the dependent variables of Jureczko and Siegmund data sets are discrete and continuous in nature, respectively. Hence, while choosing the predictor, we used Random Forest (1) as a classifier for Jureczko data and (2) as a regressor for Siegmund data.

For Siegmund data, we performed a k-fold cross validation. Given the relatively small sample sizes in Apache and BDBJ, we chose $k = 2$. This, we reasoned, would maintain a sufficiently large test data size in order for R to be measured reliably. However, in 2-fold cross validation, the division of data tends to affect the outcome significantly. To counter this, we randomized the order of the data, training on one half while identifying treatment plans on the other, repeating the process 40 times as mentioned above.

The Jureczko data, being temporal in nature, allows us to implement a validation procedure that ensures only past data is ever used to predict future values. Hence, in that data, we used the train/test sets shown in Figure 6. (Aside: note that all the SMOTEing and Random Forest tunings (discussed in §3) occurred in the *train* phase of Figure 9).

5.2 Experimental Results

Recall from our introduction that we are assessing planners on three criteria: *effectiveness*, which is how much they reduce the expected value of the changed examples; *succinctness*, which is how many things we need to change to achieve a plan; and *surprise*, which is how different are the plans from standard truisms.

5.2.1 Effectiveness Results

Measured in terms of effectiveness, some data sets were harder to optimize than others. SQL (in Figure 11) defied all our methods for reducing runtimes. XTREE was the only method that could optimize BDBJ (in Figure 11). In general, in most data sets, large reductions were observed:

- An improvement of 60%, as compared to the original baseline in Ant and Lucene of Figure 10;
- An improvement of 77% as compared to the original baseline in BDBC of Figure 11;

Overall, XTREE was most the effective. It was always the top-ranked method with the exception of SQL. Where it ranked better, it had significant improvements in the median performance values (1) In the Jureczko data sets, there was a median improvement of around 10% larger than the next ranked method; (2) In the Seigmund data sets, improvements as large as 36% greater than the next ranked method were observed.

5.2.2 Succinctness Results

Figure 12 reports the percent of times in the 40 repeats that a method proposed changing a feature. The left-hand-side plot of that figure reports results from one of the Jureczko data sets (*lucene*) and the right-hand-side shows a Siegmund data set (*BDBJ*).

In these plots, *more* succinct a planning method, *fewer* the frequency (in percent) where it recommends changing a particular feature (i.e. the vertical bars in that plot are *lower*). For example, XTREE's plans were usually succinct – in all data sets, XTREE changes around a fifth of the features (see Figure 13). On other hand, Method1 (CD) was the least succinct since it wanted to change all features (observe the change frequencies as high as 100% for all features). Method1's policy of "change everything" might be acceptable if this approach lead to the most effective changes. However, in Figure 10 and Figure 11, there is no evidence for this.

An interesting feature of Figure 12 was that fewer things were changed in the Seigmund data sets like *BDBJ* than in the Jureczko

	Rank	Treatment	Median	IQR
Ant	1	XTREE	60	8
	2	CD	52	15
	3	CD+FS	45	1
	3	BIC	45	3
	Rank	**Treatment**	**Median**	**IQR**
Lucene	1	XTREE	60	11
	2	CD+FS	51	4
	2	CD	48	4
	3	BIC	37	2
	Rank	**Treatment**	**Median**	**IQR**
Poi	1	XTREE	56	1
	2	CD	44	16
	2	BIC	43	9
	2	CD+FS	40	5
	Rank	**Treatment**	**Median**	**IQR**
Ivy	1	XTREE	46	8
	2	BIC	40	2
	3	CD	20	5
	3	CD+FS	18	0
	Rank	**Treatment**	**Median**	**IQR**
Jedit	1	XTREE	55	1
	2	CD	45	9
	2	CD+FS	45	9
	2	BIC	45	0

Figure 10: Results on Jureczko data sets. Results from 40 repeats. Ratios of (1) number of examples with defects (expected in the test examples) after they have been altered by a planner to (2) the number of examples with defects in the original test set. *Larger* median values are *better*.

BDBJ

Rank	Treatment	Median	IQR
1	XTREE	43	13
2	BIC	7	5
3	CD+FS	0	1
3	CD	0	1

Apache

Rank	Treatment	Median	IQR
1	XTREE	28	11
2	BIC	5	4
3	CD	3	5
4	CD+FS	0	1

SQL

Rank	Treatment	Median	IQR
1	XTREE	1	3
1	BIC	0	0
1	CD	0	0
1	CD+FS	-1	0

BDBC

Rank	Treatment	Median	IQR
1	XTREE	77	8
2	BIC	66	5
3	CD	-1	1
3	CD+FS	-1	1

X264

Rank	Treatment	Median	IQR
1	XTREE	28	8
2	BIC	6	2
3	CD	0	0
3	CD+FS	0	1

LLVM:

Rank	Treatment	Median	IQR
1	XTREE	12	1
2	BIC	2	1
3	CD	0	0
3	CD+FS	0	0

Figure 11: Results: Seigmund data sets. Results from 40 repeats. Ratios of (1) software runtimes expected in the test examples after alteration by a planner to (2) the sum of the software runtimes in the original test set. *Larger* median values are *better*.

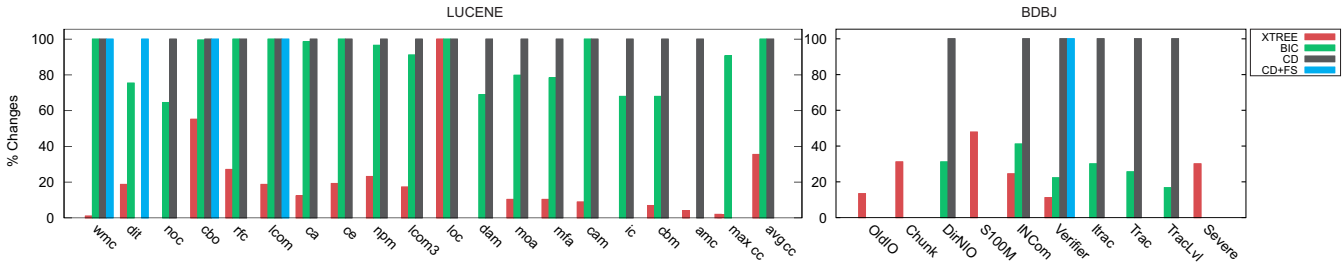

Figure 12: Percent frequency for how often certain feature was changed by a plan.

data set like *lucene*. In turns out that this holds true across nearly all our data sets. Figure 13 summarizes all the change frequencies for all data sets. As with Figure 12, there are fewer changed features in the Seigmund data than in the defect prediction data. One explanation for that is the nature of the features: the Jureczko data sets have continuous features while the Seigmund data has binary independent features (where settings can be turned "ON" or "OFF"). While the features of the Jureczko data sets are not constrained, those of Seigmund data sets are subject to several constrains (dictated by their feature models, see Figure 5). It is possible that planners in Method1, Method2, and Method3 find very few valid settings that can improve the performance scores, thereby making insignificant changes. As a result all methods, except XTREE, fail to optimize the runtime, as evidenced by the results in Figure 11.

		XTREE	BIC	CD	CD+FS
Jureczko data	Ant	20%	87%	80%	10%
	Ivy	21%	81%	70%	20%
	Jedit	19%	92%	90%	20%
	Lucene	19%	86%	95%	25%
	Poi	13%	79%	85%	20%
	mean:	*18%*	*85%*	*84%*	*19%*
Siegmund data	Apache	27%	5%	33%	22%
	BDBJ	7%	8%	3%	15%
	LLVM	33%	4%	36%	9%
	X264	24%	12%	18%	12%
	BDBC	25%	9%	16%	5%
	SQL	10%	6%	10%	23%
	mean:	*21%*	*7%*	*19%*	*14%*

Figure 13: Average number of features whose values are changed by a planner.

5.2.3 *Surprising Results*

If a planner only ever reported conclusions that were already known, then that planner offers little value over "just use established wisdom". Accordingly, we studied our results for plans that were somewhat counter-intuitive.

Such a surprising plan can be seen in *lucene*. Recall the standard advice for OO systems: build classes that are internally cohesive with low coupling to other parts of the system [15]. We can assess the relevance of this advice to specific projects by checking how often a planner changes the coupling-related features:

- *ca*: afferent couplings = # classes using this class;
- *ce*: efferent couplings = # classes used by this class.
- *cbm*: coupling between methods = # new/redefined methods to which all the inherited methods are coupled
- *cbo*: coupling between objects = a value that increases when the methods of one class access services of another.
- *ic*: inheritance coupling = # parent classes which a given class is coupled (includeing methods and variables inherited)

In many of our results with the Jureczko data, it was indeed true that the changes proposed by XTREE lead to lower coupling. However, the *lucene* results were quite different and rather surprising.

In the *lucene* XTREE results of Figure 12, the most frequent change was to alter the lines of code in a class (see the tallest red histogram in that figure on the *loc*, or lines of code). Looking at the logs of our planner, we can see that XTREE's proposed change is to *reduce* the size of a class. The only way to do that, while keeping the same functionality, is to create a network of smaller classes that

interact to produce that functionality. That is, we would need to *increase* the coupling of those classes to achieve XTREE's plan.

In theory, increasing coupling between classes complicates and confuses a class design. But the *lucene* XTREE results of Figure 12 rarely proposes changing the coupling features *ca, ce, cbm, ic* (in fact, XTREE never proposes any change to *ic*).

The only coupling issue that XTREE usually adds to its plans is *cbo* (which appears 55% of the time in Figure 12). But note that this is *object* coupling measure, not class coupling. So here XTREE is warning against, say, some factory class generating a large community of agents, all of the same class, who co-ordinate on some task. This is a different issue to the class redesign issue that would be triggered by altering *loc*.

In summary, XTREE satisfies that criteria that, sometimes, it produces surprising plans. At least for the *lucene* data set, we can see advice that recommends *increasing coupling* to reduce defects.

6. THREATS TO VALIDITY

As with any empirical study, biases can affect the final results. Therefore, any conclusions made from this work must be considered with the following issues in mind.

6.1 Learner Bias

For building the defect predictors in this study, we elected to use Random Forest and Random Forest Regressors . We chose this approach, based on its reputation for having the better performance of 21 other learners for defect prediction [31]. Data mining is a large and active field and any single study can only use a small subset of the known classification algorithms.

That said, we have taken care to document in this paper the decisions made by engineering judgement that could effect our conclusions. The above code used a set of variables which future work should vary in order to test the internal validity of our conclusions:

- All our planners divide data into groups of size $\alpha = \sqrt{N}$;
- Method2 used the top $\beta = 33\%$ most informative features (ranked using INFOGAIN);
- Method3 uses $\omega = 0.5$ to find its deltas;
- XTREE considered a sibling useful if it's score was less than $\gamma = 0.5$ times the mean score of the current leaf.

6.2 Sampling bias

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

6.3 Evaluation Bias

Another threat to validity of this work is our use of predictors learned on the training data to assess the impact of our planners. This issue was discussed in detail in §2.3.

To those notes, we add a few more details. If possible, planners should be assessed via some external oracle that can accurately assess new examples. For example, in search-based software engineering, examples can be assigned objective scores via some

Rank	Treatment	Median	IQR
1	XTREE	59	9
2	BIC	5	1
2	CD+FS	-7	100
2	CD	-9	77

Figure 14: Methods 1,2,3,4 applied to some ground-truth data (in this case, the POM3 model). Values collected from 40 repeated runs of each method with different random seeds. Results show the efficacy of XTREE in reducing the total overall cost in the original test data, when other planners fail to do so.

model. In this approach, a changed example can be assessed by generating actual objective scores from the model.

The POM3 model [38], [39] is a tool for exploring management challenges. POM3 implements the Boehm and Turner model of agile programming [40] where teams select tasks as they appear in the scrum backlog. POM3 can study the implications of different ways to adjust task lists in the face of shifting priorities. The model outputs estimated task completion rates; programmer idle rates; and total overall cost. POM3 models requirements as a dependency tree. A single requirement in the tree of a prioritization value has a cost, along with a list of child-requirements and dependencies. Before any requirement can be satisfied, its children and dependencies must first be satisfied. POM3 simulates changing priorities by making teams aware of random items in the requirements tree at random intervals, thus forcing teams to constantly readjust their "to do" lists. For further details on this model see [38], [39], and [40].

Figure 14 shows results from using POM3 as an oracle to assess our planning methods and their ability to reduce the project cost. In this experiment, we generated a training and testing set with 1000 randomly generated instances, which we passed to our four methods. 40 times, we let those methods propose changes to those projects. For assessment purposes, the changed projects were then fed back to the POM3 oracle.

Using this approach, it is possible to assess the value of a plan by measuring its effectiveness with respect to some ground truth (in this case, the POM3 model). As shown in Figure 14, XTREE reduces the cost by 59% thereby passing this assessment. These results from the POM3 model further endorse our previous conclusions; i.e. compared to three other methods, XTREE's supervised methods are best for generating plans on how to change example projects.

7. RELATED WORK

7.1 Planning in AI

The XTREE planner is somewhat different to the logic-based planners explored by classical AI. Those kinds of planners employ a logical procedure [41] that seeks an ordering on *operators* to take some domain *state* from a *start* state to a *goal* state. This classical logical approach is known to suffer from computational bottlenecks [42]. On the other hand, tools like XTREE will scale to any domain that can generate decision trees.

7.2 Evaluating Changes

Some organizations have the resources to run repeated trials to assess project changes. For example, in one study, Bente et al. reported results where the same specification was developed by four different organizations [43]. Given those kind of resources, it would be possible to (say) take a code base, assign it to different teams, make these teams adopt different polices, then check in 12

months time which teams have fewer defects than the others.

Seldom do industrial or research groups have access to the kinds of resources needed for this kind study (evidence: in the six years since the publication of that work, we know of only one similar study to Bente et al.). Also, given the diversity of modern software projects, it might be unreasonable to demand that all proposed changes for all projects are always evaluated by something like the Bente et al. study. Hence, this paper has used data miners to build an oracle that can assess changed examples. The advantage of our approach is that it required far less resources to assess the effectiveness of proposed changes to a project.

7.3 Search-based SE

Another way way to propose changes to software artifacts is via some search-based method [44,45]. Such SBSE methods are evolutionary programs that make extensive changes to some initial sample of project data (perhaps 100s to 100,000s of mutations). Each of these mutations is reassessed using some domain model. Examples of these algorithms include GALE, NSGA-II, NSGA-III, SPEA2, IBEA, particle swarm optimization, MOEA/D, etc. [21, 46–51].

One problem with these SBSE methods is that they can make extensive mutations to the data they are exploring. In the language of §2.3, these methods may not be *trust-increasing* since those algorithms make no attempt to prevent new examples from mutating away from the kinds of data used to commission the model (in which case, we would start doubting the model's output).

Another issue with standard search-based SE methods is that they require ready access to trustworthy domain model that can offer an assessment of newly generated examples. While some domains have such models (e.g. see the COCOMO effort estimation model used in the last section), our experience is that many others do not. 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.

What we would recommend is a two-pronged policy. In domains with ready access to trusted 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. [21, 46–51]. Otherwise, we recommend tools like XTREE.

8. CONCLUSIONS

The planner proposed in this paper proposes changes to software project details in order to improve the expected value of the performance scores of that part of the project. To evaluate these planners, data miners can be used to build oracles to assess planners. Such planners should be *trust-increasing*; i.e. they should propose changes that generate changed examples that are closer to the training data of the data miner. One caveat here is that the evaluations we can make on the planner are only as good as the predictive performance of the data miner. Hence, if domain data does not support satisfactory predictors, then planning in that domain cannot be evaluated.

Four planners were assessed here for the tasks of reducing defects and runtimes. Three of the methods come from our prior publications [4, 5], and the conclusion of this paper is that a novel fourth method clearly out-performs the other three (measured in terms of *effectiveness, succinctness, and surprise*). We conjecture that XTREE worked better than the rest since:

- It uses a supervised method to divide the data and;
- When planning how to move examples to better classes, it is best to reflect over differences between those classes.

9. FUTURE WORK

Future work in this area could explore numerous questions. For example, XTREE, as used here, sought improvements in a one goal (the class variable). Does XTREE work for multi-goal reasoning?

Also, the XTREE algorithm seems quite general to any data of table with rows containing weighted classes (so we can distinguish "bad" rows from "better" ones). Does XTREE works on domains (other than the defect/runtime data explored here)?

As an example of a domain that might benefit from XTREE, recent results raise doubts about the value of changing code to remove "bad smells" [52]. Can XTREE be used as a "bad smell" detector to select the subset of possible refactorings that have the most potential benefit?

As to scalability, XTREE is a post-processor to a decision tree algorithm. Hence, in theory, XTREE works on domain where data miners can generate decision trees. Given the current state of the art in Big Data, can XTREE be applied to very large data sets?

We discussed above in §2.2 the general conclusions of Passos, Jørgensen et.al [6, 7]; i.e. software developers are reluctant to surrender their old biases in the face of new data. Accordingly, it must be asked if the *mental resistance* of developers will prevent them applying XTREE's automatically generated recommendations of tools? Note this this issue is not just a concern for XTREE, but also for any automatic tool proposing refactorings.

There are many more methods for generating plans and no one paper can survey them all. For example, this paper has not explored variations to the α, β, and γ parameters that controlled XTREE. Would we get better results if we varied those parameters?

That said, the goal of this paper was not to claim that (e.g.) XTREE is some absolute optimal algorithm. Rather, it is was to offer a baseline result (with XTREE) and an evaluation strategy that can assess if alternate methods are better than XTREE. The authors of this paper would actively support other teams exploring this method (with or without using our current code base). So our last questions is this: will other researchers try to repeat and/or improve (or even refute) our results? Perhaps.

Acknowledgements

The work has partially funded by NSF award #1506586.

10. REFERENCES

[1] T. Menzies and T. Zimmermann. Goldfish bowl panel: Software development analytics. In *ICSE'12*, pages 1032–1033.

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

[3] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering*, pages 167–177. IEEE Press, 2012.

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

[5] Rahul Krishna and Tim Menzies. Actionable = cluster + constrast? *ACTION'15: an ASE'15 workshop*, 2015.

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

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

[8] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. Towards improving statistical modeling of software engineering data: think locally, act globally! *Empirical Software Engineering*, pages 1–42, 2014.

[9] Ye Yang, Lang Xie, Zhimin He, Qi Li, Vu Nguyen, Barry W. Boehm, and Ricardo Valerdi. Local bias and its impacts on the performance of parametric estimation models. In *PROMISE*, 2011.

[10] Ye Yang, Zhimin He, Ke Mao, Qi Li, Vu Nguyen, Barry W. Boehm, and Ricardo Valerdi. Analyzing and handling local bias for calibrating parametric cost estimation models. *Information & Software Technology*, 55(8):1496–1511, 2013.

[11] Leandro L. Minku and Xin Yao. Ensembles and locality: Insight on improving software effort estimation. *Information & Software Technology*, 55(8):1512–1528, 2013.

[12] Tim Menzies, Andrew Butcher, David R. Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Trans. Software Eng.*, 39(6):822–834, 2013. Available from `http://menzies.us/pdf/12localb.pdf`.

[13] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs global models for effort estimation and defect prediction. In *IEEE ASE'11*, 2011. Available from `http://menzies.us/pdf/11ase.pdf`.

[14] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of ASE'11*, 2011.

[15] Harpal Dhama. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, 29(1):65 – 74, 1995. Oregon Metric Workshop.

[16] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal Machine Learning*, 12:2825–2830, 2012.

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

[18] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[19] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: is it really necessary? *Submitted to ICSE ' 16*, 2016.

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

[21] J Krall and T Menzies. GALE: Geometric Active Learning for Search-based Software Engineering. *IEEE Transactions on Software Engineering (to appear)*, 2015.

[22] M A Hall and G Holmes. Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. *IEEE Transactions On Knowledge And Data Engineering*, 15(6):1437–1447, 2003.

[23] Ron Kohavi and George H John. Wrappers for Feature Subset Selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[24] Vasil Papakroni. Data carving: Identifying and removing irrelevancies in the data. Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia Unviersity, 2013.

[25] U M Fayyad and I H Irani. Multi-interval Discretization of Continuous-valued Attributes for Classification Learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[26] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

[27] Daniel Boley. Principal direction divisive partitioning. *Data Min. Knowl. Discov.*, 2(4):325–344, December 1998.

[28] Christos Faloutsos and King-Ip Lin. FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 163–174, 1995.

[29] John C. Platt. FastMap, MetricMap, and Landmark MDS are all Nyström algorithms. In *In Proceedings of 10th International Workshop on Artificial Intelligence and Statistics*, pages 261–268, 2005.

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

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

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

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

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

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

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

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

[38] Barry Boehm and Richard Turner. Using risk to balance agile and plan-driven methods. *Computer*, (6):57–66, 2003.

[39] Daniel Port, Alexy Olkov, and Tim Menzies. Using simulation to investigate requirements prioritization strategies. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 268–277. IEEE Computer Society, 2008.

[40] Barry Boehm and Richard Turner. *Balancing agility and discipline: A guide for the perplexed*. Addison-Wesley Professional, 2003.

[41] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving, 1971.

[42] Tom Bylander. The computational complexity of propositional STRIPS planning, 1994.

[43] Bente C D Anda, Dag I K Sjø berg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3):407–429, 2009.

[44] M Harman, Sa Mansouri, and Y Zhang. Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. *Engineering*, (TR-09-03):1–78, 2009.

[45] M Harman, P McMinn, JT De Souza, and S Yoo. Search based software engineering: Techniques, taxonomy, tutorial. *Search*, 2012:1–59, 2011.

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

[47] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In *Evolutionary Methods for Design, Optimisation, and Control*, pages 95–100. CIMNE, Barcelona, Spain, 2002.

[48] Eckart Zitzler and Simon Künzli. Indicator-Based Selection in Multiobjective Search. In Xin Yao, EdmundK. Burke, JoséA. Lozano, Jim Smith, JuanJulián Merelo-Guervós, JohnA. Bullinaria, JonathanE. Rowe, Peter TiÅĹo, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *Lecture Notes in Computer Science*, pages 832–842. Springer Berlin Heidelberg, 2004.

[49] 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*, 18(4):577–601, August 2014.

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

[51] Qingfu Zhang and Hui Li. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *Evolutionary Computation, IEEE Transactions on*, 11(6):712–731, December 2007.

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