Tuning for Software Analytics: is it Really Necessary?

Wei Fu*, Tim Menzies*, Xipeng Shen

Department of Computer Science, North Carolina State University, Raleigh, NC, USA

Abstract

Context: Data miners have been widely used in software engineering to, say, generate defect predictors from static code measures. Such static code defect predictors perform well compared to manual methods, and they are easy to use and useful to use. But one of the "black arts" of data mining is setting the tunings that control the miner.
Objective: We seek simple, automatic, and very effective method for finding those tunings.
Method: For each experiment with different data sets (from open source JAVA systems), we ran differential evolution as an optimizer to explore the tuning space (as a first step) then tested the tunings using hold-out data.
Results: Contrary to our prior expectations, we found these tunings were remarkably simple: it only required tens, not thousands, of attempts to obtain very good results. For example, when learning software defect predictors, this method can quickly find tunings that alter detection precision from 0% to 60%.
Conclusion: Since the (1) the improvements are so large, and (2) the tuning is so simple, we need to change standard methods in software analytics. At least for defect prediction, it is no longer enough to just run a data miner and present the result *without* conducting a tuning optimization study. The implication for other kinds of analytics is now an open and pressing issue.

Keywords: defect prediction, CART, random forest, differential evolution, search-based software engineering.

1. Introduction

In the 21^{st} century, it is impossible to manually browse all available software project data. The PROMISE repository of SE data has grown to 200+ projects [1] and this is just one of over a dozen open-source repositories that are readily available to researchers [2]. For example, at the time of this writing (Jan 2016), our web searches show that Mozilla Firefox has over 1.1 million bug reports, and platforms such as GitHub host over 14 million projects.

Faced with this data overload, researchers in empirical SE use data miners to generate *defect predictors from static code measures*. Such measures can be automatically extracted from the code base, with very little effort even for very large software systems [3].

One of the "black arts" of data mining is setting the tuning parameters that control the choices within a data miner. Prior to this work, our intuition was that tuning would change the behavior or a data miner, to some degree. Nevertheless, we rarely tuned our defect predictors since we reasoned that a data miner's default tunings have been well-explored by the developers of those algorithms (in which case tuning would not lead to large performance improvements). Also, we suspected that tuning would take so long time and be so CPU intensive that the benefits gained would not be worth effort.

*Corresponding author: Tel:19195345251(Wei)
Email addresses: wfu@ncsu.edu (Wei Fu), tim.menzies@gmail.com (Tim Menzies), xshen5@ncsu.edu (Xipeng Shen)

The results of this paper show that the above points are false since, at least for defect prediction from code attributes:

1. Tuning defect predictors is *remarkably simple*;
2. And can *dramatically improve the performance*.

Those results were found by exploring six research questions:

- RQ1: *Does tuning improve the performance scores of a predictor?* We will show below examples of truly dramatic improvement: usually by 5 to 20% and often by much more (in one extreme case, precision improved from 0% to 60%).
- RQ2: *Does tuning change conclusions on what learners are better than others?* Recent SE papers [4, 5] claim that some learners are better than others. Some of those conclusions are completely changed by tuning.
- RQ3: *Does tuning change conclusions about what factors are most important in software engineering?* Numerous recent SE papers (e.g. [6, 7, 8, 9, 10, 11]) use data miners to conclude that *this* is more important than *that* for reducing software project defects. Given the tuning results of this paper, we show that such conclusions need to be revisited.
- RQ4: *Is tuning easy?* We show that one of the simpler multi-objective optimizers (differential evolution [12]) works very well for tuning defect predictors.
- RQ5: *Is tuning impractically slow?* We achieved dramatic improvements in the performance scores of our data miners in less than 100 evaluations (!); i.e., very quickly.

- RQ6: *Should data miners be used "off-the-shelf" with their default tunings?* For defect prediction from static code measures, our answer is an emphatic "no" (and the implication for other kinds of analytics is now an open and urgent question).

Based on our answers to these questions, we strongly advise that:

- Data miners should not be used "off-the-shelf" with default tunings.
- Any future paper on defect prediction should include a tuning study. Here, we have found an algorithm called differential evolution to be a useful method for conducting such tunings.
- Tuning needs to be repeated whenever data or goals are changed. Fortunately, the cost of finding good tunings is not excessive since, at least for static code defect predictors, tuning is easy and fast.

2. Preliminaries

2.1. Tuning: Important and Ignored

This section argues that tuning is an under-explored software analytics– particularly in the apparently well-explored field of defect prediction.

In other fields, the impact of tuning is well understood [13]. Yet issues of tuning are rarely or poorly addressed in the defect prediction literature. When we tune a data miner, what we are really doing is changing how a learner applies its heuristics. This means tuned data miners use different heuristics, which means they ignore different possible models, which means they return different models; i.e. *how* we learn changes *what* we learn.

Are the impacts of tuning addressed in the defect prediction literature? To answer that question, in Jan 2016 we searched scholar.google.com for the conjunction of "data mining" and "software engineering" and "defect prediction"[1]. After sorting by the citation count and discarding the non-SE papers (and those without a pdf link), we read over this sample of 50 highly-cited SE defect prediction papers. What we found in that sample was that few authors acknowledged the impact of tunings (exceptions: [14, 4]). Overall, 80% of papers in our sample *did not* adjust the "off-the-shelf" configuration of the data miner (e.g. [15, 9, 16]). Of the remaining papers:

- Some papers in our sample explored data super-sampling [17] or data sub-sampling techniques via automatic methods (e.g. [14, 15, 17, 18]) or via some domain principles (e.g. [9, 19, 20]). As an example of the latter, Nagappan et al. [19] checked if metrics related to organizational structure were relatively more powerful for predicting software defects. However, it should be noted that these studies varied the input data but not the "off-the-shelf" settings of the data miner.
- A few other papers did acknowledge that one data miner may not be appropriate for all data sets. Those papers tested different "off-the-shelf" data miners on the same data set. For example, Elish et al.[16] compared support vector machines to other data miners for the purposes of defect prediction. SVM's execute via a "kernel function" which should be specially selected for different data sets and the Elish et al. paper makes no mention of any SVM tuning study. To be fair to Elish et al., we hasten to add that we ourselves have published papers using "off-the-shelf" tunings [15] since, prior to this paper it was unclear to us how to effectively navigate the large space of possible tunings.

Over our entire sample, there was only one paper that conducted a somewhat extensive tuning study. Lessmann et al.[4] tuned parameters for some of their algorithms using a *grid search*; i.e. divide all C configuration options into N values, then try all N^C combinations. This is a slow approach– we have explored grid search for defect prediction and found it takes days to terminate [15]. Not only that, we found that grid search can miss important optimizations [21]. Every grid has "gaps" between each grid division which means that a supposedly rigorous grid search can still miss important configurations [13]. Bergstra and Bengio [13] comment that for most data sets only a few of the tuning parameters really matter– which means that much of the runtimes associated with grid search is actually wasted. Worse still, Bergstra and Bengio comment that the important tunings are different for different data sets– a phenomenon makes grid search a poor choice for configuring data mining algorithms for new data sets.

Since the Lessmann et al. paper, much progress has been made in configuration algorithms and we can now report that *finding useful tunings is very easy*. This result is both novel and unexpected. A standard run of grid search (and other evolutionary algorithms) is that optimization requires thousands, if not millions, of evaluations. However, in a result that we found startling, that *differential evolution* (described below) can find useful settings for learners generating defect predictors in less than 100 evaluations (i.e. very quickly). Hence, the "problem" (that tuning changes the conclusions) is really an exciting opportunity. At least for defect prediction, learners are very amenable to tuning. Hence, they are also very amenable to significant performance improvements. Given the low number of evaluations required, then we assert that tuning should be standard practice for anyone building defect predictors.

2.2. You Can't Always Get What You Want

Having made the case that tuning needs to be explored more, but before we get into the technical details of this paper, this section discusses some general matters about setting goals during tuning experiments.

This paper characterizes tuning as an optimization problem (how to change the settings on the learner in order to best improve the output). With such optimizations, it is not always possible to optimize for all goals at the same time. For example, the following text does not show results for tuning on recall

[1] More details can be found at https://goo.gl/Inl9nF

or false alarms since optimizing *only* for those goals can lead to some undesirable side effects:

- *Recall* reports the percentage of predictions that are actual examples of what we are looking for. When we tune for *recall*, we can achieve near 100% recall– but the cost of a near 100% false alarms.
- *False alarms* is the percentage of other examples that are reported (by the learner) to be part of the targeted examples. When we tune for *false alarms*, we can achieve near zero percent false alarm rates by effectively turning off the detector (so the recall falls to nearly zero).

Accordingly, this paper explores performance measures that comment on all target classes: see the precision and "F" measures discussed below: see *Optimization Goals*. That said, we are sometimes asked what good is a learner if it optimizers for (say) precision at the expense of (say) recall.

Our reply is that software engineering is a very diverse enterprise and that different kinds of development need to optimize for different goals (which may not necessarily be "optimize for recall"):

- Anda, Sjoberg and Mockus are concerned with *reproducibility* and so assess their models using the the "coefficient of variation" $C\frac{stddev}{mean}$) [22].
- Arisholm & Briand [23], Ostrand & Weyeuker [24] and Rahman et al. [25] are concerned with reducing the work load associated with someone else reading a learned model, then applying it. Hence, they assess their models using *reward*; i.e. the fewest lines of code containing the most bugs.
- Yin et al. are concerned about *incorrect bug fixes*; i.e. those that require subsequent work in order to complete the bug fix. These bugs occur when (say) developers try to fix parts of the code where they have very little experience [26]. Hence, they assess a learned model using a measure that selects for the most number of bugs in regions that *the most programmers have worked with before*.
- For safety critical applications, high false alarm rates are acceptable if the cost of overlooking critical issues outweighs the inconvenience of inspecting a few more modules.
- When rushing a product to market, there is a business case to avoid the extra rework associated with false alarms. In that business context, managers might be willing to lower the recall somewhat in order to minimize the false alarms.
- When the second author worked with contractors at NASA's software independent verification and validation facility, he found new contractors only reported issues that were most certainly important defects; i.e. they minimized false alarms even if that damaged their precision (since, they felt, it was better to silent than wrong). Later on, once those contractors had acquired a reputation of being insightful members of the team, they improved their precision scores (even if it means some more false alarms).

Accordingly, this paper does not assume that (e.g.) minimizing false alarms is more important than maximizing precision or recall. Such a determination depends on business conditions.

Rather, what we can show examples where changing optimization goals can also change the conclusions made from that learner on that data. More generally, we caution that it is important not to overstate empirical results from analytics. Those results need to be expressed *along with* the context within which they are relevant (and by "context", we mean the optimization goal).

2.3. Notes on Defect Prediction

This section discusses defect prediction, which is the particular task explored by our optimizers. Note that this section repeats much of our standard introduction to defect prediction [27], as well as presenting some new results from Rahman et al. [28].

Human programmers are clever, but flawed. Coding adds functionality, but also defects. Hence, software sometimes crashes (perhaps at the most awkward or dangerous moment) or delivers the wrong functionality. For a very long list of software-related errors, see Peter Neumann's "Risk Digest" at catless.ncl.ac.uk/Risks.

Since programming inherently introduces defects into programs, it is important to test them before they're used. Testing is expensive. Software assessment budgets are finite while assessment effectiveness increases exponentially with assessment effort. For example, for black-box testing methods, a *linear* increase in the confidence C of finding defects can take *exponentially* more effort[2]. Exponential costs quickly exhaust finite resources so standard practice is to apply the best available methods on code sections that seem most critical. But any method that focuses on parts of the code can blind us to defects in other areas. Some *lightweight sampling policy* should be used to explore the rest of the system. This sampling policy will always be incomplete. Nevertheless, it is the only option when resources prevent a complete assessment of everything.

One such lightweight sampling policy is defect predictors learned from static code attributes. Given software described in the attributes of Table 1, data miners can learn where the probability of software defects is highest.

The rest of this section argues that such defect predictors are *easy to use*, *widely-used*, and *useful* to use.

Easy to use: Static code attributes can be automatically collected, even for very large systems [3]. Other methods, like manual code reviews, are far slower and far more labor-intensive. For example, depending on the review methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six people [29]. *Widely used:* Researchers and industrial practitioners use static attributes to guide software quality

[2] A randomly selected input to a program will find a fault with probability p. After N random black-box tests, the chances of the inputs not revealing any fault is $(1-p)^N$. Hence, the chances C of seeing the fault is $1-(1-p)^N$ which can be rearranged to $N(C,p) = log(1-C)/log(1-p)$. For example, $N(0.90, 10^{-3}) = 2301$ but $N(0.98, 10^{-3}) = 3901$; i.e. nearly double the number of tests.

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.
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 instance 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 $lcom3 = ((\frac{1}{a}\sum_j^a \mu(a_j)) - m)/(1-m)$.
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	
defect	defect	Boolean: where defects found in post-release bug-tracking systems.

Table 1: OO measures used in our defect data sets.

predictions. Defect prediction models have been reported at Google [30]. Verification and validation (V&V) textbooks [31] advise using static code complexity attributes to decide which modules are worth manual inspections.

Useful: Defect predictors often find the location of 70% (or more) of the defects in code [15]. Defect predictors have some level of generality: predictors learned at NASA [15] have also been found useful elsewhere (e.g. in Turkey [32, 33]). The success of this method in predictors in finding bugs is markedly higher than other currently-used industrial methods such as manual code reviews. For example, a panel at *IEEE Metrics 2002* [34] concluded that manual software reviews can find ≈60% of defects. In another work, Raffo documents the typical defect detection capability of industrial review methods: around 50% for full Fagan inspections [35] to 21% for less-structured inspections.

Not only do static code defect predictors perform well compared to manual methods, they also are competitive with certain automatic methods. A recent study at ICSE'14, Rahman et al. [28] compared (a) static code analysis tools FindBugs, Jlint, and Pmd and (b) static code defect predictors (which they called "statistical defect prediction") built using logistic regression. They found no significant differences in the cost-effectiveness of these approaches. Given this equivalence, it is significant to note that static code defect prediction can be quickly adapted to new languages by building lightweight parsers that find information like Table 1. The same is not true for static code analyzers– these need extensive modification before they can be used on new languages.

2.4. Notes on Data Miners

There are several ways to make defect predictors using CART [36], Random Forest [37], WHERE [38] and LR (logistic regression). For this study, we use CART, Random Forest and LR versions from SciKitLearn [39] and WHERE, which is available from github.com/ai-se/where. We use these algorithms for the following reasons.

CART and Random Forest were mentioned in a recent IEEE TSE paper by Lessmann et al. [4] that compared 22 learners for defect prediction. That study ranked CART worst and Random Forest as best. In a demonstration of the impact of tuning, this paper shows we can *refute* the conclusions of Lessmann et al. in the sense that, after tuning, CART performs just as well as Random Forest.

LR was mentioned by Hall et al. [5] as usually being as good or better as more complex learners (e.g. Random Forest). In a finding that endorses the Hall et al. result, we show that untuned LR performs better than untuned Random Forest (at least, for the data sets studied here). However, we will show that tuning raises doubts about the optimality of the Hall et al. recommendation.

Finally, this paper uses WHERE since, as shown below, it offers an interesting case study on the benefits of tuning.

2.5. Learners and Their Tunings

Our learners use the tuning parameters of Table 2. This section describes those parameters. The default parameters for CART and Random Forest are set by the SciKitLearn authors and the default parameters for WHERE-based learner are set via our own expert judgement. When we say a learner is used "off-the-shelf", we mean that they use the defaults shown in Table 2.

As to the value of those defaults, it could be argued that these defaults are not the best parameters needed for practical defect prediction. That said, prior to this paper, two things were true:

- Many data scientists in SE use the standard defaults in their data miners, without tuning (e.g. [15, 9, 11, 10]).
- The effort involved to adjust those tunings seemed so onerous, that many researchers in this field were content to take our prior advice of "do not tune... it is just too hard" [27].

As to why we used the "Tuning Range" shown in Table 2, and not some other ranges, we note that (1) those ranges included the defaults; (2) the results shown below show that by exploring those ranges, we achieved large gains in the performance of our defect predictors. This is not to say that *larger* tuning ranges might not result in *greater* improvements. However, for

Learner Name	Parameters	Default	Tuning Range	Description
Where-based Learner	threshold	0.5	[0.01,1]	The value to determine defective or not .
	infoPrune	0.33	[0.01,1]	The percentage of features to consider for the best split to build its final decision tree.
	min_sample_split	4	[1,10]	The minimum number of samples required to split an internal node of its final decision tree.
	min_Size	0.5	[0.01,1]	Finds min_samples_leaf in the initial clustering tree using $n\_samples^{min\_Size}$.
	wriggle	0.2	[0.01, 1]	The threshold to determine which branch in the initial clustering tree to be pruned
	depthMin	2	[1,6]	The minimum depth of the initial clustering tree below which no pruning for the clustering tree.
	depthMax	10	[1,20]	The maximum depth of the initial clustering tree.
	wherePrune	False	T/F	Whether or not to prune the initial clustering tree.
	treePrune	True	T/F	Whether or not to prune the final decision tree.
CART	threshold	0.5	[0,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_samples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
	max_depth	None	[1, 50]	The maximum depth of the tree.
Random Forests	threshold	0.5	[0.01,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	max_leaf_nodes	None	[1,50]	Grow trees with max_leaf_nodes in best-first fashion.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_samples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
	n_estimators	100	[50,150]	The number of trees in the forest.
Logistic Regression	This study uses untuned LR in order to check a conclusion of [5].			

Table 2: List of parameters tuned by this paper.

the goals of this paper (to show that some tunings do matter), exploring just these ranges shown in Table 2 will suffice.

As to the details of these learners, LR is a parametric modeling approach. Given $f = \beta_0 + \sum_i \beta_i x_i$, where x_i is some measurement in a data set, and β_i is learned via regression, LR converts that into a function $0 \leq g \leq 1$ using $g = 1/\left(1+e^{-f}\right)$. This function reports how much we believe in a particular class.

CART, Random Forest, and WHERE-based learners are all tree learners that divide a data set, then recur on each split. All these learners generate numeric predictions which are converted into binary "yes/no" decisions via Equation 1.

$$inspect = \begin{cases} d_i \geq T \rightarrow Yes \\ d_i < T \rightarrow No, \end{cases} \quad (1)$$

The splitting process is controlled by numerous tuning parameters. If data contains more than *min_sample_split*, then a split is attempted. On the other hand, if a split contains no more than *min_samples_leaf*, then the recursion stops. CART and Random Forest use a user-supplied constant for this parameter while WHERE-based learner firstly computes this parameter *m*=*min_samples_leaf* from the size of the data sets via $m = size^{min\_size}$ to build an initial clustering tree. Note that WHERE builds *two* trees: the initial clustering tree (to find similar sets of data) then a final decision tree (to learn rules that predict for each similar cluster)[3]. The tuning parameter *min_sample_split* controls the construction of the final decision tree (so, for WHERE-based learner, *min_size* and *min_sample_split* are the parameters to be tuned).

These learners use different techniques to explore the splits:

- CART finds the attributes whose ranges contain rows with least variance in the number of defects[4].
- Random Forest divides data like CART then builds $F > 1$ trees, each time using some random subset of the attributes.
- When building the initial cluster tree, WHERE projects the data on to a dimension it synthesizes from the raw data using a process analogous to principle component analysis[5]. WHERE divides at the median point of that projection. On recursion, this generates the initial clustering tree, the leaves of which are clusters of very similar examples. After that, when building the final decision tree, WHERE pretends its clusters are "classes", then asks the InfoGain of the Fayyad-Irani discretizer [40], to rank the attriubutes, where *infoPrune* is used. WHERE's final decision tree generator then ignores everything except the top *infoPrune* percent of the sorted attributes.

Some tuning parameters are learner specific:

- *Max_feature* is used by CART and Random Forest to select the number of attributes used to build one tree. CART's default is to use all the attributes while Random Forest usually selects the square root of the number of attributes.
- *Max_leaf_nodes* is the upper bound on leaf notes generated in a Random Forest.
- *Max_depth* is the upper bound on the depth of the CART tree.
- WHERE's tree generation will always split up to *depthMin* number of branches. After that, WHERE will only split data if the mean performance scores of the two halves is "trivially

[3] A frequently asked question is why does WHERE build two trees– would not a single tree suffice? The answer is, as shown below, tuned WHERE's twin-tree approach generates very precise predictors.

[4] If an attribute ranges r_i is found in n_i rows each with a defect count variance of v_i, then CART seeks the attributes whose ranges minimizes $\sum_i \left(\sqrt{v_i} \times n_i/(\sum_i n_i)\right)$.

[5] PCA synthesises new attributes $e_i, e_2, ...$ that extends across the dimension of greatest variance in the data with attributes d. This process combines redundant variables into a smaller set of variables (so $e \ll d$) since those redundancies become (approximately) parallel lines in e space. For all such redundancies $i, j \in d$, we can ignore j since effects that change over j also change in the same way over i. PCA is also useful for skipping over noisy variables from d– these variables are effectively ignored since they do not contribute to the variance in the data.

small" (where "trivially small" is set by the *wriggle* parameter).

- WHERE's *tree_prune* setting controls how WHERE prunes back superflous parts of the final decision tree. If a decision sub-tree and its parent have the same majority cluster (one that occurs most frequently), then if *tree_prune* is enabled, we prune that decision sub-tree.

2.6. Tuning Algorithms

How should researchers select which optimizers to apply to tuning data miners? Cohen [41] advises comparing new methods against the simplest possible alternative. Similarly, Holte [42] recommends using very simple learners as a kind of "scout" for a preliminary analysis of a data set (to check if that data really requires a more complex analysis). Accordingly, to find our "scout", we used engineering judgement to sort candidate algorithms from simplest to complex. For example, here is a list of optimizers used widely in research: *simulated annealing* [43, 44]; various *genetic algorithms* [45] augmented by techniques such as *differential evolution* [12], *tabu search* and *scatter search* [46, 47, 48, 49]; *particle swarm optimization* [50]; numerous *decomposition* approaches that use heuristics to decompose the total space into small problems, then apply a *response surface methods* [51, 52]. Of these, the simplest are simulated annealing (SA) and differential evolution (DE), each of which can be coded in less than a page of some high-level scripting language. Our reading of the current literature is that there are more advocates for differential evolution than SA. For example, Vesterstrom and Thomsen [53] found DE to be competitive with particle swarm optimization and other GAs.

DEs have been applied before for parameter tuning (e.g. see [54, 55]) but this is the first time they have been applied to optimize defect prediction from static code attributes. The pseudocode for differential evolution is shown in Algorithm 1. In the following description, superscript numbers denote lines in that pseudocode.

DE evolves a *NewGeneration* of candidates from a current *Population*. Our DE's lose one "life" when the new population is no better than current one (terminating when "life" is zero)^{L4}. Each candidate solution in the *Population* is a pair of *(Tunings, Scores)*. *Tunings* are selected from Table 2 and *Scores* come from training a learner using those parameters and applying it test data$^{L23-L27}$.

The premise of DE is that the best way to mutate the existing tunings is to *Extrapolate*^{L28} between current solutions. Three solutions a, b, c are selected at random. For each tuning parameter i, at some probability cr, we replace the old tuning x_i with y_i. For booleans, we use $y_i = \neg x_i$ (see line 36). For numerics, $y_i = a_i + f \times (b_i - c_i)$ where f is a parameter controlling crossover. The *trim* function^{L38} limits the new value to the legal range min..max of that parameter.

The main loop of DE^{L6} runs over the *Population*, replacing old items with new *Candidate*s (if new candidate is better). This means that, as the loop progresses, the *Population* is full of increasingly more valuable solutions. This, in turn, also improves the candidates, which are *Extrapolate*d from the *Population*.

Algorithm 1 Pseudocode for DE with Early Termination

Input: $np = 10$, $f = 0.75$, $cr = 0.3$, $life = 5$, $Goal \in \{pd, f, ...\}$
Output: S_{best}

```
 1: function DE(np, f, cr, life, Goal)
 2:     Population ← InitializePopulation(np)
 3:     S_best ←GetBestSolution(Population)
 4:     while life > 0 do
 5:         NewGeneration ← ∅
 6:         for i = 0 → np − 1 do
 7:             S_i ← Extrapolate(Population[i], Population, cr, f)
 8:             if Score(S_i) ¿Score(Population[i]) then
 9:                 NewGeneration.append(S_i)
10:             else
11:                 NewGeneration.append(Population[i])
12:             end if
13:         end for
14:         Population ← NewGeneration
15:         if ¬ Improve(Population) then
16:             life− = 1
17:         end if
18:         S_best ← GetBestSolution(Population)
19:     end while
20:     return S_best
21: end function
22: function SCORE(Candidate)
23:     set tuned parameters according to Candidate
24:     model ←TrainLearner()
25:     result ←TestLearner(model)
26:     return Goal(result)
27: end function
28: function EXTRAPOLATE(old, pop, cr, f)
29:     a, b, c ← threeOthers(pop, old)
30:     newf ← ∅
31:     for i = 0 → np − 1 do
32:         if cr < random() then
33:             newf.append(old[i])
34:         else
35:             if typeof(old[i]) == bool then
36:                 newf.append(not old[i])
37:             else
38:                 newf.append(trim(i,(a[i] + f ∗ (b[i] − c[i]))))
39:             end if
40:         end if
41:     end for
42:     return newf
43: end function
```

For the experiments of this paper, we collect performance values from a data mining, from which a *Goal* function extracts one performance value^{L26} (so we run this code many times, each time with a different *Goal*^{L1}). Technically, this makes a *single objective* DE (and for notes on multi-objective DEs, see [56, 57, 58]).

3. Experimental Design

3.1. Data Sets

Our defect data comes from the PROMISE repository [6] and pertains to open source Java systems defined in terms of Table 1: *ant*, *camel*, *ivy*, *jedit*, *log4j*, *lucene*, *poi*, *synapse*, *velocity* and *xerces*.

An important principle in data mining is not to test on the data used in training. There are many ways to design a experiment that satisfies this principle. Some of those methods have limitations; e.g. *leave-one-out* is too slow for large data sets and *cross-validation* mixes up older and newer data (such that data from the *past* may be used to test on *future data*).

[6] http://openscience.us/repo

Dataset	antV0	antV1	antV2	camelV0	camelV1	ivy	jeditV0	jeditV1	jeditV2
training	20/125	40/178	32/293	13/339	216/608	63/111	90/272	75/306	79/312
tuning	40/178	32/293	92/351	216/608	145/872	16/241	75/306	79/312	48/367
testing	32/293	92/351	166/745	145/872	188/965	40/352	79/312	48/367	11/492
Dataset	log4j	lucene	poiV0	poiV1	synapse	velocity	xercesV0	xercesV1	
training	34/135	91/195	141/237	37/314	16/157	147/196	77/162	71/440	
tuning	37/109	144/247	37/314	248/385	60/222	142/214	71/440	69/453	
testing	189/205	203/340	248/385	281/442	86/256	78/229	69/453	437/588	

Table 3: Data used in this experiment. E.g., the top left data set has 20 defective classes out of 125 total. See §3.1 for explanation of *training, tuning, testing* sets.

To avoid these problems, we used an incremental learning approach. The following experiment ensures that the training data was created at some time before the test data. For this experiment, we use data sets with at least three consecutive releases (where release $i+1$ was built after release i). When tuning a learner,

- The *first* release was used on line 24 of Algorithm 1 to build some model using some the tunings found in some *Candidate*.
- The *second* release was used on line 25 of Algorithm 1 to test the candidate model found on line 24.
- Finally the *third* release was used to gather the performance statistics reported below from the best model found by DE.

To be fair for the untuned learner, the *first* and *second* releases used in tuning experiments will be combined as the training data to build a model. Then the performance of this untuned learner will be evaluated by the same *third* release as in the tuning experiment.

Some data sets have more than three releases and, for those data, we could run more than one experiment. For example, *ant* has five versions in PROMISE so we ran three experiments called V0,V1,V2:

- AntV0: first, second, third = versions 1, 2, 3
- AntV1: first, second, third = versions 2, 3, 4
- AntV2: first, second, third = versions 3, 4, 5

These data sets are displayed in Table 3.

3.2. Optimization Goals

Recall from Algorithm 1 that we call differential evolution once for each optimization goal. This section lists those optimization goals. Let $\{A,B,C,D\}$ denote the true negatives, false negatives, false positives, and true positives (respectively) found by a binary detector. Certain standard measures can be computed from A,B,C,D, as shown below. Note that for pf, the *better* scores are *smaller* while for all other scores, the *better* scores are *larger*.

$$\begin{array}{ll} pd = recall = & D/(B+D) \\ pf = & C/(A+C) \\ prec = precision = & D/(D+C) \\ F = & 2*pd*prec/(pd+prec) \end{array}$$

The rest of this paper explores tuning for *prec* and *F*. As discussed in §2.2, our point is not that these are best or most important optimization goals. Indeed, the list of "most important" goals is domain-specific (see §2.2) and we only explore these two to illustrate how conclusions can change dramatically when moving from one goal to another.

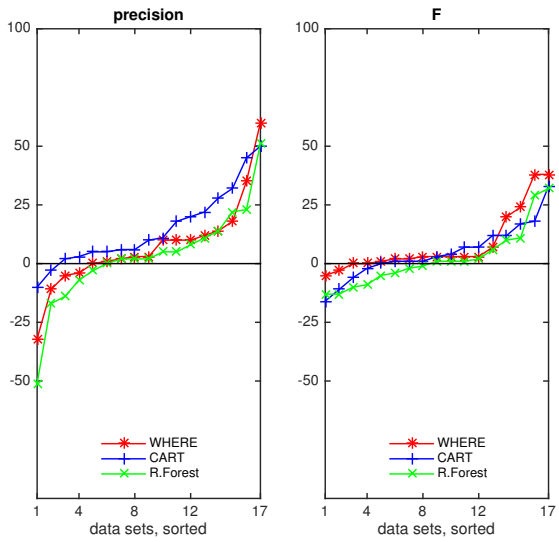

Figure 1: Deltas in performance seen in Table 4 (left) and Table 5 (right) between tuned and untuned learners. Tuning improves performance when the deltas are above zero.

4. Experimental Results

In the following, we explore the effects of tuning WHERE, Random Forest, and CART. LR will be used, untuned, in order to check one of the recommendations made by Hall et al. [5].

4.1. RQ1: Does Tuning Improve Performance?

Figure 1 says that the answer to RQ1 is "yes"– tuning has a positive effect on performance scores. This figure sorts deltas in the precision and the F-measure between tuned and untuned learners. Our reading of this figure is that, overall, tuning rarely makes performance worse and often can make it much better.

Table 4 and Table 5 show the the specific values seen before and after tuning with *precision* and *"F"* as different optimization goals(the corresponding "F" and precision values for Table 4 and Table 5 are not provided for the space limitation). For each data set, the maximum precision or "F" values for each data set are shown in **bold**. As might have been predicted by Lessmann et al. [4], untuned CART is indeed the worst learner (only one of its untuned results is best and **bold**). And, in $\frac{12}{17}$

	WHERE		CART		Random Forest	
Data set	default	Tuned	default	Tuned	default	Tuned
antV0	0	35	15	**60**	21	44
antV1	0	60	54	56	**67**	50
antV2	45	55	42	52	56	**67**
camelV0	20	30	30	50	28	**79**
camelV1	27	28	**38**	28	34	27
ivy	25	21	21	**26**	23	20
jeditV0	34	37	56	**78**	52	60
jeditV1	30	42	32	**64**	32	37
jeditV2	4	**22**	6	17	4	6
log4j	96	91	95	98	95	**100**
lucene	61	75	67	70	63	**77**
poiV0	70	70	65	**71**	67	69
poiV1	74	76	72	90	78	**100**
synapse	61	50	50	**100**	60	60
velocity	34	**44**	39	**44**	40	42
xercesV0	14	17	17	14	**28**	14
xercesV1	86	54	72	**100**	78	27

Table 4: Precision results (best results shown in **bold**).

	WHERE		CART		Random Forest	
Data set	default	Tuned	default	Tuned	default	Tuned
antV0	0	20	20	**40**	28	38
antV1	0	38	37	**49**	38	**49**
antV2	47	50	45	49	**57**	56
camelV0	31	28	39	28	**40**	30
camelV1	34	34	38	32	**42**	33
ivy	39	34	28	**40**	35	33
jeditV0	45	47	56	57	**63**	59
jeditV1	43	44	44	47	46	**48**
jeditV2	8	**11**	10	10	8	9
log4j	47	50	53	37	**60**	47
lucene	73	73	65	72	70	**76**
poiV0	50	74	31	64	45	**77**
poiV1	75	**78**	68	69	77	**78**
synapse	49	56	43	**60**	52	53
velocity	51	53	53	51	**56**	51
xercesV0	19	22	19	**26**	34	21
xercesV1	32	70	34	35	42	**71**

Table 5: F-measure results (best results shown in **bold**).

cases, the untuned Random Forest performs better than or equal to untuned CART in terms of precision.

That said, tuning can improve those poor performing detectors. In some cases, the median changes may be small (e.g. the "F" results for WHERE and Random Forests) but even in those cases, there are enough large changes to motivate the use of tuning. For example:

- For "F" improvement, there are two improvements over 25% for both WHERE and Random Forests. Also, in *poiV0*, all untuned learners report "F" of under 50%, tuning changes those scores by 25%. Finally, note the *xercesV1* result for the WHERE learner. Here, tuning changes precision from 32% to 70%.

- Regarding precision, for *antV0*, and *antV1* untuned WHERE reports precision of 0. But tuned WHERE scores 35 and 60 (the similar pattern can seen in "F").

4.2. RQ2: Does Tuning Change a Learner's Ranking ?

Researchers often use performance criteria to assert that one learner is better than another [4, 15, 5]. For example:

1. Lessmann et al. [4] conclude that Random Forest is considered to be statistically better than CART.

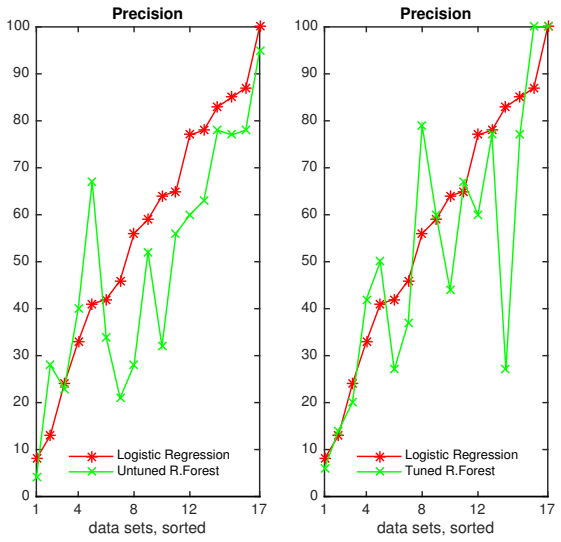

Figure 2: Comparison between Logistic Regression and Random Forest before and after tuning.

2. Also, in Hall et al.'s systematic literature review[5], it is argued that defect predictors based on simple modeling techniques such as LR perform better than complicated techniques such as Random Forest[7].

Given tuning, how stable are these conclusions? Before answering issue, we digress for two comments.

Firstly, it is important to comment on why it is so important to check the conclusions of these particular papers. These papers are prominent publications (to say the least). Hall et al. [5] is the fourth most-cited IEEE TSE paper for 2009 to 2014 with 176 citations (see goo.gl/MGrGr7) while the Lessmann et al. paper [4] has 394 citations (see goo.gl/khTp97)– which is quite remarkable for a paper published in 2009. Given the prominence of these papers, researchers might believe it is appropriate to use their advice without testing that advice on local data sets.

Secondly, while we are critical of the results of Lessmann et al. and Hall et al., it needs to be said that their analysis was excellent and exemplary given the state-of-the-art of the tools used when those papers were written. While Hall et al. did not perform any new experiments, their summarization of so many defect prediction papers have not been equalled before (or since). As to the Lessmann et al. paper, they compared 22 data miners using various data sets (mostly from NASA) [4]. In that study, some learners were tuned using manual methods (C4.5, CART and Random Forest) and some, like SVM-Type learners, were tuned by automatic grid search (for more on grid search, see §2.1).

[7] By three measures, Random Forest is more complicated than LR. Firstly, LR builds one model while Random Forest builds many models. Secondly, LR is just a model construction tool while Random Forest needs both a tool to construct its forest *and* a second tool to infer some conclusion from all the members of that forest. Thirdly, the LR model can be printed in a few lines while the multiple models learned by Random Forest model would take up multiple pages of output.

That said, our tuning results show that it is time to revise the recommendations of those papers. Figure 2 comments on the advice from Hall et al. (that LR is better than Random Forest)L

- In a result that might have been predicted by Hall et al., untuned Random Forests performs comparatively worse than Logistic Regression. Specifically, untuned Random Forest performs worse than Linear regression in in 13 out of 17 data sets.
- However, it turns out that advice is sensitive to the tunings used with Random Forest. After tuning, we find that tuned Random Forest loses to Logistic Regression in only 6 out of 17 data sets.

As to Lessmann et al.'s advice (that Random Forest is better than CART), in Table 4 and Table 5, we saw those counter-examples to that statement. Recall in those tables, tuned CART are better than or equal to tuned Random Forest in $\frac{12}{17}$ and $\frac{7}{17}$ data sets in terms of precision and F-measure, respectively. Prior to tuning experiments, those numbers are $\frac{5}{17}$ and $\frac{1}{17}$. Results from the non-parametric Kolmogorov-Smirnov(KS) Test show that the performance scores of tuned CART and tuned Random Forest are not statistically different. Note that Random Forest is not significantly better than CART, would not have been predicted by Lessmann et al.

Hence we answer RQ2 as "yes": tuning can change how data miners are comparatively ranked.

4.3. RQ3: Does Tuning Select Different Project Factors?

Researchers often use data miners to test what factors have most impact on software projects [6, 7, 8, 9, 10, 11]. Table 6 comments that such tests are unreliable since the factors selected by a data miner are much altered before and after tuning.

Table 6 shows what features are found in the trees generated by the WHERE algorithm (bold shows the features found by the trees from tuned WHERE; plain text shows the features seen in the untuned study). Note that different features are selected depending on whether or not we tune an algorithm.

For example, consider *mfa* which is the number of methods inherited by a class plus the number of methods accessible by member methods of the class. For both goals (precision and "F") *mfa* is selected for 8 and 5 data sets, for the untuned and tuned data miner (respectively). Similar differences are seen with other attributes.

As to why different tunings select for different features, recall from §2.1 that tuning changes how data miners heuristically explore a large space of possible models. As we change how that exploration proceeds, so we change what features are found by that exploration.

In any case, our answer to RQ3 is "yes", tuning changes our conclusions about what factors are most important in software engineering. Hence, many old papers need to be revisited and perhaps revised [6, 7, 8, 9, 10, 11]. For example, one of us (Menzies) used data miners to assert that some factors were more important than others for predicting successful software reuse [8]. That assertion should now be doubted since Menzies did not conduct a tuning study before reporting what factors the data miners found were most influential.

Data set	Precision	F
antV0	**rfc** mfa, loc, cam, dit, dam, lcom3	**None** mfa, loc, cam, dit, dam, lcom3
camelV0	**mfa, wmc, lcom3** mfa, wmc, rfc, loc, cam, lcom3	**None** mfa, wmc, rfc, loc, cam, lcom3
ivy	**cam, dam, npm, loc, rfc, wmc** loc, cam, dam, wmc, lcom3	**cam, dam, npm, loc, rfc, wmc** loc, cam, dam, wmc, lcom3
jeditV0	**mfa, dam, loc** mfa, lcom3, dam, dit, ic	**mfa, dam, loc** mfa, lcom3, dam, dit, ic
log4j	**loc, ic, dit** mfa, lcom3, loc, ic	**mfa, wmc, rfc, loc, npm** mfa, lcom3, loc, ic
lucene	**dit, cam, wmc, lcom3, dam, rfc, cbm, mfa, ic** dit, cam, dam, ic	**dit, lcom3, dam, mfa** dit, cam, dam, cbm, ic
poiV0	**mfa, amc, dam** mfa, loc, amc, dam, wmc, lcom	**mfa, amc, dam** mfa, loc, amc, dam, wmc, lcom
synapse	**loc, dit, rfc, cam, wmc, dam, lcom, mfa, lcom3** loc, mfa, cam, lcom, dam, lcom3	**dam** loc, mfa, cam, lcom, dam, lcom3
velocity	**dit, wmc, cam, rfc, cbo, moa, dam** dit, dam, lcom3, ic, mfa, cbm	**mfa, dit** dit, dam, lcom3, ic, mfa
xercesV0	**wmc** wmc, mfa, lcom3, cam, dam	**cam, dam, avg_cc, loc, wmc, dit, mfa, ce, lcom3** wmc, mfa, lcom3, cam, dam

Table 6: Features selected by tuned WHERE with different goals: **bold** features are those found useful by the tuned WHERE. Also, features shown in plain text are those found useful by the untuned WHERE.

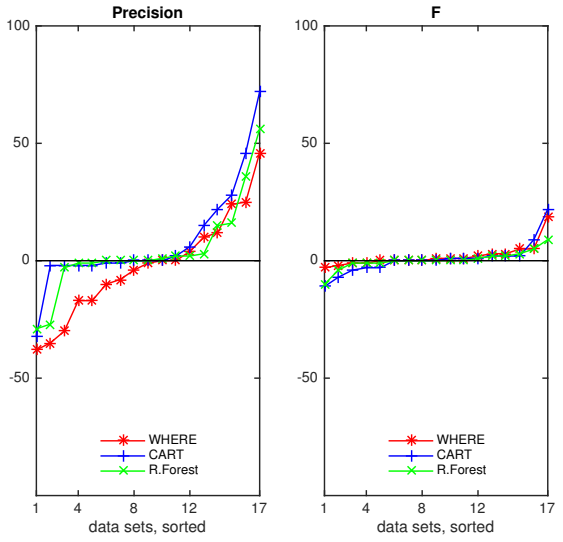

Figure 3: Deltas in performance between *np = 10* and the recommended np's. The recommended np is better when deltas are above zero. *np = 90, 50 and 60* are recommended population size for WHERE, CART and Random Forest by Storn.

4.4. RQ4: Is Tuning Easy?

In terms of the search space explored via tuning, optimizing defect prediction from static code measures is much *smaller* than the standard optimization.

To see this, recall from Algorithm 1 that DE explores a *Population* of size *np = 10*. This is a very small population size since Rainer Storn (one of the inventors of DE) recommends setting *np* to be ten times larger than the number of attributes being optimized [12].

From Table 2, we see that Storn would therefore recommend

	Precision		F	
Learner	CART	WHERE	CART	WHERE
CART	-	0.41	-	0.24
R. Forest	0.12	0.35	0.18	0.18

Table 7: Kolmogorov-Smirnov Tests for distributions of Figure 3

np values of 90, 50, 60 for WHERE, CART and Random Forest (respectively). Yet we achieve our results using a constant *np = 10*; i.e. $\frac{10}{90}, \frac{10}{50}, \frac{10}{60}$ of the recommended search space.

To justify that *np = 10* is enough, we did another tuning study, where all the settings were the same as before but we set *np = 90, np = 50* and *np = 60* for WHERE, CART and Random Forest, respectively (i.e. the settings as recommended by Storn). The tuning performance of learners was evaluated by precision and "F" as before. To compare performance of each learner with different *np*'s, we computed the delta in the performance between *np = 10* and *np* using any of $\{90, 50, 60\}$.

Those deltas, shown in Figure 3, are sorted along the x-axis. In those plots, a zero or negative y value means that *np = 10* performs as well or better than $np \in \{90, 50, 60\}$. One technical aside: the data set orderings in Figure 3 on the x-axis are not the same (that is, if $np > 10$ was useful for optimizing one data set's precision score, it was not necessary for that data set's F-measure score).

Figure 3 shows that the median improvement is zero; i.e. *np = 10* usually does as well as anything else. This observation is supported by the KS results of Table 7. At a 95% confidence, the KS threshold is $1.36\sqrt{34/(17*17)} = 0.46$, which is greater than the values in Figure 3. That is, no result in Figure 3 is significantly different to any other– which is to say that there is no evidence that *np = 10* is a poor choice of search space size.

Another measure showing that tuning is easy (for static code defect predictors) is the number of evaluations required to complete optimization (see next section). That is, we answer RQ4 as "yes", tuning is surprisingly easy– at least for defect predictors and using DE.

4.5. RQ5: Is Tuning Impractically Slow?

The number of evaluations/runtimes used by our optimizers is shown in Table 8 and Table 9. WHERE's runtimes are slower than CART and Random Forest since WHERE has yet to benefit from decades of implementation experience with these older algorithms. For example, SciKitLearn's CART and Random Forest make extensive use of an underlying C library whereas WHERE is a purely interpreted Python.

Looking over Table 8 and Table 9, the general pattern is that 50 to 80 evaluations suffice for finding the tuning improvements reported in this paper. 50 to 80 evaluations are much fewer than our pre-experimental intuition. Prior to this paper, the authors have conducted numerous explorations of evolutionary algorithms for search-based SE applications [51, 59, 43, 44, 60]. Based on that work, our expectations were that non-parametric evolutionary optimization would take thousands, if not millions, of evaluations of candidate tunings. This turned out not to be that case.

Hence, we answer RQ5 as "no": tuning is so fast that it could (and should) be used by anyone using defect predictors.

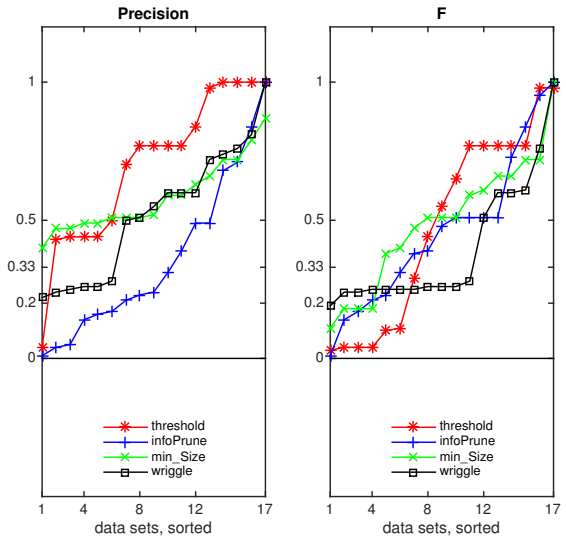

Figure 4: Four representative tuning values in WHERE with precision and F-measure as the tuning goal, respectively.

As to why DE can tune defect predictors so quickly, that is an open question. One possibility is that the search space within the control space of these data miners has many accumulative effects such that one decision can cascade into another (and the combination of decisions is better than each separate one). DE would be a natural tool for reasoning about such "cascades", due to the way it mashes candidates together, then inserts the result back into the frontier (making them available for even more mashing at the next step of the inference).

4.6. RQ6: Should we use "off-the-shelf" Tunings?

In Figure 4, we show how tuning selects the optimal values for tuned parameters. For space limitation, only four parameters from WHERE learner are selected as representatives and all the others can be found in our online support documents[8]. Note that the tunings learned were different in different data sets and for different goals. Also, the tunings learned by DE were often very different to the default (the default values for *threshold*, *infoPrune*, *min_Size* and *wriggle* are 0.5, 0.33, 0.5 and 0.2, respectively). That is, to achieve the performance improvements seen in the paper, the default tuning parameters required a wide range of adjustments.

Hence, we answer RQ6 as "no" since, to achieve the improvements seen in this paper, tuning has to be repeated whenever the goals or data sets are changed. Given this requirement to repeatedly run tuning, it is fortunate that (as shown above) tuning is so easy and so fast (at least for defect predictors from static code attributes).

5. Reliability and Validity

Reliability refers to the consistency of the results obtained

[8] https://goo.gl/aHQKtU

Datasets	Tuned_Where	Naive_Where	Tuned_CART	Naive_CART	Tuned_RanFst	Naive_RanFst
antV0	50 / 95.47	1.65	60 / 5.08	0.08	60 / 9.78	0.20
antV1	60 / 224.67	3.03	50 / 6.52	0.12	60 / 14.13	0.25
antV2	70 / 644.99	8.24	50 / 9.00	0.24	60 / 16.75	0.44
camelV0	70 / 690.62	7.93	70 / 12.68	0.24	110 / 28.49	0.34
camelV1	60 / 1596.77	23.56	60 / 17.13	0.27	70 / 33.96	0.77
ivy	60 / 66.69	0.97	60 / 4.26	0.07	60 / 8.89	0.19
jeditV0	80 / 459.30	5.33	80 / 8.69	0.11	90 / 18.40	0.32
jeditV1	60 / 421.56	6.59	80 / 9.05	0.12	80 / 17.93	0.36
jeditV2	90 / 595.56	6.88	60 / 7.90	0.14	110 / 27.34	0.38
log4j	50 / 76.09	1.33	50 / 2.60	0.06	80 / 9.69	0.15
lucene	80 / 236.45	2.60	70 / 6.07	0.10	60 / 9.77	0.25
poiV0	60 / 263.12	3.92	70 / 7.42	0.09	130 / 25.86	0.28
poiV1	50 / 398.33	6.94	70 / 9.31	0.13	50 / 12.67	0.29
synapse	70 / 144.09	1.85	50 / 3.88	0.07	50 / 8.13	0.19
velocity	60 / 184.10	2.68	50 / 4.27	0.07	100 / 15.18	0.21
xercesV0	60 / 136.87	1.98	80 / 9.17	0.10	70 / 14.17	0.22
xercesV1	80 / 1173.92	12.78	60 / 10.47	0.16	50 / 18.27	0.40

Table 8: Evaluations/runtimes and runtimes for tuned and default learners(in sec), optimizing for precision.

Datasets	Tuned_Where	Naive_Where	Tuned_CART	Naive_CART	Tuned_RanFst	Naive_RanFst
antV0	50 / 93.38	1.39	50 / 3.52	0.08	70 / 9.89	0.17
antV1	60 / 186.95	3.18	50 / 6.18	0.12	60 / 13.39	0.25
antV2	90 / 654.34	8.08	60 / 8.79	0.18	120 / 27.56	0.36
camel	50 / 543.28	9.65	80 / 17.00	0.28	70 / 22.52	0.41
camelV1	60 / 1808.03	26.98	110 / 31.92	0.28	70 / 37.00	0.85
ivy	60 / 74.50	1.18	60 / 4.72	0.08	60 / 10.39	0.21
jeditV0	80 / 518.47	6.11	60 / 7.9	0.10	60 / 14.32	0.37
jeditV1	70 / 576.29	6.89	70 / 8.13	0.10	70 / 17.42	0.34
jeditV2	80 / 657.59	7.93	70 / 10.34	0.15	80 / 20.20	0.40
log4j	70 / 123.48	1.59	50 / 2.92	0.08	50 / 7.67	0.17
lucene	60 / 219.02	3.68	60 / 6.89	0.12	70 / 13.06	0.35
poiV0	60 / 314.53	4.82	60 / 7.80	0.10	80 / 19.29	0.32
poiV1	50 / 446.05	7.55	50 / 7.62	0.14	110 / 27.23	0.36
synapse	60 / 138.75	1.83	60 / 4.87	0.08	90 / 13.29	0.17
velocity	60 / 211.88	3.13	60 / 5.51	0.10	60 / 11.58	0.27
xercesV0	80 / 178.49	2.02	60 / 7.47	0.11	80 / 17.31	0.28
xercesV1	80 / 1370.89	14.42	60 / 11.07	0.19	80 / 25.27	0.46

Table 9: Evaluations/runtimes and runtimes for tuned and default learners(in sec), optimizing for F-Measure.

from the research. For example, how well independent researchers could reproduce the study? To increase external reliability, this paper has taken care to either clearly define our algorithms or use implementations from the public domain (SciKitLearn). Also, all the data used in this work is available on-line in the PROMISE code repository and all our algorithms are online at github.com/ai-se/where.

External validity checks if the results are of relevance for other cases, or can be generalized from samples to populations. The examples of this paper only relate to precision, recall, and the F-measure but the general principle (that the search bias changes the search conclusions) holds for any set of goals. Also, the tuning results shown here only came from one software analytics task (defect prediction from static code attributes). There are many other kinds of software analytics tasks (software development effort estimation, social network mining, detecting duplicate issue reports, etc) and the implication of this study for those tasks is unclear. However, those other tasks often use the same kinds of learners explored in this paper so it is quite possible that the conclusions of this paper apply to other SE analytics tasks as well.

6. Conclusions

Our exploration of the six research questions listed in the introduction show that when learning defect predictors for static code attributes, analytics without parameter tuning are considered *harmful* and *misleading*:

- Tuning improves the performance scores of a predictor. That improvement is usually positive (see Figure 1) and sometimes it can be quite dramatic (e.g. precision changing from 0 to 60%).
- Tuning changes conclusions on what learners are better than others. Hence, it is time to revisit numerous prior publications of our own [15] and others [4, 5].
- Also, tuning changes conclusions on what factors are most important in software development. Once again, this means that old papers may need to be revised including those some of our own [8] and others [6, 7, 9, 10, 11].

As to future work, it is now important to explore the implications of these conclusions to other kinds of software analytics. This paper has investigated *some* learners using *one* optimizer. Hence, we can make no claim that DE is the *best* optimizer for *all* learners. Rather, our point is that there exists at least some learners whose performance can be dramatically improved by at least one simple optimization scheme. We hope that this work inspires much future work as this community develops and debugs best practices for tuning software analytics.

Finally, on a more general note, we point out that Fürnkranz [61] says data mining is inherently a multi-objective optimization problem that seeks the smallest model with the highest performance, that generalizes best for future examples (perhaps learned in minimal time using the least amount of data). In this view, we are using DE to optimize an optimizer. Perhaps a better approach might be to dispense with the separation of "optimizer" and "learner" and combine them both into

one system that learns how to tune itself as it executes. If this view is useful, then instead of adding elaborations to data miners (as done in this paper, or by researchers exploring hyper-heuristics [62]), it should be possible to radically simplify optimization and data mining with a single system that rapidly performs both tasks.

Acknowledgments

The work has partially funded by a National Science Foundation CISE CCF award #1506586.

References

[1] T. Menzies, C. Pape, M. Rees-Jones, The promise repository of empirical software engineering data (Feb 2015).
URL http://openscience.us/repo

[2] D. Rodriguez, I. Herraiz, R. Harrison, On software engineering repositories and their open problems, in: Proceedings RAISE'12, 2012.

[3] N. Nagappan, T. Ball, Static analysis tools as early indicators of pre-release defect density, in: ICSE '05, ACM, 2005, pp. 580–586.

[4] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Trans. Softw Eng. 34 (4) (2008) 485–496.

[5] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic review of fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2012) 1276–1304.

[6] R. M. Bell, T. J. Ostrand, E. J. Weyuker, The limited impact of individual developer data on software defect prediction, Empirical Software Engineering 18 (3) (2013) 478–505.

[7] F. Rahman, P. Devanbu, How, and why, process metrics are better, in: ICSE '13, IEEE Press, 2013, pp. 432–441.

[8] T. Menzies, J. D. Stefano, More success and failure factors in software reuse, IEEE Trans. Softw Eng. 29 (5) (2003) 474–477, available from http://menzies.us/pdf/02sereuse.pdf.

[9] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: ICSE '08, ACM, 2008, pp. 181–190. doi:10.1145/1368088.1368114.
URL http://doi.acm.org/10.1145/1368088.1368114

[10] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: PROMISE'07, IEEE, 2007, pp. 9–9.

[11] K. Herzig, S. Just, A. Rau, A. Zeller, Predicting defects using change genealogies, in: ISSRE '13, IEEE, 2013, pp. 118–127.

[12] R. Storn, K. Price, Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces, Journal of global optimization 11 (4) (1997) 341–359.

[13] J. Bergstra, Y. Bengio, Random Search for Hyper-Parameter Optimization, Journal of Machine Learning Research 13 (2012) 281–305.

[14] K. Gao, T. M. Khoshgoftaar, H. Wang, N. Seliya, Choosing software metrics for defect prediction: An investigation on feature selection techniques, Softw. Pract. Exper. 41 (5) (2011) 579–606. doi:10.1002/spe.1043.
URL http://dx.doi.org/10.1002/spe.1043

[15] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw Eng. 33 (1) (2007) 2–13, available from http://menzies.us/pdf/06learnPredict.pdf.

[16] K. O. Elish, M. O. Elish, Predicting defect-prone software modules using support vector machines, Journal of Systems and Software 81 (5) (2008) 649 – 660. doi:http://dx.doi.org/10.1016/j.jss.2007.07.040.
URL http://www.sciencedirect.com/science/article/pii/S016412120700235X

[17] L. Pelayo, S. Dick, Applying novel resampling strategies to software defect prediction, in: Fuzzy Information Processing Society, 2007. NAFIPS '07. Annual Meeting of the North American, 2007, pp. 69–72. doi:10.1109/NAFIPS.2007.383813.

[18] S. Kim, H. Zhang, R. Wu, L. Gong, Dealing with noise in defect prediction, in: ICSE '11, ACM, 2011, pp. 481–490. doi:10.1145/1985793.1985859.
URL http://doi.acm.org/10.1145/1985793.1985859

[19] N. Nagappan, B. Murphy, V. Basili, The influence of organizational structure on software quality: An empirical case study, in: ICSE '08, ACM, 2008, pp. 521–530. doi:10.1145/1368088.1368160.
URL http://doi.acm.org/10.1145/1368088.1368160

[20] A. E. Hassan, Predicting faults using the complexity of code changes, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 78–88. doi:10.1109/ICSE.2009.5070510.
URL http://dx.doi.org/10.1109/ICSE.2009.5070510

[21] D. Baker, A Hybrid Approach to Expert and Model-based Effort Estimation, Ph.D. thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University (2007).

[22] B. Anda, D. I. K. Sjøberg, A. Mockus, Variability and reproducibility in software engineering: A study of four companies that developed the same system, IEEE Trans. Softw Eng. 35 (3) (2009) 407–429.

[23] E. Arisholm, L. Briand, Predicting fault-prone components in a java legacy system, in: ISESE '06, 2006, available from http://simula.no/research/engineering/publications/Arisholm.2006.4.

[24] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Where the bugs are, in: ISSTA '04, ACM, 2004, pp. 86–96.

[25] F. Rahman, D. Posnett, P. Devanbu, Recalling the 'imprecision' of cross-project defect prediction, in: FSE'12, ACM, 2012, pp. 61:1–61:11. doi:10.1145/2393596.2393669.
URL http://doi.acm.org/10.1145/2393596.2393669

[26] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, L. Bairavasundaram, How do fixes become bugs?, in: ESEC/FSE '11, 2011, pp. 26–36.

[27] T. Menzies, E. Kocaguneli, L. Minku, F. Peters, B. Turhan, Sharing Data and Models in Software Engineering, Morgan Kaufmann, 2015.

[28] F. Rahman, S. Khatri, E. Barr, P. Devanbu, Comparing static bug finders and statistical prediction, in: ICSE 2014, ACM, 2014, pp. 424–434. doi:10.1145/2568225.2568269.
URL http://doi.acm.org/10.1145/2568225.2568269

[29] T. Menzies, D. Raffo, S. Setamanit, Y. Hu, S. Tootoonian, Model-based tests of truisms, in: ASE '02, 2002, available from http://menzies.us/pdf/02truisms.pdf.

[30] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, E. J. W. Jr, Does bug prediction support human developers? findings from a google case study, in: ICSE '13, IEEE, 2013, pp. 372–381.

[31] S. Rakitin, Software Verification and Validation for Practitioners and Managers, Second Edition, Artech House, 2001.

[32] A. Tosun, A. Bener, R. Kale, AI-based software defect predictors: Applications and benefits in a case study, in: IAAI, 2010.

[33] A. Tosun, A. Bener, B. Turhan, Practical considerations of deploying ai in defect prediction: A case study within the Turkish telecommunication industry, in: PROMISE'09, 2009.

[34] F. Shull, V. B. ad B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, M. Zelkowitz, What we have learned about fighting defects, in: Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada, IEEE, 2002, pp. 249–258.

[35] M. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal 15 (3).

[36] L. Breiman, A. Cutler, Random forests, https://www.stat.berkeley.edu/ breiman/RandomForests (2001).

[37] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification and Regression Trees, 1984.

[38] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, T. Zimmermann, Local versus global lessons for defect prediction and effort estimation, IEEE Trans. Softw Eng. 39 (6) (2013) 822–834.

[39] 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, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[40] U. M. Fayyad, I. H. Irani, Multi-interval discretization of continuous-valued attributes for classification learning, in: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 1993, pp. 1022–1027.

[41] P. R. Cohen, Empirical Methods for Artificial Intelligence, MIT Press, 1995.

[42] R. Holte, Very simple classification rules perform well on most commonly used datasets, Machine Learning 11 (1993) 63.

[43] M. S. Feather, T. Menzies, Converging on the optimal attainment of requirements, in: IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 2002, available from http://menzies.us/pdf/02re02.pdf.

[44] T. Menzies, O. El-Rawas, J. Hihn, M. Feather, B. Boehm, R. Madachy, The business case for automated software engineerng, in: ASE '07, ACM, 2007, pp. 303–312, available from http://menzies.us/pdf/07casease-v0.pdf.

[45] A. Goldberg, On the complexity of the satisfiability problem, in: Courant Computer Science conference, No. 16, New York University, NY, 1979.

[46] F. Glover, C. McMillan, The general employee scheduling problem. an integration of ms and ai, Computers & Operations Research 13 (5) (1986) 563 – 573.

[47] R. P. Beausoleil, MOSS: multiobjective scatter search applied to nonlinear multiple criteria optimization, European Journal of Operational Research 169 (2) (2006) 426 – 449.

[48] J. Molina, M. Laguna, R. Marti, R. Caballero, Sspmo: A scatter tabu search procedure for non-linear multiobjective optimization, INFORMS Journal on Computing.

[49] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, A. Beham, Abyss: Adapting scatter search to multiobjective optimization, IEEE Trans. Evol. Comp. 12 (4) (2008) 439–457.

[50] H. Pan, M. Zheng, X. Han, Particle swarm-simulated annealing fusion algorithm and its application in function optimization, in: International Conference on Computer Science and Software Engineering, 2008, pp. 78–81.

[51] J. Krall, T. Menzies, M. Davies, Gale: Geometric active learning for search-based software engineering, To appear, IEEE Trans. Softw Eng.

[52] M. Zuluaga, A. Krause, G. Sergent, M. Püschel, Active learning for multi-objective optimization, in: International Conference on Machine Learning (ICML), 2013.

[53] J. Vesterstrom, R. Thomsen, A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems, in: IEEE Congress on Evolutionary Computation '04, 2004. doi:10.1109/CEC.2004.1331139.

[54] M. Omran, A. P. Engelbrecht, A. Salman, Differential evolution methods for unsupervised image classification, in: IEEE Congress on Evolutionary Computation '05, Vol. 2, 2005, pp. 966–973.

[55] I. Chiha, J. Ghabi, N. Liouane, Tuning pid controller with multi-objective differential evolution, in: ISCCSP '12, IEEE, 2012, pp. 1–4.

[56] T. Robič, B. Filipič, Demo: Differential evolution for multiobjective optimization, in: Evolutionary Multi-Criterion Optimization, Springer, 2005, pp. 520–533.

[57] Q. Zhang, H. Li, Moea/d: A multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evol. Comp 11 (6) (2007) 712–731. doi:10.1109/TEVC.2007.892759.
URL http://dx.doi.org/10.1109/TEVC.2007.892759

[58] W. Huang, H. Li, On the differential evolution schemes in moea/d, in: ICNC '10, Vol. 6, 2010, pp. 2788–2792.

[59] J. Krall, T. Menzies, M. Davies, Better model-based analysis of human factors for safe aircraft approach, To appear, IEEE Transactions on Human Machine Systems.

[60] P. G. II, T. Menzies, S. Williams, O. El-Rawas, Understanding the value of software engineering technologies, in: ASE '09, IEEE, 2009, pp. 52–61. doi:10.1109/ASE.2009.93.
URL http://dx.doi.org/10.1109/ASE.2009.93

[61] J. Fürnkranz, P. Flach, Roc 'n' rule learning: towards a better understanding of covering algorithms, Machine Learning 58 (1) (2005) 39–77. doi:http://dx.doi.org/10.1007/s10994-005-5011-x.

[62] Y. Jia, M. B. Cohen, M. Harman, J. Petke, Learning combinatorial interaction testing strategies using hyperheuristic search, in: ICSE '15, 2015.