Analytics Without Parameter Tuning Considered Harmful?

Wei Fu, Tim Menzies, Vivek Nair
Computer Science, North Carolina State University, Raleigh, USA
fuwei.ee, tim.menzies, vivekaxl@gmail.com

ABSTRACT

One of the "black arts" of data mining is setting the tuning parameters that control choices within a data miner. We offer a simple, automatic, and very effective method for finding those tunings.

For the purposes of learning software defect predictors this optimization strategy can quickly find good tunings that dramatically change the performance of a learner. For example, in this paper we show tunings that alter detection precision from 2% to 98%.

These results prompt for a change to 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* first conducting a tuning optimization study. The implications for other kinds of software analytics is now an open and pressing questions.

Categories/Subject Descriptors: D.2.8 [Software Engineering]: Product metrics; I.2.6 [Artificial Intelligence]: Induction

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

1 Introduction

"So those who are last now will be first then,
and those who are first will be last."
– Matthew 20:16

In the 21st century, it is now impossible to manually browse all the available software project data. The PROMISE repository of SE data has grown to 200+ projects [?] and this is just one of over a dozen open-source repositories that are readily available to researchers [?]. For example, the time of this writing (Feb 2015), 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 information overload, researchers in empirical SE use data miners to (e.g.) 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 [?]. In addition, they have some generality across multiple projects: e.g. defect predictors developed at NASA [?] have also been successfully applied in Turkey [?]. Such detectors reduce the effort required for defect prediction: if inspection teams let themselves be guided by defect predictors then they can find 80% to 88% of the defects after inspecting 20% to 25% of the code [?, ?].

One of the "black arts" of data mining is setting the tuning parameters that control the choices within that data miner. Many researchers have applied automatic optimizers to find good tunings. The search-based SE community has developed techniques for tuning that treat parameters as a configuration search space [?, ?]. The field of *hyper-heuristics* explores methods for auto-adapting options within the device searching for solutions [?] in applications like test-case generation and code refactorings.

To the best of our knowledge, this paper is the first extensive exploration of applying automatic optimizers to tune data miners for defect prediction, (though see [?, ?, ?, ?] for tuning effort estimators). The one similar study we can find in the period 2004 to 2014[1] was Bouktiff et al. [?] who used simulated annealing to tune Bayes classifiers. That was a very limited study, applied to a single defect data set. The analysis of this paper is far more extensive and offers the following novel result:

- Tuning static code defect predictors is *remarkably simple* and can *dramatically improve the performance* of those learners.

Prior to these experiments, our expectation was tuning would be an extensive and expensive evolutionary optimization procedure. A standard run of such evolutionary optimizers requires thousands, if not millions, of evaluations. To our surprise, we achieved dramatic improvements in the performance scores of our data miners after mere 50 to 80 evaluations (!!) of a very simple evolutionary optimizer called differential evolution [?]. Better yet, those tunings (found so quickly) have a dramatic change to the performance of a learner. For example, in this paper we show that tuning can alter the precision of a software defect predictor from 2% to 98%.

Our tuning results challenge much prior work in software analytics. Firstly, there exist research papers that use data miners to show that certain factors are more influential than others for (say) predicting defects. As shown below, such conclusions can be dramatically changed by the tuning process since those "influential" factors are very different pre- and post- tuning. Also, those factors tend to change from project to project or if the goal of the tuning is altered. Hence, many old papers need to be revisted and perhaps revised [?, ?, ?, ?, ?, ?]. For example, one of us (Menzies) used data miners to assert that some factors were more important than others for predicting successful software reuse [?]. That assertion should now be doubted since that Menzies study did not conduct a tuning study before reporting what factors the data miners found where most influential.

Secondly, several prominent IEEE TSE papers [?,?,?] have claimed that learnerX is better than learnerY for some software analytics task. For example, a recent IEEE TSE article claimed that the CART decision tree learner was far worse than Random Forests for software defect prediction [?]. Such conclusions do not survive tuning.

[1] See the GECCO and SSBSE proceedings at goo.gl/2MY602 and goo.gl/uzvU8e.

For example, after tuning, the worst untuned learner (CART) can out-perform the supposedly best learner (Random Forests). Hence, all those prior results that ranked learners for software analytics now need to be revisited and perhaps revised.

Thirdly, it is standard practice to use the default "off-the-shelf" tunings for data mining tools (previously we have defended that approach on methodological grounds arguing that it encourages reproducibility [**?**]). That "off-the-shelf" policy can no longer be condoned. For example, one such default setting in the Python SciKitLearn toolkit [**?**] is to use $F = 10$ decision trees in Random Forest classifiers. Our optimizer settled on F values that were nowhere near that default:

$$F \in \left\{ \begin{array}{l} 55, 65, 70, 82, 88, 96, 100, 102, 104, 107, \\ 108, 119, 133, 140, 140, 147, 145, 142 \end{array} \right\}$$

In summary,it is now an open and pressing research issue to check if analytics without parameter tuning is considered *harmful* or, at the very least, *misleading*. Clearly, we must now doubt conclusions based on "off-the-shelf" tunings. Further, it is no longer enough to just run a data miner and report the result *without* first conducting an tuning optimizations study.

1.1 Preliminaries

Before beginning, we offer four caveats on these results.

Firstly, we are *not* saying that *all* learning requires tuning. For example, when proposing fixes to software, Weimar et al. use a genetic learner to propose patches [**?**]. Consider one performance criteria for that work; i.e. that that that method can find and fix known bugs. Note that this is a *competency criteria* which does not include the phrase "*better than*". Such performance criteria do not requiring tuning. However, once *better than* enters the performance criteria (as done in [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]) then this becomes a race between competing methods (or attributes). In such a race, it is unfair to hobble one competitor with poor tunings.

Secondly, 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.

Thirdly, this paper explores *some* learners using *one* optimizer. Hence, it makes no claim that this 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.

Fourthly, it would be incorrect to say that this paper is arguing that software analytics is somehow wrong-headed, misguided, and we should not do it anymore. In the age of the Internet and global access to software engineering data, there exists the problem of information overload. *Something* must be done to allow analysts to make conclusions via an automatic analysis over a lot of data. The results of this paper is that for a particular local context (a specific data set and a specific goal) there exists methods for optimizing the conclusions reached in that context. Those conclusions may not generalize to other contexts but this is not a council for despair. While there may not exist general conclusions, there does seem to exist general methods for finding local conclusions in a particular context. Further, as shown below, those methods may be very simple to implement and very fast to execute.

2 Motivating Example

This section offers a small demonstration of the impact of tuning parameters. It also demonstrates that this issue of tuning effects not just the complex data miners discussed later in the paper but also applies for even very simple learning schemes.

One way to generate a defect predictor from static code is to use linear regression. This is a standard statistical method that fits a straight line to a set of points. The line offers a set of predicted values. If the points are somewhat scattered, then a single regression line cannot pass through all points and the distance from these predicted values to the actual values is a measure of the error associated with that line. Linear regression search for a line that minimizes that error and maximizes the correlation[2] denoted as c.

Suppose a researcher wants to use linear regression to test if Halstead's [**?**] measures of function complexity (number of symbols programmers has to understand) are *better than* mere lines of code for predicting software defects. That researcher might argue that Halstead's cognitive approach to software bugs is better suited to code refactoring tools since it offers more ways to alter functions that just some coarse grain lines of code measure.

That researcher might test that belief by using studying historical defect logs with linear regression. Here are two equations (learned from the NASA data at goo.gl/pGDfvp) that use just lines of code or the Halstead measures N, V, L, D, I, E, B, T seen in a software module (in this case, a function). Note that the Halstead correlation c_2 is worse than those from lines of code c_1. This result seems to suggest that despite their potential support for refactoring, our researchers should not use Halstead.

$measures$	$d = \#defects$	$correlation$
LOC	$d_i = 0.0164 + 0.0114LOC$	$c_1 = 0.65$
$Halstead$	$d_2 = 0.231 + 0.00344N$	
	$+0.0009V - 0.185L$	
	$-0.0343D - 0.00541I$	
	$+0.000019E + 0.711B$	
	$-0.00047T$	$c_2 = -0.36$

Enter tuning. Suppose the defect predictors d_1 and d_2 learned from LOC or Halstead are used to to call an inspection team to check for errors in certain parts of the code using the rule

$$inspect = \left\{ \begin{array}{l} d_i \geq T \rightarrow Yes \\ d_i < T \rightarrow No \end{array} \right. \tag{1}$$

Figure **??** shows the effects of tuning the T variable. Not surprisingly, at $T = 0$, all modules get inspected so the false alarm rate is very high. To reduce that problem, we can increase T. Figure **??** reports that the false alarm rate falls below 20% at $T = 0.45$ (for Halstead).

One important lesson from Figure **??** is that the "best" tunings are context specific. For mission critical systems (e.g. a nuclear power plant), management might accept the cost of high false alarm rates if it meant increasing the probability of detecting errors. For such contexts, we might recommend some very small value of T.

The other important lesson from Figure **??** is that, with tuning, a seemingly poor detector can work just as well as seemingly better ones. Note that either the Halstead or LOC detector can reach some desired level of recall, regardless of their correlations, just by selecting the appropriate threshold value. For example, in Figure **??**, see the recall=75% values found at *either* $d_i \geq 0.65$ or $d_2 \geq 0.45$

[2]$Correlation$ measures how closely two variables co-vary. It ranges from from -1 (perfect negative correlation) through 0 (no correlation) to +1 (perfect positive correlation). Let a_i and p_i denote some actual and predicted values respectively; and n and $\overline{x}$ denote the number of observations and the mean of the n observations, respectively. Then: $S_{PA} = (\sum_i (p_i - \overline{p})(a_i - \overline{a}))/(n-1)$ and $S_p = (\sum_i (p_i - \overline{p})^2)(n-1)$ and $S_a = (\sum_i (a_i - \overline{a})^2)/(n-1)$ and correlation is $c = S_{PA}/\sqrt{SpSa}$.

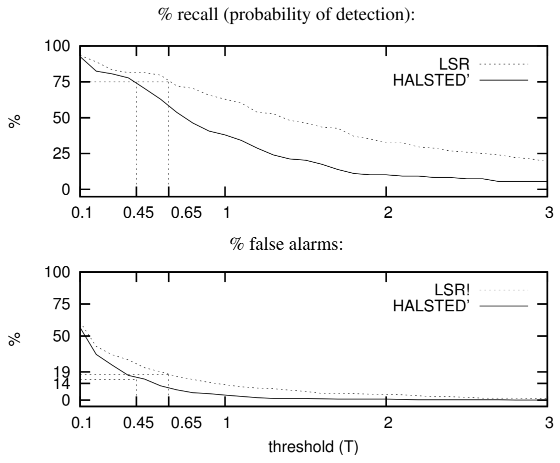

Figure 1: Y-axis shows probability of false alarm, and probability of recognizing defective modules seen using $d_i \geq T$. Curves calculated from the KC2 dataset from the PROMISE repository goo.gl/pGDfvp.

(and at the threshold, the false alarm rates were very similar: 14% and 19%).

The core point of this example is that the true value of a detector could not be assessed *without* conducting a tuning study in the context of some business case (in this case, issuing a request to an inspection team to review some module). This is strong motivation to explore the issue of tunings.

3 Background Notes

The rest of this paper repeats the analysis of the last section, but for much more complex learners. In those results, shown below, we will find examples of *rank reversals*; i.e. after tuning the top ranked defector becomes last and vice versa. Such examples motivate the extensive exploration of tuning in software analytics. Before presenting those results, this section offers some background notes on defect prediction, performance measures, and optimization with differential evolution.

3.1 Defect Prediction

This section is our standard introduction to defect prediction [**?**], plus some new results from Rahman et al.'s 2014 FSE paper [**?**].

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, is important to thoroughly *test* software before it is *used*. Such testing can be very expensive. Software assessment budgets are finite while assessment effectiveness increases exponentially with assessment effort. Lowry et al. warn that the state space explosion problem imposes strict limits on how much a system can be explored via automatic formal methods [**?**]. For standard black-box testing methods, a *linear* increase in the confidence C that we have found all defects can take *exponentially* more effort. For example, for one-in-a-thousand detects, moving C from 90% to 98% nearly doubles the number of tests (2301 to 3910 black box probes, respectively)[3].

[3] 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, $N(0.90, 10^{-3}) = 2301$.

Exponential costs quickly exhaust finite resources. Standard practice is to apply the best available assessment methods on the sections of the program that the best available domain knowledge declares is most critical. We endorse this approach. Clearly, the most critical sections require the best known assessment methods. However, this focus on certain sections can blind us to defects in other areas. Therefore, standard practice should be augmented with a *lightweight sampling policy* to explore the rest of the system. This sampling policy will always be incomplete. Nevertheless, it is the recommended option when resources do not permit a complete assessment of the whole system.

One such lightweight sampling policy is defect predictors learned from static code attributes. Given software described using (for example) the attributes of Figure **??**, it is possible to use standard data mining technology to learn where the probability of software defects is highest. Such defect defect predictors learned from static code attributes 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 [**?**]. 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 [**?**].

Widely used: Many researchers and industrial practitioners use static attributes to guide software quality predictions. Defect prediction models have been reported to have been used at Google [**?**]. Verification and validation (V&V) textbooks ([**?**]) advise using static code complexity attributes to decide which modules are worthy of manual inspections. For several years, one of us (Menzies) worked on-site at the NASA software Independent Verification and Validation facility and he knows of several large government software contractors that won't review software modules *unless* tools like McCabe predict that they are fault prone.

Useful: A standard result for defect predictors is that they find the location of 70% (or more) of the defects in code [**?**]. Defect predictors developed at NASA [**?**] have also been used in software development companies outside the US (in Turkey). When the inspection teams focused on the modules that trigger the defect predictors, they found up to 70% of the defects using just 40% of their QA effort (measured in staff hours) [**?**]. A subsequent study on the Turkish software compared how much code needs to be inspected using random selection vs. selection via defect predictors. Using random testing, 87% of the files would have to be inspected in order to detect 87% of the defects. Further, if the inspection process was restricted to the 25% of the files that trigger the defect predictors, then 88% of the defects could be found. That is, the same level of defect detection (after inspection) can be achieved using $(87-25)/87 = 71\%$less effort [**?**].

Defect prediction scales well to a commercial context. Defect predicting technology has been commercialized in many tools including *Predictive* [**?**], a product suite to analyze and predict defects in software projects. Predictive was observed to highlight simimilar issues to those found with the more expensive tools. Significantly, Predictive was able to faster process a larger code base than the more expensive tool [**?**].

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* [**?**] concluded that manual software reviews can find ≈60% of defects. In other work, Raffo documents the typical defect detection capability of industrial review methods: around 50% for full Fagan inspections [**?**] to 21% for less-structured inspections.

Note only do static code defect predictors perform well compared

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.

Figure 2: OO measures used in our defect data sets. Last line is the dependent attribute (whether a defect is reported to a post-release bug-tracking system).

to manual methods, they also are competititve with certain automatic methods. At ICSE'14, Rahman et al. [?] offered an extensive comparison of:

- The static code analysis tools FindBugs, Jlint, and Pmd;
- Against static code defect predictors (which they called "statistical defect prediction") build using logistic regression.

That study assessed the cost-effectiveness of finding bugs using either method. They found no significant differences in the cost-effectiveness of the two approaches. Given this equivalence, it is significant to note that static code defect prediction can be quickly adapted to new languages, just by building lightweight parsers that can extract information like Figure **??**. The same cannot be said for static code analyzers that require extensive modification before they can be applied to a new language.

3.2 Data Mining Algorithms

This section describes this study's learners (CART [?], Random Forest [?], and WHERE [?]) and their tuning parameters (summarised in Figure **??**).

Our implementations for CART and Random Forest comes from SciKitLearn [?]. WHERE is available from github.com/ai-se/where[4]. WHERE has the most tuning parameters. This turns out to be important since, in the experiments shown below, tuned WHERE outperformed the other learners– a result suggesting that if you are going to tune learners, then use one with many tuning options.

3.3 Why Study These Algorithms?

This paper studies WHERE since this was the first learner we tried to tune and, as shown below, it offers an interesting case study on the benefits of tuning.

This paper also studies CART and Random Forest since these were used in a recent IEEE TSE paper by Lessmann et al. [?] that compared 21 different learners for software defect prediction:

- *Statistical classifiers:* Linear discriminant analysis, Quadratic discriminant analysis, Logistic regression, Naive Bayes, Bayesian networks, Least-angle regression, Relevance vector machine,
- *Nearest neighbor methods:* k-nearest neighbor, K-Star
- *Neural networks:* Multi-Layer Perceptron, Radial bias function network,
- *Support vector machine-based classifiers:* Support vector machine, Lagrangian SVM Least squares SVM, Linear programming, Voted perceptron,
- *Decision-tree approaches:* C4.5 decision tree, CART, Alternating decision tree.
- *Ensemble methods:* Random Forest, Logistic Model Tree.

In that study, CART was severely trounced (was ranked last) and Random Forest was the standout best method. The experiments shown below both confirm and refute that ranking. In a result consistent with the prior result, untuned Random Forest performs best. However, after tuning, the worst learner found by Lessmann et al. (CART) performed better than Random Forest.

3.4 Learners and Their Tunings

CART, Random Forest, and WHERE are all tree learners that divide a data set, then recurse on each split. 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 recursion stops. For CART and Random Forest use a user-supplied constant for this parameter while WHERE computes m=*min samples leaf* from the size of the data sets via $m = size^{minsize}$ (so, for WHERE, *min size* is the parameter to be tuned).

These learners generate numeric predictions which are converted into binary "yes/no" decisions via Equation **??**. Hence, they all use the *threshold* value T discussed in §**??**.

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[6].
- Random Forest divides data like CART, but it builds $F > 1$ trees, each time with a subset of the attributes (selected at random).
- WHERE projects the data on to a dimension it synthesizes from the raw data using a process analogous to principle component analysis[7]. WHERE divides at the median point of that projec-

[4]Wei: need a nice where repo. with an data and code examples sub-directory. clean code. throw out anything not needed. before april1.

[6]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 $\sum_i \left(\sqrt{v_i} \times n_i/(\sum_i n_i)\right)$.

[7]PCA synthesises new attributes $e_i, e_2, \ldots$ 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

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 CART tree[5].
	min_sample_split	4	[1,10]	The minimum number of samples required to split an internal node of CART tree.
	min_Size	0.5	[0.01,1]	The value to determine the minimum number of samples to be a Where-clustering tree based on $n\_samples^{min\_Size}$.
	wriggle	0.2	[0.01, 1]	The threshold to determine which branch in Where tree to be pruned
	depthMin	2	[1,6]	The minimum depth of the tree below which no pruning for Where- clustering tree.
	depthMax	10	[1,20]	The maximum depth of the Where-clustering tree.
	wherePrune	False	T/F	Whether or not to prune the Where-clustering tree.
	treePrune	True	T/F	Whether or not to prune the classification tree built by CART.
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.
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.

Figure 3: List of parameters to be tuned.

tion. On recursion, this generates a dendogram, the leaves of which are clusters of very similar examples.

WHERE's *infoPrune* tuning parameter then choices the attributes that best select different clusters. WHERE pretends its clusters are "classes", then asks the InfoGain of the Fayyad-Irani discretizer [?], to rank the attriubutes. WHERE then ignores everything except the top *infoPrune* percent of the sorted attributes.

Optionally, if the *where prune* option is set, WHERE continues to applies infogain criteria recursively to build a tree that selects for the different clusters. If WHERE's *tree prune* parameter is enabled, then WHERE also prunes superfluous sub-trees. For example, if a sub-tree and its parent have the same majority cluster (one that occurs most frequently), then we prune the sub-tree. This tree pruning sometimes prunes aways all cluster selectors branches. To tame this effect, the *wriggle* parameter blocks tree pruning for at least the first *wriggle* number of initial branches.

Other tuning parameters are learner specific. For example, *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. Also, *max leaf nodes* is the upper bound on leaf notes generated in a Random Forest.

3.5 Tuning Algorithms

3.5.1 Parametric Tuning Algorithms

The goal of this paper is to adjust the tuning parameters of Figure **??** in order to optimize (improve) some particular performance scores generated by a particular learner being applied to a particular data set. For this task, we do not use traditional parametric numeric optimizer such as gradient descent optimizers [?] that require models comprise differential functions (i.e. functions of real-valued variables whose derivative exists at each point in its domain). This is impractical for our learners since their internal states are not a smoothly differential continuous function. Rather, learners being tuned contains many regions with many different properties (tuning options can drive the learner into very different modes with very different performance properties).

for skipping over noisy variables from d– these variables are effectively ignored since they do not contribute to the variance in the data.

3.5.2 Non-Parametric Tuning Algorithms

Non-parametric optimizers make no assumption about the model being only differential functions. One such optimizer is simulated annealing. SA generates *new* solutions by randomly perturbing (a.k.a. "mutating") some part of an *old* solution. *New* replaces *old* if (a) it scores higher; or (b) it reaches some probability set by a "temperature" constant. Initially, temperature is high so SA jumps to sub-optimal solutions (this allows the algorithm to escape from local minima). Subsequently, the "temperature" cools and SA only ever moves to better *new* solutions. SA is often used in search-based SE e.g. [?, ?], perhaps due to its simplicity.

SA was invented in the 1950s, when computer RAM was very small [?]. A standard SA algorithm needs only space for three solutions *new, old* and the *best* seen so far. In the 1960s, when more RAM became available, it became standard to generate many *new* mutants, and then combine together parts of promising solutions [?]. Such *evolutionary algorithms* (EA) work in *generations* over a population of candidate solutions. Initially, the population is created at random. Subsequently, each generation makes use of select+crossover+mutate operators to pick promising solutions, mix them up in some way, and then slightly adjust them. EAs are also often used in search-based software engineering, particularly in test case generation [?, ?] or refactoring [?]

Later work focused on creative ways to control the mutation process. Tabu search and scatter search work to bias new mutations away from prior mutations [?, ?, ?, ?]. Particle swarm optimization randomly mutates multiple solutions (which are called "particles"), but biases those mutations towards the best solution seen by one particle and/or by the neighborhood around that particle [?]. Differential evolution mutates solutions by interpolating between members of the current population [?].

Another more recent technique that has claimed much attention are heuristics that decompose the total space into many smaller problems, and then which use a simpler optimizer for each region. For example, in $\mathcal{E}$-domination [?], the user is asked 'what is the lower threshold $\mathcal{E}$ on the size of a useful effect?". The solution space is then divided into boxes of size $\mathcal{E}$ and linked such that the set $X.lower$ contains boxes with worse objective scores that X. Solutions in pairs boxes are quickly compared using small samples

Algorithm 1 Pesudocode for DE with Early Termination

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

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

from each and, if some box X is found to be inferior, then it is quickly pruned along with all solutions in the $X.lower$ boxes. Later research generalized this approach. MOEA/D (multiobjective evolutionary algorithm based on decomposition [**?**]) is a generic framework that decomposes a multiobjective optimization problem into many smaller single problems, then applies a second optimizer to each smaller subproblem, simultaneously. Other work in this arena are the response surface methods that quickly find multiple approximations to the problem, each of which holds for a very tiny region. Each region has a "slope" and examples in that region are pushed along the slope towards better solutions [**?**, **?**].

3.5.3 Selecting a Tuning Algorithm

From all the above methods, how do we select which optimizers to apply to tuning data miners. Cohen [**?**] advises comparing any supposedly more sophisticated method against the simplest possible alternative. For example, in one study with "floor effects", Holte showed that, often, much of the performance of complex multi-level decision trees could be easily achieved using a much simpler single-level decision tree learner called 1R [**?**]. He therefore recommends a very simple rule learner (called "1R") as a kind of "scout" that can do a quick preliminary analysis of a data set and which can report back if that data really requires a more complex analysis.

To find our "scout", we used engineering judgement to sort the above algorithms from simplest to most complex. The three simplest optimizers are SA, $\mathcal{E}$-domination, and differential evolution (each 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 or $\mathcal{E}$-domination:

- When the MOEA/D community requires a secondary optimizer, they often use differential evolution [**?**, **?**].
- Vesterstrom and Thomsen [**?**] report that DE is competitive with particle swarm optimization and a genetic algorithm.

DEs have been applied before for parameter tuning (e.g. see [**?**, **?**]) but this is the first time they have been applied to optimizing defect prediction from static code attributes.

3.5.4 Differential Evolution: The Details

The psuedocode for differential evolution is shown in Algorithm **??**. Note that, as we describe the algorithm, any superscript number denotes a line in that algorithm.

DE is an evolutionary algorithm; i.e. the next *NewGeneration* is learnt from a current *Population*. If the new is no better than the current, then we lose one life, terminating when we run out of lives^{5}.

Each candidate solution in the *Population* is a pair of *(Tunings, Scores)*. In this paper, *Tunings* are selected from Figure **??** and *Scores* come from training a learner using those parameters and applying it to some test data$^{23-28}$.

The premise of this algorithm is that the best way to mutate existing tunings is to *Extrapolate*^{29} 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 found as follows:

- (For numerics) $y_i = a_i + f \times (b_i - c_i)$ where f is a parameter controlling the cross-over amount. The *trim* function^{39} limits the new value to the legal range min..max of that parameter.
- (For booleans) $y_i = \neg x_i$ (see line 37).

The main loop of DE^{7} runs over the *Population*, replacing old items with new *Candidate*s (if the new candidate is better than the old item). 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 generated from the *Population*.

For this experiments of this paper, we collect performance values from a data mining, from which a *Goal* function extracts one performance value^{27} (so we tun this code many times, each time with a different *Goal*^{2}). Technically, this makes this a *single objective* DE (and for notes on multi-objective DEs, see [**?**, **?**, **?**]).

4 Experimental Design

The following experimental aims to compare the performance of three learners, tuned and untuned, on 17 sets of data.

4.1 Data Sets

Our defect data comes from the PROMISE repository[8]. It pertains to open source Java systems defined in terms of Figure **??**: *ant*, *camel*, *ivy*, *jedit*, *log4j*, *lucene*, *synapse*, *velocity*, *xalan* 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 certain drawbacks:

- Leave-one-out is too slow for large data sets;
- Cross-validation can mix up older and newer data sets so it may well be that data from the *future* is used to test on *past data*.

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.

[8] 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

Figure 4: Ratios of defective instances in each experimental data set. E.g., the top left data set has 20 defective classes out of 125 total. This paper runs one experiment for each column shown in this figure. The *training* set is used by an untuned learner to build a model, which is then tested on the *testing* data. *Training* data is used by differential evolution when it builds a model (using one set of possible tunings) and that model is tested on *tunings*. Finally, the best model from by DE is applied to *testing*. This information is continued in Figure **??**.

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

Figure 5: More ratios of defective instances in each experimental data set. Same format as Figure **??**.

For this experiment, we looked for data sets that had at least three releases in PROMISE. Note that the following ensures that all treatments get assessed on the same test set.

- The *first* release was used for some *training*, to collect a baseline using an untuned learner.
- The *first* release was also used on line 24 of Algorithm **??** to build some model using some the tunings found in some *Candidate*.
- The *second* release was used on 25 of Algorithm **??** to test the model found on line 24.
- Finally the *third* release was used to gather the performance statistics reported below from (a) the model generated by the untuned learner or (b) the best model found by differential evolution.

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 Figure **??** and Figure **??**.

4.2 Optimization Goals

Recall from Algorithm ! that we call differential evolution one time for each goal we are trying to optimize. 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:

$$\begin{array}{ll}
pd = recall = & \frac{D}{B+D} \\
pf = & \frac{C}{A+C} \\
pf' = & 1 - pf \\
prec = precision = & \frac{D}{D+C} \\
g = & \frac{2*pd*pf'}{pd+pf'}
\end{array}$$

All the above vary from zero to one. For pf, the *better* scores and *smaller*. For all other scores, the *better* scores are *larger*.

The following results make no assumption that (e.g.) minimizing false alarms are more important that maximizing recall or precision. That determination should be make with respect to current business conditions:

- For safety critical applications, low false alarm rates are irrelevant since the cost of overlooking any critical might outweigh the inconvenience of having to inspect a few more modules.
- On the other hand, when rushing a product to market before a competing product is released, 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.

What the falling results do show is that once the optimization goal changes, so does everything else. Hence, it is important not to overstate empirical results from software analytics. Rather, those results need to be expressed *along with* the context within which they are relevant.

5 Experimental Results

The introduction of this article made several claims about tuning defect predictor that use static code attributes. This section presents our results in order of those claims:

1. Such tuning was easy (see §**??**);
2. Such tuning is fast (see §**??**);
3. Such tuning is can dramatically change the performance scores of a predictor; e.g. one result where precision changes from 2% to 98% (see §**??**);
4. Tuning changes conclusions about what factors are most important (see §**??**);
5. Tuning changes conclusions about what learners are better than others (see §**??**;)

5.1 Tuning is Easy

All the following results are generated with a a very simple optimizer (differential evolution). Further, our DE is actually much simpler than a standard implementation. Recall from Algorithm 1 that DE explores a *Population* of size *np=10*. This is a very small population size: the standard recommendation is to set *np* to be ten times larger than the number of attributes being optimized [**?**]

The results of this paper are presented

6 Discussion

7 Related Work

Tuning in efforts estimation, software engineering.

Defect Prediction

8 Threats to Validity

Internal and external threats

9 Conclusion

10 Acknowledgments