Information and Software Technology 76 (2016) 135–146

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

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

ARTICLE INFO

Article history:
Received 25 January 2016
Revised 26 April 2016
Accepted 29 April 2016
Available online 30 April 2016

Keywords:
Defect prediction
CART
Random forest
Differential evolution
Search-based software engineering

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 (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.

1. Introduction

In the 21st 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.

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 5–20% 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. [6–11]) use data miners to conclude that *this* is more important than *that* for reducing software project defects. Given

* Corresponding author.
E-mail addresses: wfu@ncsu.edu (W. Fu), tim.menzies@gmail.com (T. Menzies), xshen5@ncsu.edu (X. Shen).

http://dx.doi.org/10.1016/j.infsof.2016.04.017
0950-5849/

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" (more details can be found at https://goo.gl/Inl9nF). 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: [4,14]). Overall, 80% of papers in our sample *did not* adjust the "off-the-shelf" configuration of the data miner (e.g. [9,15,16]). 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 runtime 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 that 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 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 at 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 optimizes 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