When, and Why, Simple Methods Fail. Lessons Learned from Hyperparameter Tuning in Software Analytics (and Elsewhere)

Anonymous Author(s)

ABSTRACT

Tuning a data miner for for software analytics is something of a black art. Recent research has shown that some of that tuning can be achieved via automatic tools, called "hyperparameter optimizers". Much of that research has used tools developed from outside of SE. Hence, here, we ask how and when we can exploit the special properties of SE data to build faster and better optimizers.

Specifically, we apply hyperparameter optimization for 120 data sets addressing problems like bad smell detection, predicting Github issue close time, bug report analysis, defect prediction and dozens of other non-SE problems. To these, we applied a tool developed using SE data which (a) out-performs the state-of-the-art for these SE problems yet (b) fails badly on non-SE problems. From this experience, we can infer a simple rule for when to use/avoid different kinds of optimizers. SE data is often about infrequent issues, like the occasional defect, or the rarely exploited security violation, or the requirement that holds for one special case. But as we show, the same was not observed when we applied it on non-SE data.

Our conclusion will be that we can exploit this special properties of SE to great effect; specifically, to find better optimizations for SE tasks via a tactic called "dodging" (explained in this paper).

1 INTRODUCTION

The last 15 years have seen a dramatic increase in the application of machine learning and hyperparameter optimization (HPO) to software analytics– as evidenced by numerous highly cited publications [30, 38, 43, 47, 48, 51, 54, 69, 75] and the growing prominence of venues such as the Mining Software Repositories conference (now ranked in the top dozen software systems venue by Google Scholar).

Researchers reported that these techniques/algorithms/tools are often used in a "black box" manner without reflecting on the merits of choices associated with a particular tool [9, 23]. Such black-box usage is risky since it means SE practitioners might be applying the wrong tool. An emerging question in the application of machine learning techniques to software engineering is the validity of the assumptions that underlie the creation of those tools and techniques [3, 9, 23, 58, 71]. These articles show superior results can be obtained by adapting machine learning tools to the particulars of software engineering problem (see examples in next section).

One way to specialize machine learning tools for software engineering is *hyperparameter optimization* that automatically searches through the space of possible learner settings to discover, for example, how many clusters work best for K-means. Table 1 shows some of the hyperparameter options seen in recent SE papers. The search space is very large. Assuming that the numerics of Table 1 are divided into ten bins, then there are billions of different combinations in that table. Hence, these optimizers can be slow to execute. Also, even when they work, these optimizers **do not offer any generalization across their results**. Hyperparameter optimization does not offer general rules for when some machine learning tools are more useful than others. This is a significant deficiency. If such an insight was available, we could offer engineers a simple rule that lets them quickly select appropriate tools.

Accordingly, this paper applies various tools to numerous problems from SE and elsewhere. Some of these tools are general machine learning tools, while another, called DODGE [1], is a hyperparameter optimizer that auto-adjusts learner settings in a somewhat unusual way (see the "relax" heuristics of §3.2.2). Agrawal et al. [1] justified that approach via certain properties of software projects and some limited experimentation of 16 data sets from two SE problems (bug report analysis and defect prediction).

Agrawal et al. did not test their supposed general tool on non-SE problems. Hence, we apply DODGE to 120 data sets from SE and elsewhere: 37 non-SE problems from the standard machine learning literature as well as 83 SE problems which includes 6 issue tracking data sets, 10 defect prediction data sets 63 data sets exploring Github issue close time; and 4 data sets exploring bad smell detection.

We see that DODGE **fails badly** on non-SE problems but **performs very well** on SE problems. Using that experience, this paper:

- Learns when to use/avoid general machine learning tools.
- Describes a case study utilizing that methodology.
- Shows that this method can indeed recommend when to use a general machine learning tool or tools built using SE data.
- Shows that this recommendation is useful and if we follow them, we can find better optimizations for SE problems.

The rest of this paper is structured as follows. The next section offers notes on related work and some standard machine learning tools. This is followed by a description of DODGE in §3.2.2. After that, in §3.3, we describe the 120 data sets used to test DODGE as well as other more standard tools. The experiments of §3.4 illustrate how well DODGE performs for SE problems and how badly it performs otherwise in §4. The meta-analysis of §5 generates the rule that explains why DODGE works so well/poorly for SE/other problem types.

Our conclusion is that the algorithms which we call "general machine learning tools" may not be "general" at all. Rather, they are tools which are powerful in their home domain but need to be used with care if applied to some new domains like software engineering. Hence, we argue that it is not good enough to just take tools developed elsewhere, then apply them verbatim to SE problems. Software engineers need to develop specialized machine learning tools that are better suited to the particulars of SE problems.

1.1 Data and Code Availability

All the data and scripts used in this paper are available on-line in the reproduction package at http://tiny.cc/dodge2020.

Table 1: Hyperparameter options seen in recent SE papers [2, 3, 23, 25] and in the documentation of a widely-used data mining library (Scikit-learn [62]). While this list is incomplete, it does include many of the hyperparameters being explored in the literature.

Learners:

- DecisionTreeClassifier(criterion=b, splitter=c, min_samples_split=a)
  - a, b, c= randuniform(0.0,1.0), randchoice([‘gini’,‘entropy’]), randchoice([‘best’,‘random’])
- RandomForestClassifier(n_estimators=a,criterion=b, min_samples_split=c)
  - a,b,c = randint(50, 150), randchoice([’gini’, ’entropy’]), randuniform(0.0, 1.0)
- LogisticRegression(penalty=a, tol=b, C=float(c))
  - a,b,c=randchoice([‘l1’,‘l2’]), randuniform(0.0,0.1), randint(1,500)
- MultinomialNB(alpha=a) = randuniform(0.0,0.1)
- KNeighborsClassifier(n_neighbors=a, weights=b, p=d, metric=c)
  - a, b,c = randint(2, 25), randchoice([‘uniform’, ‘distance’]), randchoice([‘minkowski’,‘chebyshev’])
  - if c==’minkowski’: d= randint(1,15) else: d=2

Pre-processors for defect prediction, Issue lifetime, Bad Smells:

- StandardScaler
- MinMaxScaler
- MaxAbsScaler
- RobustScaler(quantile_range=(a, b)) = randint(0,50), randint(51,100)
- KernelCenterer
- QuantileTransformer(n_quantiles=a, output_distribution=c, subsample=b)
  - a, b = randint(100, 1000), randint(1000, 1e5)
  - c = randchoice([‘normal’,‘uniform’])
- Normalizer(norm=a) = randchoice([‘l1’, ‘l2’,‘max’])
- Binarizer(threshold=a) = randuniform(0,100)
- SMOTE(a=n_neighbors, b=n_synthetics, c=Minkowski_exponent)
  - a,b = randit(1,20),randchoice(50,100,200,400)
  - c = randuniform(0.1,5)

Pre-Processors for Text mining:

- CountVectorizer(max_df=a, min_df=b) = randint(100, 1000), randint(1, 10)
- TfidfVectorizer(max_df=a, min_df=b, norm=c)
  - a, b,c = randint(100, 1000), randint(1, 10), randchoice([‘l1’, ‘l2’, None])
- HashingVectorizer(n_features=a, norm=b)
  - a = randchoice([1000, 2000, 4000, 6000, 8000, 10000])
  - b = randchoice([‘l1’, ‘l2’, None])
- LatentDirichletAllocation(n_components=a, doc_topic_prior=b, topic_word_prior=c, learning_decay=d, learning_offset=e,batch_size=f)
  - a, b, c = randint(10, 50), randuniform(0, 1), randuniform(0, 1)
  - d, e = randuniform(0.51, 1.0), randuniform(1, 50),
  - f = randchoice([150,180,210,250,300])

1.2 Relationship to Prior Work

Since this paper builds on prior work by Agrawal et al. [1], we take care to highlight where we repeat or extend that prior work. Table 1 and §3 are summaries of notes from Agrawal et al. [1]. Also, of the 120 data sets processed here, only 16 come from that prior study: see §4.1 and §4.2. (Aside: and just as a quality assurance measure, we reran all the experiments for those 16 data sets.)

Apart from that, nearly all of this paper has not appeared before. The new material in this paper includes:

- The new experiments on 104 data sets,
- The meta-analysis of §5.

2 RELATED WORK

This section offers examples where other SE researchers have argued that software engineering is different (so we need our own kind of specialized SE tools).

Binkley et al. [9] note that information retrieval tools for SE often equates word frequency with word importance, even though the number of occurrences of a variable name such as“tmp” is not necessarily indicative of its importance. They argue that the negative impacts of such differences manifest themselves when off-the-shelf IR tools are applied in the software domain.

Software analytics usually applies standard data mining algorithms as “black boxes” [9, 23], where researchers do not tinker with internal design choices of those techniques. This is not ideal. Much recent research has shown that such automatic hyperparameter optimizers can automate that tinkering process, and greatly improve predictive performances [3, 4, 23, 61, 71].

Another example comes from sentiment analysis. Standard sentiment analysis tools are usually trained on non-SE data (e.g., the Wall Street Journal or Wikipedia). Novielli et al. [58] recently developed their own sentiment analysis for the software engineering domain. After re-training those tools on an SE corpus, they found not only better performance at predicting sentiment, but also more agreement between different sentiment analysis tools.

Yet another example of “SE needs specialized tools” comes from the “naturalness” work of Devanbu et al [32]. Software has the property that some terms are used very often and everything else is used with exponentially less frequency. This means that n-gram “language models” (where frequency counts of the last N terms are used to predict the next term) can be very insightful for (a) recommending what should be used next; or (b) predicting that the next token is unusual and possibly erroneous [67]. Note that this “natural” approach can be more insightful than standard tools such as deep learning [31].

While the above work illustrates the point of this paper, it does not offer a simple rule for looking at new data to decide (a) when to use pre-existing standard machine learners; or (b) when to move on to other technologies. This rest of this paper describes methods used to answer these two questions.

3 METHODS

3.1 Data mining tools

Hyperparameter optimizers adjust the control parameters of data miners. This section reviews the data mining tools used in this study: grid search, DE, GA, SVM, Random Forests, decision tree learners, logistic regression, Naive Bayes, LDA, and EM.

Before doing that, it is reasonable to ask “why did we select these tools, and not some other set?”. This paper does not compare DODGE against all other learners and all other hyperparameter optimizers (since such a comparison would not fit into a single paper). Instead, we use baselines as found in the SE literature for bad smell detection, predicting Github issue close time, bug report analysis, and defect prediction.

For example, for defect prediction, our classifiers come from a study by Ghotra et al. [25]. They found that the performance of dozens of data miners (applied to defect prediction) clustered into just few groups. By sampling a few algorithms from each group, we can explore the range of data miners seen in defect prediction.

Clustering algorithms like EM [16] divide the data into related groups, then check the properties of each group. Another clustering method used in text mining, is Latent Dirichlet Allocation [11] that infers "topics" (commonly associated words). After documents are scored according to how often they use some topic, a secondary classifier can then be used to distinguish the different topics.

Clustering algorithms like EM and LDA do not make use of any class variable. Naive Bayes classifiers [18], on the other hand, always divide the data on the class. New examples are then classified according to which class is most similar. Also, logistic regression fits the data to a particular parametric form (the logistic function).

Another learner that uses class variables are decision tree algorithms [14, 65]. These learners divide data on attribute whose values most separate the classes. The learner then recurses on each division. Random Forests [13] build a "committee" of multiple decision trees, using different sub-samples of the data. Conclusions then come from a voting procedure across all the trees in the forest.

Standard clustering and decision tree algorithms base their analysis using the raw problem data. But what if some extra derived attribute is best at separating the classes? To address that issue, SVMs use a "kernel" to infer those extra dimensions [12].

3.2 Hyperparameter Optimizers

Hyperparameter optimizers are tools to find a learner's control settings (e.g., see Table 1).

3.2.1 Standard Optimizers.

3.2.1 Standard Optimizers. As said above, we do not compare against all hyperparameter optimization methods. Instead, we use hyperparameter optimization methods seen in SE literature.

For example, when text mining SE data, Panichella et al. [61] used genetic algorithm [27] (GA) to "evolve" a set of randomly generated control settings for SE text miners by repeating the following procedure, across many "generations": (a) mutate a large population of alternate settings; (b) prune the worse performing settings; (c) combine pairs of the better, mutated options.

An alternative hyperparameter optimization strategy is the differential evolution [70] used by Wu et al. [23] and others [3]. DE generates mutants by interpolating between the better ranked settings. These better settings are kept in a "frontier list". Differential evolution iterates over the frontier, checking each candidate against a new mutant. If the new mutant is better, it replaces the frontier item, thus improving the space of examples used for subsequent mutant interpolation.

Tantithamthavorn et al. [71] used a grid search for their hyperparameter optimization study. Grid search runs nested "for-loops" over the range of each control options. Fu et al. [24] found that for defect prediction, grid search ran 100 to 1000 times slower than DE.

3.2.2 Optimizing with DODGE. DODGE is a hyperparameter optimizer proposed by Agrawal et al. [1]. For the purposes of this paper, the essential point about DODGE is that it was commissioned using data from SE problems. As we shall see, while DODGE was very successful for SE problems, this algorithm fails very badly when applied to data sets taken from outside the SE domain.

Given an ever evolving set of tools, languages, platforms, tasks, user expectations, development population, development practices, etc, we might expect that any prediction about an SE project will only ever be approximately accurate, i.e., within ϵ of the true value. Agrawal et al. reasoned that ϵ is not a problem to be solved, but a resource that could be exploited, as follows:

The RELAX heuristic: *Ignore anything less than ϵ.*

DODGE applies this RELAX heuristic to do hyperparameter optimization. To illustrate this process, consider the following example. We are exploring the hyperparameter space of Table 1 and we are scoring each option via the recall and false alarm rate of the learner that uses those settings (for a definition of these scores, see §3.4).

Given performance goals with the range $0 \leq g \leq 1$, ϵ divides the performance output space into $(1/\epsilon)^g$ cells. For example, consider the $g = 2$ goals of recall and false alarm. These have minimum and maximum values of zero and one. Hence, if $\epsilon = 0.2$, then these scores divide into five regions (at 0.2, 0.4, 0.6, 0.8). As shown in Figure 1, these divided scores separate a two-dimensional plot of recall vs false alarm scores into $(1/0.2)^2 = 25$ cells. In those cells, green denotes good performance (high recall, low false alarm) and red denotes cells with relatively worse performance.

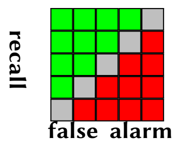

Figure 1: 25 cells of width $\epsilon = 0.2$.

When billions of inputs (in Table 1) are mapped into the 25 cells of Figure 1, then many inputs are *redundant*, i.e., lead to the same outputs. The faster we can "dodge" redundant options, the faster we can move on to explore the other $(1/\epsilon)^g$ possible outputs.

To implement "dodging", DODGE models Table 1 as a tree where all nodes have initial weights $w = 0$. Next, N_1 times, DODGE selects branches at random. We evaluate the options in a branch and if the resulting scores are within ϵ of any previous scores, then DODGE deprecates those options via $w = w - 1$, else $w = w + 1$.

After that, DODGE freezes the selected branches found so far. N_2 times, DODGE then makes random selection to restrict any numeric ranges. When a range is initially evaluated, a random number $r = random(lo, hi)$ is selected and its weight $w(r)$ is set to zero. Subsequently, this weight is adjusted (as described above). When a new value is required (i.e., when the branch is evaluated again) then if the best, worst weights seen so far (in this range) are x, y (respectively) then we reset lo, hi to:

$$IF\ x \leq y\ THEN\ lo, hi = x, (x + y)/2\ ELSE\ lo, hi = (x + y)/2, x$$

When ϵ is large, a few samples should suffice to find good results. Hence, Agrawal et al. [1] recommends $\epsilon = 0.2$ and $N_1 = N_2 = 15$.

DODGE can be recommended for two reasons. Firstly, for SE problems, DODGE's optimizations are better than the prior state-of-the-art (evidence: see [1], and the rest of this paper). Secondly, DODGE achieves those results very quickly. Based on the default parameters suggested by Goldberg [27], Storn [70], and using some empirical results from Fu et al. [23], we can compute the number of times a hyperparameter optimizer would have to call a data miner. Assuming that, 25 times[1] we are analyzing 10 data sets, then hyperparameter optimization with grid search or genetic algorithms or differential evolution or DODGE would need to call a data miner thousands to millions of times (respectively).

3.3 Data

This section describes the problem data used to evaluate DODGE.

Agrawal et al. say DODGE is an "SE-based" algorithm since (a) it was commissioned on SE data where (b) the inherent uncertainties of SE developed learn to high variability of predictions, in which case (c) the RELAX might become relevant and useful.

Their approach is so unusual that it is reasonable to ask "when won't it work?". To answer this question, the rest of this paper applies DODGE to 120 data sets, 83 of which come from SE while the others are from the Non-SE literature. Agrawal et al. previously studied 16 of these data sets (and none of their data came from non-SE problems).

We will find that DODGE works very well for SE case studies and very badly for non-SE case studies. Later in this paper, we precisely characterize the kinds of data for which DODGE is not recommended.

3.3.1 *Defect Prediction.*

Software developers are smart, but sometimes make mistakes. Hence, it is essential to test software before the deployment [8, 56, 59, 80]. Software bugs are not evenly distributed across the project [29, 39, 53, 60]. Hence, a useful way to perform software testing is to allocate most assessment budgets to the more defect-prone parts in software projects. Data miners can learn a predictor for defect proneness using, for e.g., the static code metrics of Table 2.

In a recent study, Rahman et al. [66] 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. This result is interesting since code defect prediction can be quickly adapted to new languages by building lightweight parsers to extract static code metrics. The same is not true for static code analyzers that need much modification when languages update.

A recent survey of 395 practitioners from 33 countries and five continents [72] found that over 90% of the respondents were willing to adopt defect prediction techniques. Also, when Misirli et al. [53] built a defect prediction model for a telecommunications company, those models could predict 87% of files with defects. Those models also decreased inspection efforts by 72%, and reduced post-release defects by 44%.

Table 3 shows the static code data used in this paper. All these projects have multiple versions and we use older versions to predict the properties of the latest version. Note the fluctuating frequencies of the target class in the training and testing data (sometimes increasing, sometimes decreasing), for e.g., xerces has target frequency changes between 16 to 74% while in jedit it changes from 23 to 2%. One of the challenges of doing data mining in such domains is finding learner settings that can cope with some wide fluctuations.

[1]Why 25? In a 5x5 cross-val experiment, the data set order is randomized five times. Each time, the data is divided into five bins. Then, for each bin, that bin becomes a test set of a model learned from the other bins.

amc	average method complexity
avg_cc	average McCabe
ca	afferent couplings
cam	cohesion amongst classes
cbm	coupling between methods
cbo	coupling between objects
ce	efferent couplings
dam	data access
dit	depth of inheritance tree
ic	inheritance coupling
lcom (lcom3)	2 measures of lack of cohesion in methods
loc	lines of code
max_cc	maximum McCabe
mfa	functional abstraction
moa	aggregation
noc	number of children
npm	number of public methods
rfc	response for a class
wmc	weighted methods per class
defects	Boolean: where defects found in bug-tracking

Table 2: Static code metrics for defect prediction. For details, see [42].

Table 3: Defect prediction data from http://tiny.cc/seacraft. Uses metrics from Table 2.

Project	Training Data		Testing Data	
	Versions	% of Defects	Versions	% of Defects
Poi	1.5, 2.0, 2.5	426/936 = 46%	3.0	281/442 = 64%
Lucene	2.0, 2.2	235/442 = 53%	2.4	203/340 = 60%
Camel	1.0, 1.2, 1.4	374/1819 = 21%	1.6	188/965 = 19%
Log4j	1.0, 1.1	71/244 = 29%	1.2	189/205 = 92%
Xerces	1.2, 1.3	140/893 = 16%	1.4	437/588 = 74%
Velocity	1.4, 1.5	289/410 = 70%	1.6	78/229 = 34%
Xalan	2.4, 2.5, 2.6	908/2411 = 38%	2.7	898/909 = 99%
Ivy	1.1, 1.4	79/352 = 22%	2.0	40/352 = 11%
Synapse	1.0, 1.1	76/379 = 20%	1.2	86/256 = 34%
Jedit	3.2,4.0, 4.1,4.2	292/1257 = 23%	4.3	11/492 = 2%

Table 4: Issue tracking data (from http://tiny.cc/seacraft).

Dataset	No. of Documents	No. of Unique Words	Severe %
PitsA	965	155,165	39
PitsB	1650	104,052	40
PitsC	323	23,799	56
PitsD	182	15,517	92
PitsE	825	93,750	63
PitsF	744	28,620	64

3.3.2 *Text Mining Issue Reports.*

Many SE project artifacts come in the form of *unstructured text* such as word processing files, slide presentations, comments, Github issue reports, etc. According to White [73], 80% of business is conducted on unstructured data, 85% of all data stored is held in an unstructured format and unstructured data doubles every three months. Nadkarni and Yezhkova [57] say that 1600 Exabytes of data appears in unstructured sources and that each year, humans generate more unstructured artifacts than structured.

Mining such unstructured data is complicated by the presence of free form natural language which is semantically very complex and may not conform to any known grammar. In practice, text documents require tens of thousands of attributes (one for each word). For example Table 4 shows the number of unique words found in the issue tracking system for six NASA projects PitsA, PitsB, PitsC, etc. [44, 49]. Our PITS dataset contains tens to hundreds of thousands of words (even when reduced to unique words, there are still 10,000+ unique words). One other thing to note in Table 4 is that the target class frequencies are much higher than with defect prediction (median=60%).

For large vocabulary problems, text miners apply dimensionality reduction. (see Table 1 for the list of dimensionality reduction pre-processing methods used here). After pre-processing, one of the learners from Table 1 was applied to predict for issue severity.

While these data mention five classes of severity, two of them comprise nearly all the examples. Hence, for this study we use the most common class and combine all the others into “other”. Agrawal et al. [1] showed that using Table 1, they could auto-configure classifiers to better predict for this binary severity problem.

3.3.3 *Issue Lifetime Estimation.*

Issue tracking systems collect information about system failures, feature requests, and system improvements. Based on this information and actual project planing, developers select the issues to be fixed.

Predicting the time it may take to close an issue has multiple benefits for the developers, managers, and stakeholders involved in a software project. Such predictions helps software developers to better prioritize work. For an issue close time prediction generated at issue creation time can be used, for example, to auto-categorize the issue or send a notification if it is predicted to be an easy fix. Also, such predictions helps managers to effectively allocate resources and improve consistency of release cycles. Lastly, such predictions helps project stakeholders understand changes in project timelines.

Such predictions can be generated via data mining. Rees-Jones et al. [68] analyzed the Giger et al. [26] data using Hall’s CFS feature selector [28] and the C4.5 decision tree learner [65]. They found that the attributes of Table 5 could be used to generate very accurate predictions for issue lifetime. Table 6 shows information about the nine projects used in the Rees-Jones study. Note here that the target class frequencies vary greatly from 2 to 42%.

Table 5: Metrics used in Issue lifetime data

Commit	Comment	Issue
nCommitsByActorsT	meanCommentSizeT	issueCleanedBodyLen
nCommitsByCreator	nComments	nIssuesByCreator
nCommitsByUniqueActorsT		nIssuesByCreatorClosed
nCommitsInProject		nIssuesCreatedInProject
nCommitsProjectT		nIssuesCreatedInProjectClosed
		nIssuesCreatedProjectClosedT
		nIssuesCreatedProjectT
Misc.	nActors, nLabels, nSubscribedByT	

3.3.4 *Bad Code Smell Detection.*

According to Fowler [21], bad smells (i.e., code smells) are “a surface indication that usually corresponds to a deeper problem”. Studies suggest a relationship between code smells and poor maintainability or defect proneness [76, 77, 81]. Research on software refactoring endorses the use of code-smells as a guide for improving the quality of code as a preventative maintenance. Consequently, code smells are captured by popular static analysis tools, like PMD, CheckStyle, FindBugs, and SonarQube. Much recent progress has been made towards adopting data miners to classify code smells. Kreimer [40] proposes an adaptive detection to combine known methods for finding design flaws. Khomh et al. [36] proposed a Bayesian approach to detect occurrences of the Blob antipattern on open-source programs. Khomh et al. [37] also proposed an approach to build Bayesian Belief Networks. Yang et al. [78] studied the judgment of individual users by applying machine learning algorithms on code clones.

Table 6: Issue Lifetime Estimation Data from [68]

Project Name	Dataset	# of instances		# metrics (see Table 5).
		Total	Closed (%)	
camel	1 day	5056	698 (14.0)	18
	7 days		437 (9.0)	
	14 days		148 (3.0)	
	30 days		167 (3.0)	
	90 days		298 (6.0)	
	180 days		657 (13.0)	
	365 days		2052 (41.0)	
cloudstack	1 day	1551	658 (42.0)	18
	7 days		457 (29.0)	
	14 days		101 (7.0)	
	30 days		107 (7.0)	
	90 days		133 (9.0)	
	180 days		65 (4.0)	
	365 days		23 (2.0)	
cocoon	1 day	2045	125 (6.0)	18
	7 days		92 (4.0)	
	14 days		32 (2.0)	
	30 days		45 (2.0)	
	90 days		86 (4.0)	
	180 days		51 (3.0)	
	365 days		73 (3.5)	
node	1 day	6207	2426 (39.0)	18
	7 days		1800 (29.0)	
	14 days		521 (8.0)	
	30 days		453 (7.0)	
	90 days		552 (9.0)	
	180 days		254 (4.0)	
	365 days		180 (3.0)	
deeplearning	1 day	1434	931 (65.0)	18
	7 days		214 (15.0)	
	14 days		76 (5.0)	
	30 days		72 (5.0)	
	90 days		69 (5.0)	
	180 days		39 (3.0)	
	365 days		32 (2.0)	
hadoop	1 day	12191	40 (0.0)	18
	7 days		65 (1.0)	
	14 days		107 (1.0)	
	30 days		396 (3.0)	
	90 days		1743 (14.0)	
	180 days		2182 (18.0)	
	365 days		2133 (17.5)	
hive	1 day	5648	18 (0.0)	18
	7 days		22 (0.0)	
	14 days		58 (1.0)	
	30 days		178 (3.0)	
	90 days		1050 (19.0)	
	180 days		1356 (24.0)	
	365 days		1440 (25.0)	
ofbiz	1 day	6177	1515 (25.0)	18
	7 days		1169 (19.0)	
	14 days		467 (8.0)	
	30 days		477 (8.0)	
	90 days		574 (9.0)	
	180 days		469 (7.5)	
	365 days		402 (6.5)	
qpid	1 day	5475	203 (4.0)	18
	7 days		188 (3.0)	
	14 days		84 (2.0)	
	30 days		178 (3.0)	
	90 days		558 (10.0)	
	180 days		860 (16.0)	
	365 days		531 (10.0)	

Table 7: Bad code smell detection data from [5]

Nature	Dataset	No. of instances	No. of attributes	Smelly %
Method	Feature Envy	109	82	45
Method	Long Method	109	82	43.1
Class	God Class	139	61	43.9
Class	Data Class	119	61	42

Recently, Fontana et al. [5] considered 74 systems in data mining analysis. Table 7 shows the data used in that analysis. This corpus

Table 8: NON-SE problems: 37 UCI Datasets statistics.

Area	Dataset	# of instances	# of attributes	Class %
Computer	optdigits	1143	64	50
Physical	satellite	2159	36	28
Physical	climate-sim	540	18	91
Financial	credit-approval	653	15	45
Medicine	cancer	569	30	37
Business	shop-intention	12330	17	15
Computer Vision	image	660	19	50
Life	covtype	12240	54	22
Computer	hand	29876	15	47
Social	drug-consumption	1885	30	23
Environment	biodegrade	1055	41	34
Social	adult	45222	14	25
Physical	crowdsource	1887	28	24
Medicine	blood-transfusion	748	4	24
Financial	credit-default	30000	23	22
Medicine	cervical-cancer	668	33	7
Social	autism	609	19	30
Marketing	bank	3090	20	12
Financial	bankrupt	4769	64	3
Financial	audit	775	25	39
Life	contraceptive	1473	9	56
Life	mushroom	5644	22	38
Computer	pendigits	2288	16	50
Security	phishing	11055	30	56
Automobile	car	1728	6	30
Medicine	diabetic	1151	19	53
Physical	hepmass	2000	27	50
Physical	htru2	17898	8	9
Computer	kddcup	3203	41	69
Automobile	sensorless-drive	10638	48	50
Physical	waveform	3304	21	50
Physical	annealing	716	10	13
Medicine	cardiotocography	2126	40	22
Phyical	shuttle	54489	9	16
Electrical	electric-stable	10000	12	36
Physical	gamma	19020	10	35
Medicine	liver	579	10	72

comes from 11 systems written in Java, characterized by different sizes and belonging to different application domains. The authors computed a large set of object-oriented metrics belonging at a class, method, package, and project level. A detailed list of metrics are available in appendices of [5]. Note in Table 7, how the target class frequencies are all around 43%,

3.3.5 Non-SE Problems. The UCI machine learning repository [7, 17, 22] was created in 1987 to foster experimental research in machine learning. To say the least, this repository is commonly used by industrial and academic researchers (evidence: the 2007, 2010, and 2017 version of the repository are cited 4020, 3179 and 4179 times respectively [7, 17, 22]). Many of machine learning tools used in SE were certified using data from UCI. This repository holds 100s of data mining problems from many problem areas including engineering, molecular biology, medicine, finance and politics. Using a recent state-of-the-art machine learning paper [74] we identified 37 UCI data sets that machine learning researchers often used in their analysis (see Table 8).

One issue with comparing Table 8 to the SE problems is that the former often have $N > 2$ classes whereas the SE problems use binary classification. Also, sometimes, the SE data exhibits large class imbalances (where the target is less than 25% of the total). Such imbalances are acute in the issue lifetime data in Table 6 but it also appears sometimes in the test data of Table 3.

We considered various ways to remove the above threat to validity including (a) clustering and sub-sampling each cluster; (b) some biased sampling of the UCI data. In then end, we adopted a very simple method (lest anything more complex introduced its own biases). For each UCI dataset, we selected:

- The UCI rows from the most frequent and rarest class;
- And declared that the UCI rarest class is the target class.

3.4 Experimental Controls

To comparatively evaluate the performance of machine learning tools, we need performance measures (§3.4.1), appropriate experimental rigs (§3.4.2), and statistical tests (§3.4.3).

3.4.1 Performance Measures. *D2h*, or "distance to heaven", shows how close a classifier falls to "heaven" (where recall=1 and false alarms (FPR)=0) [15]:

$$\begin{aligned} Recall &= TruePositives/(TruePositives + FalseNegatives) && (1)\\ FPR &= FalsePositives/(FalsePositives + TrueNegatives) && (2)\\ d2h &= \left(\sqrt{(1 - Recall)^2 + (0 - FPR)^2}\right)/\sqrt{2} && (3) \end{aligned}$$

Here, the $\sqrt{2}$ term normalizes *d2h* to the range zero to one.

For defect prediction, *Popt(20)* comments on the inspection effort required *after* a defect predictor is triggered. $Popt(20) = 1 - \Delta_{opt}$, where Δ_{opt} is the area between the effort (code-churn-based) cumulative lift charts of the optimal learner and the proposed learner. To calculate *Popt(20)*, we divide all the code modules into those predicted to be defective (D) or not (N). Both sets are then sorted in ascending order of lines of code. The two sorted sets are then laid out across the x-axis, with D before N. On such a chart, the y-axis shows what percent of the defects would be recalled if we traverse the code sorted that x-axis order. Following from Ostrand et al. [60], *Popt* is reported at the 20% point. Futher, following Kamei, Yang et al. [35, 55, 79] we normalize *Popt* using: $P_{opt}(m) = 1 - \frac{S(optimal) - S(m)}{S(optimal) - S(worst)}$ where $S(optimal)$, $S(m)$ and $S(worst)$ represent the area of curve under the optimal learner, proposed learner, and worst learner. Note that the worst model is built by sorting all the changes according to the actual defect density in ascending order. After normalization, *Popt(20)* (like *d2h*) has the range zero to one. Note that *larger* values of *Popt(20)* are *better*; but *smaller* values of *d2h* are *better*.

3.4.2 Control Rig. Jimeneze et al. [33] recommended that train/test data be labelled in their natural temporal sequence; i.e. apply training and hyperparameter optimization to the prior versions, then tested on latter version. We will call this **RIG0**.

When temporal markers are missing, we use a cross-val method (which is also standard in literature [79]). Given one data set and N possible treatments, then 25 times we use 80% of the data (selected at random) for training and hyperparameter optimization, then the remaining 20% for testing. We will call this **RIG1**.

3.4.3 Statistical Tests. When comparing results from two samples, we need a statistical significance test (to certify that the distributions are indeed different) and an effect size test (to check that the differences are more than a "small effect"). Here, we used tests which have been past peer reviewed in the literature [2, 3]. Specifically, we use Efron's 95% confidence bootstrap procedure [19] and the A12 effect test endorsed by Acuri & Briand in their ICSE paper [6].

4 RESULTS

In the following, when we say "DODGE", that is shorthand for DODGE using Table 1 with $N_1 + N_2 = 15, \epsilon = 0.2$. Also, when we say "DODGE performed better", we mean that, according to a 95% bootstrap and the A12 test, DODGE performed significantly better by more than a small effect.

4.1 Defect Prediction Results

Table 9 shows which tools found best predictors for defects, using the data of §3.3.1.

When the target class is not common (as in camel, ivy, jedit and to a lesser extent velocity and synapse), it can be difficult for a data mining algorithm to generate a model that can locate it. Researchers have used class balancing techniques such as SMOTE to address this problem [3].

Table 9: Ten defect prediction results. Smote+ means SMOTE+ DE tuning + best of the Ghotra'15 learners.

data	Best tool	
	D2h	Popt
Poi	Dodge	Smote+
Lucene	Dodge	Dodge
Camel	Dodge	Dodge
Log4j	Smote+	Dodge
Xerces	Dodge	Dodge
Velocity	Dodge	Dodge
Xalan	Smote+	Dodge
Ivy	Dodge	Dodge
Synapse	Dodge	Dodge
Jedit	Smote+	Dodge

Table 9 study compares DODGE versus methods selected from prior state-of-the-art SE papers. An ICSE'18 paper [3] reported that hyperparameter tuning (using DE) of SMOTE usually produces best results (and this result holds across multiple learners, applied after class rebalancing). We used SMOTE tuning (for data-processing) plus learners taken from Ghotra et al. [25] (who found that the performance of dozens of data miners can be clustered into just a few groups). We used learners sampled across those clusters (Random Forests, CART, SVM, KNN ($k = 5$), Naive Bayes, Logistic Regression).

Table 9 results were generated using **RIG0** with *d2h* and *Popt(20)* as the performance goal. DODGE performed statistically better than the prior state-of-the-art in sixteen out of twenty results of 10 data sets.

4.2 Text Mining Results

Table 10 shows which techniques found best predictors for the data of §3.3.2. In this study, all data were preprocessed using the usual text mining filters [20]. We implemented stop words removal using NLTK toolkit [10] (to ignore very common short words such as "and" or "the"). Next, Porter's stemming filter [64] was used to delete uninformative word endings (e.g., after performing stemming, all the following words would be rewritten to "connect": "connection", "connections", "connective", "connected", "connecting").

Table 10: Six text mining results.

Data	Best tool
PitsA	Dodge
PitsB	LDA + DE
PitsC	Dodge
PitsD	Dodge
PitsE	Dodge
PitsF	Dodge

Table 10 compares DODGE versus methods seen in prior state-of-the-art SE papers: specifically, SVM plus Latent Dirichlet allocation [11] with hyperparameter optimization via differential evolution [2] or genetic algorithms [61].

Table 11: Sixty three issue lifetime prediction results. DODGE loses to random forests in colored cells.

Data	Days till closed						
	> 365	< 180	< 90	< 30	< 14	< 7	< 1
cloudstack	Dodge	Dodge	Dodge	RF	RF	Dodge	RF
node	Dodge	Dodge	Dodge	Dodge	RF	Dodge	Dodge
deeplearning	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge	RF
cocoon	RF	RF	Dodge	Dodge	Dodge	Dodge	Dodge
ofbiz	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge
camel	RF	Dodge	Dodge	RF	RF	Dodge	Dodge
hadoop	Dodge	Dodge	Dodge	RF	RF	Dodge	RF
qpid	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge
hive	RF	RF	Dodge	Dodge	Dodge	Dodge	Dodge
DODGE wins	6/9	7/9	9/9	6/9	5/9	9/9	6/9

Table 10 results were generated using **RIG1** with *d2h* as the performance goal. In these results, DODGE performed statistically better than the prior state-of-the-art (in five out of six data sets).

4.2.1 Issue Lifetime Estimation. Table 11 shows what techniques found the best predictors for the data of §3.3.3. The table compares DODGE versus the methods in a recent study on issue lifetime estimation [68], (feature selection with the Correlation Feature Selection [28] followed by classification via Random Forests).

Table 11 results were generated using **RIG1** with *d2h* as the performance goal. In these results, DODGE performed statistically better than prior work (in 47/63=75% of the datasets).

4.3 Bad Code Smell Results

Figure 2 shows which techniques found best predictors, using the data of §3.3.4. The figure compares DODGE versus bad smell detectors from a TSE'18 paper [41] that studied bad smells. The TSE

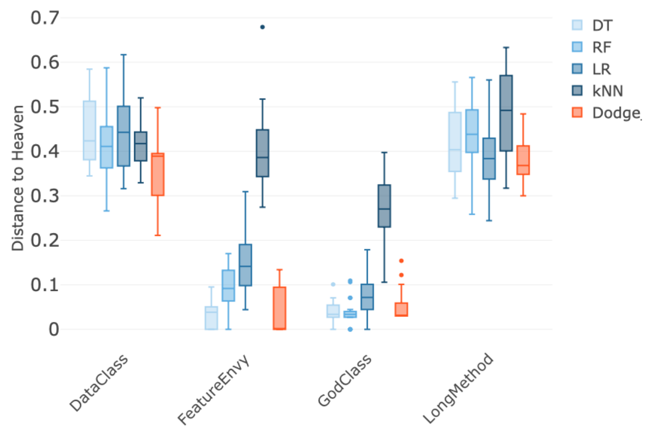

Figure 2: Four bad smell prediction results. Boxes show 75th-25th ranges fro 25 repeats. Whiskers extend min to max. Horizontal line in the middle of each box show median value. *Lower* values are *better*.

Table 12: Thirty seven results from non-SE problems. Colored cells show where DODGE performs best.

Data	Best tool	Data	Best tool
optdigits	RF	satellite	RF
climate-sim	SVM	credit-approval	Dodge
cancer	SVM	shop-intention	RF
image	RF	covtype	RF
hand	RF	drug-consumption	Dodge
biodegrade	RF	adult	RF
crowdsource	RF	blood-transfusion	Dodge
credit-default	SVM	cervical-cancer	Dodge
autism	RF	bank	SVM
bankrupt	Dodge	audit	RF
contraceptive	SVM	mushroom	RF
pendigits	RF	phishing	RF
car	RF	diabetic	SVM
hepmass	RF	htru2	SVM
kddcup	RF	sensorless-drive	RF
waveform	SVM	annealing	RF
cardiotocography	RF	shuttle	RF
electric-stable	RF	gamma	RF
liver	Dodge		

article used Decision Trees (CART), Random Forests, Logistic Regression and KNN($k = 5$). To the best of our knowledge, there has not been any prior case study applying hyperparameter optimizer to bad smell prediction.

Figure 2 results were generated using **RIG1** with *d2h* as the performance goal. In those results, DODGE has the same median performance as prior work for two data sets (FeatureEnvy and GodClass) and performed statistically better that the prior state-of-the-art (for DataClass and LongMethod). That is, compared to the other algorithms used in this study, DODGE statistically performs as well or better than anything else.

4.4 Results from Non SE-Problems

All the above problems come from the SE domain. Now we turn to non-SE problems. Table 12 shows which techniques found best predictors for the 37 different problems from §3.3.5.

In Table 12, DODGE was compared against standard data miners (CART, Random Forests, Logistic Regression and KNN($k = 5$)). Table 12 results were generated using **RIG1** with *d2h* as the performance goal. Each cell of that table lists the best performing learner. Note that despite its use of hyperparameter optimization (which should have given some advantage) DODGE performs very badly (only succeeds in 6/31 problems).

5 META-ANALYSIS

DODGE works very well for SE problems and works very poorly for non-SE problems. Why? To answer that question, this section uses the following meta-analysis:

(1) Given M treatments for data, divide those treatments into those from SE and those from elsewhere. In our case, DODDE is the SE-based tool. Also, as discussed in §3.1, grid search, DE, GA, SVM, Random Forests, decision tree learners, logistic regression, Naive Bayes, LDA, and EM are representative of standard tool.
(2) Apply those M treatments to P data sets (from SE and elsewhere).
(3) (Optional): As done in this paper, we recommend guiding that exploration by adopting the methods seen in recent publications in high-profile SE venues (since such methods have some pedigree in the literature).
(4) Using statistical methods and experimental rigs appropriate to those problems, build a table of data showing what techniques work best for each data sets. In that table:
   - For the dependent class, use the name of the best technique.
   - For the independent variables, use some data characteristics shared by all your problems. For this study, we used the independent attributes described in the next section.
(5) Run a decision tree learner (CART or C4.5) over that table.
(6) Read the generated tree to learn what kind of data is best processed by standard tools or SE-based tools.

5.1 Independent Variables for the Meta-Analysis

To understand when DODGE performs well/fail, we need some characterization of the data sets that holds across most of the above data. All of the above problems are stored in tables of data and can be described via:

- N = Number of samples (rows of example instances);
- F= Frequency of our selected target class;

Of course, our data has more independent information than these two points. For example, we tried using some other attributes (the percentile distributions of the expected value of the attribute entropy). But when we applied the CFS [28] feature selector, all those other independent attributes kept being deleted. Hence, to build a rule for determining when to use/avoid DODGE, we used (N, F).

The data generated this way is at http://tiny.cc/variance_defense.

5.2 Meta-Analysis Results

For this analysis, anything not labelled as "DODGE" was renamed "standard tool". The following rule was generated. The number of brackets after each conclusion shows the percentage of the data that falls to each branch.

```
IF    F <= 21.15%
THEN  IF  N <= 12215
      THEN use DODGE  (55)
      ELSE use standard tool (3)
THEN  use standard tool (42)
```

In summary, if the target class is not rare (more than 21%) then use standard tools. Otherwise, except for one infrequent case (3% of our data), use DODGE.

Since this rule was learned via data mining, it is not a categorical statement that, always, anything confirming to one of its branches will 100% benefit best from DODGE or some standard tool. There are certainly counter-examples to this rule in our data (particularly with the text mining and bad smell data where the target classes are quite common). Hence, in a 5-way cross-validation experiment, this rule did not report 100% recalls or precisions for selecting which tools:

class	recall	precision	false alarm
DODGE	77	72	28
Standard Tool	72	70	27

Nevertheless, given the complexity of the phenomena being explored here, the observed recall, precision, and false alarm rates are remarkably good. Also, as discussed in the next section, we found this rule insightful.

6 DISCUSSION

We posted the rule learned above to a machine learning discussion forum and received back the following insight. For many decades, the community has been using the UCI repository as a source of data for certifying their algorithms. Several of the data sets of Table 8 have appeared in publications dating back at least to 1995 i.e., nearly 25 years ago [52]. Quirks of that data have hence led to the development of algorithms that are skewed toward particular kinds of data.

To say that another way, the algorithms that we have called "standard tool" may not be "general" at all. Rather, they may be tools that are powerful in their home domain but need to be used with care if applied to new domains such as software engineering. Many data mining problems in software engineering target infrequent phenomena. Hence, when the target class is relatively uncommon (say, less than 20%), we can do better than just using any standard tools built and tested for data with much more frequent classes.

As to the connection of infrequent classes to the RELAX heuristic of §3.2.2, it seems reasonable to assume that the more common the target class, then:

- The easier it is to learn about the target;
- The finer the control the learner has about its generated models;
- The smaller the ϵ inaccuracy measure in the predictions;
- The fewer the redundant tuning options;
- And the less likely that the limited sampling of DODGE will find good tunings;

On the other hand, when the target class becomes less frequent, then:

- It is harder to find the target;
- The larger the observed ϵ in the results;
- The greater the number of redundant tunings;
- The more likely that DODE will work.

That is, since software engineering often deals with relatively infrequent target classes, then we should expect to see a large ϵ uncertainty in our conclusions.

7 THREATS TO VALIDITY

7.1 Sampling Bias

Sampling bias threatens any classification experiment since what matters for some data sets may or may not hold for others.

But in our case, sampling bias may be mitigated since we applied our frameworks to many data sets. In fact, our reading of the literature is that the above study uses much more data than most other publications. Also, we assert we did not "cherry pick" our data sets. All the non-SE data from [74] were applied here. As to the SE data, we used everything we could access in the time frame of this production of this paper.

That said, in future work, it would be important to check the results of this paper against yet more data sets from yet more problems from SE and elsewhere.

7.2 Learner Bias

When comparing DODGE against other tools, we did not explore all other tools. As stated above, such a comparison would not fit into a single paper. Instead, we use baselines taken from SE literature about bad smell detection, predicting Github issue close time, bug report analysis, and defect prediction. From that work we used tools that have some pedigree in the literature [3, 25, 61, 71].

7.3 Evaluation Bias

This paper used two performance measures, i.e., P_{opt} and $dist2heaven$ and many others exist [34, 46, 50]. Note that just because other papers use a particular evalaution bias, then it need not follow that it must be applied in all other papers. For example, precision is a widely used evaluation method even though it is known to peform badly when data sets have rare target classes [45].

7.4 Order Bias

For the performance evaluation part, the order that the data trained and predicted can affects the results. To mitigate for order bias, we used a cross-validation procedure that (multiple times) randomizes the order of the data.

7.5 Construct Validity

At various stages of data collection by different researchers, they must have made engineering decisions about what attributes to be extracted from Github for Issue lifetime data sets, or what object-oriented metrics need to be extracted. While this can inject issues of construct validity, we note that the data sets used here have also appeared in other SE publications, i.e., the class labels used here have verified by other researchers. So at the very least, using this data, we have no more construct validity bias than other researchers.

7.6 Statistical Validity

To increase the validity of our results, we applied two statistical tests, bootstrap and the a12 effect size test. Both of these are non-parametric tests so the above results are not susceptible to issues of parametric bias.

7.7 External Validity

One threat to external validity is that this article compares DODGE against existing baselines for traditional machine learning algorithms. We do not compare our new approach against the kinds of optimizers we might find in the search-based SE literature [63]. The reason for this is that the main point of this paper is to document a previously unobserved feature of the output space of software analytics. It is an open question whether or not DODGE is the best way to explore output space In order to motivate the community to explore that space, some work must demonstrate its existence and offer baseline results that, using the knowledge of output space, it is possible to do better than past work. We hope that this paper provides that motivation.

8 FUTURE WORK

Further to the discussion of in the *Threats to Validity* section, the analysis of this paper should be extended with studies on different data sets and other tools/algorithms/techniques.

Another useful extension to the above would be to explore problems with three or more goals (e.g., reduce false alarms while at the same time improving precision and recall). Currently, we have only explored classification tasks. DODGE needs to be applied for other tasks such as regression task.

Right now, DODGE only deprecate tunings that lead to similar results. Another approach would be to also depreciate tunings that lead to similar *and worse* results (perhaps to rule out larger parts of the output space, sooner).

Further, for pragmatically reasons it would be useful if the Table 1 list could be reduced to a smaller, faster to run, set of learners. That is, here we would select learners that run fastest while generating the most variable kinds of models.

Moving beyond DODGE, it would be useful to test the generality of the meta-analysis of §5. For example, Helendoorn and Devanbu [31] report cases where a standard technique (deep learning) is defeated by their own SE-based method (based on software "naturalness"). To repeat the analysis of this paper, it would be useful to collect a large library of problems where deep learning does/does not work best, then look for problem data features that predict for the success of deep learning.

9 CONCLUSION

One way to view SE is that it is just a problem domain where Non-SE/Standard tools can be deployed. This paper takes another view. We say software engineering is different and that those differences mean that best results arise from tools specialized to the particulars of software engineering.

For example, SE problem data is often about infrequent issues, for e.g., the occasional defect, the rarely exploited security violation, the requirement that only holds for one small part of a system. But as shown by the rule in §5.2, standard tools work best when the target is relatively more frequent. That is:

- SE problem data may be inherently different to the problem data used to develop standard tools.
- And that difference can be used to (e.g.,) find better ways to build predictions for SE classification problems such as defect prediction, issue text mining, Github issue lifetime estimation, and bad code smell detection.

The tool used to find these better predictions, called DODGE, runs orders of magnitude faster than other methods (see calculations at the end of §3.4). We conjecture that DODGE works better and faster than those other tools since those other general tools do not appreciate the simplicity of the output space of SE problems (Figure 1):

- Tools such as grid search, differential evolution, or genetic algorithms waste much CPU as they struggle to cover billions of tuning options like Table 1 (most of which yield indistinguishably different results).
- This is a waste of effort since when billions of possibilities (Table 1) are mapped into just a few outputs (Figure 1), then many of those possibilities are *redundant*; i.e., they lead to the same outputs.

DODGE knows that the faster we move away from these redundant possibilities, the faster we can move on to explore the remaining options. Hence, one lesson from the DODGE research is to say:

- Stop treating conclusion uncertainty as a problem;
- Instead, treat large conclusion uncertainty as a resource that can be used to simplify the analysis of software systems.

In conclusion, it is not good enough to just take tools developed elsewhwere, then apply them without modification to SE problems. Understanding SE is a different problem to understanding other problems that are more precisely controlled and restrained. The algorithms that might be called "standard tools" may not be "general" at all. Rather, they are tools which are powerful in their home domain but need to be used with care if applied to some new domains like software engineering. Software engineers need to develop tools that are better suited to the particulars of SE problems. As shown in this paper:

- Such new algorithms can exploit the peculiarities of SE data to dramatically simplify the analysis of software systems.
- Further, as shown in §5.2, it is possible to define a rule that precisely defines when *not* to use tools commissioned using SE data.

ACKNOWLEDGEMENTS

This work was partially funded by a NSF Grant #1703487.

REFERENCES

[1] Amritanshu Agrawal, Wei Fu, Di Chen, Xipeng Shen, and Tim Menzies. 2019. How to "DODGE" Complex Software Analytics? *IEEE Transactions on Software Engineering* (2019).
[2] Amritanshu Agrawal, Wei Fu, and Tim Menzies. 2018. What is wrong with topic modeling? And how to fix it using search-based software engineering. *Information and Software Technology* (2018).
[3] Amritanshu Agrawal and Tim Menzies. 2018. Is better data better than better data miners?: on the benefits of tuning SMOTE for defect prediction. In *International Conference on Software Engineering*.
[4] Amritanshu Agrawal, Tim Menzies, Leandro L. Minku, Markus Wagner, and Zhe Yu. 2018. Better Software Analytics via "DUO": Data Mining Algorithms Using/Used-by Optimizers. *CoRR* abs/1812.01550 (2018). http://arxiv.org/abs/1812.01550 Available online at https://arxiv.org/abs/1812.01550.
[5] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* (2016).
[6] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *International Conference on Software Engineering*.
[7] Arthur Asuncion and David Newman. 2007. UCI machine learning repository.
[8] Earl T Barr et al. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* (2015).
[9] Dave Binkley, Dawn Lawrie, and Christopher Morrell. 2018. The Need for Software Specific Natural Language Techniques. *Empirical Softw. Engg.* 23, 4 (Aug. 2018), 2398–2425. https://doi.org/10.1007/s10664-017-9566-5
[10] Steven Bird. 2006. NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*.
[11] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (March 2003), 993–1022. http://dl.acm.org/citation.cfm?id=944919.944937
[12] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. 1992. A Training Algorithm for Optimal Margin Classifiers *(COLT '92)*. ACM, 144–152. https://doi.org/10.1145/130385.130401
[13] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[14] Leo Breiman. 2017. *Classification and regression trees.* Routledge.
[15] Di Chen et al. 2018. Applications of Psychological Science for Actionable Analytics. *Foundations of Software Engineering* (2018).

[16] A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B* 39, 1 (1977), 1–38.

[17] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[18] Richard O. Duda, Peter E. Hart, and David G. Stork. 2000. *Pattern Classification (2Nd Edition)*. Wiley.

[19] B. Efron and R. J. Tibshirani. 1993. *An Introduction to the Bootstrap*. Chapman & Hall.

[20] R-SANGER Feldman. 2006. *J, The Text Mining Handbook*. Cambridge University Press.

[21] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman.

[22] Andrew Frank. 2010. UCI machine learning repository. *http://archive. ics. uci. edu/ml* (2010).

[23] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *Information and Software Technology* 76 (2016), 135–146.

[24] Wei Fu, Vivek Nair, and Tim Menzies. 2016. Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors? *CoRR* abs/1609.02613 (2016). arXiv:1609.02613 http://arxiv.org/abs/1609.02613

[25] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *International Conference on Software Engineering*.

[26] Emanuel Giger, Martin Pinzger, and Harald Gall. 2010. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 52–56.

[27] David E Goldberg and John H Holland. 1988. Genetic algorithms and machine learning. *Machine learning* 3, 2 (1988), 95–99.

[28] Mark A Hall and Geoffrey Holmes. 2002. Benchmarking attribute selection techniques for discrete class data mining. (2002).

[29] Maggie Hamill and Katerina Goseva-Popstojanova. 2009. Common trends in software fault and failure data. *IEEE Transactions on Software Engineering* (2009).

[30] Ahmed E Hassan. 2008. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*. IEEE, 48–57.

[31] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Foundations of Software Engineering*. ACM.

[32] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[33] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The Importance of Accounting for Real-world Labelling when Predicting Software Vulnerabilities *(ESEC/FSE 2019)*. ACM, 695–705. https://doi.org/10.1145/3338906.3338941

[34] Magne Jorgensen. 2004. Realism in assessment of effort estimation uncertainty: It matters how you ask. *IEEE Transactions on Software Engineering* (2004).

[35] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* (2013).

[36] F. Khomh, S. Vaucher, Y. G. Gueheneuc, and H. Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In *International Conference on Quality Software*.

[37] Foutse Khomh, Stephane Vaucher, Yann-Gael Gueheneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* (2011).

[38] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 96–107.

[39] A Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. 2009. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering* (2009).

[40] Jochen Kreimer. 2005. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science* (2005).

[41] Rahul Krishna and Tim Menzies. 2018. Bellwethers: A Baseline Method For Transfer Learning. *IEEE Transactions on Software Engineering* (2018).

[42] Rahul Krishna, Tim Menzies, and Lucas Layman. 2017. Less is more: Minimizing code reorganization using XTREE. *Information and Software Technology* (mar 2017). https://doi.org/10.1016/j.infsof.2017.03.012 arXiv:1609.03614

[43] Mark Last, Menahem Friedman, and Abraham Kandel. 2003. The data mining approach to automated software testing. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 388–396.

[44] Tim Menzies. 2008. Improving IV&V Techniques Through the Analysis of Project Anomalies: Text Mining PITS issue reports-final report. *Citeseer* (2008).

[45] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. 2007. Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'". *IEEE TSE* (2007).

[46] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. 2007. Problems with Precision: A Response to" comments on'data mining static code attributes to learn defect predictors'". *IEEE TSE* 33, 9 (2007).

[47] Tim Menzies, Jeremy Greenwald, and Art Frank. 2006. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering* 33, 1 (2006), 2–13.

[48] Tim Menzies and Ying Hu. 2003. Data mining for very busy people. *Computer* 36, 11 (2003), 22–29.

[49] Tim Menzies and Andrian Marcus. 2008. Automated severity assessment of software defect reports. In *International Conference on Software Maintenance*. IEEE.

[50] Tim Menzies, Dan Port, Zhihao Chen, and Jairus Hihn. 2005. Simple software cost analysis: safe or unsafe?. In *ACM SIGSOFT Software Engineering Notes*. ACM.

[51] Tim Menzies and Thomas Zimmermann. 2018. Software Analytics: What's Next? *IEEE Software* 35, 5 (2018), 64–70.

[52] Donald Michie, D. J. Spiegelhalter, C. C. Taylor, and John Campbell (Eds.). 1994. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, Upper Saddle River, NJ, USA.

[53] Ayse Tosun Misirli et al. 2011. Ai-based software defect predictors: Applications and benefits in a case study. *AI Magazine* (2011).

[54] Julie Moeyersoms, Enric Junqué de Fortuny, Karel Dejaeger, Bart Baesens, and David Martens. 2015. Comprehensible software fault and effort prediction: A data mining approach. *Journal of Systems and Software* 100 (2015), 80–90.

[55] Akito Monden et al. 2013. Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Transactions on Software Engineering* (2013).

[56] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[57] Ashish Nadkarni and Natalya Yezhkova. 2014. Structured Versus Unstructured Data: The Balance of Power Continues to Shift. *IDC (Industry Development and Models) Mar* (2014).

[58] Nicole Novielli, Daniela Girardi, and Filippo Lanubile. [n.d.]. A Benchmark Study on Sentiment Analysis for Software Engineering Research *(MSR '18)*. ACM, 364–375. https://doi.org/10.1145/3196398.3196403

[59] Alessandro Orso and Gregg Rothermel. 2014. Software testing: a research travelogue (2000–2014). In *Future of Software Engineering*. ACM.

[60] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. 2004. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*. ACM.

[61] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. 2013. How to Effectively Use Topic Models for Software Engineering Tasks? An Approach Based on Genetic Algorithms. In *ICSE*.

[62] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* (2011).

[63] Justyna Petke and Tim Menzies. 2018. Guest Editorial for the Special Section from the 9th International Symposium on Search Based Software Engineering. *Information and Software Technology* (2018).

[64] M Porter. 1980. The Porter Stemming Algorithm. , 130–137 pages. http://tartarus.org/martin/PorterStemmer/

[65] J. R. Quinlan. 1986. Induction of decision trees. *Machine Learning* 1, 1 (01 Mar 1986), 81–106. https://doi.org/10.1007/BF00116251

[66] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *ICSE*. ACM.

[67] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 428–439.

[68] Mitch Rees-Jones, Matthew Martin, and Tim Menzies. 2017. Better Predictors for Issue Lifetime. *CoRR* arxiv.org/abs/1702.07735 (2017).

[69] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2009. Recommendation systems for software engineering. *IEEE software* 27, 4 (2009), 80–86.

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

[71] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 321–332.

[72] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang. 2018. Perceptions, Expectations, and Challenges in Defect Prediction. *IEEE Transactions on Software Engineering* (2018), 1–1.

[73] Colin White. 2005. Consolidating, Accessing and Analyzing Unstructured Data. http://www.b-eye-network.com/view/2098.

[74] Leland Wilkinson, Anushka Anand, and Dang Nhon Tuan. 2011. CHIRP: A New Classifier Based on Composite Hypercubes on Iterated Random Projections *(KDD '11)*. ACM, 6–14. https://doi.org/10.1145/2020408.2020418

[75] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. 2009. Data mining for software engineering. *Computer* 42, 8 (2009), 55–62.

[76] Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software* (2013).

[77] Aiko Yamashita and Leon Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *ICSE*.

[78] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. 2012. Filtering clones for individual user based on machine learning analysis. In *International Workshop on Software Clones (IWSC)*.

[79] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Foundations of Software Engineering*. ACM.

[80] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[81] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Workshop on Managing Technical Debt*. ACM.