Is AI different for SE?

Amritanshu Agrawal, Tim Menzies

aagrawa8@ncsu.edu, timm@ieee.org
North Carolina State University, USA

ABSTRACT

What AI tools are needed for SE? Ideally, we should have simple rules that peek at data, then say “use this tool” or “use that tool”.

To find such a rule, we explored 120 different data sets addressing numerous problems, including bad smell detection, predicting Github issue close time, bug report analysis, defect prediction and dozens of other non-SE problems. To this data, we apply a SE-based tool that (a) out-performs the state-of-the-art for these SE problems yet (b) fails very badly on standard AI problems.

In those results, we can find a simple rule for when to use/avoid the SE-based tool. 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, standard AI tools work best when the target is relatively more frequent. Also, we can exploit this special properties of SE, to great effect (to rapidly find better optimizations for SE tasks via a tactic called “dodging”, explained in this paper).

More generally, this result says we need a new kind of SE research for developing new AI tools that are more suited to SE problems.

1 INTRODUCTION

The last 15 years have seen a dramatic increase in the application of AI tools in software engineering 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 AI 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 AI 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 specializing AI tools to the particulars of software engineering problem (see examples in next section).

One way to specialize AI tools for software engineering is hyperparameter optimization that automatically search 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 very 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 AI tools are more useful than others. This is significant deficiency. If such an insight was available, we could offer engineers a simple rule that lets them quickly select appropriate AI tools.

To address this deficiency, this paper applies various tools to numerous problems from SE and elsewhere. Some of these tools are general AI tools while another, called DODGE [1], was developed by Agrawal et al. specifically for SE applications. This tool is a hyperparameter optimizer that auto-adjusts learner settings in a somewhat unusual way (see the “relax” heuristics of §3). Agrawal et al. [1] justified that approach via certain SE-based properties 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 “SE-based” tool on non-SE problems. Therefore we apply DODGE to 120 data sets from SE and elsewhere. Specifically, 37 non-SE problems from the standard AI literature (taken from a standard machine learning data repository) 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. From that experience, we can:

• Offer a method for deciding when to use/avoid general AI tools.
• Describe a case study utilizing that methodology.
• Show that this method can indeed recommend when to use a general AI tool or a SE-based tool.
• Show that this recommendation is useful and if we follow them, we can find better optimizations for SE problems.
• Show that SE data is very different to the data used to develop many AI tools. 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 our results, many standard AI tools work best when the target is relatively more frequent.

The rest of this paper is structured as follows. The next section offers notes on related work and some standard AI tools. This is followed by a description of DODGE in §3. After that, in §4, we describe the 120 data sets used to test DODGE as well as other more standard AI tools. The experiments of §5 illustrate how well DODGE performs for SE problems and how badly it performs otherwise. The meta-analysis of §6 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 AI 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 AI tools developed elsewhere, then apply them verbatim to SE problems. Software engineers need to develop AI tools that are better suited to the particulars of SE problems.

As to the relationship to prior work, Table 1 and §3 are shortened version of material from Agrawal et al. [1]. Also, of the 120 data sets...
processed here, 16 come from that prior study—see the results in §5.41 and §5.4.2 (but just as a quality assurance measure, we reran all the experiments for those 16 data sets). The rest of this paper is new, including the experiments on 104 new data sets, and the meta-analysis of §6.

2 BACKGROUND

2.1 Lessons from Applying Standard AI Tools

This section offers examples where other SE researchers have argued that software engineering is different (so we need our own kind of specialized AI 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 AI data mining algorithms as “black boxes” [9, 23], where researchers do not tinker with internal design choices of those AI tools. 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 different AI 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 AI tools such as deep learning [31].

While the above work certainly illustrates the main point of this paper, it does not offer a simple general rule to look at new data and decide when to use pre-existing standard AI tools or when to move on to other technologies. This paper tried answering the same.

2.2 Standard AI Tools

This section describes some standard AI tools: grid search, DE, GA, SVM, Random Forests, decision tree learners, logistic regression, Naive Bayes, LDA, and EM. All these algorithms are examples of general AI tools developed outside of software engineering and then applied to SE. That is, one way to view SE is that it is just a problem domain where general AI tools can be deployed.

This paper takes another view. We say software engineering is new, including the experiments on 104 new data sets, and the meta-analysis of §6.

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]) for a list of options not explored in those studies. While this list is incomplete, it does include many of the hyperparameter optimizations being explored in the literature.

We comparatively evaluated DODGE against AI tools which have been published in prior SE research. 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 about 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 AI tools 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 the class variable are decision tree algorithms [14, 65]. These learners divide the data using the attribute whose values most separate the classes. This 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 some voting procedure across all the decision 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 some “kernel” to infer those extra dimension [12].

As for hyperparameter optimization, recall that these are tools to find a learner’s control settings (e.g., see Table 1). As mentioned above, we do not compare DODGE against all other hyperparameter optimization methods. Instead, we compare DODGE against prior hyperparameter optimization methods seen in SE literature. For example, for text mining of 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. In Table 1, 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 §5).

Given performance goals with the range $0 \leq g \leq 1$, $\epsilon$ divides the performance output space into $(1/\epsilon)^2$ 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.

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 $(1/\epsilon)^2$ possible outputs.

To implement “dodging”, DODGE models Table 1 as a tree where all nodes have initial weights $w = 0$. Next, $N_t$ times, DODGE selects branches at random. We evaluate the options in a branch and if the resulting scores are within $\epsilon$ 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. $N_t$ times, DODGE then makes random selection to restrict any numeric ranges. When a range is initially evaluated, a random number $r = \text{random}(lo, hi)$ 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:

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

When $\epsilon$ is large, a few samples should suffice to find good results. Hence, Agrawal et al. [1] recommends $\epsilon = 0.2$ and $N_1 + N_2 = 30$.

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 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 $10^7$ or $10^6$ or $10^5$ or $10^4$ times (respectively).

4 BUT WHEN WILL DODGE FAIL?

This section describes the problem data used to evaluate DODGE. Agrawal et al. say DODGE is an “SE-based” algorithm whose success depends on RELAX heuristic which, in turn, depends on the high variability of predictions for software. Their approach is so unusual that it is reasonable to ask “when won’t it work?”. To

---

1Why 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.
Table 2: Static code metrics for defect prediction. For details, see [42].

| Metric  | Description                                      | Version | Defects | Response for a class |
|---------|-------------------------------------------------|---------|---------|----------------------|
| amc     | Average method complexity                       |         | 0.00    | 3.39                 |
| avg cc  | Average McCabe                                   |         | 0.00    | 5.24                 |
| ca      | Overll coupling                                 |         | 0.00    | 0.00                 |
| cam     | Coupling among classes                          |         | 0.00    | 0.00                 |
| cmcb     | Coupling between methods                         |         | 0.00    | 0.00                 |
| ce      | Inheritance coupling                            |         | 0.00    | 0.00                 |
| dam     | Data access                                     |         | 0.00    | 0.00                 |
| ddt     | Depth of inheritance tree                       |         | 0.00    | 0.00                 |
| itc      | Inheritance coupling                             |         | 0.00    | 0.00                 |
| loc      | Lines of code                                   |         | 0.00    | 0.00                 |
| max cc  | Maximum McCabe                                   |         | 0.00    | 0.00                 |
| max fa  | Maximum functional abstraction                   |         | 0.00    | 0.00                 |
| max gc  | Maximum Group Contribution                       |         | 0.00    | 0.00                 |
| mea     | Method aggregation                               |         | 0.00    | 0.00                 |
| ncp     | Number of children                               |         | 0.00    | 0.00                 |
| rpm     | Number of public methods                         |         | 0.00    | 0.00                 |
| fc      | Response for a class                             |         | 0.00    | 0.00                 |
| wmc     | Weighted method complexity                       |         | 0.00    | 0.00                 |
| defects | Boolean: where defects found in bug-tracking     |         | 0.00    | 0.00                 |

4.1 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.

4.2 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 such large vocabulary problems, one of the tasks of a text miner is dimensionality reduction. See Table 1 for a list of dimensionality reduction pre-processing methods applied to the Table 4. 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.

### 4.3 Issue Lifetime Estimation

Issue tracking systems collect information about system failures, feature requests, and system improvements. Based on this information and actual project planning, 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 help 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 help 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%.

### 4.4 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 mining AI tools 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.

Recently, Fontana et al. [5] considered 74 systems in data mining analysis. Table 7 shows the data used in that analysis. This corpus 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%,
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-sun           | 540            | 15              | 91     |
| Financial     | credit-approval       | 586            | 15              | 85     |
| Medicine      | cancer                | 569            | 50              | 37     |
| Business      | shop-intention        | 12350          | 17              | 15     |
| Computer Vision| image                | 1608           | 19              | 85     |
| Life          | credit                | 12240          | 24              | 32     |
| Computer      | hand                  | 29076          | 15              | 37     |
| Social        | drug-consumption      | 1800           | 30              | 22     |
| Environment   | bio degrade           | 1055           | 51              | 34     |
| Social        | adult                 | 45222          | 14              | 25     |
| Physical      | crew-attendance       | 1800           | 28              | 24     |
| Medicine      | blood-transfusion     | 748            | 4               | 4      |
| Financial     | credit-default        | 30000          | 23              | 22     |
| Medicine      | cervical-cancer       | 668            | 33              |        |
| Social        | autism                | 609            | 19              | 50     |
| Marketing     | bank                  | 3090           | 20              | 12     |
| Financial     | bankruptcy            | 4759           | 64              | 3      |
| Financial     | audit                 | 775            | 25              | 72     |
| Life          | contraceptive         | 1249           | 9               | 96     |
| Life          | mushroom              | 5644           | 22              | 58     |
| Computer      | penguins              | 2955           | 16              | 76     |
| Security      | phishing              | 11055          | 30              | 56     |
| Automotive    | car                   | 1728           | 6               | 39     |
| Medicine      | diabetes              | 12192          | 17              | 99     |
| Physical      | hopisma               | 2000           | 27              | 50     |
| Computer      | hitsia                | 17858          | 8               | 9      |
| Computer      | kickoff               | 5201           | 41              | 59     |
| Automotive    | sensorless-drive      | 10658          | 15              | 85     |
| Physical      | waveform              | 3304           | 21              | 50     |
| Physical      | annealing             | 716            | 10              | 15     |
| Medicine      | cardiology            | 2126           | 20              | 85     |
| Physical      | sustle                | 54489          | 9               | 16     |
| Electrical    | electric-stable       | 10000          | 12              | 36     |
| Physical      | gamma                 | 19200          | 10              | 35     |
| Medicine      | liver                 | 379            | 10              | 72     |

5 EXPERIMENTS AND RESULTS

To comparatively evaluate the performance of AI tools, we need performance measures (§5.1), appropriate experimental rigs (§5.2), and statistical tests (§5.3).

5.1 Performance Measures

\[ D2h = \sqrt{\frac{1 - \text{Recall}^2 + (0 - \text{FPR})^2}{2}} \] (3)

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

For defect prediction, $P_{opt}(20)$ comments on the inspection effort required after a defect predictor is triggered. $P_{opt}(20) = 1 - \Delta_{opt}$, where $\Delta_{opt}$ is the area between the effort (code-churn-based) cumulative lift charts of the optimal learner and the proposed learner. To calculate $P_{opt}(20)$, 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 Ostrander et al. [60], $P_{opt}(20)$ is reported at the 20% point. Futher, following Kamei, Yang et al. [35, 55, 79] we normalize $P_{opt}$ using:

\[ P_{opt}(m) = \frac{S_{(optimal) - S(m)}}{S_{(optimal) - S(worst)}} \]

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, $P_{opt}(20)$ (like $d2h$) has the range zero to one. Note that larger values of $P_{opt}(20)$ are better, but smaller values of $d2h$ are better.

5.2 Experimental Rigs

Jimenez et al. [33] recommended that train/test data be labelled in their natural temporal sequence. E.g. for defect data, we apply training and hyperparameter optimization to the prior versions, then tested on the latest version. We call this approach RIG0.

When such temporal markers are absent, we applied a cross-val method (which is also standard in literature, e.g., see [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 use the remaining 20% of the data for testing. We call this approach RIG1.

5.3 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 & Brand in their ICSE paper [6].
5.4 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.

5.4.1 Defect Prediction. Table 9 shows which AI tools found best predictors for defects, using the data of §4.1.

One issue with defect prediction is the occasional presence of class imbalance. If 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 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 \( d_{2h} \) and \( \text{PopT}(20) \) as the performance goal. DODGE performed statistically better than the prior state-of-the-art in sixteen out of twenty data sets.

5.4.2 Text Mining Issues Reports. Table 10 shows which AI tools found best predictors for the data of §4.2. 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 study 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 9: Ten defect prediction results. Smote+ means SMOTE + DE tuning + best of the Ghotra’15 learners.

| Data   | Best tool   |
|--------|-------------|
| D2h    | RF          |
| Poi    | Dodge       |
| Lucene | Dodge       |
| Camel  | Dodge       |
| Log4j  | SMOTE+      |
| Xerces | Dodge       |
| Velocity| SMOTE+  |
| Xalan  | SMOTE+      |
| Ivy    | Dodge       |
| Synapse| Dodge       |
| Jedit  | SMOTE+      |

Table 10: Six text mining results.

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

Table 10 results were generated using RIG1 with \( d_{2h} \) 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).

5.4.3 Issue Lifetime Estimation. Table 11 shows which AI tools found best predictors, using the data of §4.3. The table compares DODGE versus the methods in a recent study on issue lifetime estimation [68], i.e., feature selection with the Correlation Feature Selection algorithm [28] followed by classification using Random Forests.

Table 11 results were generated using RIG1 with \( d_{2h} \) as the performance goal. In these results, DODGE performed statistically better than prior work (in 47/63=75% of the datasets).

5.4.4 Bad Code Smell Detection. Figure 2 shows which AI tools found best predictors, using the data of §4.4. The figure compares DODGE versus bad smell detectors from a TSE’18 paper [41] that studied bad smells. The TSE article used Decision Trees (CART), Random Forests, Logistic Regression and KNN \((k = 5)\). To the best
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        | cvtype        | 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       |
| kiddcup       | SVM       | sensorless-drive| RF |
| waveform      | SVM       | annealing     | RF        |
| cardiotocography | RF  | shuttle       | RF        |
| electric       | RF        | gamma         | RF        |
| liver         | Dodge     |               |           |

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.

5.4.5 Non SE-Problems. The results so far support DODGE’s approach to hyperparameter optimization. But all the above problems come from the SE domain. When we turn to non-SE problems, a very different pattern was observed.

Table 12 shows which AI tools found best predictors for the 37 different problems from §4.5.

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).

6 META-ANALYSIS

This section addresses the puzzle raised by the above results. DODGE works very well for SE problems and works very poorly for non-SE problems. Why?

6.1 Methodology

This section describes our method for learning what kind of problems works best for SE/AI-based tools.

(1) Given \(M\) treatments for data, divide those treatments into those from SE and those from elsewhere. In our case, DODGE is the SE-based tool. Also, as discussed in §2.2, grid search, DE, GA, SVM, Random Forests, decision tree learners, logistic regression, Naive Bayes, LDA, and EM are representative of standard AI.

(2) Apply those \(M\) treatments to \(P\) data sets (some SE, some from elsewhere). In the case of this paper, our data was described above.

(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 AI tools work best for each data sets. In that table:

- For the dependent class, use the name of the best AI tool.
- 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 general AI tools or SE-based tools.

6.2 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 in this way is shown at http://tiny.cc/variance-defense.

6.3 Meta-Analysis Results

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

\[
\text{IF } F \leq 21.15\% \text{ THEN }
\]
\[
\text{IF } N \leq 12215 \text{ THEN use DODGE (55) ELSE use general AI tools (3)} \text{ THEN use general AI tools (42)}
\]

In summary, if the target class is not rare (more than 21%) then use general AI 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% 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       | 72     | 72        | 28          |
| General AI  | 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.

6.4 Discussion

We posted the rule learned above to a machine learning discussion forum and received back the following insight. For many decades, the AI 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 "general AI tools" 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 AI tools built and tested for data with much more frequent classes.

As to the connection of infrequent classes to the RELAX heuristic of §3, 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 AI tools, we did not explore all other AI 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 evaluation 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 perform 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 different AI tools.

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 tasks.

Right now, DODGE only deprecate tunings that lead to similar results. Another approach would be to also deprecate 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 §6. For example, Helendoorn and Devanbu [31] report cases where a standard AI tool (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 general AI 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 §6.3, standard AI 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 AI 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 §5). We conjecture that DODGE works better and faster than those other tools since those other general AI tools do not appreciate the simplicity of the output space of SE problems (Figure 1):

- AI 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 AI tools developed elsewhere, 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 “general AI 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 AI 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 §6.3, it is possible to define a rule that precisely defines when not to use those SE-based tools.

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

[5] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandra 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 Morrill. 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
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. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’11)* ACM, New York, NY, USA, 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 *International Conference on Software Engineering*.

[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, Jimping 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] Shun 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.