Noname manuscript No.
(will be inserted by the editor)

Better Software Analytics via "DUO": Data Mining Algorithms Using/Used-by Optimizers

Amritanshu Agrawal · Tim Menzies · Leandro L. Minku · Markus Wagner · Zhe Yu

the date of receipt and acceptance should be inserted later

Abstract This paper claims that a new field of empirical software engineering research and practice is emerging: data mining using/used-by optimizers for empirical studies, or DUO. For example, data miners can generate models that are explored by optimizers. Also, optimizers can advise how to best adjust the control parameters of a data miner. This combined approach acts like an agent leaning over the shoulder of an analyst that advises "ask this question next" or "ignore that problem, it is not relevant to your goals". Further, those agents can help us build "better" predictive models, where "better" can be either greater predictive accuracy or faster modeling time (which, in turn, enables the exploration of a wider range of options). We also caution that the era of papers that just use data miners is coming to an end. Results obtained from an unoptimized data miner can be quickly refuted, just by applying an optimizer to produce a different (and better performing) model. Our conclusion, hence, is that for software analytics it is possible, useful and necessary to combine data mining and optimization using DUO.

Keywords Software analytics, data mining, optimization, evolutionary algorithms

Authors listed alphabetically.

Amritanshu Agrawal, Tim Menzies, Zhe Yu
Department of Computer Science, North Carolina State University, Raleigh, NC, USA. E-mail: zyu9@ncsu.edu

Leandro L. Minku
School of Computer Science, University of Birmingham, Edgbaston, Birmingham, UK. E-mail: l.l.minku@cs.bham.ac.uk

Markus Wagner
School of Computer Science, University of Adelaide, Adelaide, SA, Australia. E-mail: markus.wagner@adelaide.edu.au

1 Introduction

After *collecting data* about software projects, and before *making conclusions* about those projects, there is a middle step in empirical software engineering where the data is *interpreted*. When the data is very large and/or is expressed in terms of some complex model of software projects, then interpretation is often accomplished, in part, via some automatic algorithm. For example, an increasing number of empirical studies base their conclusions on data mining algorithms (e.g. see [10, 76, 78–80]) or model-intensive algorithms such as optimizers (e.g. see the recent section on Search-Based Software Engineering in the December 2016 issue of this journal [62]).

This paper asserts that, for software analytics it is *possible*, *useful* and *necessary* to combine data mining and optimization. We call this combination DUO, short for data miners using/used-by optimizers. Once data miners and optimizers are combined, this results in a very different and interesting class of interpretation methods for empirical SE data. DUO acts like an agent leaning over the shoulder of an analyst that advises "ask this question next" or "ignore that problem, it is not relevant to your goals". Further, DUO helps us build "better" predictive models, where "better" can be for instance greater predictive accuracy, or models that generalize better, or faster modeling time (which, in turn, enables the faster exploration of a wider range of options). Therefore, DUO can speed up and produce more reliable analyses in empirical studies.

This paper makes the following claims about DUO:

- **Claim1:** *For software engineering tasks, optimization and data mining are very similar*. Hence, it is natural and simple to combine the two methods.
- **Claim2:** *For software engineering tasks. optimizers can greatly improve data miners*. A data miner's default tuners can lead to sub-optimal performance. Automatic optimizers can find tunings that dramatically improve that performance [3, 4, 41].
- **Claim3:** *For software engineering tasks, data miners can greatly improve optimization.* If a data miner groups together related items, an optimizer can explore and report conclusions that are general across a set of solutions. Further, optimization for SE problems can be very slow (e.g. consider the 15 years of CPU needed by Wang et al. [123]). But if that optimization executes over the groupings found by a data miner, that inference can terminate orders of magnitude faster [65, 91].
- **Claim4:** *For software engineering tasks, data mining without optimization is not recommended.* Conclusions reached from an unoptimized data miner can be changed, sometimes even dramatically improved, by running the same tuned learner on the same data [41]. Researchers in data mining should, therefore, consider adding an optimization step to their analysis.

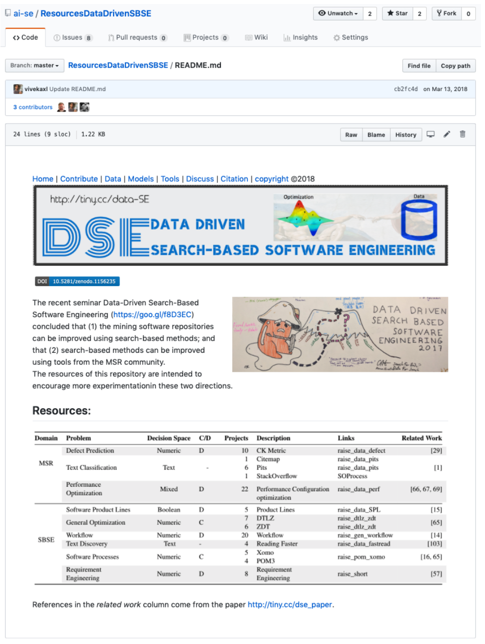
This paper extends a prior conference paper on the same topic [88]. That prior paper focused on case study material for DUO-like applications (see Figure 1). While a useful resource, it did not place DUO in a broader context. Nor did it contain the literature review of this paper. That is, this paper is both more general (discussing the broader context) and more specific (containing a detailed literature review) than prior work.

The rest of this paper is organized as follows. The next section describes some related work. After that, we devote one section to each of **Claim1, Claim2, Claim3** and **Claim4**. To defend these claims we use evidence drawn from a literature review of applications of DUO, described in Table 1 (Table 2 and Table 3 offer notes on the data miners and optimizers seen in the literature review). Finally, a *Research Directions* section discusses numerous open research issues that could be better explored within the context of DUO.

2 Tutorial

Before discussing combinations of data miners and optimizers, we should start by defining each term separately, and discussing their relationship. *Optimizers* reflect over a model to learn inputs that most improve model output. *Data miners*, on the other hand, reflect over data to return a summary of that data. Data miners usually explore a fixed set of data while optimizers can generate more data by re-running the model N times using different inputs. Data miners "slice" data such that similar patterns are found within each division. Optimizers "zoom" into interesting regions of the data, using a model to fill in missing details about those regions.

There are many different kinds of data miners, each of which might produce a different kind of model. For example, regression methods generate equations; nearest-neighbor-based methods might yield a small set of most interesting attributes and examples; and neural net methods return a weighted directed graph. Some data miners generate combinations of models. For example, the M5 "model tree learner" returns a set of equations and a tree that decides when to use each equation [101].


ai-se / ResourcesDataDrivenSBSE
Unwatch 2
Star 2
Fork 0
Code
Issues 8
Pull requests 0
Projects 0
Wiki
Insights
Settings
Branch: master
ResourcesDataDrivenSBSE / README.md
Find file
Copy path
vivekaxl Update README.md
cb2fc4d on Mar 13, 2018
3 contributors
24 lines (9 sloc) | 1.22 KB
Raw
Blame
History
Home | Contribute | Data | Models | Tools | Discuss | Citation | copyright ©2018
http://tiny.cc/data-SE
Optimization
Data
DSE DATA DRIVEN SEARCH-BASED SOFTWARE ENGINEERING
DOI 10.5281/zenodo.1156235
DATA DRIVEN SEARCH BASED SOFTWARE ENGINEERING 2017

The recent seminar Data-Driven Search-Based Software Engineering (https://goo.gl/f8D3EC) concluded that (1) the mining software repositories can be improved using search-based methods; and that (2) search-based methods can be improved using tools from the MSR community.
The resources of this repository are intended to encourage more experimentationin these two directions.

Resources:

Domain	Problem	Decision Space	C/D	Projects	Description	Links	Related Work
MSR	Defect Prediction	Numeric	D	10	CK Metric	raise_data_defect	[29]
	Text Classification	Text	-	1 6 1	Citemap Pits StackOverflow	raise_data_pits raise_data_pits SOProcess	[1]
	Performance Optimization	Mixed	D	22	Performance Configuration optimization	raise_data_perf	[66, 67, 69]
SBSE	Software Product Lines	Boolean	D	5	Product Lines	raise_data_SPL	[15]
	General Optimization	Numeric	C	7 6	DTLZ ZDT	raise_dtlz_zdt raise_dtlz_zdt	[65]
	Workflow	Numeric	D	20	Workflow	raise_gen_workflow	[14]
	Text Discovery	Text	-	4	Reading Faster	raise_data_fastread	[103]
	Software Processes	Numeric	C	5 4	Xomo POM3	raise_pom_xomo	[16, 65]
	Requirement Engineering	Numeric	D	8	Requirement Engineering	raise_short	[57]

References in the *related work* column come from the paper http://tiny.cc/dse_paper.

Fig. 1: Training and teaching resources for this work: http://tiny.cc/data-se. The authors of this paper invite the international research community to issue pull requests on this material.

Whereas data miners usually have "hard-wired goals" (e.g. improve accuracy), the goals of an optimizer can be adjusted from problem to problem. In this way, an optimizer can be tuned to the needs of different business users. For example:

– For requirements engineers, we can find the *least* cost mitigations that enable the delivery of *most* requirements [33].
– For project managers, we can apply optimizers to software process models to find options that deliver *more* code in *less* time with *fewer bugs* [75].
– For developers, our optimizers can tune data miners looking for ways to find *more* bugs in *fewer* lines of code (thereby reducing the human inspection effort required once the learner has finished [42].
– Etc.

In any engineering discipline, including software engineering, it is common to trade-off between multiple completing goals (e.g. all the examples in the above list are competing for multi-objective goals). Hence, for the rest of this tutorial section, we focus on multi-objective optimization.

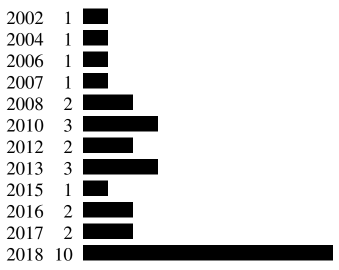
Research questions	Q1. What papers have used DUO in the past? Q2. When were they published? Q3. What problems from what software engineering domains have they tackled? Q4. What optimizers have they used? Q5. What data miners have they used? Q6. What were the advantages offered by DUO?	
Search query	software engineering AND ("optimization" OR "evolutionary algorithm") AND ("data mining" OR "analytics" OR "machine learning")	
Search engines	Three widely used literature sources were adopted: – ACM Guide to Computing Literature (`https://dl.acm.org/advsearch.cfm?coll=DL&dl=ACM`) – IEEEXplore (`https://ieeexplore.ieee.org/search/advsearch.jsp`) – Google Scholar (`https://scholar.google.com/#d=gs_asd&p=&u=`)	
Inclusion criteria	– ACM and IEEEXplore: >3 citations per year OR published in 2017/2018. – Google Scholar: (>10 cites/year OR published in 2017/2018) AND in first 20 result pages. – For all search engines: paper relates to software engineering and must use DUO. The number of citations was more strict in Google Scholar, because this search engine usually retrieves more citations than the others. We restricted the Google Scholar results to the first 20 pages (200 papers) because Google Scholar allows papers that do not match the search query completely to be retrieved, resulting in 28,000 results that would need to be manually filtered for the software engineering domain and number of citations per year.	
Q1	The search query retrieved 393 results when considering all ACM, IEEEXplore and the first 20 pages of Google Scholar results. After excluding papers that did not match the citation and year of publication criteria, we obtained 90 papers. After excluding any duplicates and papers that were not in the software engineering domain, we obtained 72 papers. Finally, among these 72, 29 used DUO. These are: [1,3–5,21,23,24,27,30,33,39,41,55,56,69,73,74,84,85,90,91,93,94,107,109–111,118,134]. In addition, we found a literature review related to DUO [2]. This review is on the topic of genetic programming for predictive modeling in software engineering. Genetic programming can be seen as an optimization algorithm.	
Q2	The number of DUO papers published per year is shown below, with 2018 having the largest number of papers. One might claim that this is because we ignore the number of citations in 2018. However, this is also the case for 2017, which had a considerably smaller number of DUO papers. Therefore, DUO seems to have been recently attracting increased research interest. 	
Q3	Domain where DUO is applied:	Specific problem:
	Project management	Software effort estimation [21,24,55,56,69,84,85,93,110,111], managing human resources [23].
	Requirements	Requirements optimization [33].
	Design	Software architecture optimization [30], extraction of products from very large product lines [21].
	Security	Intrusion detection [5,107]
	Software quality	Software detect prediction [4,27,41,73,109,117,134], test case generation [1].
	Software configuration	Software configuration optimization [23,90,91].
	Text mining: StackOverflow	Linking posts [3,39], topic modeling [3,118].
	Text mining: Defect Reports	Defect reports topic modeling [3].
	Text mining: Software Artifact Search, Linking and Labeling	Traceability link recovery [94], locate features in source code [94], software artifacts labeling [94], topic modeling of software engineering papers [3], Stack Overflow/GitHub topic modeling [118].
Q4	See Table 2.	
Q5	See Table 3.	
Q6	See Sections 4, 5 and 6.	

Table 1: Exploring the DUO literature.

2.1 Multi-Objective Optimization

All the optimizers of Table 2 seek the maxima or minima of an objective functions $y_i = f_i(x)$, $1 \leq i \leq m$ (where f_i are the objective or “evaluation” functions, m is the number of objectives, x is called the *independent variable*, and y_i are the *dependent variables*).

For single-objective ($m = 1$) problems, algorithms aim at finding a single solution able to optimize the objective function, i.e., a **global maximum/minimum**.

For multi-objective ($m > 1$) problems, there is no single ‘best’ solution, but a number of ‘best’ solutions. These best solutions, also known as the **Pareto Front**, are the non-dominated solutions found using a *dominance* relation. This **domination criterion** can be defined many ways:

Genetic Algorithms (GAs) execute over multiple generations. Generation zero is usually initialized at random. After that, in each generation, candidate items are subject to *select* (prune away the less interesting solutions), *crossover* (build new items by combining parts of selected items), and *mutate* (randomly perturb part of the new solutions). Modern GAs take different approaches to the *select* operator e.g. dominance rank, dominance count, and dominance depth. Notable exceptions are MOEA/D that use a decomposition operator to divide all the solutions into many small neighborhoods where if anyone finds a better solution, all its neighbors move there as well [7, 24, 30, 55, 84, 93, 94, 109, 111].

Different evolution (DE) execute over multiple generations. Generation zero is usually initialized at random. After that, in each generation, candidate items are subject to *select* (prune away the less interesting solutions), *mutate* (build new items by combining with 3 other random candidates from the same generation) [3, 4, 39, 41, 74, 116].

MOEAs contains different types of multi-objective evolutionary algorithms such as MOEA/D, NSGA-II, and more. They differ based on either the diversity mechanism (such as crowding distance or hypervolume contribution), or their sorting algorithm, or their use of target vectors, etc. [1, 18, 23, 28, 57, 85, 111].

Particle Swarm Optimization (PSO) works by having a swarm of candidates where these candidates move around in the search space using simple formulae. The movements of the particles are guided by their own best-known position in the search-space as well as the entire swarm’s best-known position. When improved positions are being discovered these will then come to guide the movements of the swarm [5, 27, 99, 107].

Genetic programs (GPs) are like GAs except that while GAs manipulate candidates that are lists of options, GPs manipulate items that are trees. GPs can, therefore, be better for problems with some recursive structure (e.g. learning the parse tree of a useful equation) or when human-readable models are sought [2, 8, 64].

SWAY first randomly generates a large number of candidates, recursively divides the candidates and only selects one. SWAY quits after the initial generation while GA reasons over multiple generations. It makes no use of reproduction operators so there is no way for lessons learned to accumulate as it executes [23].

Tabu Search uses a local or neighborhood search procedure to iteratively move from one potential solution x to an improved solution x’ in the neighborhood of x, until some stopping criterion has been satisfied [23, 46].

FLASH, a sequential model-based method such as Bayesian optimization, is a useful strategy to find extremes of an unknown objective. FLASH is efficient because of its ability to incorporate prior belief as already measured solutions (or configurations), to help direct further sampling. Here, the prior represents the already known areas of the search (or performance optimization) problem. The prior can be used to estimate the rest of the points (or unevaluated configurations). Once one (or many) points are evaluated based on the prior, the posterior can be defined. The posterior captures the updated belief in the objective function. This step is performed by using a machine learning model, also called surrogate model [91].

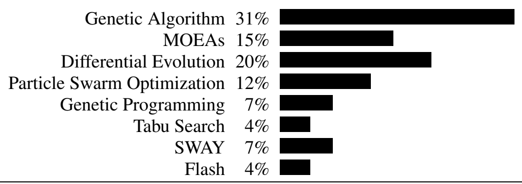


Table 2: Notes on different optimizers. Bar chart at bottom shows the frequency of use in papers of Table 1’s literature review.

Decision Tree learners such as CART and C4.5 seek attributes which, if we split on their ranges, most *reduces* the expected value of the diversity in splits. These algorithms then recurse over each split to find further useful divisions. For numeric classes, diversity may be measured in terms of variance. For discrete classes, the Gini or entropy measures can assess diversity. Decision tree learners are widely applied in software engineering due to the simplicity and interpretability [1, 4, 17, 24, 27, 41, 73, 84, 90, 91, 93, 111, 117, 120].
Support Vector Machines (SVMs) are supervised learning trying to separate training items from two classes by a clear gap [115]. For linearly non-separable problems, SVMs either allow but penalize misclassification of training items (soft-margin) or utilize kernel tricks to map input data to a higher-dimensional feature space before learning a hyperplane to separate the two classes. Many software engineering researchers have explored using SVM models to predict software artifacts [4, 27, 39, 74, 93, 109, 117, 120].
Instance-based algorithms: instead of fitting a model on the training data, instance-based algorithms stores all the training data as a database. When a new test item comes, the similarities between the test item and every training item are measured. The test item is then classified to the class where most of its similar training items belong to. Examples of instance-based algorithms include k-nearest neighbor [4, 111, 117, 120] and analogy algorithms [24, 55, 84, 111].
Ensemble algorithms use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone [100]. Usually, an ensemble algorithm builds multiple weak models that are independently trained and combines the output of each weak model in some way to make the overall prediction. Examples of ensemble algorithms include boosting [36, 117], bagging [15, 84, 117], and random forest [4, 17, 41, 72, 118].
Bayesian algorithms collect separate statistics for each class. Those statistics are used to estimate the prior distribution and the likelihood of each item in each class. When classifying a new test item, the estimated prior distribution and likelihood are applied to calculate its posterior distribution, which is then used to predict the class of the test item. Bayesian algorithms are widely applied in solving classification problems in software engineering [4, 5, 17, 27, 93, 107, 117].
Regression algorithms use predefined functions to model the mapping from input space to output space. Parameters of the predefined function are estimated by minimizing the error between the ground truth outputs and the function outputs. Regression algorithms can be applied to solve both regression (e.g. linear regression [111]) and classification problems (e.g. logistic regression [4, 41, 111, 117, 120]).
Artificial Neural Networks (ANNs) are models that are inspired by the structure and function of biological neural networks [44]. An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it. Such ANNs are usually applied in software engineering as baseline algorithms [24, 27, 84, 117].
Dimensionality reduction algorithms transform the data in the high-dimensional space to a space of fewer dimensions [104]. The resulting low-dimensional space can be used as features to train other data mining models or directly used as a clustering result. Examples of dimensionality reduction algorithms include principal component analysis [60], linear discriminant analysis [103, 117], and latent Dirichlet allocation [3, 12, 94, 118].
Covering (rule-based) algorithms provide mechanisms that generate rules in a bottom-up, separate-and-conquer manner by concentrating on a specific class at a time and maximizing the probability of the desired classification. Examples of rule-based algorithms include PRISM [70] and RIPPER [26, 27, 117, 120].
Deep Learning methods are a modern update to ANNs that exploit abundant cheap computation. Deep learning uses a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input. In a supervised or unsupervised manner, it learns multiple levels of representations that correspond to different levels of abstraction; the levels form a hierarchy of concepts [29]. Examples of instance-based algorithms include deep belief networks and convolutional neural network [39].

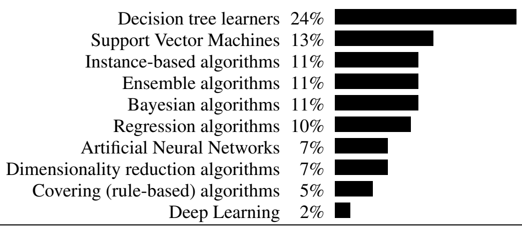


Table 3: Notes on the different data miners found by the literature review of Table 1. Bar chart at bottom shows the frequency of use in papers of Table 1's literature review.

Algorithm 1: Zitzler's indicator domination predicate [135]. Most useful when reasoning about more than two objectives [113]. In the weights array, -1,1 means "minimize, maximize" respectively. Objective scores are normalized 0..1 since, otherwise, the exponential calculation might explode.

```
def x_better_than_y(
            x,y,      # lists of candidate solutions
            weights, # dictionary of objective weights,
            goals,    # list of objective indexes
            lo, hi): # lists of low,high values of objectives
    xloss, yloss, n = 0, 0, len(w)
    for g in goals:
        a      = normalize( x[g], lo[g], hi[g] )
        b      = normalize( y[g], lo[g], hi[g] )
        w      = weights[g]
        xloss -= 10**( w * (a-b)/n
        yloss -= 10**( w * (b-a)/n )
    end
    return xloss < yloss

def normalize(z,lo,hi): return  (z  - lo) / (hi - lo + 0.00001)
```

- The standard "Boolean dominance" predicates says that x_1 dominates another x_2, if x_1 is better than x_2 in at least one objective and if x_1 is no worse than x_2 in all other objectives.
- Another style of domination predicate is the "Zitler indicator" shown in Algorithm 1. This method reports what loses least: moving from x to y or y to x (and the solution x is preferred if moving to x results is the least loss; see Algorithm 1).
- Regardless of how domination is defined, a solution is called non-dominated if no other solution dominates it.

As shown in Figure 2, solutions found by an optimization algorithm are called the **Predicted Pareto Front** (PF). The list of best solutions of a space found so far is known as the **Actual Pareto Front**. As this can be unknowable in practice or prohibitively expensive to generate, it is common to build the actual front using the union of all optimization outcomes all non-dominated solutions.

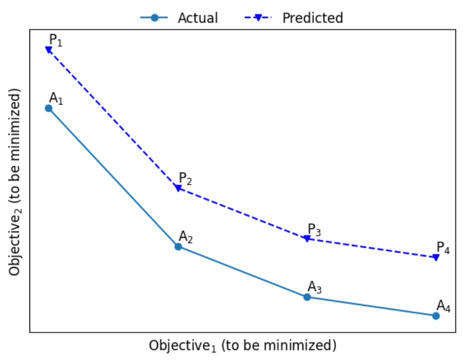


Fig. 2: Exploring multi-objective goals. Here, "predicted" is what is achieved and "actual" is some idealized set of goals (which may not be reachable).

2.2 Meta-heuristic Optimizers

Any non-trivial software system contains thousands to millions of condition statements (if, case, etc). Each of these conditions subdivides the internal state space of a software system. From a technical perspective, this means that the state space of software is *not* a continuous function (since each conditional creates a different sub-system with different properties). Accordingly, to optimize a software system, simplistic numeric optimizers are not appropriate. Instead, a common approach is to use the search-based meta-heuristic methods shown in Figure 3. The rest of this section offers notes on that pseudo-code.

```
# A Traditional bare−bone algorithm
def SBSE(problem, pop_size, generation):
  # Generate the decision space by randomly generating solutions
  initial_pop = [initialize(random=True) for _ in range(pop_size)]
  # Evaluate all the individuals in the initial_pop using problem
  # specific fitness function
  population = [individual.fitness(problem) for individual in initial_pop]
  while generation > 0:
    # Generate mutants by recombining individuals from population
    # Size of new_pop is equal to size of pop.
    new_population = operator.recombine(population)
    # Evaluate the mutants (ind) from the new_pop
    new_population = [ind.fitness(problem) for ind in new_population]
    # Select the best performing individual from pop + new_population
    pop = operator.elitism(new_population + population)
    # Reduce the budget by 1.
    generation −= 1
  return population
```

Fig. 3: Meta-heuristic search (high-level view).

Multi-objective optimizers for SE use either a model (which represent a software process [13]) or can be directly applied to any software engineering system (including problems that require evaluating a solution by running a specific benchmark [65]). However the model or software system is created, it can be represented in the **decision space** in a myriad of ways, for example, as Boolean or numerical vectors, as strings, and as graphs. (and the space of all solutions that can be represented in this way is the decision space).

Search-based meta-heuristic methods explore and refine a **population** of candidate solutions using the decision space representation. The process of search typically starts by creating a population of random solutions (valid or invalid) [21, 22, 54, 106]. A **fitness function** then maps the solution (which is represented using numerics) to a numeric scale (also known as *Objective Space*) which is used to distinguish between good and not so good solutions Simply put, the fitness function is a transformation function which converts a point in the decision space to the objective space.

Some **variation operator** is then applied to **mutate** the solutions (that is, to generate new candidate solutions). Typically, unary operators are called mutation operators, and operators with higher arity are called crossover operators.

Then, it is common to use operators that apply some pressure towards selecting better solutions from the union of the current population and the newly created solutions. This selection is done either deterministically (e.g., elitism operator) or stochastically. An important class of operator is *Elitism* which simulates the 'survival of the fittest' strategy, i.e., it eliminates not so good solutions thereby preserving the good solutions in the population.

As shown by the `while` loop of Figure 3, a meta-heuristic algorithm iteratively improves the population (set of solutions) as, at each iteration, each member of the population might be mutated or eliminated via elitism. Each step of this process, which includes generation of new solutions using recombination of the existing population and selecting solutions using the elitism operator, is called a **generation**. Over successive generations, the population 'evolves' (in the best case) toward a globally optimal solution.

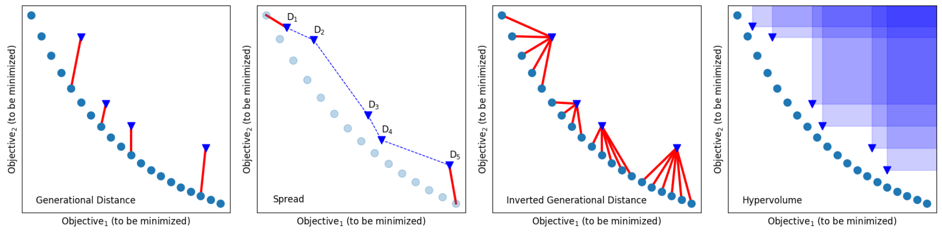


Fig. 4: Optimization evaluation criteria.

2.3 How is it all assessed?

For single-objective problems, measures such as *absolute residual* or *rank-difference* can be very useful. For multi-objective problems, the evaluation scores must explore trade-offs between (potentially) competing goals. Figure 4 illustrates some of the standard evaluation measures used for multi-objective reasoning. The rest of this section explains that figure.

Generational distance is the measure of convergence—how close is the predicted Pareto front to the actual Pareto front. It is defined to measure (using Euclidean distance) how far are the solutions that exist in the population P from the nearest solutions in the Actual Pareto front A. In an ideal case, the GD is 0, which means the predicted PF is a subset of the actual PF. Note that it ignores how well the solutions are spread out.

Spread is a measure of diversity—how well the solutions in P are spread. An ideal case is when the solutions in P is spread evenly across the Predicted Pareto Front.

Inverted Generational Distance measures both convergence as well as the diversity of the solutions—it measures the shortest distance from each solution in the Actual PF to the closest solution in Predicted PF. Like Generational distance, the distance is measured in Euclidean space. In an ideal case, IGD is 0, which means the Predicted PF is same as the Actual PF.

The **Hypervolume** indicator is used to measure both convergence as well as the diversity of the solutions—hypervolume is the union of the cuboids w.r.t. to a reference point. Note that the hypervolume implicitly defines an arbitrary aim of optimization. Also, it is not efficiently computable when the number of dimensions is large, however, approximations exist.

3 Claim1: For software engineering tasks, optimization and data mining are very similar

At first glance, the obvious connection between data miners and optimizers is that the former build models from data while the latter can be used to exercise those models (in order to find good choices within a model). Nevertheless, as far as we can tell, these two areas are currently being explored by different research teams. While counter-examples exist, data miners are used by *software analytics* workers and optimizers are used by researchers into *search-based SE* for the most part. As shown in Figure 5, the field we call "DUO" combines software engineering work from both fields.

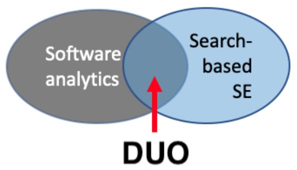


Fig. 5: About DUO.

Search-based software engineering [51, 52] characterizes SE tasks as optimizing for (potentially competing) goals; e.g. designing a product such that it delivers the *most* features at *least* cost [113]. Software analytics [10,79,80], on the other hand, is a workflow that distills large amounts of low-value data into small chunks of very high-value data. For example, software analytics might build a model predicting where defects might be found in source code [47].

For the most part, researchers in these two areas work separately (as witnessed by, say the annual Mining Software Repositories conference and the separate symposium on Search-based Software Engineering). What will be argued in this paper is that insights and methods from these two fields can be usefully combined. We say this because papers that combined data miners and optimizers explore important SE tasks:

– Project management [21,23,24,55,56,69,84,85,93,110,111];
– Requirements engineering [33];
– Software design [21,30];
– Software security [5,107];
– Software quality [1,4,27,41,73,109,117,134];
– Software configuration [23,90,91];
– Text mining of software-related textual artifacts [3,94,94,118].

In our literature review, we have seen four different flavors of this combined DUO approach:

– *Mash-ups of data miners and optimizers:* In this approach, data miners and optimizers can be seen as separate executables. For example, Abdessalem et al. [1] generate test cases for autonomous cars via a cyclic approach where an optimizer reflects on the output of data miners that reflect on the output of an optimizer (and so on).
– *Data miners acting as optimizers:* In this approach, there is no separation between the data miner and optimizer. For example, Chen et al. [21] show that their recursive descent bi-clustering algorithm (which is a data mining technique) outperforms traditional evolutionary algorithms for the purposes of optimizing SE models.
– *Optimizers control the data miners:* In this approach, the data miner is a subroutine called by the optimizer. For example, several recent papers improve predictive performance via optimizers that tune the control parameters of the data miner [4,41,117].
– *Data miners control the optimizers:* In this approach, the optimizer is a subroutine called by the data miner. For example, Majumder et al. [74] use k-means clustering to divide up a complex text mining problem, then apply optimizers within each cluster. They report that this method speeds up their processing by up to three orders of magnitude.

To understand why data mining technology is so useful for optimization, and vice versa, we must dive deeper into the formal underpinnings of work in this area. Without loss of generality, an optimization problem is of the following format [14]:

$$\begin{aligned} \text{minimize } & f_i(\mathbf{a}), i = 1, 2, \cdots, n \\ \text{subject to } & g'_j(\mathbf{a}) \leq 0, j = 1, 2, \cdots, n' \\ & g''_k(\mathbf{a}) = 0, k = 1, 2, \cdots, n'' \end{aligned} \tag{1}$$

where $\mathbf{a} = (a_1, \cdots, a_m) \in \mathbf{A}$ is the optimization variable of the problem,[1] $f_i(\mathbf{a}) : \mathbf{A} \to \mathbb{R}$. are the objective functions (goals) to be minimized, $g'_j(\mathbf{a})$ are inequality constraints, and

[1] This definition has been generalized with respect to [14], not to be restricted to continuous optimization problems, where $\forall a_i,\ a_i \in \mathbb{R}$

$g''_k(\mathbf{a})$ are equality constraints.[2] Sometimes, there are no constraints (so $n' = 0$ and $n'' = 0$). Also, we say a *multi-objective problem* has $n > 1$ objectives, as opposed to a *single-objective problem*, where $n = 1$.

One obvious question about this general definition is “where do the functions f, g', g'' come from?”. Traditionally, these have been built by hand but, as we shall see below, f, g', g'' can be learned via data mining. That is, optimizers can *explore* the functions *proposed* by a learner.

An example of an optimization problem in the area of software engineering is to find a subset **a** of requirements that maximizes value $f(\mathbf{a})$ if implemented[3], given a constrained budget $g'_0(\mathbf{a}) \leq 0$, where $g'_0(\mathbf{a}) = cost(\mathbf{a}) - budget$ [108]. Many different algorithms exist to search for solutions to optimization problems. Table 2 shows the optimization algorithms that have been used by the software engineering community when applying DUO.

Data mining is a problem that involves finding an approximation $\hat{h}(\mathbf{x})$ of a function of the following format:

$$\mathbf{y} = h(\mathbf{x}) \tag{2}$$

where $\mathbf{x} = (x_1, \cdots, x_p) \in \mathbf{X}$ are the input variables, $\mathbf{y} = (y_1, \cdots, y_q) \in \mathbf{Y}$ are the output variables of the function $h(\mathbf{x}) : \mathbf{X} \rightarrow \mathbf{Y}$, $\mathbf{X}$ is the input space and $\mathbf{Y}$ is the output space. The input variables **x** are frequently referred to as the independent variables or input features, whereas **y** are referred to as the dependent variables or output features.

An example of a data mining problem in software engineering is software defect prediction [50]. Here, the input features could be a software component’s size and complexity, and the output feature could be a label identifying the component as defective or non-defective. Many different machine learning algorithms can be used for data mining. Table 3 shows data mining algorithms used by the software engineering community when applying DUO.

The functions $h(\mathbf{x})$ and $\hat{h}(\mathbf{x})$ do *not* necessarily correspond to the optimization functions $f_i(\mathbf{a})$ depicted in Eq. 1. However, the true function $h(\mathbf{x})$ is unknown. Therefore, data mining frequently relies on machine learning algorithms to learn an approximation $\hat{h}(\mathbf{x})$ based on a set $D = \{(x_i, y_i)\}_{i=1}^{|D|}$ of known examples (data points) from $h(\mathbf{x})$. And, learning this approximation typically consists of searching for a function $\hat{h}(\mathbf{x})$ that minimizes the error (or other predictive performance metrics) on examples from D. Therefore, learning such approximation *is* an optimization problem of the following format:

$$\text{minimize } f_i(\mathbf{a}), i = 1, \cdots, n \tag{3}$$

where $\mathbf{a} = (\hat{h}(\mathbf{x}), D)$, and f_i are the predictive performance metrics obtained by $\hat{h}(\mathbf{x})$ on D. The functions $f_i(\mathbf{a})$ depicted in Eq. 3 thus correspond to the functions $f_i(\mathbf{a})$ depicted in Eq. 1. An example of performance metric function would be the mean squared error, defined as follows:

$$f(\hat{h}(\mathbf{x}), D) = \frac{1}{|D|} \sum_{(x_i, y_i) \in D} (y_i - \hat{h}(x_i))^2 \tag{4}$$

As we can see from the above, solving a data mining problem means solving an optimization problem, i.e., optimization and data mining are very similar. Indeed, several popular machine learning algorithms *are* optimization algorithms. For example, gradient descent

[2] The optimization variable is usually identified by the symbol **x**, and the inequality and equality constraints are frequently identified by the symbols g and h in the optimization literature. However, we use the symbols **a**, g and g'' here to avoid confusion with the terminology used in data mining, which is introduced later in this section.

[3] Any maximization problem can be re-written as a minimization problem.

for training artificial neural networks, quadratic programming for training support vectors machines and least squares for training linear regression are optimization algorithms.

From the above, we can already see that optimization is of interest to data mining researchers, even though this connection between the two fields is not always made explicit in software engineering research. More explicit examples of how optimization is relevant to data mining in software engineering include the use of optimization algorithms to tune the parameters of the data mining algorithms, as mentioned at the beginning of this section and further explained in Sections 4 and 6. We consider such more explicitly posed connections between data mining and optimization as a form of DUO.

A typical distinction made between the optimization and data mining fields is data mining's need for generalization. Despite the fact that data mining uses machine learning to search for approximations $\hat{h}(\mathbf{x})$ that minimize the error on a given dataset D, the true intention behind data mining is to search for approximations $\hat{h}(\mathbf{x})$ that minimize the error on unseen data D' from $h(\mathbf{x})$, i.e., being able to generalize. As D' is unavailable for learning, data mining has to rely on a given known data set D to find a good approximation $\hat{h}(\mathbf{x})$. Several strategies can be adopted by machine learning to avoid poor generalization despite the unavailability of D'. For instance, the performance metric functions $f_i(\hat{h}(\mathbf{x}), D)$ may use regularization terms [11], which encourage the parameters that compose $\hat{h}(\mathbf{x})$ to adopt small values, making $\hat{h}(\mathbf{x})$ less complex and thus generalize better. Another strategy is early stopping [11], where the learning process stops early, before finding an optimal solution that minimizes the error on the whole set D.

However, even the distinction above starts to become blurry when considering that generalization can also frequently be of interest to optimization researchers. For example, in software configuration optimization (Section 5), the true optimization functions $f_i(\mathbf{a})$ are often too expensive to compute, requiring machine learning algorithms to learn approximations of such functions. Optimization functions approximated by machine learning algorithms are referred to as surrogate models, and correspond to the approximation $\hat{h}(\mathbf{x})$ of Eq. 3. These approximation functions are then the one optimized, rather than the true underlying optimization function. Even though an optimization algorithm to solve this problem does not attempt to generalize, a data mining technique can do so on its behalf. We consider this as another form of DUO. Sections 4 and 5 explain how several other examples of software engineering problems are indeed both optimization and data mining problems at the same time, and how different forms of DUO can help solving these problems.

Overall, this section shows that optimization and data mining are very similar to each other, and that the typical distinction made between them can become very blurry when considering real world problems. Several software engineering problems require both optimization and generalization at the same time. Therefore, many of the ideas independently developed by the field of optimization are applicable to improve the field of data mining, and vice-versa. Sections 4 to 6 explain how useful DUO can and could be.

4 Claim2: For software engineering tasks, optimizers can greatly improve data miners

One of the most frequent ways to integrate data mining and optimization is via *hyperparameter optimization*. This is the art of tuning the parameters that control the choices within a data miner. While these can be set manually[4] we found that several papers in our literature review

[4] Using a process called "engineering judgement"; i.e. guessing.

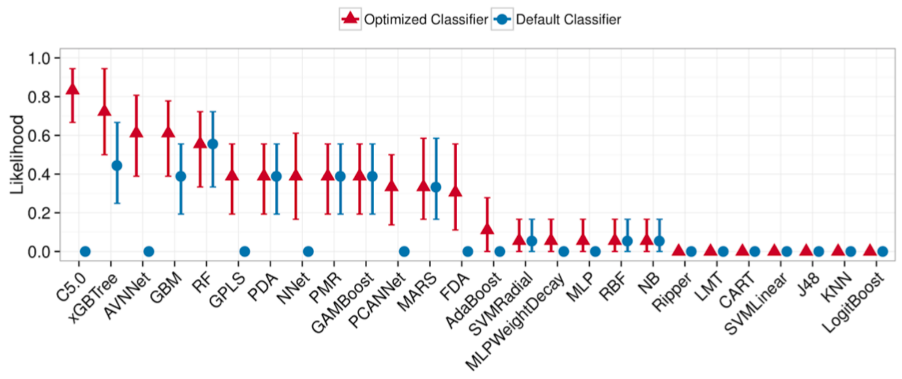


Fig. 6: Effects of hyperparameter optimization on control parameters of a learner from [117]. Blue dots and red triangle show the mean performance before and after tuning (respectively). X-axis shows different learners. Y-axis shows the frequency at which a learner was selected to be "top-ranked" (by a statistical analysis). Vertical lines show the variance of that selection process over repeated runs.

used optimizers to automatically find the best parameters [4, 41, 73, 93, 109, 118, 134]. There are many reasons why this is so:

– These control parameters are many and varied. Even supposedly simple algorithms come with a daunting number of options. For example, the scitkit-learn toolkit lists over a dozen configuration options for Logistic Regression[5]. This is an important point since recent research shows that the *more* settings we add to software, the *harder* it becomes for humans to use that software [130].
– Manually fine-tuning these parameters to obtain the best results for a new data set, is not only tedious, but also can be biased by a human's (mis-)understanding of the inner workings of the data miner.
– The hurdle to implement or apply a "successful" heuristic for automated algorithm tuning is low since (a) the default settings are often not optimal for the situation at hand, and (b) a large number of optimization packages are readily available [31, 102]. Some data mining tools now come with built-in optimizers or tuners; e.g the SMAC implementation built into the latest versions of Weka [48, 58]; or the CARET package in "R" [68].
– Several results report spectacular improvements in the performance of data miners after tuning [4, 41, 73, 109, 117, 118, 134].

As evidence to the last point, we offer two examples. Tantithamthavorn et al. [117] applied the CARET grid search [68] to improve the predictive performance of classifiers. Grid search is an exhaustive search across a pre-defined set of hyperparameter values. It is implemented as a set of for-loops, one for each hyperparameter. Inside the inner-most for-loop, some learner is applied to some data to assess the merits of a particular set of hyperparameters. Based on statistical methods, Tantithamthavorn et al. ranked all the learners in their study to find the top-ranked tuned learner. As shown in Figure 6, there is some variability in the likelihood of being top-ranked (since their analysis was repeated for multiple runs).
From Figure 6 we can see that:

[5] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html, accessed 30 November 2018.

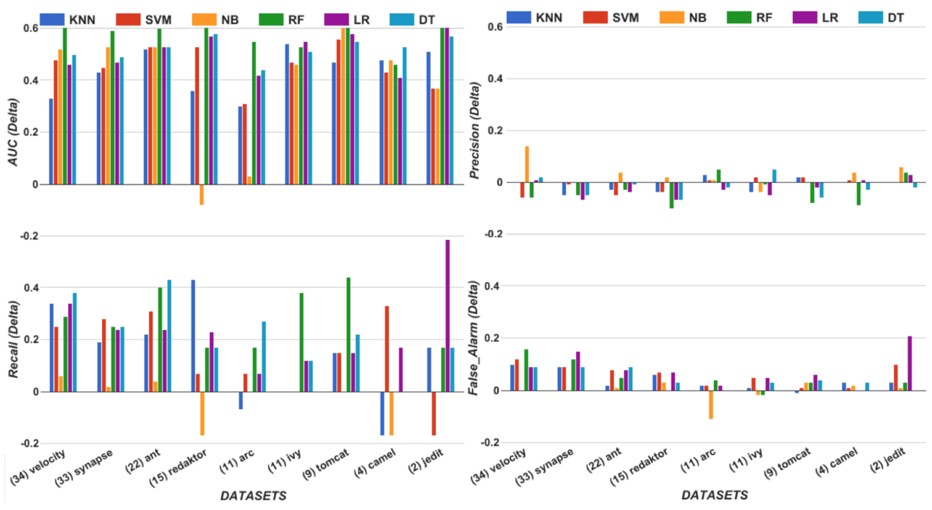


Fig. 7: Effects of hyperparameter optimization on control parameters of a data pre-processor from [4]. Different colored vertical bars refer to different learners: KNN=nearest neighbor; SVM= support vector machine; NB=naive Bayes; RF=random forest; LR=logistic regression; DT=decision trees. X-axis shows different datasets. Y-axis shows the *after - before* values of four different performance scores: recall, precision, false alarm and AUC (area under the false positive vs true positive rate). For false alarms, *better* deltas are *smaller*. For all other measures, the *larger* the *better*.

– Hyperparameter optimization *never makes performance worse*. We say this since the red triangles (tuned results) are never lower than their blue dot counterpart (untuned results).
– Hyperparameter optimization is *associated with some spectacular performance improvements*. Consider the first six left-hand-side blue dots at $Y = 0$. These show all untuned learners that are never top-ranked. After tuning, however, the ranking of these learners is very different. Note once we tune two of these seemingly “bad” learners (C5.0 and AVNNet), they become part of the top three best learners.

For another example of the benefits of hyperparameter optimization, consider Figure 7. This work shown by Agrawal et al. [4] where differential evolution tuned a data pre-processor called SMOTE [19]. SMOTE rebalances training data by discarding items of the majority class and synthesizing artificial members of the minority class. As discussed in Table 2, differential evolution [116] is an evolutionary algorithm that uses a unique mutator that builds new candidates by combining three older candidates. From Figure 7 we can observe the same patterns seen in Figure 6:

– Hyperparameter optimization *rarely makes performance much worse*. There are some losses in precision and false alarms grow slightly. But overall, these changes are very small.
– Hyperparameter optimization is *associated with some spectacular performance improvements*. The improvements in recall can be large and the improvements in AUC are the largest we have ever seen of any method, ever, in the software analytics literature.

Another advantage of hyperparameter optimizers is that it can tune learners such that they succeed on multiple criteria. Standard learners have their goals “hard-wired” (e.g. minimize entropy). This can be a useful design choice since it allows algorithm designers to produce very fast systems that scale to very large data sets. That said, there are many situations where the business users have *multiple* competing goals; e.g. deliver *more* functionality, in *less* time, with *fewer* bugs. For further examples of multiple business goals in software analytics, see Table 23.2 of [76].

While the goals of data miners are often hard-wired, optimizers can accept multiple goals as part of the specification of a problem (see the g terms within Equation 1). This means optimizers can explore a broader range of goals than data miners. For example:

- Minku and Yao [81] used multi-objective evolutionary algorithms to generate neural networks for software effort estimation, with the objectives of minimizing different error metrics.
- Sarro et al. [111] used multi-objective genetic programming for software effort estimation, with the objectives of minimizing the error and maximizing the confidence on the estimates.

An interesting variant of the above is optimization (in the form of reinforcement learning) is model selection over time [82, 83]. Depending on the problem being tackled, the best predictive model to be used for a given problem may change over time, due to changes suffered by the underlying data generating process of this problem. For such problems, model choice has to be continuously performed over time, rather than performed only once, prior to model usage. To deal with that in the context of software effort estimation, Minku and Yao [82, 83] monitor the predictive performance of software effort estimation models created using software effort data from different companies over time. This predictive performance was monitored based on a time-decayed performance metric derived from the reinforcement learning literature, computed over their effort estimations provided for a given company of interest over time. The models whose predictive performances are recently the best are then selected to be emphasized when performing software effort estimations to the company of interest. When combined with transfer learning [82], this strategy enabled a reduction of 90% in the number of within-company effort data required to perform software effort estimation, while maintaining or sometimes even slightly improving predictive performance. This is a significant achievement, given that the cost of collecting within-company effort data is typically very high.

4.1 A Dozen Tips for Using Optimizers

The next section describes some of the problems associated with using optimizers. Before that, this section offers some rules of thumb for software engineers wishing to use *optimizers* in the manner recommended by this paper.

Just to say the obvious, we cannot *prove* the utility of the following heuristics. That said, when we work with our industrial partners or graduate students, we often say the following.

1. Start with a clear and detailed problem formulation. If you do not understand the problem well, then the proposed approach to solve the problem may not deliver what is desired.

2. Visualize first, optimize second. That is, try to visualize the trade-offs between objectives before going into optimization by (e.g.) randomly generating some candidate solutions and plotting the generated performance scores (and for multi-goal reasoning, visualize the principal components of the objective space). We say this since (sometimes) glancing at such

a visualization can lead to insights such as "this problem divides into two separate problems that we should explore separately".

3. Optimizers of the kind explored here are found in many open source toolkits written in various languages (e.g. JAVA, C++, Python) such as jMETAL [31] or PAGMO/PyGMO (esa.github.io/pagmo2).

4. No optimizer works best for all applications [126]. That said, once it can be shown that one optimizer is better than another then the set of better optimizers is exponentially smaller [86]. Hence, when faced with a new problem, it is useful to try several optimizers and stop after two significant improvements have been achieved over some initial baseline result. In terms of algorithms, a useful set to try first are NSGA-2 [28] (since everyone tries this one), MOEA/D [133] (since it is fast), and differential evolution [116] (since it is so simple to use).

5. Further to the last point, any industrial application of this paper should try several optimizers. For that purpose, it is useful to apply faster optimizers (e.g. MOEA/D) before slower methods (e.g. NSGA-II). Note that if your preferred method is very slow, then it can be speeded up via data mining (see point#1).

6. As said in §4, if optimizers run too slowly, use data miners to divide the data then and apply optimization to each segment [74].

7. If your users cannot understand what the optimizer is saying, use data mining to produce a summary of the results.

8. Watch out for changes in the problem over time – they may cause a previous optimal solution to become poor.

9. More specifically, insights can change with the computational budget. That is, conclusions that seem most useful after 1,000 evaluations might be superseded by the results from 5,000 evaluations on. Hence, if possible, before reporting a conclusion to users, try doubling the number of evaluations to see if your current conclusions still hold.

10. True multi-objective formulations are less biased than linear combinations of objectives. We say this since, sometimes, it is suggested to reduce a multi-optimization problem to a simpler single-objective problem by adding "magic weights" to each objective (e.g. "three times the speed of the car plus twice times the cost of the car"). Such "magic weights" introduce an unnecessary bias to the analysis. These magic weights can be avoided by using a true multi-objective algorithm (e.g. NSGA-II, MOEA/D, differential evolution (augmented with Algorithm 1).

11. When optimizing for one or two goals, a simple predicate is enough to select which solution is better (specifically: x is not worse than y on both objectives; and x is better on at least one).

12. But when optimizing for more goals, Zitzler's indicator method [135] might be needed [113]. This indicator method was shown in Algorithm 1.

13. Finally, for multiple goals, sometimes it is useful to focus first on a small number of most difficult goals (then use the results of that first study to "seed" a second study that explores the remaining goals) [112].

4.2 Problems with Hyperparameter Optimization

In summary, optimization is associated with some spectacular improvements in data mining. Also, by applying optimizers to data miners, they can better address the domain-specific and goal-specific queries of different users.

One pragmatic drawback with hyperparameter optimization is its associated runtime. Each time a new hyperparameter setting is evaluated, a learner must be called on some training data, then tested on some separate “hold-out” data. This must be repeated, many times. In practice, this can take a long time to terminate:

– When replicating the Tantithamthavorn et al. [117] experiment, Fu et al. [43] implemented tuning using grid search and differential evolution. That study used 20 repeats for tuning random forests (as the target learner), and optimized four different measures of AUC, recall, precision, false alarm, that grid search required 109 days of CPU. Differential evolution and grid search required 10^4 and 10^7 seconds to terminate, respectively[6].
– Xia et al. [128] reports experiments with hyperparameter optimization for software effort estimation. In their domain, depending on what data set was processed, it took 140 to 700 minutes (median to maximum) to compare seven ways to optimize two data miners. If that experiment is repeated 30 times for statistical validity, then the full experiment would take 70 to 340 hours (median to max).

While the above runtimes might seem practical to some researchers, we note that other hyperparameter optimization tasks take a very long time to terminate. Here are the two worst (i.e. slowest) examples that we know of, seen in the recent SE literature:

– E.g. decades of CPU time were needed by Treude et al. [118] to achieve a 12% improvement over the default settings;
– E.g. 15 years of CPU were needed in the hyperparameter optimization of software clone detectors by Wang et al. [123].

One way to address these slow runtimes is via (say) cloud-based CPU farms. Cross-validation experiments can be easily parallelized just by running each cross-val on a separate core. But the cumulative costs of that approach can be large. For example, recently while developing a half million US dollar research proposal, we estimate how much it would cost to run the same kind of hardware as seen in related work. For that three year project, two graduate students could easily use $100,000 in CPU time – which is a large percentage of the grant (especially since, after university extracts its overhead change, that $500K grant would effectively be $250K).

Since using optimizers for hyperparameter optimization can be very resource-intensive, the next section discusses DUO to significantly reduce that cost.

5 Claim3: For software engineering tasks, data miners can greatly improve optimization

The previous section mentions the benefits of optimizers for data mining, but warned that such optimization can be slow. One way to speed up optimization is to divide the total problem into many small sub-problems. As discussed in this section, this can be done using data mining. That is, data mining can be used to optimize optimizers.

Another benefit of data mining is that, as discussed below, it can generalize and summarize the results of optimization. That is, data mining can make optimization results more comprehensible.

[6] Total time to process 20 repeated runs across multiple subsets of the data, for multiple data sets.

5.1 Faster Optimization

It can be a very simple matter to implement data miners improving optimizers. Consider, for example, Majumder et al. [74] who were looking for the connections between posts to StackOverflow (which is a popular online question and answer forum for programmers):

- An existing deep learning approach [129] to that problem was so slow that it was hard to reproduce prior results.
- Majumder et al. found that they could get equivalent results 500 to 900 times faster (using one core or eight cores working in parallel) just by applying k-means clustering to the data, then running their hyperparameter optimizer (differential evolution) within each cluster.

Another example of data mining significantly improving optimization, consider the *sampling* methods of Chen et al. [21, 23]. This team explored optimizers for a variety of (a) software process models as well as the task of (b) extracting products from a product line description. These are multi-objective problems that struggle to find solutions that (e.g.) minimize development cost while maximizing the delivered functionality (and several other goals as well). The product extracting task was particularly difficult. Product lines were expressed as trees of options with "cross-tree constraints" between sub-trees. These constraints mean that decisions in one sub-tree have to be carefully considered, given their potential effects on decisions in other sub-trees. Formally, this makes the problem NP-hard and in practice, this product extraction process was known to defeat state-of-the-art theorem provers [98], particularly for large product line models (e.g. the "LVAT" product line model of a LINUX kernel contained 6888 variables within a network of 343,944 constraints [21]).

Chen et al. tackled this optimization problem using data mining to look at just a small subset of the most informative examples. Chen et al. call this approach a "sampling" method. Specifically, they used a recursive bi-clustering algorithm over a large initial population to isolate the superior candidates. As shown in the following list, this approach is somewhat different to the more standard genetic algorithms approach:

- Genetic algorithms (in SE) often start with a population of $N = 10^2$ individuals.
- On the other hand, samplers start with a much larger population of $N = 10^4$ individuals.
- Genetic algorithms run for multiple *generations* where useful variations of individuals in generation i are used to seed generation $i + 1$.
- On the other hand, samplers run for a single generation, then terminate.
- Genetic algorithms evaluate all N individuals in all generations.
- On the other hand, the samplers of Chen et al. evaluate pairs of distant points. If one point proves to be inferior then it is pruned along with all individuals in that half of the data. Samplers then recursively prune the surviving half. In this way, samplers only evaluate $O(2\log_2 N)$ of the population,

Regardless of the above differences, the goal of genetic algorithms and samplers is the same: find options that best optimize some competing set of goals. In comparisons with NSGA-II (a widely used genetic algorithm [28]), Chen et al.'s sampler usually optimized the same, or better, as the genetic approach. Further, since samplers only evaluate $O(2\log_2 N)$ individuals, sampling's median cost was just 3% of runtimes and 1% of the number of model evaluations (compared to only running the genetic optimizer) [21].

For another example of data miners speeding up optimizers, see the work of Nair et al. [90]. That work characterized the software configuration optimization problem as ranking a (very large) space of configuration options, without having to run tests on all those options. For example, such configuration optimizers might find a parameter setting to SQLite’s configuration files that maximized query throughput. Testing each configuration requires re-compiling the whole system, then re-running the entire test suite. Hence, testing the three million valid configurations for SQLite is an impractically long process.

The key to quickly exploring such a large space of options, is *surrogate modeling*; i.e. learning an approximation to the response variable being studied. The two most important properties of such surrogates are that they are much faster to evaluate than the actual model, and that the evaluations are precise. Once this approximation is available then configurations can be ranked by generating estimates from the surrogates. Nair et al. built their surrogates using a data miner; specifically, a regression tree learner called CART [16]. An initial tree is built using a few randomly selected configurations. Then, while the error in the tree’s predictions decreases, a few more examples are selected (at random) and evaluated. Nair et al. report that this scheme can build an adequate surrogate for SQLlite after 30-40 evaluated examples.

For this paper, the key point of the Nair et al. work is that this data mining approach scales to much larger problems then what can be handled via standard optimization technology. For example, the prior state-of-the-art result in this area was work by Zuluga et al. [136] who used a Gaussian process model (GPM) to build their surrogate. GPMs have the advantage that they can be queried to find the regions of maximum variance (which can be an insightful region within which to make the next query). However, GPMs have the disadvantage that they do not scale to large models. Nair et al. found that the data mining approach scaled to models orders of magnitude larger than the more standard optimization approach of Zuluga et al.

5.2 Better Comprehension of User Goals

Aside from speeding up optimization, there are other benefits of adding data miners to optimizers. If we combine data miners and optimizers then we can (a) better understand user goals to (b) produce results that are more relevant to our clients.

To understand this point, we first note that modern data miners run so quickly since they are highly optimized to achieve a single goal (e.g. minimize class entropy or variance). But there are many situations where the business users have *multiple* competing goals; e.g. deliver *more* functionality, in *less* time, with *fewer* bugs. A standard data miner (e.g. CART) can be kludged to handle multiple goals reasoning, as follows: compute the class attribute via some *aggregation function* that uses some “magic weights” β, e.g., $class\_value = \beta_1 \times goal1 + \beta_2 \times goal2 + ...$ But using an aggregate function for the class variable is a kludge, for three reasons. Firstly, when users change their preferences about β_i, then the whole inference must be repeated.

Secondly, the β_i goals may be inconsistent and conflicting. A repeated result in decision theory is that user preferences may be nontransitive [34] (e.g. users rank $\beta_1 < \beta_2$ and $\beta_2 < \beta_3$ but also $\beta_3 < \beta_1$). Such intransitivity means that a debate about how to set β to a range of goals may never terminate.

Thirdly, rather than to restrict an inference to the whims of one user, it can be insightful to let an algorithm generate solutions across the space of possible preferences. This approach was used by Veerappa et al. [121] when exploring the requirements of the London Ambulance system. They found that when they optimized those requirements, the result was a *frontier* of hundreds of solutions like that shown in Figure 8. Each member of that frontier is trying to

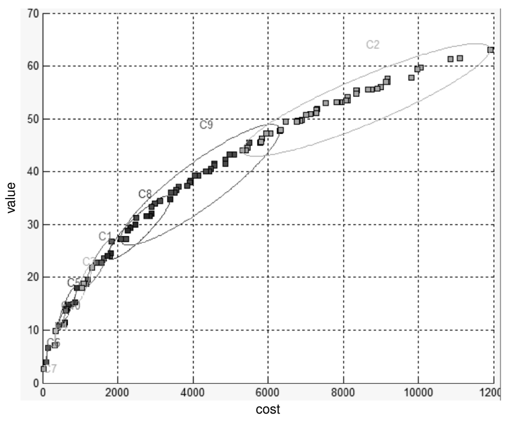


Fig. 8: Clusters of the solution frontier. Frontier generated by reasoning over the goals (in this case, *minimize* cost and *maximize* value). Clusters generated by reasoning over the decisions that lead to those goals. From [121].

push out on all objectives (and perhaps failing on some). Mathematicians and economists call this frontier the *Pareto frontier [95]*. Others call it the *trade-off space* since it allows users to survey the range of compromises (trade-offs) that must be made when struggling to achieve multiple, possibly competing, goals.

Figure 8 is an illustrative example of how data mining can help optimizers. In that figure, the results are grouped by a data miner (a clusterer). The decisions used to reach the centroid of each cluster are a specific example, each with demonstrably different effects. When showing these results to users, Veerappa et al. can say (e.g.) "given all that you've told us, there a less than a dozen kinds of solutions to your problem". That is:

- Step#1: Optimize: here are the possible decisions;
- Step#2: Data mine: here is a summary of those decisions;
- Step#3: Business users need only debate the options in the summary.

Note that Figure 8 lets us comment on the merits of global vs local reasoning. *Global* reasoning might return the average properties of all the black points in that figure. But if we apply *local* reasoning, we can find specialized models within each cluster of Figure 8. The merits of local vs global reasoning are domain-specific but in the specific case of Figure 8, there seem to be some key differences between the upper right and lower left clusters.

Another approach to understanding trade-off space is contrast-rule generation as done by (e.g.) Menzies et al.'s STAR algorithm [75]. STAR divided results into a 10% "best" set and a 90% "rest" set. A Bayesian contrast set procedure ranks all N decisions in descending order (best to worst). STAR then re-runs the optimizer $i \in N$ times, each time pre-asserting the first i-the ranges. For example, suppose 10,000 evaluations lead to $B = 1000$ 'best' solutions and $R = 9000$ rest.

- Let a be "analyst capability" and let "a=high" appear 50 times in best and 90 times in rest.
- Let u be "use of software tools" and let "u=high" appear 100 times in best and 180 times in rest. So $b_{a.high} = 50/B = 50/1000 = 0.05$; $r_{a.high} = 90/R = 90/9000 = 0.01$; $b_{u.high} = 100/1000 = 0.1$; and $r_{u.high} = 180/9000 = 0.02$.

- STAR sorts ranges via $s = b^n/(b + r)$, where n is a constant used to reward ranges with high support in "best". E.g. if $n = 2$ then $s_{a.high} = 0.042$ and $s_{u.high} = 0.083$. That is, STAR thinks that the high use of software tools is more important than high analyst capability.

The output of STAR result is a graph showing the effects of taking the first best decision, the first two best decisions, and so on. In this way, STAR would report to users a succinct rule set advising them what they can do if they are willing to change just one thing, just two things, etc [75].

Before ending this section, we stress the following point: *it can be very simple to add a data miner to an optimizer*. For example:

- STAR's contrast set procedure described above, is very simple to code (around 30 lines of code in Python).
- Recall from the above that Majumder et al. speed up their optimzer by 500 to 900 times, just by prepending a k-means clusterer to an existing optimization process.

5.3 A Dozen Tips for Using Data Mining

This section offers some rules of thumb for software engineers wishing to use *data miners* in the manner recommend by this paper.

As said above, just to say the obvious, we cannot *prove* the utility of the following heuristics. That said, when we work with our industrial partners or graduate students, we often say the following.

1. Data miners of the kind explored here are found in many open source toolkits written in various languages (e.g. JAVA, Python) such a WEKA [48] or Scikit-Learn [96].

2. If your data mining problem has many goals, consider replacing your data mining algorithms with an optimizer.

3. Avoid the use of the off-the-shelf parameters. Instead, use optimizers to select better settings for the local problem. For more on this point, see **Claim4**, below.

4. Check for conclusion stability. Once you make a conclusion, repeat the entire process ten times using a 90% random sample of the data each time. Do not tell business users about effects that are unstable across different samples. This point is particularly relevant for systems that combine data miners with optimizers that make use of any stochastic component.

5. No data miner works best for all applications [125]. Hence, we offer the same advice as with point#4,5 in §4.1. That is, when faced with a new problem, it is useful to try several data miners and stop after two significant improvements have been achieved over some initial baseline result. In terms of what data miners to try first, there is a large candidate list. For software analytics, we refer the reader to table IX of Ghotra et al. [45] that ranks dozens of different data mining algorithms into four "ranks". To sample a wide range of algorithms, we suggest applying one algorithm for each rank.

6. Watch out for the temporal effect of data – it may cause past models to become inadequate.

7. Strive to avoid overfitting. Try to test on data not used in training. If the data has timestamps, train on earlier data and test on later data. If no timestamps, then ten times randomly reorganized the data and divide it into ten bins. Next, make each bin the test set and all the other bins the train set.

8. Ignore spurious distinctions in the data. For example, the Fayyad-Irani discretizer [32] can simplify numeric columns by dividing up regions that best divide up the target class.

9. Ignore spurious columns. If a column ended up being poorly discretized, that is a symptom that that column is uninformative. By pruning the columns with low discretization scores, spurious data can be ignored [49].

10. Ignore spurious rows. Similarly, instance and range pruning can be useful. After discretization and feature selection, numeric ranges can be scored by how well they achieve specific target classes. If we delete rows that have few interesting ranges, we can reduce and simplify any process that visualizes or searches the data [97].

11. If there is very little data, consider asking the model inside the data miner to generate more examples. If that is not practical (e.g. the model is too slow to execute) then try transfer learning [66], active learning [131], or semi-supervised learning.

12. For more advice about using data miners, see "Bad Smells for Software Analytics"' [77].

6 Claim4: For software engineering tasks, data mining without optimization is not recommended

There are many reports in the empirical SE literature where the results of a data miner are used to defend claims such as:

- *"In this domain, the most important concepts are X."* For example, Barua et al. [9] used text mining called to conclude what topics are most discussed at Stack Overflow.
- *"In this domain, learnerX is better than learnerY for building models."* For example, Lessmann et al. [71] reported that the CART decision tree performs much worse than random forests for defect prediction.

All the above results were generated using the default values for CART, random forests and a particular text mining algorithm. We note that these conclusions are now questionable given that tuned learners produce very different results to untuned learners. For example:

- Claims like *"In this domain, the most important concepts are X"* can be changed by applying an optimizer to a data miner. For example, Tables 3 and 8 of [3] show what was found before/after tuned text miners were applied to Stack Overflow data. In many cases, the pre-tuned topics just disappeared after tuning. Also, in defense of the tuned results, we note that, in "order effects experiments", the pre-tuned topics were far more "unstable" than the tuned topics[7].
- Claims like *"In this domain, learnerX is better than learnerY for building models"* can be changed by tuning. One example Fu et al. reversed some of the Lessmann et al. conclusions by showing that tuned CART performs much better than random forest [41]. For another example, recall Figure 6 where, before/after tuning the C5.0 algorithm was the worst/best learner (respectively).

For a small example of this effect (that optimizing a data miner leads to different results), see Figure 9. This figure ranks the importance of different static code features in a defect prediction decision tree. Here "importance" is computed as the (normalized) total reduction of the Gini index for a feature[8]. In this case, tuning significantly improved the performance of the learner (by 16%, measured in terms of the "utopia" index[9]). After tuning:

[7] In "order effects experiments", the training data is re-arranged at random before running the learner again. In such experiments, a result is "unstable" if the learned model changes just by re-ordering the training data.

[8] The Gini index measures class diversity after a set of examples is divided by some criteria – in this case, the values of an attribute.

[9] The distance to of a predictor's performance to the "utopia" point of recall=1, false alarms=0.

Before Tuning		After Tuning	
features	value	features	value
rfc	26	rfc	64
avg_cc	9	moa	10
amc	8	avg_cc	8
lcom3	7	dam	7
cam	7	loc	6
loc	6	mfa	2
cbo	5	cam	2
ce	5	npm	1
npm	5	amc	0
moa	4	ca	0
dam	4	cbm	0
mfa	3	cbo	0
ca	3	ce	0
cbm	2	dit	0
wmc	2	ic	0
lcom	2	lcom	0
max_cc	1	lcom3	0
dit	1	max_cc	0
ic	0	noc	0
noc	0	wmc	0

Fig. 9: Features importance shown for decision tree before and after tuning on jEdit defect prediction where it is optimized for *minimizing* distance to "utopia" (where "utopia" is the point recall=1, false alarms=0).

– Features that seemed irrelevant (before tuning) are found to be very important (after tuning); e.g. moa, dam.
– Many features received much lower ranking (e.g. amc, lcom3, cbo).

Overall, the number of "interesting" features that we might choose to report as "conclusions" in this study is greatly reduced.

Apart from the examples in this section, there are many other examples where (a) hyper-parameter optimization selects models with better performance and (b) those selected models report very different things to alternate models. For example, in §5.1, we saw the following:

– In the Nair et al. [90] case study, the CART decision tree was used to summarize the data seen so far. Such decision trees usually prune away most variables so that any human reading a CART model would see different things than if they read a logistic regression model (where every variable may appear in the logistic equation).
– In the Tantithamthavorn et al. [117] case study of Figure 6, depending on what learner was selected by what optimizer, that analysis would have reported models as a single decision tree, a forest of trees, a set of rules, the probability distributions within a Bayes classifiers, or as an opaque neural net model.
– In the Majumder et al. [74] case study, tuning made us select SVM over a deep learner. Note that that change (from one learner to another) also changes what we would report from that model. SVM models can be reported as the difference between their support vectors (which are few in number). A report of the deep learning model may be much more complicated (since such a summary would require any number of complex transforms, none of which are guaranteed to endorse the same model as the SVM),
– In the Chen et al. [21, 23] case study, tuning used a recursive clustering algorithm (that prune away most of the details in the original model). Such pruning is a very different

approach to that seen in other kinds of optimizers. Hence, any model learned from the Chen et al. methods would be very different to models learned from other optimizers.

Note that the above effect, where the nature of the model generated by a learner is effected by the optimizer attached to that learner, is quite general to all machine learning algorithms. Fürnkranz and Flach characterize learners as "surfing" a landscape of modeling options looking for a "sweet spot" that balances different criteria (e.g. false alarms vs recall). Figure 2 offers an insightful example of this process. In that figure, a learner adds more and more conditions to a model, thereby driving it to different places on the landscape. Depending on the hyperparameters of a learner, that learner will "surf" that space in different ways, terminate at different locations, and return different models.

The important point here is that, in domains where data miners can be optimized very quickly, it would take just a few minutes to hours to refine and improve untuned results. Further, when humans lean in to read what has been learned, these different tunings lead to different kinds of models and hence different kinds of conclusions. Hence, we do not recommend reporting on the models learned via data mining without optimization.

It is worth noting that the use of optimizers to tune the learners' hyperparameters does not mean that we should ignore information on the range of results achieved using different hyperparameter choices. Analyses of sensitivity to hyperparameters are still important, especially considering that the best hyperparameters "right now" may not be the best hyperparameters for "later". Moreover, optimizers have themselves' hyperparameters, which could potentially affect their ability to suggest good hyperparameters to learners. Therefore, it is important to understand how sensitive the learners are to hyperparameter choice.

7 Research Directions

One way to assess any proposed framework such as DUO is as follows: is it sufficient to guide the on-going work of a large community of researchers? As argued in this section, DUO scores very well on this criterion.

Firstly, given that *for software engineering tasks, data mining without optimization should be deprecated*, it is time to revisit and recheck every prior software analytics result based on untuned data miners. This will be a very large task that could consume the time of hundreds of graduate students for at least a decade.

Secondly, given that *for software engineering tasks, data miners can greatly improve optimizers*, there are many research directions:

– *Better explanation and comprehension tools for AI systems:* Use the data miners to summarize complex optimizer output in order to convert opaque inference into something comprehensible. Some methods for that were seen above (Figure 8 and the STAR algorithm of §5.2) but they are just two early prototypes. Adding comprehension tools to AI systems that use optimization is a fertile ground for much future research. For a discussion on criteria to assess comprehensibility in software analytics, see [20, 87].
– *Auditable AI:* There is much recent interest in allowing humans to query AI systems for their biases and, where appropriate, to adjust them. Sampling tools like those of Chen and Nair et al. [21, 23, 91] offer functionality that might be particularly suited for that task. Recall that these tools optimized their models using just a few dozen examples – which is a number small enough to enable human inspection. Perhaps we could build human-in-the-loop systems where humans watch the samplers' explored options – in the field of optimization, the concepts of interactive optimization, dynamic optimization, and

the multi-objective encoding of user preferences is well-established. At the very least, this might allow humans to understand the reasoning. And at the very most, these kinds of tools might allow humans to "reach in" and better guide the reasoning.
- *Faster Deep Learning:* One open and pressing issue in software analytics is the tuning of deep learning networks. Right now, deep learning training is so slow that it is common practice to download pre-tuned networks. This means that deep learning for software analytics may be prone to all the problems we discussed in claims 2 and 4. We gave some ideas here on how data mining can divide up and simplify the deep learning training problem (but, at this time, no definitive results).
- *Avoiding Hyper-hyperparameter Optimization:* While hyperparameter optimization is useful, it should be noted that the default parameters of the hyperparameter optimizers may themselves need optimizing by other optimizers. This is a problem since if hyper-parameter optimization is slow, then hyper-hyperparameter optimization would be even slower. So how can we avoid hyper-hyperparameter optimization?
  - One possible approach is *transfer learning*. Typically when something is tuned, we do so because it will be run multiple times. So instead of search *taula rasa* for good tunings, perhaps it is possible to partially combine parts of old solutions to speed up the search for good hyperparameter values [89].
  - An analogous approach is to select from a portfolio of algorithms. This typically involves the training of machine learning models on performance data of algorithms in combination with instances given as feature data. In software engineering, this has been recently used for the Software Project Scheduling Problem [114, 127]. The field of per-instance configuration has received much attention recently, and we refer the interested reader to a recent updated survey article [63].
  - Another approach is to find shortcuts around the optimization process. For recent work on that, which we would characterize as highly speculative, see [40].

Thirdly, given that *for software engineering tasks, optimizers can greatly improve data miners*, it is time to apply hyperparameter optimization to more data mining tasks within software analytics. As shown by this paper, such an application can lead to dramatically better predictors. Moreover, given that multi-objective perspective that can be given by optimizers, we can use optimizers to enable data mining to explore multiple objectives. For example:

- For software analytics, we could try to learn data miners that find the highest priority bugs after the *fewest* tests, found in the *smallest* methods in code that is *most familiar* to the current human inspector. Such a data miner would return the most important bugs and easiest to fix (thus reducing issue resolution time for important issues).
- For project management, when crowdsourcing large software projects, we could allocate tasks to programmers in order to *minimize* development time while *maximizing* work assignments to programmers that have the most familiarity with that area of the code.
- For refereeing new research results in SE, the tools described here could assign reviewers to new results in order to *minimize* the number of reviews per reviewer while *maximizing* the number of reviewers who work in the domain of that paper.

We could also extend SE to make it about using DUO for providing software engineering support for artificial intelligence systems. For example, using the optimization discussed above, Weise et al. [124] have run over 157 million experiments on over 200 instances of two classical AI combinatorial problems. They found that the naive configuration is a good baseline approach, but with effort, it was possible to outperform it. Friedrich et al. [37, 38] studied a particular family of heuristic hill-climbers problems. Their empirical optimization results sparked extensive theoretical investigations (i.e., proper computational complexity analyses) that showed that the new algorithm configuration is provably faster than what has been state-of-the-art. These can be seen as examples of software engineering to find better configurations. Other work in this area includes Neshat et al. [92] who studied wave energy equations. Their optimization results were translated into well-performing algorithms for large problems. This can be seen as software engineering to improve algorithms' performance.

Lastly, given *that for software engineering tasks, data mining and optimization is essentially the same thing*, it is time to explore engineering principles for creating unified data mining/optimizer toolkits. We already have one exemplar of such a next-generation toolkit: see `www.automl.org` for the work on AutoML and its connections to Weka and scikit-learn. That said, this research area is wide open for experimentation. For two reasons, we would encourage researchers to "roll their own" DUO implementations before automatically turning to tools like AutoML:

– Some initial results suggest it may not be enough to just use AutoML [119] (in summary, given N hyperparameter optimizers, AutoML was "best" for a minority of goals and datasets).
– In terms of training ourselves on how easy it is to combine optimizers and data miners, there is no substitute for "rolling your own" implementation (at least once, then perhaps moving on to tools like AutoML).

8 Related Work

As discussed in §3, this paper is a reflection of two related research areas which, currently, are explored separately by different research teams. It is not the only attempt to do that. At the 2017 NII Shonan Meeting on "Data-Driven Search-Based Software Engineering" [122], three dozen researchers from the software analytics and search-based software engineering communities have met to revisit the 2003 technical note "Reformulating software engineering as a search problem" by Clarke et al. [25]. While some of the attendees discussed the mechanics of search-based methods and how they can help software analytics, most of the discussions focused on how to exploit all the data about software projects that has recently become available (e.g. at online sites like Github, etc) to help search-based software engineering. Such data could form valuable priors. For example:

– Researchers exploring cross-project defect prediction report that lessons learned from one project can now be applied to another [61, 67, 97, 105, 132].
– Researchers exploring language models in SE reported highly repetitive regularities in source code, making that artifact most amenable to prediction of nominal and off-nominal behaviours [6, 53, 59].

This paper differs from the above in that we explore the advantages of new algorithms (optimizers) for software analytics, while much of the discussions at Shonan were about the advantages of new data. Given that, a natural future direction would be to combine both the new algorithms discussed here with the new data sources.

9 Conclusion

For software analytics it is *possible*, *useful* and *recommended* to combine data mining and optimization using DUO. Such combination can lead to better (e.g., faster, more accurate or reliable, more interpretable, and multi-goals) analyses in empirical software engineering studies, in particular those studies that require automated tools for analyzing (large quantities of) data. We support our arguments based on a literature review of applications of DUO.

We say it is *possible* to combine data mining and optimization since data mining and optimization perform essentially the same task (Section 3). Hence, it is hardly surprising that it can be a relatively simple matter to build a unified data mining/optimizer tool. For instance:

– Nair et al.’s approach [91] was just a for-loop around CART.
– STAR [75] was also a very simple learner (see §5.2) wrapped around a simulated annealer, then re-ran the simulated annealer after setting the first i-best ranges.
– Sampling is just a simple bi-recursive clustering algorithm (but see §3.4 of [21] for a discussion on some of the nuances of that process).
– There are many mature open source data mining and optimization frameworks[10]. Also, some of the data mining and optimization algorithms are inherently very simple to implement (e.g. naive Bayes [35], differential evolution [116]). Hence it is easy to implement optimizer+data miner combinations.

As to *usefulness*, this paper has listed several benefits of DUO:

– Data miners can speed up optimizers by dividing large problems into several simpler and smaller ones.
– Also, when optimizers return many example solutions, data miners can generalize and summarize those into a very small set of exemplars (see for example Figure 8) or rules (see for example the STAR algorithm of §5.2).
– Optimization technology lets data miners explore a much broader set of competing goals than just (e.g.) precision, recall, and false alarms. Using those goals, it is possible to better explore an interesting range of SE problems such as project management, requirements engineering, design, security problems, software quality, software configuration, mining textual SE artifacts, just to name a few.

Finally, as to *recommended*, we warn that it can be misleading to report conclusions from an untuned learner since those conclusions can changed by tuning. Since papers that use untuned learners can be so easily refuted, this community should be wary of publishing analytics papers that lack an optimization component.

Having made this case, we acknowledge that DUO would require a paradigm shift for the software analytics community. Graduate subjects would have to be changed (to focus on different kinds of algorithms and case studies); toolkits would need to be reorganized; and journals and conferences will have to adjust their paper selection criteria. Looking into the future, we anticipate several years where DUO is explored by a minority of software analytics researchers. That said, by 2025, we predict that DUO will be standard practice in software analytics.

Acknowledgements

Earlier work ultimately leading to the present one was inspired by the NII Shonan Meeting on Data-Driven Search-based Software Engineering (goo.gl/f8D3EC), December 11-14, 2017. We thank the organizers of that workshop (Markus Wagner, Leandro L. Minku, Ahmed E. Hassan, and John Clark) for their academic leadership and inspiration.

[10] E.g. in Python: scikit-learn and DEAP [96, 102]. E.g. in Java: Weka and (jMetal or SMAC) [31, 48, 58].

Dr Menzies' work was partially supported by NSF grant No. 1703487.

Dr Minku's work was partially supported by EPSRC grant Nos. EP/R006660/1 and EP/R006660/2.

Dr Wagner's work was partially supported by the ARC grant DE160100850.

References

1. Abdessalem, R.B., Nejati, S., Briand, L.C., Stifter, T.: Testing vision-based control systems using learnable evolutionary algorithms. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 1016–1026. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180160
2. Afzal, W., Torkar, R.: On the application of genetic programming for software engineering predictive modeling: A systematic review. Expert Systems with Applications **38**(9), 11,984–11,997 (2011)
3. Agrawal, A., Fu, W., Menzies, T.: What is wrong with topic modeling? and how to fix it using search-based software engineering. Information and Software Technology **98**, 74–88 (2018)
4. Agrawal, A., Menzies, T.: Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In: Proceedings of the 40th International Conference on Software Engineering, pp. 1050–1061. ACM (2018)
5. Ali, M.H., Al Mohammed, B.A.D., Ismail, A., Zolkipli, M.F.: A new intrusion detection system based on fast learning network and particle swarm optimization. IEEE Access **6**, 20,255–20,261 (2018)
6. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) **51**(4), 81 (2018)
7. Anderson-Cook, C.M.: Practical genetic algorithms (2005)
8. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic programming: an introduction, vol. 1. Morgan Kaufmann San Francisco (1998)
9. Barua, A., Thomas, S.W., Hassan, A.E.: What are developers talking about? an analysis of topics and trends in stack overflow. Empirical Software Engineering **19**, 619–654 (2012)
10. Bird, C., Menzies, T., Zimmermann, T. (eds.): The Art and Science of Analyzing Software Data. Morgan Kaufmann, Boston (2015). DOI https://doi.org/10.1016/B978-0-12-411519-4.09996-1
11. Bishop, C.: Pattern Recognition and Machine Learning. Springer (2006)
12. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. Journal of machine Learning research **3**(Jan), 993–1022 (2003)
13. Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., Selby, R.: Cost models for future software life cycle processes: Cocomo 2.0. Annals of software engineering (1995)
14. Boyd, S.P., Vandenberghe, L.: Section 4.1 – optimization problems. In: Convex Optimization. Cambridge University Press (2004)
15. Breiman, L.: Bagging predictors. Machine learning **24**(2), 123–140 (1996)
16. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth and Brooks, Monterey, CA (1984)
17. Catal, C., Diri, B.: Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Information Sciences **179**(8), 1040–1058 (2009)
18. Chand, S., Wagner, M.: Evolutionary many-objective optimization: A quick-start guide. Surveys in Operations Research and Management Science **20**(2), 35 – 42 (2015). DOI https://doi.org/10.1016/j.sorms.2015.08.001
19. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)
20. Chen, D., Fu, W., Krishna, R., Menzies, T.: Applications of psychological science for actionable analytics. In: ESEC/SIGSOFT FSE (2018)
21. Chen, J., Nair, V., Krishna, R., Menzies, T.: "Sampling" as a baseline optimizer for search-based software engineering. IEEE Transactions on Software Engineering (2018)
22. Chen, J., Nair, V., Menzies, T.: Beyond evolutionary algorithms for search-based software engineering. Information and Software Technology (2017)
23. Chen, J., Nair, V., Menzies, T.: Beyond evolutionary algorithms for search-based software engineering. Information and Software Technology **95**, 281–294 (2018)
24. Chiu, N.H., Huang, S.J.: The adjusted analogy-based software effort estimation based on similarity distances. Journal of Systems and Software **80**(4), 628–640 (2007)
25. Clarke, J., Dolado, J.J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., et al.: Reformulating software engineering as a search problem. IEE Proceedings-software **150**(3), 161–175 (2003)

26. Cohen, W.W.: Fast effective rule induction. In: Machine Learning Proceedings 1995, pp. 115–123. Elsevier (1995)
27. De Carvalho, A.B., Pozo, A., Vergilio, S.R.: A symbolic fault-prediction model based on multiobjective particle swarm optimization. Journal of Systems and Software **83**(5), 868–882 (2010)
28. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation **6**(2), 182–197 (2002). DOI 10.1109/4235.996017
29. Deng, L., Yu, D., et al.: Deep learning: methods and applications. Foundations and Trends® in Signal Processing **7**(3–4), 197–387 (2014)
30. Du, X., Yao, X., Ni, Y., Minku, L.L., Ye, P., Xiao, R.: An evolutionary algorithm for performance optimization at software architecture level. In: Evolutionary Computation (CEC), 2015 IEEE Congress on, pp. 2129–2136. IEEE (2015)
31. Durillo, J.J., Nebro, A.J.: jmetal: A java framework for multi-objective optimization. Advances in Engineering Software **42**, 760–771 (2011). DOI DOI:10.1016/j.advengsoft.2011.05.014
32. Fayyad, U., Irani, K.: Multi-interval discretization of continuous-valued attributes for classification learning (1993)
33. Feather, M.S., Menzies, T.: Converging on the optimal attainment of requirements. In: Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on, pp. 263–270. IEEE (2002)
34. Fishburn, P.C.: Nontransitive preferences in decision theory. Journal of Risk and Uncertainty **4**(2), 113–134 (1991). DOI 10.1007/BF00056121
35. Frank, E., Trigg, L., Holmes, G., Witten, I.H.: Technical note: Naive bayes for regression. Machine Learning **41**(1), 5–25 (2000). DOI 10.1023/A:1007670802811
36. Freund, Y., Schapire, R.E., et al.: Experiments with a new boosting algorithm. In: Icml, vol. 96, pp. 148–156. Citeseer (1996)
37. Friedrich, T., Göbel, A., Quinzan, F., Wagner, M.: Heavy-tailed mutation operators in single-objective combinatorial optimization. In: A. Auger, C.M. Fonseca, N. Lourenço, P. Machado, L. Paquete, D. Whitley (eds.) Parallel Problem Solving from Nature – PPSN XV, pp. 134–145. Springer International Publishing, Cham (2018)
38. Friedrich, T., Quinzan, F., Wagner, M.: Escaping large deceptive basins of attraction with heavy-tailed mutation operators. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '18, pp. 293–300. ACM, New York, NY, USA (2018). DOI 10.1145/3205455.3205515
39. Fu, W., Menzies, T.: Easy over hard: A case study on deep learning. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 49–60. ACM (2017)
40. Fu, W., Menzies, T., Chen, D., Agrawal, A.: Building better quality predictors using" $\epsilon - dominance$". arXiv preprint arXiv:1803.04608 (2018)
41. Fu, W., Menzies, T., Shen, X.: Tuning for software analytics: Is it really necessary? Information and Software Technology **76**, 135–146 (2016)
42. Fu, W., Menzies, T., Shen, X.: Tuning for software analytics: Is it really necessary? Information and Software Technology **76**, 135–146 (2016)
43. Fu, W., Nair, V., Menzies, T.: Why is differential evolution better than grid search for tuning defect predictors? arXiv preprint arXiv:1609.02613 (2016)
44. van Gerven, M., Bohte, S.: Artificial neural networks as models of neural information processing. Frontiers Media SA (2018)
45. Ghotra, B., McIntosh, S., Hassan, A.E.: Revisiting the impact of classification techniques on the performance of defect prediction models. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 789–800 (2015)
46. Glover, F., Laguna, M.: Tabu search. In: Handbook of combinatorial optimization, pp. 2093–2229. Springer (1998)
47. Gondra, I.: Applying machine learning to software fault-proneness prediction. Journal of Systems and Software **81**(2), 186–195 (2008)
48. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. **11**(1), 10–18 (2009). DOI 10.1145/1656274.1656278
49. Hall, M.A., Holmes, G.: Benchmarking attribute selection techniques for discrete class data mining. IEEE Transactions on Knowledge and Data Engineering **15**(6), 1437–1447 (2003)
50. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering **38**(6), 1276–1304 (2012)
51. Harman, M., Jones, B.F.: Search-based software engineering. Information and software Technology **43**(14), 833–839 (2001)
52. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR) **45**(1), 11 (2012)

53. Hellendoorn, V.J., Devanbu, P.T., Alipour, M.A.: On the naturalness of proofs. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 724–728. ACM (2018)
54. Henard, C., Papadakis, M., Harman, M., Le Traon, Y.: Combining multi-objective search and constraint solving for configuring large software product lines. In: International Conference on Software Engineering (2015)
55. Huang, S.J., Chiu, N.H.: Optimization of analogy weights by genetic algorithm for software effort estimation. Information and software technology **48**(11), 1034–1045 (2006)
56. Huang, S.J., Chiu, N.H., Chen, L.W.: Integration of the grey relational analysis with genetic algorithm for software effort estimation. European Journal of Operational Research **188**(3), 898–909 (2008)
57. Huang, V.L., Suganthan, P.N., Qin, A.K., Baskar, S.: Multiobjective differential evolution with external archive and harmonic distance-based diversity measure. School of Electrical and Electronic Engineering Nanyang, Technological University Technical Report (2005)
58. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: International Conference on Learning and Intelligent Optimization, pp. 507–523. Springer (2011)
59. Jensen, I.H.: Naturalness of software: Science and applications, by prem devanbu (2019)
60. Jolliffe, I.: Principal component analysis. In: International encyclopedia of statistical science, pp. 1094–1096. Springer (2011)
61. Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E.: Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering **21**(5), 2072–2106 (2016)
62. Kessentini, M., Ruhe, G.: A guest editorial: special section on search-based software engineering. Empirical Software Engineering **21**(6), 2456–2458 (2016). DOI 10.1007/s10664-016-9474-0
63. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Data Mining and Constraint Programming, pp. 149–190. Springer (2016)
64. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. Statistics and computing **4**(2), 87–112 (1994)
65. Krall, J., Menzies, T., Davies, M.: Gale: Geometric active learning for search-based software engineering. IEEE Transactions on Software Engineering **41**(10), 1001–1018 (2015)
66. Krishna, R., Menzies, T.: Bellwethers: A baseline method for transfer learning. IEEE Transactions on Software Engineering pp. 1–1 (2018)
67. Krishna, R., Menzies, T.: Bellwethers: A baseline method for transfer learning. IEEE Transactions on Software Engineering (2018)
68. Kuhn, M.: Building predictive models in r using the caret package. Journal of Statistical Software, Articles **28**(5), 1–26 (2008). DOI 10.18637/jss.v028.i05
69. Kumar, K.V., Ravi, V., Carr, M., Kiran, N.R.: Software development cost estimation using wavelet neural networks. Journal of Systems and Software **81**(11), 1853–1867 (2008)
70. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: International conference on computer aided verification, pp. 585–591. Springer (2011)
71. Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering **34**(4), 485–496 (2008). DOI 10.1109/TSE.2008.35
72. Liaw, A., Wiener, M., et al.: Classification and regression by randomforest. R news **2**(3), 18–22 (2002)
73. Liu, Y., Khoshgoftaar, T.M., Seliya, N.: Evolutionary optimization of software quality modeling with multiple repositories. IEEE Transactions on Software Engineering **36**(6), 852–864 (2010)
74. Majumder, S., Balaji, N., Brey, K., Fu, W., Menzies, T.: 500+ times faster than deep learning (a case study exploring faster methods for text mining stackoverflow). arXiv preprint arXiv:1802.05319 (2018)
75. Menzies, T., Elrawas, O., Hihn, J., Feather, M., Madachy, R., Boehm, B.: The business case for automated software engineering. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 303–312. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321676
76. Menzies, T., Kocagüneli, E., Minku, L., Peters, F., Turhan, B.: Data science for software engineering: Sharing data and models (2013)
77. Menzies, T., Shepperd, M.: 'bad smells' in software analytics papers. Information and Software Technology **112**, 35 – 47 (2019)
78. Menzies, T., Williams, L., Zimmermann, T.: Perspectives on Data Science for Software Engineering. Morgan Kaufmann, Boston (2016)
79. Menzies, T., Zimmermann, T.: Software analytics: so what? IEEE Software **4**, 31–37 (2013)
80. Menzies, T., Zimmermann, T.: Software analytics: Whats next? IEEE Software **35**(5), 64–70 (2018). DOI 10.1109/MS.2018.290111035

81. Minku, L., Yao, X.: Software effort estimation as a multi-objective learning problem. ACM Transactions on Software Engineering and Methodology **22**(4) (2013)
82. Minku, L., Yao, X.: How to make best use of cross-company data in software effort estimation? In: ICSE, pp. 446–456. Hyderabad (2014)
83. Minku, L., Yao, X.: Which models of the past are relevant to the present? a software effort estimation approach to exploiting useful past models. Automated Software Engineering Journal **24**(7), 499–542 (2017)
84. Minku, L.L., Yao, X.: An analysis of multi-objective evolutionary algorithms for training ensemble models based on different performance measures in software effort estimation. In: Proceedings of the 9th international conference on predictive models in software engineering, p. 8. ACM (2013)
85. Minku, L.L., Yao, X.: Software effort estimation as a multiobjective learning problem. ACM Transactions on Software Engineering and Methodology (TOSEM) **22**(4), 35 (2013)
86. Montaez, G.D.: Bounding the number of favorable functions in stochastic search. In: 2013 IEEE Congress on Evolutionary Computation, pp. 3019–3026 (2013). DOI 10.1109/CEC.2013.6557937
87. Mori, T., Uchihira, N.: Balancing the trade-off between accuracy and interpretability in software defect prediction. Empirical Software Engineering (2018). DOI 10.1007/s10664-018-9638-1
88. Nair, V., Agrawal, A., Chen, J., Fu, W., Mathew, G., Menzies, T., Minku, L., Wagner, M., Yu, Z.: Data-driven search-based software engineering. In: Proceedings of the 15th International Conference on Mining Software Repositories, MSR ’18, pp. 341–352. ACM, New York, NY, USA (2018). DOI 10.1145/3196398.3196442
89. Nair, V., Krishna, R., Menzies, T., Jamshidi, P.: Transfer learning with bellwethers to find good configurations. CoRR **abs/1803.03900** (2018)
90. Nair, V., Menzies, T., Siegmund, N., Apel, S.: Using bad learners to find good configurations. arXiv preprint arXiv:1702.05701 (2017)
91. Nair, V., Yu, Z., Menzies, T., Siegmund, N., Apel, S.: Finding faster configurations using Flash. arXiv preprint arXiv:1801.02175 (2018)
92. Neshat, M., Alexander, B., Wagner, M., Xia, Y.: A detailed comparison of meta-heuristic methods for optimising wave energy converter placements. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO ’18, pp. 1318–1325. ACM, New York, NY, USA (2018). DOI 10.1145/3205455.3205492
93. Oliveira, A.L., Braga, P.L., Lima, R.M., Cornélio, M.L.: GA-based method for feature selection and parameters optimization for machine learning regression applied to software effort estimation. information and Software Technology **52**(11), 1155–1166 (2010)
94. Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A.: How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 522–531. IEEE Press (2013)
95. Pareto, V.: Manuale di economia politica, vol. 13. Societa Editrice (1906)
96. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
97. Peters, F., Menzies, T., Layman, L.: Lace2: Better privacy-preserving data sharing for cross project defect prediction. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 801–811. IEEE (2015)
98. Pohl, R., Lauenroth, K., Pohl, K.: A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 313–322 (2011). DOI 10.1109/ASE.2011.6100068
99. Poli, R., Kennedy, J., Blackwell, T.: Particle swarm optimization. Swarm intelligence **1**(1), 33–57 (2007)
100. Polikar, R.: Ensemble based systems in decision making. IEEE Circuits and systems magazine **6**(3), 21–45 (2006)
101. Quinlan, J.R.: Learning with continuous classes. In: Proceedings AI’92, pp. 343–348. World Scientific (1992)
102. Rainville, D., Fortin, F.A., Gardner, M.A., Parizeau, M., Gagné, C., et al.: Deap: A python framework for evolutionary algorithms. In: Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, pp. 85–92. ACM (2012)
103. Riffenburgh, R.H.: Linear discriminant analysis. Ph.D. thesis, Virginia Polytechnic Institute (1957)
104. Roweis, S.T., Saul, L.K.: Nonlinear dimensionality reduction by locally linear embedding. science **290**(5500), 2323–2326 (2000)
105. Ryu, D., Choi, O., Baik, J.: Value-cognitive boosting with a support vector machine for cross-project defect prediction. Empirical Software Engineering **21**(1), 43–71 (2016)

106. Saber, T., Brevet, D., Botterweck, G., Ventresque, A.: Is seeding a good strategy in multi-objective feature selection when feature models evolve? Information and Software Technology (2017)
107. Sadiq, A.S., Alkazemi, B., Mirjalili, S., Ahmed, N., Khan, S., Ali, I., Pathan, A.S.K., Ghafoor, K.Z.: An efficient ids using hybrid magnetic swarm optimization in wanets. IEEE Access **6**, 29,041–29,053 (2018)
108. del Sagrado, J., ÁAguila, I.M., Orellana, F.J.: Requirements interaction in the next release problem. In: Proceedings of the 13th annual conference companion on Genetic and evolutionary computation, pp. 241–242. ACM (2011)
109. Sarro, F., Di Martino, S., Ferrucci, F., Gravino, C.: A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In: Proceedings of the 27th annual ACM symposium on applied computing, pp. 1215–1220. ACM (2012)
110. Sarro, F., Ferrucci, F., Gravino, C.: Single and multi objective genetic programming for software development effort estimation. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, pp. 1221–1226. ACM, New York, NY, USA (2012). DOI 10.1145/2245276.2231968
111. Sarro, F., Petrozziello, A., Harman, M.: Multi-objective software effort estimation. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, pp. 619–630. IEEE (2016)
112. Sayyad, A.S., Ingram, J., Menzies, T., Ammar, H.: Scalable product line configuration: A straw to break the camel's back. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 465–474 (2013)
113. Sayyad, A.S., Menzies, T., Ammar, H.: On the value of user preferences in search-based software engineering: a case study in software product lines. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 492–501. IEEE Press (2013)
114. Shen, X.N., Minku, L.L., Marturi, N., Guo, Y.N., Han, Y.: A q-learning-based memetic algorithm for multi-objective dynamic software project scheduling. Information Sciences **428**, 1 – 29 (2018). DOI https://doi.org/10.1016/j.ins.2017.10.041
115. Steinwart, I., Christmann, A.: Support vector machines. Springer Science & Business Media (2008)
116. Storn, R., Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. J. of Global Optimization **11**(4), 341–359 (1997). DOI 10.1023/A:1008202821328
117. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: Automated parameter optimization of classification techniques for defect prediction models. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, pp. 321–332. IEEE (2016)
118. Treude, C., Wagner, M.: Predicting good configurations for github and stack overflow topic models. In: Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19, pp. 84–95. IEEE Press, Piscataway, NJ, USA (2019). DOI 10.1109/MSR.2019.00022
119. Tu, H., Nair, V.: Is one hyperparameter optimizer enough? In: SWAN 2018 (2018)
120. Vandecruys, O., Martens, D., Baesens, B., Mues, C., De Backer, M., Haesen, R.: Mining software repositories for comprehensible software fault prediction models. Journal of Systems and software **81**(5), 823–839 (2008)
121. Veerappa, V., Letier, E.: Understanding clusters of optimal solutions in multi-objective decision problems. In: 2011 IEEE 19th International Requirements Engineering Conference, pp. 89–98 (2011). DOI 10.1109/RE.2011.6051654
122. Wagner, M., Minku, L., Hassan, A.E., Clark, J.: NII Shonan Meeting #2017-19: Data-driven search-based software engineering. Available online at `https://shonan.nii.ac.jp/docs/No-105.pdf`. Tech. Rep. 2017-19, NII Shonan Meeting Report (2017)
123. Wang, T., Harman, M., Jia, Y., Krinke, J.: Searching for better configurations: a rigorous approach to clone evaluation. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 455–465. ACM (2013)
124. Weise, T., Wu, Z., Wagner, M.: An improved generic bet-and-run strategy for speeding up stochastic local search. CoRR **abs/1806.08984** (2018). Accepted for publication at AAAI 2019.
125. Wolpert, D.H.: The lack of a priori distinctions between learning algorithms. Neural Computation **8**(7), 1341–1390 (1996). DOI 10.1162/neco.1996.8.7.1341
126. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1), 67–82 (1997). DOI 10.1109/4235.585893
127. Wu, X., Consoli, P., Minku, L., Ochoa, G., Yao, X.: An evolutionary hyper-heuristic for the software project scheduling problem. In: J. Handl, E. Hart, P.R. Lewis, M. López-Ibáñez, G. Ochoa, B. Paechter (eds.) Parallel Problem Solving from Nature – PPSN XIV, pp. 37–47. Springer, Cham (2016)
128. Xia, T., Krishna, R., Chen, J., Mathew, G., Shen, X., Menzies, T.: Hyperparameter optimization for effort estimation. CoRR **abs/1805.00336** (2018)
129. Xu, B., Ye, D., Xing, Z., Xia, X., Chen, G., Li, S.: Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 51–62 (2016)

130. Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., Talwadker, R.: Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 307–319. ACM, New York, NY, USA (2015). DOI 10.1145/2786805.2786852
131. Yu, Z., Kraft, N.A., Menzies, T.: Finding better active learners for faster literature reviews. Empirical Software Engineering **23**(6), 3161–3186 (2018)
132. Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E.: Cross-project defect prediction using a connectivity-based unsupervised classifier. In: Proceedings of the 38th International Conference on Software Engineering, pp. 309–320. ACM (2016)
133. Zhang, Q., Li, H.: Moea/d: A multiobjective evolutionary algorithm based on decomposition. IEEE Transactions on evolutionary computation **11**(6), 712–731 (2007)
134. Zhong, S., Khoshgoftaar, T.M., Seliya, N.: Analyzing software measurement data with clustering techniques. IEEE Intelligent Systems **19**(2), 20–27 (2004)
135. Zitzler, E., Künzli, S.: Indicator-based selection in multiobjective search. In: PPSN (2004)
136. Zuluaga, M., Krause, A., Sergent, G., Püschel, M.: Active learning for multi-objective optimization. In: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML’13, pp. I–462–I–470. JMLR.org (2013)