Differential Evolution and FLASH for Software Effort Estimation

Foundations of Software Science Final Project Report

Tianpei Xia
North Carolina State University
txia4@ncsu.edu

ABSTRACT

Software analytics has been widely used in software engineering for many tasks such as generating effort estimates for software projects. One of the "black arts" of software analytics is tuning the parameters controlling predicting algorithms. Such hyperparameter optimization has been widely studied in some software analytics domains (e.g. defect prediction and text mining). But, so far, has not been extensively explored for effort estimation. Accordingly, I try to seek simple, automatic, effective and fast methods for finding hyperparameter options for automatic software effort estimation.

I introduce a hyperparameter optimization architecture called RATE (Rapid Automatic Tuning for Effort-estimation), a novel configuration tool for effort estimation based on the combination of regression tree learners (e.g. CART) and optimizers (e.g. Differential Evolution). I test RATE on a wide range of hyperparameter optimizers using data from 945 software projects, and compare my work against multiple baseline methods for software effort estimation including Automatically Transformed Linear Model (ATLM) and LP4EE. Results show that after tuning, large improvements in effort estimation accuracy were observed (measured in terms of the magnitude of the relative error (MRE) and standardized accuracy (SA)) with relatively very small computational cost. From those results, I recommend to use RATE as a standard component of software effort estimation in the future.

An important part of this analysis is its reproducibility and refutability. All my scripts and data are on-line. It is hoped that this paper will prompt and enable much more research on better methods to tune software effort estimators.

KEYWORDS

Effort Estimation, Evolutionary Algorithms, Bayesian Optimization

1 INTRODUCTION

Estimating the effort cost of a software project is one of the most important tasks in software project management. Accurate cost estimates are critical to software development since they can be used for proposal request, contract negotiations, scheduling, monitoring and control. With one or more wrong factors, the effort estimate results could be inaccurate which affect the allocated funds for the projects [36]. In general, both over and underestimations of cost can cause serious issues. Underestimating the effort costs may result in management approving proposed systems that then exceed their budgets, with underdeveloped functions and poor quality, and failure to complete the project on time. On the other hand, overestimations may result in many extra resources wasted to the projects and which can lead the software developers to loss in contract bidding. Unfortunately, although researchers in the software engineering community keep proposing new models to achieve effort prediction accuracy, it's hard to find effort estimation model that can be consistently successful at predicting software project effort in all situations [85].

There are a number of estimating software effort cost methods available for software developers to predict effort and test effort required for software development, these approaches basically fall into different categories: parametric models, which are derived from the statistical and/or numerical analysis of historical project data, and machine learning models, which are based on a set of artificial intelligence techniques such as artificial neural networks, genetic algorithms, analogy based or case based reasoning, decision trees, and genetic programming, etc. Effort estimators based on machine learning approaches such as multilayer perceptrons, radial basis function networks and regression trees [8, 17, 30, 77, 87] have been receiving increased attention [34], as they can model the complex relationship between effort and software attributes, especially when this relationship is not linear and does not seem to have any predetermined form. However, based on Wolpert's no-free lunch theorems, for machine learning algorithms [88], despite the large number of empirical studies, inconsistent results have been reported repeatedly regarding the estimation accuracy of these machine learning models, no single method works best on all datasets.

In this paper, I explore methods to improve algorithms for software effort estimation since the procedures of effort estimations can be wildly inaccurate [36]. Menzies et al. conclude that effort estimations need to be accurate since many government organizations demand that the budgets allocated to large publicly funded projects be double-checked by some estimation model [53]. In addition, non-algorithm methods, like techniques that rely on human judgment [33] are much harder to audit or dispute (e.g., when the estimate is generated by a senior colleague but disputed by others), which may not be ideal for effective use compared to those machine learning methods.

Minku et al. explain that the way used to choose parameter settings is frequently omitted from the experimental framework reported in software effort estimation papers [54], which thus seem to make an implicit assumption that parameter settings would not change the outcomes of the algorithms significantly. Nevertheless, it is not known to what extent different parameter settings affect the performance of several of the machine learning approaches that have been used for software effort estimation. Such knowledge could be very useful for the software effort estimation community,

Foundations of Software Science, Fall 2018, Computer Science

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$9999.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

as it could guide the choice of machine learning approaches for software effort estimation. *Hyperparameter optimizers* tuning the control parameters of a machine learning algorithm. It is well established that classification tasks like software defect prediction or text classification are improved by such tuning [1, 2, 24, 82] but not widely explored in effort estimation field yet. In this paper, I investigate hyperparameter optimization using effort data from 945 projects in SeaCraft repository.

Overall, the contributions of this paper are:

- A demonstration that default settings are not the best way to perform effort estimation. Hence, when new data is encountered, some tuning process is required to learn the best settings for generating estimates from that data.
- A recognition of the inherent difficulty associated with effort estimation. Since there is not one universally best effort estimation method, commisioning a new effort estimator requires extensive testing. As shown below, this can take hours to days to weeks of CPU time.
- A new criteria for assessing effort estimators. Given that inherent cost of commissioning an effort estimator, it is now important to assess effort estimation methods not only on their predictive accuracy, but also on the time required to generate those estimates.
- The identification of a combination of learner and optimizer that works as well as anything else, and which takes just an hour or two to learn an effort estimator.
- An extensible open-source architecture called RATE that enables the commissioning of effort estimation methods. RATE makes my results repeatable and refutable.

The rest of this paper is structured as follows. The next two sections discuss different methods for effort estimation and how to optimize the parameters of effort estimation methods, with a related work about hyperparameter optimization. Then introduce RATE (short for *rapid automatic turning for effort-estimation*), a CPU-lite search-based SE method based on differential evolution [79] and FLASH [59]. This is followed by a description of my data, my experimental methods, my results, and a discussion section explores open issues/future work with this work. The results from this study let me comment on four research questions:

- **RQ1: Is it best to just use "off-the-shelf" defaults?** I will find that tuned learners provide better estimates than untuned learners. Hence, for effort estimation, "off-the-shelf" defaults should be deprecated.
- **RQ2: Can I replace the old defaults with new defaults?** This checks if I can run tuning only once then use those new defaults ever after. I will observe that effort estimation tunings differ extensively from dataset to dataset. Hence, for effort estimation, there are no "best" default settings.
- **RQ3: Can I avoid slow hyperparameter optimization?** The answer will be "yes" since my results show that for effort estimation: Overall, my slowest optimizers perform no better than certain faster ones.
- **RQ4: What hyperparameter optimizers to use for effort estimation?** Here, I report that a certain combination of learners and optimizers usually produce best results. Further, this particular combination often achieves in a few hours what other optimizers need days to weeks of CPU to achieve. Hence I will recommend the following combination for effort estimation: For new datasets, try a combination of *CART* with the optimizers *differential evolution* and *flash*.

Note that RATE and all the data used in this study is freely available for download from https://github.com/ai-se/magic101

2 BACKGROUND

Software effort estimation is the process of predicting the most realistic amount of human effort (usually expressed in terms of hours, days or months of human work) required to plan, design and develop a software project based on the information collected in previous related software projects. It is important to allocate resources properly in software projects to avoid waste. In some cases, inadequate or overfull funding can cause a considerable waste of resource and time. For example, NASA canceled its Check-out Launch Control System project after the initial $200M estimate was exceeded by another $200M [16]. As shown below, effort estimation can be categorized into different methods, including algorithm-based methods [41, 71].

2.1 Algorithm-based Methods

There are many algorithmic estimation methods. Some, such as COCOMO [11], making assumptions about the attributes in the model. For example, COCOMO requires that data includes 22 specific attributes such as analyst capability (acap) and software complexity (cplx). This attribute assumptions restricts how much data is available for studies like this paper. For example, here I explore 945 projects expressed using a wide range of attributes. If I used COCOMO, I could only have accessed an order of magnitude fewer projects.

Due to its attribute assumptions, this paper does not study COCOMO data. All the following learners can accept projects described using any attributes, just as long as one of those is some measure of project development effort.

Whigham et al.'s ATLM method [86] is a multiple linear regression model which calculate the effort as $effort = \beta_0 + \sum_i \beta_i \times a_i + \varepsilon_i$, where a_i are explanatory attributes and ε_i are errors to the actual value. The prediction weights β_i are determined using least square error estimation [61]. Additionally, transformations are applied on the attributes to further minimize the error in the model. In case of categorical attributes the standard approach of "dummy variables" [29] is applied. While, for continuous attributes, transformations such as logarithmic, square root, or no transformation is employed such that the skewness of the attribute is minimum. It should be noted that, ATLM does not consider relatively complex techniques like using model residuals, box transformations or step-wise regression (which are standard) when developing a linear regression model. The authors make this decision since they intend ATLM to be a simple baseline model rather than the "best" model.

Sarro et al. proposed a method named Linear Programming for Effort Estimation (LP4EE) [67], which aims to achieve the best outcome from a mathematical model with a linear objective function

Table 1: CART's parameters.

Parameter	Default	Tuning Range	Notes
max_feature	None	[0.01, 1]	The number of feature to consider when looking for the best split.
max_depth	None	[1, 12]	The maximum depth of the tree.
min_sample_split	2	[0, 20]	The minimum number of samples required to split an internal node.
min_samples_leaf	1	[1, 12]	The minimum number of samples required to be at a leaf node.

subject to linear equality and inequality constraints. The feasible region is given by the intersection of the constraints and the Simplex (linear programming algorithm) is able to find a point in the polyhedron where the function has the smallest error in polynomial time. In effort estimation problem, this model minimises the Sum of Absolute Residual (SAR), when a new project is presented to the model, LP4EE predicts the effort as $effort = a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n$, where x is the value of given project feature and a is the corresponding coefficient evaluated by linear programming. LP4EE is suggested to be used as another baseline model for effort estimation since it provides similar or more accurate estimates than ATLM and is much less sensitive than ATLM to multiple data splits and different cross-validation methods.

Another kind of algorithm-based estimation method are regression trees such as CART [47]. CART is a tree learner that divides a dataset, then recurses on each split. If data contains more than *min_sample_split*, then a split is attempted. On the other hand, if a split contains no more than *min_samples_leaf*, then the recursion stops. CART finds the attributes whose ranges contain rows with least variance in the number of defects. If an attribute ranges r_i is found in n_i rows each with an effort variance of v_i, then CART seeks the attribute with a split that most minimizes $\sum_i \left(\sqrt{v_i} \times n_i/(\sum_i n_i)\right)$. For more details on the CART parameters, see Table 1.

Yet another algorithm-based estimator are the analogy-based Estimation (ABE) methods advocted by Shepperd and Schofield [74]. ABE is widely-used [31, 40, 42, 53, 63], in many forms. I say that "ABE0" is the standard form seen in the literature and "ABEN" are the 6,000+ variants of ABE defined below. The general form of ABE (which applies to ABE0 or ABEN) is to first form a table of rows of past projects. The *columns* of this table are composed of independent variables (the *features* that define projects) and one dependent *feature* (project effort). From this table, I learn what similar projects (analogies) to use from the training set when examining a new test instance. For each test instance, ABE then selects k analogies out of the training set. Analogies are selected via a *similarity measure*. Before calculating similarity, ABE normalizes numerics min..max to 0..1 (so all numerics get equal chance to influence the dependent). Then, ABE uses *feature* weighting to reduce the influence of less informative *features*. Finally, some *adaption strategy* is applied return a combination of the dependent effort values seen in the k nearest analogies. For details on ABE0 and ABEN, see Figure 1 & Table 2.

Table 2: Variations on analogy. Visualized in Figure 1.

- To measure similarity between x, y, ABE uses $\sqrt{\sum_{i=1}^{n} w_i(x_i - y_i)^2}$ where w_i corresponds to *feature* weights applied to independent *features*. ABE0 uses a uniform weighting where $w_i = 1$. ABE0's *adaptation strategy* is to return the effort of the nearest $k = 1$ item.
- *Two ways to find training subsets*: (a) Remove nothing: Usually, effort estimators use all training projects [13]. Our ABE0 is using this variant; (b) Outlier methods: prune training projects with (say) suspiciously large values [38]. Typically, this removes a small percentage of the training data.
- *Eight ways to make feature weighting*: Li *et al.* [49] and Hall and Holmes [28] review 8 different *feature* weighting schemes.
- *Three ways to discretize* (summarize numeric ranges into a few bins): Some *feature* weighting schemes require an initial discretization of continuous columns. There are many discretization policies in the literature, including: (1) equal frequency, (2) equal width, (3) do nothing.
- *Six ways to choose similarity measurements:* Mendes *et al.* [50] discuss three similarity measures, including the weighted Euclidean measure described above, an unweighted variant (where $w_i = 1$), and a "maximum distance" measure that focuses on the single *feature* that maximizes interproject distance. Frank *et al.* [23] use a triangular distribution that sets to the weight to zero after the distance is more than "k" neighbors away from the test instance. A fifth and sixth similarity measure are the Minkowski distance measure used in [4] and the mean value of the ranking of each project *feature* used in [84].
- *Four ways for adaption mechanisms:* (1) median effort value, (2) mean dependent value, (3) summarize the adaptations via a second learner (e.g., linear regression) [7, 49, 51, 65], (4) weighted mean [50].
- *Six ways to select analogies:* Analogy selectors are fixed or dynamic [41]. Fixed methods use $k \in \{1, 2, 3, 4, 5\}$ nearest neighbors while dynamic methods use the training set to find which $1 \leq k \leq N - 1$ is best for N examples.

2.2 Effort Estimation and Hyperparameter Optimization

Note that I do <u>not</u> claim that the above represents all methods for effort estimation. Rather, I say that (a) all the above are either prominent in the literature or widely used; and (b) anyone with knowledge of the current effort estimation literature would be tempted to try some of the above.

While the above list is incomplete, it is certainly very long. Consider, for example, just the ABEN variants documented in Table 2. There are $2 \times 8 \times 3 \times 6 \times 4 \times 6 = 6,912$ such variants. Some can be ignored; e.g. at $k = 1$, adaptation mechanisms return the same result, so they are not necessary. Also, not all *feature* weighting techniques use discretization. But even after those discards, there are still thousands of possibilities.

Given the space to exploration is so large, some researchers have offered automatic support for that exploration. Some of that prior work suffered from being applied to limited data [49] or optimizing with algorithms that are not representative of the state-of-the-art in multi-objective optimization [49].

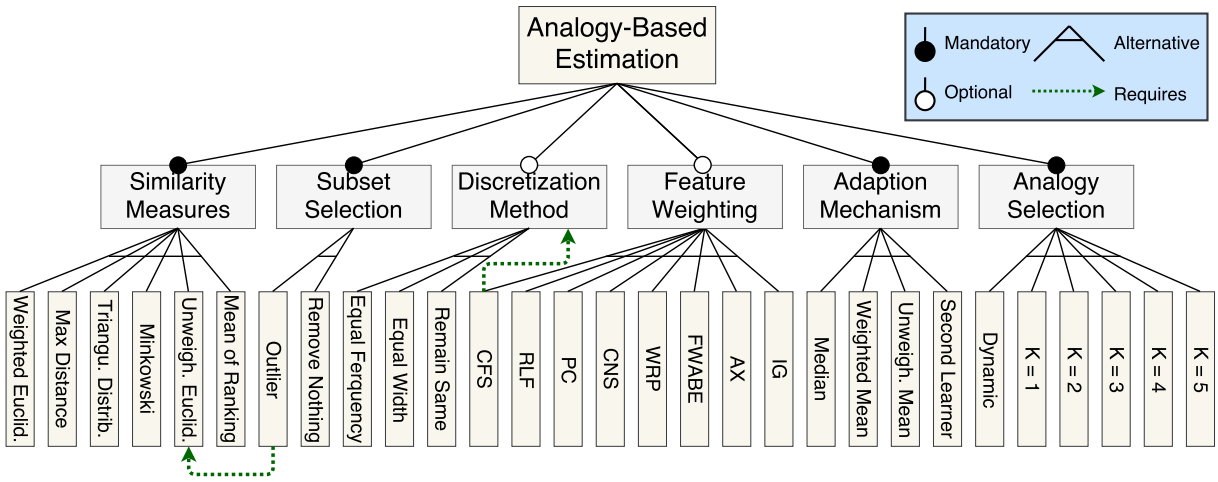


Figure 1: RATE's feature model of the space of machine learning options for ABEN. In this model, *SubsetSelection*, *Similarity*, *AdaptionMechanism* and *AnalogySelection* are the mandatory *features*, while the *FeatureWeighting* and *DiscretizationMethod* *features* are optimal. To avoid making the graph too complex, some cross-tree constrains are not presented.

Another issue with prior work is that researchers use methods deprecated in the literature. For example, some use grid search [19, 76] which is a set of nested for-loops that iterate over a range of options. Grid search is slow and, due to the size of the increment in each loop, can miss important settings [9].

Some research also recommends tabu search (TS) for hypeparameter optimisation in software effort estimation [15]. Tabu search, as proposed by Glover, aims to overcome some limitations of local search [27]. It is a meta-heuristic relying on adaptive memory and responsive exploration of the search space. However, to use TS, good understanding of the problem structure is required, and domain specific knowledge is needed for selection for tabus and aspiration criteria. I had planned to include TS in this analysis, then realized that of the earlier advocates of TS for effort estimation were still using it (e.g. Sarro and Corazza et al. [15] explored TS in their 2013 paper but elected not to use it in any of their subsequent experiments [67, 68].

Other researchers assume that the effort model is a specific parametric form (e.g. the COCOMO equation) and propose mutation methods to adjust the parameters of that equation [3, 12, 56, 66, 75]. As mentioned above, this approach is hard to test since there are very few datasets using the pre-specified COCOMO attributes.

Further, all that prior work needs to be revisited given the existence of recent and very prominent methods; i.e. ATLM from TOSEM'15 [86] or LP4EE from TOSEM'18 [67].

Accordingly, this paper conducts a more thorough investigation of hyperparameter optimization for effort estimation.

- I use methods with no data feature assumptions (i.e. no COCOMO data);
- That vary many parameters (6,000+ combinations);
- That also tests results on 9 different sources with data on 945 software projects;
- Which uses optimizers representative of the state-of-the-art (NSGA-II [18], MOEA/D [89], FLASH [59], DE [79]);
- And which benchmark results against prominent methods such as ATLM and LP4EE.

3 RELATED WORK

In software engineering, hyperparameter optimization techniques have been applied to some sub-domains, but yet to be adopted in many others. One way to characterize this paper is an attempt to adapt recent work in hyperparameter optimization in software defect prediction to effort estimation. Note that, like in defect prediction, this article has also concluded that Differential Evolution is an useful method.

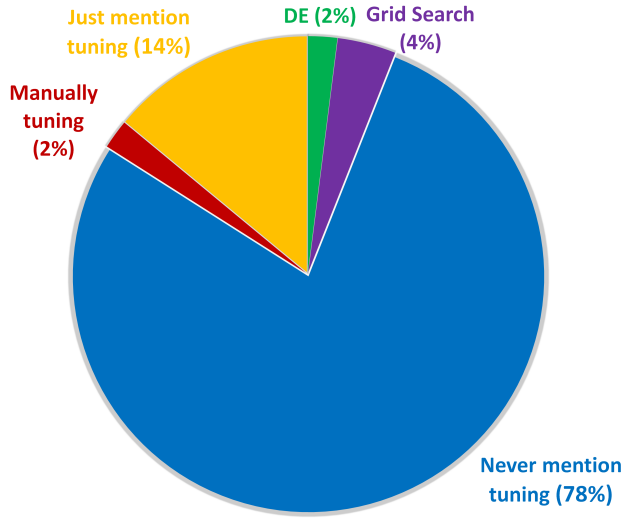
Several SE defect prediction techniques rely on static code attributes [45, 60, 80]. Much of that work has focused of finding and employing complex and "off-the-shelf" machine learning models [21, 52, 57], without any hyperparameter optimization. According to a literature review done by Fu et al. [25], as shown in Figure 2, nearly 80% of highly cited papers in defect prediction do not mention parameters tuning (so they rely on the default parameters setting of the data miners).

Gao et al. [26] acknowledged the impacts of the parameter tuning for software quality prediction. For example, in their study, "*distanceWeighting*" parameter was set to "*Weight by 1/distance*", the *KNN* parameter "*k*" was set to "30", and the "*crossValidate*" parameter was set to "*true*". However, they did not provide any further explanation about their tuning strategies.

As to methods of tuning, Bergstra and Bengio [9] comment that *grid search*[1] is very popular since (a) such a simple search to gives researchers some degree of insight; (b) grid search has very little technical overhead for its implementation; (c) it is simple to automate and parallize; (d) on a computing cluster, it can find better tunings than sequential optimization (in the same amount of time). That said, Bergstra and Bengio deprecate grid search since that style

[1]For N tunable option, run N nested for-loops to explore their ranges.

Figure 2: Literature review of hyperparameters tuning on 52 top defect prediction papers [25]



of search is not more effective than more randomized searchers if the underlying search space is inherently low dimensional.

Lessmann et al. [48] used grid search to tune parameters as part of their extensive analysis of different algorithms for defect prediction. However, they only tuned a small set of their learners while they used the default settings for the rest. my conjecture is that the overall cost of their tuning was too expensive so they chose only to tune the most critical part.

Two recent studies about investigating the effects of parameter tuning on defect prediction were conducted by Tantithamthavorn et al. [81, 82] and Fu et al. [24]. Tantithamthavorn et al. also used grid search while Fu et al. used differential evolution. Both of the papers concluded that tuning rarely makes performance worse across a range of performance measures (precision, recall, etc.). Fu et al. [24] also report that different datasets require different hyperparameters to maximize performance.

One major difference between the studies of Fu et al. [24] and Tantithamthavorn et al. [81] was the computational costs of their experiments. Since Fu et al.'s differential evolution based method had a strict stopping criterion, it was significantly faster.

Note that there are several other methods for hyperparameter optimization and I aim to explore several other method as a part of future work. But as shown here, it requires much work to create and extract conclusions from a hyperparameter optimizer. One goal of this work, which I think I have achieved, to identify a simple baseline method against which subsequent work can be benchmarked.

4 RATE

RATE is my architecture for exploring hyperparameter optimization and effort estimation, initially, my plan was to use standard hyperparameter tuning for this task. Then I learned that standard data mining toolkits like Scikit-learn [62] did not include many of the effort estimation techniques; and (b) standard hyperparameter tuners can be slow (Scikit-learn recommends a default runtime of 24 hours . Hence, I build RATE:

- At the base *library layer*, I use Scikit-learn [62].
- Above that, RATE has a *utilities layer* containing all the algorithms missing in Scikit-Learn (e.g., ABEN required numerous additions at the utilities layer).
- Higher up, RATE's *modelling layer* uses an XML-based domain-specific language to specify a *feature* map of data mining options. These feature models are single-parent and-or graphs with (optional) cross-tree constraints showing what options require or exclude other options. A graphical representation of the feature model used in this paper is shown in Figure 1.
- Finally, at top-most *optimizer layer*, there are some optimizers (e.g. differential evolution and FLASH) that make decisions across the *feature* map. An automatic *mapper* facility then links those decisions down to the lower layers to run the selected algorithms.

4.1 FLASH

FLASH, proposed by Nair et. al, is a decision tree based optimizer that incrementally grows one decision tree per objective [59]. The design of FLASH as a combination of other methods:

- From the evolutionary algorithm community, I took the *dominance counters* Ω_B and Ω_I since these can help to find good candidate(s) within a large space of multiple objective options.
- From the Bayesian optimization work, I took the technique of *minimizing the calls to the fitness functions*; i.e. build a model from the current evaluations then use that model as a surrogate for the real world evaluations.
- From the software analytics community, I took *regression tree learning with CART* since such trees offer a succinct representation of multiple examples. Another advantage of CART is that this algorithm does scale to large dimensional models (whereas the Gaussian process models used in, say, ePAL struggle to build models for more than ten dimensions).

Standard CART has the disadvantage that it only builds models for a single goal. Hence, FLASH uses CART to build a separate tree for each objective in a multi-objective problem.

The resulting algorithm is shown in Algorithm 1. For each loop of the algorithm, the *best* set of solutions is repeatedly pruned by Ω_B (the NSGA-II fast non-dominating sort algorithm). FLASH tries to grow *best* by running its CART models (one for each objective) over all the data in order to find the single example that looks most promising. However, if that new example fails to grow the *best* set, FLASH looses a life.

FLASH only executes a full evaluation, once per cycle of its **while** loop; i.e. measured in terms of number of evaluations, FLASH should run very fast. Further, since it uses CART and not Gaussian process models, it should scale to models with much more than ten dimensions.

FLASH offers certain novel innovations over other work that tried optimizing Bayesian optimizers by replacing Gaussian process models with other learners. Researchers exploring methods to optimize Bayesian optimizers beyond ten decisions typically assumed single objective tasks [10, 32, 83]. Also, for FLASH, I strive to

generate succinct descriptions of its processing (one small decision tree per objective). Other researchers never even attempt to explore comprehensible so for their Bayesian optimizers variants, they use random forests– which typically generate 10s to 100s of trees [32].

Algorithm 1 Pseudocode of FLASH

```
def FLASH(uneval_configs, fitness, size, life):
  # Add |size| number of randomly selected configurations to training data.
  # All the randomly selected configurations are measured
  eval_configs = [measure(x) for x in sample(uneval_configs, size)]
  # Remove the evaluations configuration from data
  uneval_configs.remove(eval_configs)
  # Till all the lives has been lost
  while life > 0:
    # build one CART model per objective
    for o in objectives: model[o] = CART(eval_configs)
    # Find and measure another point based on acquisition function
    acquired_point = measure(acquisition_fn(uneval_configs, model))
    eval_configs += acquired_point # Add acquired point
    uneval_config −= acquired_point # Remove acquired point
    # Stopping Criteria
    life −= 1
  return best

def acquisition_fn(uneval_configs, model, no_directions=10):
  # Predict the value of all the unevaluated configurations using model
  predicted = model.predict(uneval_configs)
  # If number of objectives is greater than 1 (In my setting len(objectives) = 2)
  if len(objectives) > 1: # For multi−objective problems
    return Bazza( predicted )
  else: # For single−objective problems
    return max(predicted)
```

4.2 Other Optimizers

Once RATE's layers were built, it was simple to "pop the top" and replace the top layer with another optimizer. Nair et al. [58] advise that for search-based SE studies, optimizers should be selecting via the a "dumb+two+next" rule. Here:

- "Dumb" is some baseline method;
- "Two" are some well-established optimizers;
- "Next" is a more recent method which may not have been applied before to this domain.

For my "dumb" optimizer, I used Random Choice (hereafter, RD). To find N valid configurations, RD selects leaves at random from Figure 1. All these N variants are executed and the best one is selected for application to the test set. To maintain parity with DE2 and DE8 systems described below, RATE uses $N \in \{40, 160\}$ (denoted RD40 and RD160).

Moving on, my "two" well-established optimizer are differential evolution (hereafter, DE [79]) and NSGA-II [18]. These have been used frequently in the SE literature [1, 2, 24, 69, 70]. NSGA-II is a standard genetic algorithm (for N generations, mutate, crossover, select best candidates for the next generation) with a fast select operator. All candidates that dominated i other items are grouped into together in "band" i. When selecting C candidates for the next generation, the top bands $1..i$ with $M < C$ candidates are all selected. Next, using a near-linear time pruning operator, the $i + 1$ band is pruned down to $C - M$ items, all of which are selected.

The premise of DE is that the best way to mutate the existing tunings is to extrapolate between current solutions. Three solutions a, b, c are selected at random. For each tuning parameter k, at some probability cr, I replace the old tuning x_k with y_k. For booleans $y_k = \neg x_k$ and for numerics, $y_k = a_k + f \times (b_k - c_k)$ where f is a parameter controlling differential weight. The main loop of DE runs over the population of size np, replacing old items with new candidates (if new candidate is better). This means that, as the loop progresses, the population is full of increasingly more valuable solutions (which, in turn, helps extrapolation). As to the control parameters of DE, using advice from Storn [79], I set $\{np, g, cr\} = \{20, 0.75, 0.3\}$. The number of generations $gen \in \{2, 8\}$ was set as follows. A small number (2) was used to test the effects of a very CPU-light effort estimator. A larger number (8) was used to check if anything was lost by restricting the inference to just two generations. These two versions were denoted DE2 and DE8.

There are many other variants of DE besides above DE2 and DE8, with different mutation and crossover strategies, number of generations, with or without early stopping rule, the performance of DE can be very different. my third DE method uses same parameters as DE2 and DE8 ($\{20, 0.75, 0.3\}$), but for the mutation strategy, a new equation is used to find replacement y_k:

$$y_k = best_k + f \times (a_k - b_k) + f \times (c_k - d_k)$$

instead of using the original random choice $y_k = a_k + f \times (b_k - c_k)$. In this new equation, four mutually exclusive solutions a, b, c, d are randomly selected, and $best_k$ is the one has the best performance in current solutions. For the crossover strategy, I use binomial, which is performed on each of the d variables whenever a randomly generated number between 0 and 1 is less than or equal to crossover rate (cr).

I call this DE method "DENM".

As to my "next" optimizer, I used MOEA/D [89]. This is a decomposition approach that runs simultaneous problems at once, as follows. Prior to inference, all candidates are assigned random weights to all goals. Candidates with similar weights are said to be in the same "neighborhood". When any candidate finds a useful mutation, then this candidate's values are copied to all neighbors that are further away than the candidate from the best "Utopian point". Note that these neighborhoods can be pre-computed and cached prior to evolution. Hence, MOEA/D runs very quickly. MOEA/D has not been previously applied to effort estimation.

Table 3: Data in this study. For details on the features, see Table 4.

	Projects	Features
kemerer	15	6
albrecht	24	7
isbsg10	37	11
finnish	38	7
miyazaki	48	7
maxwell	62	25
desharnais	77	6
kitchenham	145	6
china	499	16
total	945	

For these optimizers I used, DE and RD have the fixed evaluation budget described above. The other evolutionary treatments (NSGA-II, MOEA/D, FLASH) were ran till the results from new generations were no better than before.

Table 4: Descriptive Statistics of the Datasets

	feature	min	max	mean	std
kemerer	Lang.	1	3	1.2	0.6
	Hdware	1	6	2.3	1.7
	Duration	5	31	14.3	7.5
	KSLOC	39	450	186.6	136.8
	AdjFP	100	2307	999.1	589.6
	RAWFP	97	2284	993.9	597.4
	Effort	23	1107	219.2	263.1
albrecht	Input	7	193	40.2	36.9
	Output	12	150	47.2	35.2
	Inquiry	0	75	16.9	19.3
	File	3	60	17.4	15.5
	FPAdj	1	1	1.0	0.1
	RawFPs	190	1902	638.5	452.7
	AdjFP	199	1902	647.6	488.0
	Effort	0	105	21.9	28.4
isbsg10	UFP	1	2	1.2	0.4
	IS	1	10	3.2	3.0
	DP	1	5	2.6	1.1
	LT	1	3	1.6	0.8
	PPL	1	14	5.1	4.1
	CA	1	2	1.1	0.3
	FS	44	1371	343.8	304.2
	RS	1	4	1.7	0.9
	FPS	1	5	3.5	0.7
	Effort	87	14453	2959	3518
finnish	hw	1	3	1.3	0.6
	at	1	5	2.2	1.5
	FP	65	1814	763.6	510.8
	co	2	10	6.3	2.7
	prod	1	29	10.1	7.1
	lnsize	4	8	6.4	0.8
	lneff	6	10	8.4	1.2
	Effort	460	26670	7678	7135

	feature	min	max	mean	std
miyazaki	KLOC	7	390	63.4	71.9
	SCRN	0	150	28.4	30.4
	FORM	0	76	20.9	18.1
	FILE	2	100	27.7	20.4
	ESCRN	0	2113	473.0	514.3
	EFORM	0	1566	447.1	389.6
	EFILE	57	3800	936.6	709.4
	Effort	6	340	55.6	60.1
maxwell	App	1	5	2.4	1.0
	Har	1	5	2.6	1.0
	Dba	0	4	1.0	0.4
	Ifc	1	2	1.9	0.2
	Source	1	2	1.9	0.3
	Telon.	0	1	0.2	0.4
	Nlan	1	4	2.5	1.0
	T01	1	5	3.0	1.0
	T02	1	5	3.0	0.7
	T03	2	5	3.0	0.9
	T04	2	5	3.2	0.7
	T05	1	5	3.0	0.7
	T06	1	4	2.9	0.7
	T07	1	5	3.2	0.9
	T08	2	5	3.8	1.0
	T09	2	5	4.1	0.7
	T10	2	5	3.6	0.9
	T11	2	5	3.4	1.0
	T12	2	5	3.8	0.7
	T13	1	5	3.1	1.0
	T14	1	5	3.3	1.0
	Dura.	4	54	17.2	10.7
	Size	48	3643	673.3	784.1
	Time	1	9	5.6	2.1
	Effort	583	63694	8223	10500

	feature	min	max	mean	std
desharnais	TeamExp	0	4	2.3	1.3
	MngExp	0	7	2.6	1.5
	Length	1	36	11.3	6.8
	Trans.s	9	886	177.5	146.1
	Entities	7	387	120.5	86.1
	AdjPts	73	1127	298.0	182.3
	Effort	546	23940	4834	4188
kitchenham	code	1	6	2.1	0.9
	type	0	6	2.4	0.9
	duration	37	946	206.4	134.1
	fun_pts	15	18137	527.7	1522
	estimate	121	79870	2856	6789
	esti_mtd	1	5	2.5	0.9
	Effort	219	113930	3113	9598
china	ID	1	499	250.0	144.2
	AFP	9	17518	486.9	1059
	Input	0	9404	167.1	486.3
	Output	0	2455	113.6	221.3
	Enquiry	0	952	61.6	105.4
	File	0	2955	91.2	210.3
	Interface	0	1572	24.2	85.0
	Added	0	13580	360.4	829.8
	Changed	0	5193	85.1	290.9
	Deleted	0	2657	12.4	124.2
	PDR_A	0	84	11.8	12.1
	PDR_U	0	97	12.1	12.8
	NPDR_A	0	101	13.3	14.0
	NPDU_U	0	108	13.6	14.8
	Resource	1	4	1.5	0.8
	Dev.Type	0	0	0.0	0.0
	Duration	1	84	8.7	7.3
	N_effort	31	54620	4278	7071
	Effort	26	54620	3921	6481

5 EMPIRICAL STUDY

5.1 Data

To assess RATE, I applied it to the 945 projects seen in nine datasets from the SeaCraft repository (http://tiny.cc/seacraft); see Table 3 and Table 4. This data was selected since it has been widely used in previous estimation research. Also, it is quite diverse since it differs for:

- Observation number (from 15 to 499 projects).
- Number and type of *features* (from 6 to 25 *features*, including a variety of *features* describing the software projects, such as number of developers involved in the project and their experience, technologies used, size in terms of Function Points, etc.). Note that some features of the original datasets are not used in my experiment because they are naturally irrelevant to their effort values. For example, feature "ID" in "china" dataset is just the order number of these projects, I drop these kinds of feature in experiments. Other cases are "Syear" in "maxwell" (which denotes "start year"), "ID" in "miyazaki/kemerer/china" and "defects/months" in "nasa93/coc81".
- Technical characteristics (software projects developed in different programming languages and for different application domains, ranging from telecommunications to commercial information systems).
- Geographical locations (software projects coming from Canada, China, Finland).

5.2 Cross-Validation

Each datasets was treated in a variety of ways. Each *treatment* is an *M*N-way* cross-validation test of some learner or some learner and optimizer. That is, M times, shuffle the data randomly (using a different random number seed) then divide the data into N bins. For $i \in N$, bin i is used to test a model build from the other bins. Following the advice of Nair et al. [58], for the smaller datasets (with 40 rows or less), I use $N = 3$ bins while for the others, I use $N = 10$ bins.

As a procedural detail, first I divided the data and then I applied the treatments. That is, all treatments saw the same training and test data.

5.3 Scoring Metrics

The results from each an effort estimator can be scored many ways including the magnitude of the relative error (MRE) [14] and standardized accuracy (SA) measure defined in Table 5. I represent results in these metrics since they are widely used in the literature. Note that for these evaluation measures:

- MRE: *smaller* values are *better*;
- SA: *larger* values are *better*.

From the cross-vals, I report the *median* (termed *med*) which is the 50th percentile of the test scores seen in the *M*N results*. Also reported are the *inter-quartile range* (termed *IQR*) which is the (75-25)th percentile. The IQR is a non-parametric description of the variability about the median value.

For each datasets, the results from a *M*N-way* are sorted by their *median* value, then *ranked* using the Scott-Knott test recommended for ranking effort estimation experiments by Mittas et al. in TSE'13 [55]. For full details on Scott-Knott test, see Table 6. In summary, Scott-Knott is a top-down bi-clustering method that recursively divides sorted treatments. Division stops when there is only one treatment left or when a division of numerous treatments generates splits that are statistically *indistinguishable*. To judge when two sets of treatments are indistinguishable, I use a conjunction of *both* a 95% bootstrap significance test [20] *and* a A12 test for a non-small effect size difference in the distributions [53]. These tests were used since their non-parametric nature avoids issues with non-Gaussian distributions.

Figure 3 shows an example of the report generated by my Scott-Knott procedure. Note that when multiple treatments tie for *Rank=1*,

Table 5: Performance scores: MRE and SA

The performance for our effort estimators are measured in terms of MRE and SA, defined in this table.

MRE: MRE is defined in terms of AR, the magnitude of the absolute residual. This is computed from the difference between predicted and actual effort values:

$$AR = |actual_i - predicted_i|$$

MRE is the magnitude of the relative error calculated by expressing AR as a ratio of the actual effort value; i.e.,

$$MRE = \frac{|actual_i - predicted_i|}{actual_i}$$

MRE has been criticized [22, 39, 43, 64, 72, 78] as being biased towards error underestimations.

SA: Because of issues with MRE, some researchers prefer the use of other (more standardized) measures, such as Standardized Accuracy (SA) [46, 73]. SA is defined in terms of

$$MAE = \frac{1}{N}\sum_{i=1}^{n}|RE_i - EE_i|$$

where N is the number of projects used for evaluating the performance, and RE_i and EE_i are the actual and estimated effort, respectively, for the project i. SA uses MAE as follows:

$$SA = (1 - \frac{MAE_{P_j}}{MAE_{r_{guess}}}) \times 100$$

where MAE_{P_j} is the MAE of the approach P_j being evaluated and $MAE_{r_{guess}}$ is the MAE of a large number (e.g., 1000 runs) of random guesses. Over many runs, $MAE_{r_{guess}}$ will converge on simply using the sample mean [73]. That is, SA represents how much better P_j is than random guessing. Values near zero means that the prediction model P_j is practically useless, performing little better than random guesses [73].

Table 6: Explanation of Scott-Knott test.

This study ranks methods using the Scott-Knott procedure recommended by Mittas & Angelis in their 2013 IEEE TSE paper [55]. This method sorts a list of l treatments with ls measurements by their median score. It then splits l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. For example, we could sort $ls = 4$ methods based on their median score, then divide them into three sub-lists of size ms, $ns \in \{(1, 3), (2, 2), (3, 1)\}$. Scott-Knott would declare one of these divisions to be "best" as follows. For lists l, m, n of size ls, ms, ns where $l = m \cup n$, the "best" division maximizes $E(\Delta)$; i.e. the difference in the expected mean value before and after the spit:

$$E(\Delta) = \frac{ms}{ls}abs(m.\mu - l.\mu)^2 + \frac{ns}{ls}abs(n.\mu - l.\mu)^2$$

Scott-Knott then checks if that "best" division is actually useful. To implement that check, Scott-Knott would apply some statistical hypothesis test H to check if m, n are significantly different. If so, Scott-Knott then recurses on each half of the "best" division.

For a more specific example, consider the results from $l = 5$ treatments:

```
rx1 = [0.34, 0.49, 0.51, 0.6]
rx2 = [0.6,  0.7,  0.8,  0.9]
rx3 = [0.15, 0.25, 0.4,  0.35]
rx4=  [0.6,  0.7,  0.8,  0.9]
rx5=  [0.1,  0.2,  0.3,  0.4]
```

After sorting and division, Scott-Knott declares:

- Ranked #1 is rx5 with median= 0.25
- Ranked #1 is rx3 with median= 0.3
- Ranked #2 is rx1 with median= 0.5
- Ranked #3 is rx2 with median= 0.75
- Ranked #3 is rx4 with median= 0.75

Note that Scott-Knott found little difference between rx5 and rx3. Hence, they have the same rank, even though their medians differ.

Scott-Knott is prefered to, say, an all-pairs hypothesis test of all methods; e.g. six treatments can be compared $(6^2 - 6)/2 = 15$ ways. A 95% confidence test run for each comparison has a very low total confidence: $0.95^{15} = 46\%$. To avoid an all-pairs comparison, Scott-Knott only calls on hypothesis tests *after* it has found splits that maximize the performance differences.

For this study, our hypothesis test H was a conjunction of the A12 effect size test of and non-parametric bootstrap sampling; i.e. our Scott-Knott divided the data if *both* bootstrapping and an effect size test agreed that the division was statistically significant (95% confidence) and not a "small" effect ($A12 \geq 0.6$).

For a justification of the use of non-parametric bootstrapping, see Efron & Tibshirani [20, p220-223]. For a justification of the use of effect size tests see Shepperd & MacDonell [73]; Kampenes [35]; and Kocaguneli et al. [37]. These researchers warn that even if an hypothesis test declares two populations to be "significantly" different, then that result is misleading if the "effect size" is very small. Hence, to assess the performance differences we first must rule out small effects. Vargha and Delaney's non-parametric A12 effect size test explores two lists M and N of size m and n:

$$A12 = \left(\sum_{x \in M, y \in N} \begin{cases} 1 & if\ x > y \\ 0.5 & if\ x == y \end{cases}\right)/(mn)$$

This expression computes the probability that numbers in one sample are bigger than in another. This test was recently endorsed by Arcuri and Briand at ICSE'11 [5].

then I use the treatment's runtimes to break the tie. Specifically, for all treatments in *Rank=1*, I mark the faster ones as *Rank=1**.

5.4 Terminology for Optimizers

Some treatments are named "X_Y" which denote learner "X" tuned by optimizer "Y". In the following:

$$X \in \{CART, ABE\}$$
$$Y \in \{DE2, DE8, DENM, FLASH, MOEA/D, NSGA2\}$$

Note that I do not tune ATLM and LP4EE since they were designed to be used "off-the-shelf". Whigham et al. [86] declare that one of ATLM's most important features is that if does not need tuning.

6 RESULTS

6.1 Observations

Table 7 shows the runtimes (in minutes) for one of my 20 N*M experiments for each dataset. From the last column of that table, I see that the median to maximum runtimes per dataset range are:

- 140 to 680 minutes, for one-way;
- Hence 47 to 227 hours, for the 20 repeats of my N*M experiments.

Performance scores for all datasets are shown in Figure 4. For space reasons, all the slower treatments (as defined in Table 7) that were not ranked first have been deleted. Rationale: such sub-optimal and slower treatments need not be discussed further. Please see https://ibb.co/ftCOD9 for a display of all the results.

In Figure 4, I observe that ATLM and LP4EE performed as expected. Whigham et al. [86] and Sarro et al. [67] designed these methods to serve as baselines against which other treatments can be compared. Hence, it might be expected that these methods will perform comparatively below other methods. This was certainly the

Rank	Methods	SA Med	IQR
1*	CART_FLASH	8	1
1*	CART_DENM	8	1
2	CART_DE8	81	14
2	CART_DE2	74	10
2	CART_RD160	74	16
2	CART_NSGA2	73	14
2	CART_RD40	71	12
2	CART_MOEAD	71	11
2	CART0	69	19
3	ABEN_DE2	65	31
3	ABEN_DE8	64	34
3	ABEN_RD160	61	29
3	ABEN_RD40	59	26
3	ABEN_NSGA2	54	32
3	ABE0	49	22
4	ATLM	40	49
4	LP4EE	36	41

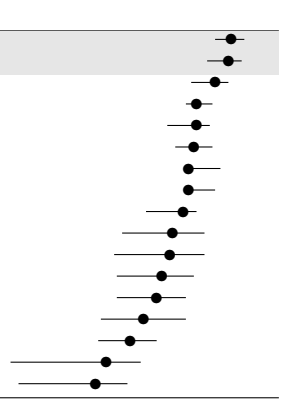


Figure 3: Example of Scott-Knott results. SA scores seen in the finnish dataset. sorted by their median value. Here, *larger* values are *better*. Med is the 50th percentile and IQR is the *inter-quartile range*; i.e., 75th-25th percentile. Lines with a dot in the middle shows median values with the IQR. For the Ranks, *smaller* values are *better*. Ranks are computed via the Scott-Knot procedure from TSE'13 [55]. Rows with the same ranks are statistically indistinguishable. 1* denotes rows of fastest best-ranked treatments.

case here, as seen in Figure 4, these baseline methods are top-ranked in only 5/18 datasets.

Another thing to observe in Figure 4 is that random search (RD) performed as expected; i.e. it was never top-ranked. This is a gratifying result since if random otherwise, then that tend to negate the value of hyperparamter optimization.

Another interesting result is that traditional estimation-by-analogy might also be termed a baseline method. Note that ABE0 scores well in 5/18 datasets; i.e. just similar as LP4EE and ATLM.

6.2 Answers to Research Questions

Finally, I turn to the research questions listed in the introduction.

RQ1: Is it best just to use the "off-the-shelf" defaults?

Arcuri & Fraser note that for test case generation, using the default settings can work just as well as anything else [6]. I can see some evidence of this effect in Figure 4. Observe, for example, the isbsg10 results where the untuned ABE0 treatment achieves *Rank=1** in both MRE and SA metrics.

However, overall, Figure 4 is negative on the use of default settings. If I just used any For example, in datasets "china/finnish/miyazaki", not even one treatments that use the default found in *Rank=1**. Overall, if I always used just *one* of the methods using defaults (LP4EE, ATLM, ABE0) then that would achieve best ranks in 9/18 datasets.

Another aspect to note in the Figure 4 results are the large differences in performance scores between the best and worst treatments (exceptions: desharnais and isbsg10's SA scores do not vary much). That is, there is much to be gained by using the *Rank=1* treatments* and deprecating the rest.

In summary, using the defaults is recommended only in a part of datasets. Also, in terms of better test scores, there is much to be gained from tuning. Hence:

> ***Lesson1:*** "Off-the-shelf" defaults should be deprecated.

RQ2: Can I replace the old defaults with new defaults?

If the hyperparameter tunings found by this paper were nearly always the same, then this study could conclude by recommending better values for default settings. This would be a most convenient result since, in future when new data arrives, the complexities of this study would not be needed.

Unfortunately, this turns out not to be the case. Table 8 shows the percent frequencies with which some tuning decision appears in my *M*N-way* cross validations (this table uses results from DE8 tuning CART since, as shown below, this usually leads to best results). Note that in those results it it not true that across most datasets there is a setting that is usually selected (thought min_samples_leaf less than 3 is often a popular setting). Accordingly, I say that Table 8 shows that there is much variations of the best tunings. Hence, for effort estimation:

> ***Lesson2:*** Overall, there are no "best" default settings.

Before going on, one curious aspect of the Table 8 results are the max_features results; it was rarely most useful to use all features. Except for finnish and china), best results were often obtained after discarding (at random) a quarter to three-quarters of the features. This is a clear indication that, in future work, it might be advantageous to explore more feature selection for CART models.

RQ3: Can I avoid slow hyperparameter optimization?

"Kilo-optimizers" such as NSGA-II examine 10^3 candidates or more since they explore population sizes of 10^2 for many generations. Hence, as shown in Table 7, they can be very slow.

Is it possible to avoid such slow runtimes? There are many heuristic methods for speeding up kilo-optimization:

- Active learners select explore a few most informative candidates [44];
- Decomposition learners like MOEA/D convert large objective problems into multiple smaller problems;
- Other optimizers explore fewer candidates (DE & RD).

Kilo-optimization is necessary when their exploration of more candidates leads to better solutions that heuristic exploration. Such better solutions from kilo-optimization are rarely found in Figure 4 (only in 1/18 cases). Further, the size of the improvements seen with kilo-optimizers over the best Rank=2 treatments is very small. Those improvements come at significant runtime cost (in Table 7), the kilo-optimizers are one to two orders of magnitude slower than other methods). Hence I say that for effort estimation:

> ***Lesson3:*** Overall, my slowest optimizers perform no better than certain faster ones.

RQ4: What hyperparatmeter optimizers to use for effort estimation?

When I discuss this work with my industrial colleagues, they want to know "the bottom line"; i.e. what they should use or, at the very least, what they should not use. This section offers that advice. I stress that this section is based on the above results so, clearly these recommendations are something that would need to be revised whenever new results come to hand.

a. % MRE (*smaller* values are *better*).

Rank	Using	Med.	IQR
albrecht			
1*	CART_FLASH	46	21
1	ABEN_RD160	48	29
1	ABEN_DE2	48	20
1*	ABE0	51	28
1	ABEN_DE8	51	30
1*	CART_DENM	51	33
1*	CART_DE8	52	28
2	CART_DE2	56	23
2	CART_MOEAD	56	23
2	CART0	59	21
4	LP4EE	143	87
4	ATLM	178	122
china			
1*	CART_DE8	8	1
1*	CART_FLASH	8	3
1*	CART_DE2	9	2
2	CART_DENM	14	3
2	CART0	15	2
3	CART_MOEAD	40	10
4	ABE0	54	10
4	LP4EE	56	17
5	ATLM	69	18
desharnais			
1*	CART_DE8	39	19
1*	CART_DENM	41	18
1*	CART_DE2	41	19
1*	CART_FLASH	43	18
2	CART_MOEAD	45	18
2	LP4EE	45	19
2	ATLM	46	18
3	CART0	51	17
3	ABE0	56	33
finnish			
1	CART_NSGA2	14	12
1*	CART_FLASH	17	18
2	CART_DENM	23	14
2	CART_DE8	24	16
2	CART_DE2	30	11
2	CART0	32	16
3	ABE0	43	26
4	CART_MOEAD	53	32
6	LP4EE	73	54
6	ATLM	87	72
kemerer			
1*	CART_NSGA2	36	21
2	CART_MOEAD	46	15
2	CART_DENM	52	24
2	CART_DE2	54	22
2	ABE0	55	38
3	LP4EE	59	23
3	CART_DE8	61	21
3	CART0	61	20
3	CART_FLASH	62	15
4	ATLM	99	444
maxwell			
1*	CART_DE8	43	26
1*	CART_DE2	45	23
1*	CART_DENM	46	22
1*	CART_FLASH	47	24
2	LP4EE	58	32
2	ABE0	67	32
2	CART0	68	20
2	CART_MOEAD	75	28
3	ATLM	540	917
miyazaki			
1*	CART_FLASH	35	25
2	CART_DENM	42	24
2	CART_DE8	42	27
2	CART_DE2	45	31
2	LP4EE	49	33
3	ABE0	56	35
3	CART0	56	20
4	CART_MOEAD	70	30
5	ATLM	158	102
isbsg10			
1*	CART_DE2	72	15
1*	CART_DE8	73	15
1*	ABE0	75	19
1*	CART0	75	10
2	CART_DENM	83	13
2	CART_FLASH	83	16
2	LP4EE	87	21
2	CART_MOEAD	89	11
3	ATLM	127	124
kitchenham			
1*	LP4EE	24	13
1*	CART_DENM	25	12
1*	ATLM	26	12
1*	CART_DE2	26	10
1*	CART_DE8	27	11
1*	CART_FLASH	28	14
1*	CART0	34	11
3	ABE0	47	20
4	CART_MOEAD	69	32

b. % SA (*larger* values are *better*)

Rank	Using	Med.	IQR
albrecht			
1*	CART_MOEAD	65	21
1	ABEN_DE2	63	11
1	ABEN_DE8	62	18
1	ABEN_RD160	60	18
1*	ABE0	60	16
1*	CART_FLASH	59	39
2	CART_DENM	54	32
2	CART_DE8	53	19
2	CART_DE2	50	52
2	CART0	45	40
3	ATLM	18	30
3	LP4EE	17	34
china			
1*	CART_DE8	93	2
1*	CART_DE2	93	2
2	CART_FLASH	89	5
2	CART_DENM	87	9
3	CART0	85	7
4	ABE0	61	11
5	CART_MOEAD	54	19
5	LP4EE	51	13
5	ATLM	48	7
desharnais			
1*	CART_FLASH	47	16
1*	CART_DENM	44	24
1	ABEN_DE8	44	24
1	ABEN_NSGA2	43	27
1*	ATLM	42	13
1	ABEN_RD40	42	27
1*	CART_DE8	42	27
1	ABEN_DE2	41	28
1*	CART_DE2	40	26
1*	LP4EE	40	24
1	ABEN_RD160	40	31
2	CART_MOEAD	35	31
2	ABE0	30	42
2	CART0	23	18
finnish			
1*	CART_FLASH	87	11
1*	CART_DENM	86	13
2	CART_DE8	81	14
2	CART_DE2	74	10
2	CART_MOEAD	71	11
2	CART0	69	19
3	ABE0	49	22
4	ATLM	40	49
4	LP4EE	36	41
kemerer			
1*	CART_MOEAD	47	47
1*	CART_FLASH	46	39
1	ABEN_DE8	41	37
1	ABEN_DE2	39	30
1*	ABE0	37	41
1	ABEN_RD160	37	46
1	ABEN_RD40	36	18
2	CART_DENM	31	72
2	CART_DE2	30	82
2	CART_DE8	26	102
2	LP4EE	14	28
2	CART0	12	48
3	ATLM	-48	507
maxwell			
1*	CART_MOEAD	60	33
1*	CART_DENM	58	31
1*	LP4EE	57	23
1*	CART_DE2	56	28
1*	CART_DE8	56	31
2	CART_FLASH	50	39
2	ABE0	39	37
2	ATLM	33	21
3	CART0	16	21
miyazaki			
1*	CART_FLASH	57	34
1*	CART_MOEAD	51	31
1*	CART_DENM	49	31
1	ABEN_NSGA2	49	34
2	CART_DE8	45	36
2	LP4EE	42	34
2	CART_DE2	41	36
2	ABE0	40	35
3	ATLM	23	56
4	CART0	10	21
isbsg10			
1*	CART_MOEAD	38	11
1	CART_NSGA2	33	21
1	ABEN_RD40	33	30
1	ABEN_DE2	31	28
1	ABEN_RD160	30	49
1*	ATLM	28	19
1*	ABE0	28	29
1	ABEN_DE8	28	57
1	ABEN_NSGA2	26	33
2	CART_DENM	22	26
2	CART_FLASH	21	34
2	CART_DE2	19	40
2	LP4EE	17	37
2	CART_DE8	16	20
2	CART0	15	24
kitchenham			
1*	LP4EE	81	10
1*	ATLM	79	8
1*	CART_DENM	79	14
1*	CART_DE2	77	12
1*	CART_DE8	76	12
2	ABE0	64	14
2	CART_FLASH	62	21
3	CART_MOEAD	58	22
3	CART0	56	15

Figure 4: % MRE and % SA results from our cross-validation studies. Same format as Figure 3. The gray rows show the *Rank=1+* results recommended for each data set. The phrase " *out-of-range* " denotes results that are so bad that they fall outside of the 0%..100% range shown here. For space reasons, these results are slightly truncated. Not shown here are any *Rank*> 1 methods that are listed as *slower* in Table 7 since such sub-optimal and slower treatments need not be discussed further) For all results, see tiny.cc/oil-18.

Table 7: Mean runtime (in minutes), for one-way out of an N*M cross-validation experiment. cross-validation (minutes). Executing on a 2GHz processor, with 8GB RAM, running Windows 10.

	faster											slower						
	ABE0	*CART0*	*ATLM*	*LP4EE*	*FLASH_CART*	*CART_RD40*	*CART_RD160*	*CART_DE2*	*CART_DE8*	*CART_DENM*	*CART_MOEAD*	*CART_NSGA2*	*ABEN_RD40*	*ABEN_RD160*	*ABEN_DE2*	*ABEN_DE8*	*ABEN_NSGA2*	total
kemerer	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	5	2	5	2	4	17	46
albrecht	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	5	2	8	3	6	24	59
finnish	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	5	3	10	4	8	30	71
miyazaki	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	5	4	14	5	10	37	86
desharnais	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	4	8	19	12	22	64	140
isbsg10	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	5	4	11	4	95	33	163
maxwell	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	6	13	38	16	29	111	224
kitchenham	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	6	15	32	26	48	126	264
china	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1	6	36	86	82	131	329	681
total	3	4	4	4	4	4	5	5	5	5	6	47	87	223	154	267	771	

Based on the above I can assert that using *all* the estimators mentioned above is contraindicated:

- For one thing, many of them never appear in my top-ranked results.
- For another thing, testing all of them on new datasets would be needleesly expensive. Recall my rig: 20 repeats over the data where each of those repeats include the very slow estimators shown in Table 7. As seen in that table, the median to maximum runtimes for such an analysis for a single dataset would take 46 to 227 hours (i.e. days to weeks).

Similarly, using just *one* of the methods is also contraindicated. The following lists the best that can be expected if an engineer chooses just one of our estimators, and applied it to all my datasets. The fractions shown at left come from counting optimizer frequencies in the top-ranks of Figure 4:

$$\begin{aligned} 9/18: &\quad A = \{CART\_DE2, CART\_DENM\} \\ 10/18: &\quad B = \{CART\_DE8\} \\ 12/18: &\quad C = \{FLASH\_CART\} \end{aligned}$$

Note that:

- None of these best solo estimators are the untuned baseline methods (*ATLM*, *LP4EE*). Hence, I cannot endorse their use for generating estimates to be shown to bsuiness managers. That said, I do still endorse their use as a baseline methods, for methodological reasons in effort estimation research (they are useful for generating a quick result against which I can compare other, better, methods).
- These best solo methods barely cover half my datasets. Hence, below, I will recommend *combinations* of a minimal number of estimators.

Since I cannot endorse the use of *all* of my estimators, and I cannot endorse the use of just *one*, I now turn to discussing what minimal *combination* of estimators offer the most value. Suppose I are allow ourselves just *two* estimators. From Figure 4, I can count what pairs of estimators appear in *Rank=1** for most datasets. These are:

$$\begin{aligned} 16/18: &\quad D = \{CART\_DE2 + FLASH\_CART\} \\ 16/18: &\quad E = \{CART\_DE8 + FLASH\_CART\} \end{aligned}$$

Also, if I are allow ourselves just *three* estimators, then at most, these would appear in *Rank=1** results the following number of times:

$$\begin{aligned} 17/18: &\quad F = \{ABE0 + FLASH\_CART + CART\_DE2\} \\ 17/18: &\quad G = \{ABE0 + FLASH\_CART + CART\_DE8\} \\ 17/18: &\quad H = \{ATLM + FLASH\_CART + CART\_DE2\} \\ 17/18: &\quad I = \{ATLM + FLASH\_CART + CART\_DE8\} \\ 17/18: &\quad J = \{CART\_MOEAD + FLASH\_CART + CART\_DE2\} \\ 17/18: &\quad K = \{CART\_NSGA2 + FLASH\_CART + CART\_DE2\} \\ 17/18: &\quad L = \{CART\_MOEAD + FLASH\_CART + CART\_DE8\} \\ 17/18: &\quad M = \{CART\_NSGA2 + FLASH\_CART + CART\_DE8\} \end{aligned}$$

Note that, to simplify the implementation of my estimators, I might want to avoid CART_DE8, MOEA/D and NSGA-II. Rationale: the same level of results (17/18) can be acquired without adding a optimizer with more evaluations or complexity. So the best triple combinations with simplest implementations are the sets $\{F, H\}$:

$$\begin{aligned} 17/18: &\quad F = \{ABE0 + FLASH\_CART + CART\_DE2\} \\ 17/18: &\quad H = \{ATLM + FLASH\_CART + CART\_DE2\} \end{aligned}$$

Applying the same "simplify the implementation" approach to my *two* estimator results, I might select the set

$$D = \{CART\_DE2 + FLASH\_CART\}$$

as my preferred simplest pairs. Note that, on my machines,

- This fastest preferred pair of D takes 3 hour to complete a 20x3-way study;
- The faster preferred triple F takes 4 hours.

Table 8: Tunings discovered by hyperparameter selections (CART+DE8). Table rows sorted by number of rows in data sets (smallest on top). Cells in this table show the percent of times a particular choice was made. White text on black denotes choices made in more than 50% of tunings.

	max_features (selected at random; 100% means "use all")				max_depth (of trees)				min_sample_split (continuation criteria)				min_samples_leaf (termination criteria)			
	25%	50%	75%	100%	≤03	≤06	≤09	≤12	≤5	≤10	≤15	≤20	≤03	≤06	≤09	≤12
kemerer	18	32	23	27	57	37	05	00	95	02	03	00	92	02	05	02
albrecht	13	23	20	43	63	28	08	00	68	32	00	00	83	15	02	00
isbsg10	12	35	28	25	57	33	08	00	47	23	15	15	60	27	10	03
finnish	07	03	27	63	32	56	12	00	73	18	05	03	78	17	05	00
miyazaki	10	22	27	40	31	46	20	03	42	24	18	16	78	13	07	02
maxwell	04	16	40	40	18	60	20	02	44	27	17	12	50	33	14	04
desharnais	25	23	27	25	40	46	11	02	36	26	13	25	32	26	24	19
kitchenham	01	12	32	56	03	42	45	10	43	30	17	10	48	35	12	04
china	00	04	25	71	00	00	25	75	56	30	10	02	68	28	04	00

KEY: 10 20 30 40 50 60 70 80 90 100 %

I see here that the fastest preferred double D is 25% faster than preferred triple F. And consider about their similar performance (16/18 vs. 17/18) and to avoid over-fitting, I recommend the D pair:

> ***Lesson4:*** For new datasets, try a combination of *CART* with the optimizers *Differential Evolution* and *FLASH*.

7 THREATS TO VALIDITY

Internal Bias: Many of my methods contain stochastic random operators. To reduce the bias from random operators, I repeated my experiment in 20 times and applied statistical tests to remove spurious distinctions.

Parameter Bias: For other studies, this is a significant question since (as shown above) the settings to the control parameters of the learners can have a dramatic effect on the efficacy of the estimation. That said, recall that much of the technology of this paper concerned methods to explore the space of possible parameters. Hence I assert that this study suffers much less paramter bias than other studies.

Sampling Bias: While I tested RATE on the nine datasets, it would be inappropriate to conclude that RATE tuning always perform better than others methods for all datasets. As researchers, what I can do to mitigate this problem is to carefully document out method, release out code, and encourage the community to try this method on more datasets, as the occasion arises.

8 CONCLUSIONS

Hyperparameter optimization is known to dramatically improve the performance of many software analytics tasks such as software defect prediction or text classification [1, 2, 24, 82]. But as discussed in §2.2, the benefits of hyperparameter optimization for effort estimation have not been extensively studied. Prior work in this area only explored very small datasets [49] or used optimization algorithms that are not representative of the state-of-the-art in multi-objective optimization [19, 49, 76]. Other researchers assume that the effort model is a specific parametric form (e.g. the COCOMO equation), which greatly limits the amount of data that can be studied. Further, all that prior work needs to be revisited given the existence of recent and very prominent methods; i.e. ATLM from TOSEM'15 [86] and LP4EE from TOSEM'18 [67].

Accordingly, this paper conducts a more thorough investigation of hyperparameter optimization for effort estimation using methods (a) with no data feature assumptions (i.e. no COCOMO data); (b) that vary many parameters (6,000+ combinations); that tests its results on 9 different sources with data on 945 software projects; (c) which uses optimizers representative of the state-of-the-art (NSGA-II [18], MOEA/D [89], DE [79], FLASH [59]); and which (d) benchmark results against prominent methods such as ATLM and LP4EE.

Sometimes hyperparamter optimization can be both too slow and not effective. Such pessimism may indeed apply to the test data generation domain. However, the results of this paper show that there exists other domains like effort estimation where hyperparameter optimization is both useful and fast. After applying hyperparamter optimization, large improvements in effort estimation accuracy were observed (measured in terms of the standardized accuracy). From those results, I can recommend using a combination of regression trees (CART) tuned by different evolution or FLASH. This particular combination of learner and optimizers can achieve in a few hours what other optimizers need days to weeks of CPU to accomplish.

9 FUTURE WORK

This section will discuss potential opportunities and challenges based on this paper's results, which will be the next steps of my study.

9.1 More Exploration on RATE

RATE predicts the software effort by using a multilayer architecture. It will be an interesting task to explore the potential variants. There are some different options that can be applied to RATE, such as:

- In optimizier layer, when using FLASH as my optimizer, I use "max predicted mean" as my acquisition function. One variant of this can be max predicted variance.
- Also when using FLASH as my optimizer, instead of building predicting model by using CART, I can also use other learners, e.g. Random Forest.
- When thinking about sampling methods, RATE only use random way to do the sampling. Several novel sampling methods have been proposed recently, some state-of-art methods (e.g. SWAY) may be able to improve the performance of RATE.

9.2 Find Better Optimizers

My RATE's has investigated *two* learners using *four* different type of optimizers. I make no claim that which is the *best* optimizer for *all* learners. Rather, my point is that there exists at least some learners whose performance can be dramatically improved by at least one simple optimization scheme like DE or FLASH.

In data mining and machine learning field, grid search combined with cross-validation is still the actual practice for parameter tuning. How does grid search compare with RATE's optimizers in terms of run time/evaluations and performance? In addition, there are many other evolutionary algorithms, is it possible to find even better optimizers to replace what I currently have?

9.3 New Data Collections

The experiment data I used in this paper are 945 projects from the SEACRAFT repository (http://tiny.cc/seacraft). It could be a good idea to collect effort data from other source to validate my methods.

With millions of git repositories, GitHub is becoming one of the most important source of software development. Researchers are starting to mine the information stored in GitHub's event logs for research purposes. Mining specific information from Github's logs will provide me a tremendous amount of new data. For the research of effort estimation, finding ways to explain and re-assemble these data can be very worthy in the near future.

9.4 Beyond Hyperparameter Tuning

The optimiziers like DE or FLASH in RATE have their own magic parameters, changing them can influence the tuning performance of these optimizers. Is it possible to get better settings for the optimizers via hyper-hyperparameter optimization?

Hyper-hyperparameter optimization could be a very slow process. Hence, results like this paper could be useful since here I have identified optimizers that are very fast and very slow (and the latter would not be suitable for hyper-hyperparamter optimization).

REFERENCES

[1] A. Agrawal, W. Fu, and T. Menzies. 2018. What is wrong with topic modeling? And how to fix it using search-based software engineering. *IST Journal* (2018).

[2] A. Agrawal and T. Menzies. 2018. " Better Data" is Better than" Better Data Miners"(Benefits of Tuning SMOTE for Defect Prediction). In *ICSE'18*.

[3] Sultan Aljahdali and Alaa F Sheta. 2010. Software effort estimation by tuning COOCMO model parameters using differential evolution. In *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*. IEEE, 1–6.

[4] L. Angelis and I. Stamelos. 2000. A simulation tool for efficient analogy based cost estimation. *EMSE* 5, 1 (2000), 35–68.

[5] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 1–10.

[6] A. Arcuri and G. Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *ESE* 18, 3 (01 Jun 2013), 594–623.

[7] D. R. Baker. 2007. *A hybrid approach to expert and model based effort estimation*. West Virginia University.

[8] Bilge Baskeles, Burak Turhan, and Ayse Bener. 2007. Software effort estimation using machine learning methods. In *Computer and information sciences, 2007. iscis 2007. 22nd international symposium on*. IEEE, 1–6.

[9] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13, 1 (Feb. 2012), 281–305. http://dl.acm.org/citation.cfm?id=2503308.2188395

[10] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*. 2546–2554.

[11] B. W. Boehm. 1981. *Software engineering economics*. Prentice-Hall.

[12] S. Chalotra, S. K. Sehra, Y. S. Brar, and N. Kaur. 2015. Tuning of COCOMO Model Parameters by using Bee Colony Optimization. *Indian Journal of Science and Technology* 8, 14 (2015).

[13] C. L. Chang. 1974. Finding prototypes for nearest neighbor classifiers. *TC* 100, 11 (1974).

[14] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. 1986. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

[15] Anna Corazza, Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, Federica Sarro, and Emilia Mendes. 2013. Using tabu search to configure support vector regression for effort estimation. *Empirical Software Engineering* 18, 3 (2013), 506–546.

[16] K. Cowing. 2002. NASA to Shut Down Checkout & Launch Control System. http://www.spaceref.com/news/viewnews.html?id=475. (2002).

[17] Iris Fabiana de Barcelos Tronto, José Demisio Simões da Silva, and Nilson Sant'Anna. 2007. Comparison of artificial neural network and regression models in software effort estimation. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*. IEEE, 771–776.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (Apr 2002), 182–197.

[19] Karel Dejaeger, Wouter Verbeke, David Martens, and Bart Baesens. 2012. Data Mining Techniques for Software Effort Estimation: A Comparative Study. *IEEE Trans SE* 38, 2 (Mar 2012), 375–397.

[20] B. Efron and J. Tibshirani. 1993. *Introduction to bootstrap*. Chapman & Hall.

[21] K. O. Elish and M. O. Elish. 2008. Predicting Defect-prone Software Modules Using Support Vector Machines. *J. Syst. Softw.* 81, 5 (May 2008), 649–660. https://doi.org/10.1016/j.jss.2007.07.040

[22] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. 2003. A simulation study of the model evaluation criterion MMRE. *TSE* 29, 11 (2003), 985–995.

[23] E. Frank, M. Hall, and B. Pfahringer. 2002. Locally weighted naive bayes. In *19th conference on Uncertainty in Artificial Intelligence*. 249–256.

[24] W. Fu, T. Menzies, and X. Shen. 2016. Tuning for software analytics: Is it really necessary? *IST Journal* 76 (2016), 135–146.

[25] Wei Fu, Vivek Nair, and Tim Menzies. 2016. Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors? *arXiv preprint arXiv:1609.02613* (2016).

[26] Kehan Gao, Taghi M Khoshgoftaar, Huanjing Wang, and Naeem Seliya. 2011. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience* 41, 5 (2011), 579–606.

[27] Fred Glover and Manuel Laguna. 1997. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA.

[28] M. A. Hall and G. Holmes. 2003. Benchmarking attribute selection techniques. *TKDE* 15, 6 (2003), 1437–1447.

[29] Melissa A Hardy. 1993. *Regression with dummy variables*. Vol. 93. Sage.

[30] Abbas Heiat. 2002. Comparison of artificial neural network and regression models for estimating software development effort. *Information and software Technology* 44, 15 (2002), 911–922.

[31] J. Hihn and T. Menzies. 2015. Data Mining Methods and Cost Estimation Models: Why is it So Hard to Infuse New Ideas?. In *ASEW*. 5–9. https://doi.org/10.1109/

ASEW.2015.27
[32] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.
[33] M. Jørgensen. 2004. A review of studies on expert estimation of software development effort. *JSS* 70, 1-2 (2004), 37–60.
[34] M. Jørgensen and M. Shepperd. 2007. A systematic review of software development cost estimation studies. *TSE* 33, 1 (2007).
[35] Vigdis By Kampenes, Tore Dybå, Jo E Hannay, and Dag IK Sjøberg. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 49, 11-12 (2007), 1073–1086.
[36] C. F. Kemerer. 1987. An empirical validation of software cost estimation models. *CACM* 30, 5 (1987), 416–429.
[37] J. Keung, E. Kocaguneli, and T. Menzies. 2013. Finding conclusion stability for selecting the best effort predictor in software effort estimation. *ASE* 20, 4 (01 Dec 2013), 543–567. https://doi.org/10.1007/s10515-012-0108-5
[38] J. W. Keung, B. A. Kitchenham, and D. R. Jeffery. 2008. Analogy-X: Providing statistical inference to analogy-based software cost estimation. *TSE* 34, 4 (2008), 471–484.
[39] B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd. 2001. What accuracy statistics really measure. *IEEE Software* 148, 3 (2001), 81–85.
[40] E. Kocaguneli and T. Menzies. 2011. How to Find Relevant Data for Effort Estimation?. In *ESEM*. 255–264. https://doi.org/10.1109/ESEM.2011.34
[41] E. Kocaguneli, T. Menzies, A. Bener, and J. W. Keung. 2012. Exploiting the essential assumptions of analogy-based effort estimation. *TSE* 38, 2 (2012), 425–438.
[42] E. Kocaguneli, T. Menzies, and E. Mendes. 2015. Transfer learning in effort estimation. *ESE* 20, 3 (01 Jun 2015), 813–843. https://doi.org/10.1007/s10664-014-9300-5
[43] M. Korte and D. Port. 2008. Confidence in software cost estimation results based on MMRE and PRED. In *PROMISE'08*. 63–70.
[44] J. Krall, T. Menzies, and M. Davies. 2015. GALE: Geometric Active Learning for Search-Based Software Engineering. *IEEE Transactions on Software Engineering* 41, 10 (Oct 2015), 1001–1018. https://doi.org/10.1109/TSE.2015.2432024
[45] R. Krishna, T. Menzies, and W. Fu. 2016. Too Much Automation? The Bellwether Effect and Its Implications for Transfer Learning. In *IEEE/ACM ICSE (ASE 2016)*. https://doi.org/10.1145/2970276.2970339
[46] W. B. Langdon, J. Dolado, F. Sarro, and M. Harman. 2016. Exact mean absolute error of baseline predictor, MARP0. *IST* 73 (2016), 16–18.
[47] R. Olshen C. Stone L.Breiman, J. Friedman. 1984. *Classification and Regression Trees*. Wadsworth.
[48] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering* 34, 4 (July 2008), 485–496. https://doi.org/10.1109/TSE.2008.35
[49] Y. Li, M. Xie, and T. N. Goh. 2009. A study of project selection and feature weighting for analogy based software cost estimation. *JSS* 82, 2 (2009), 241–252.
[50] E. Mendes, I. Watson, C. Triggs, and S. Mosley, N. Counsell. 2003. A comparative study of cost estimation models for web hypermedia applications. *ESE* 8, 2 (2003), 163–196.
[51] T. Menzies, Z. Chen, J. Hihn, and K. Lum. 2006. Selecting best practices for effort estimation. *TSE* 32, 11 (2006), 883–895.
[52] T. Menzies, J. Greenwald, and A. Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1 (Jan 2007), 2–13. https://doi.org/10.1109/TSE.2007.256941
[53] T. Menzies, Y. Yang, G. Mathew, B.W. Boehm, and J. Hihn. 2017. Negative Results for Software Effort Estimation. *ESE* 22, 5 (2017), 2658–2683. https://doi.org/10.1007/s10664-016-9472-2
[54] Leandro L Minku and Xin Yao. 2013. Ensembles and locality: Insight on improving software effort estimation. *Information and Software Technology* 55, 8 (2013), 1512–1528.
[55] N. Mittas and L. Angelis. 2013. Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm. *IEEE Trans SE* 39, 4 (April 2013), 537–551. https://doi.org/10.1109/TSE.2012.45
[56] Julie Moeyersoms, Enric Junqué de Fortuny, Karel Dejaeger, Bart Baesens, and David Martens. 2015. Comprehensible Software Fault and Effort Prediction. *J. Syst. Softw.* 100, C (Feb. 2015), 80–90. https://doi.org/10.1016/j.jss.2014.10.032
[57] R. Moser, W. Pedrycz, and G. Succi. 2008. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *30th ICSE*. https://doi.org/10.1145/1368088.1368114
[58] V. Nair, A. Agrawal, J. Chen, W. Fu, G. Mathew, T. Menzies, L. L. Minku, M. Wagner, and Z. Yu. 2018. Data-Driven Search-based Software Engineering. In *MSR*.
[59] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding faster configurations using flash. *arXiv preprint arXiv:1801.02175* (2018).
[60] J. Nam and S. Kim. 2015. Heterogeneous Defect Prediction. In *10th FSE (ESEC/FSE 2015)*. https://doi.org/10.1145/2786805.2786814
[61] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. 1996. *Applied linear statistical models*. Vol. 4. Irwin Chicago.
[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, and J. Vanderplas. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12, Oct (2011), 2825–2830.
[63] T. Peters, T. Menzies, and L. Layman. 2015. LACE2: Better Privacy-Preserving Data Sharing for Cross Project Defect Prediction. In *ICSE*, Vol. 1. 801–811. https://doi.org/10.1109/ICSE.2015.92
[64] D. Port and M. Korte. 2008. Comparative studies of the model evaluation criterion mmre and pred in software cost estimation research. In *ESEM'08*. 51–60.
[65] J. R. Quinlan. 1992. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, Vol. 92. Singapore, 343–348.
[66] G. S. Rao, C. V. P. Krishna, and K. R. Rao. 2014. Multi objective particle swarm optimization for software cost estimation. In *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India-Vol I*. Springer, 125–132.
[67] Federica Sarro and Alessio Petrozziello. 2018. Linear Programming as a Baseline for Software Effort Estimation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2018), to appear.
[68] F. Sarro, A. Petrozziello, and M. Harman. 2016. Multi-objective software effort estimation. In *ICSE*. ACM, 619–630.
[69] Abdel Salam Sayyad and Hany Ammar. 2013. Pareto-optimal search-based software engineering (POSBSE): A literature survey. In *RAISE'13*. IEEE, 21–27.
[70] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. 2013. On the value of user preferences in search-based software engineering: a case study in software product lines. In *ICSE'13*. IEEE Press, 492–501.
[71] M. Shepperd. 2007. Software project economics: a roadmap. In *2007 Future of Software Engineering*. IEEE Computer Society, 304–315.
[72] M. Shepperd, M. Cartwright, and G. Kadoda. 2000. On building prediction systems for software engineers. *EMSE* 5, 3 (2000), 175–182.
[73] M. Shepperd and S. MacDonell. 2012. Evaluating prediction systems in software project estimation. *IST* 54, 8 (2012), 820–827.
[74] M. Shepperd and C. Schofield. 1997. Estimating software project effort using analogies. *TSE* 23, 11 (1997), 736–743.
[75] Brajesh Kumar Singh and AK Misra. 2012. Software effort estimation by genetic algorithm tuned parameters of modified constructive cost model for nasa software projects. *International Journal of Computer Applications* 59, 9 (2012).
[76] Liyan Song, Leandro L. Minku, and Xin Yao. 2013. The Impact of Parameter Tuning on Software Effort Estimation Using Learning Machines. In *PROMISE'13*. ACM, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/2499393.2499394
[77] Krishnamoorthy Srinivasan and Douglas Fisher. 1995. Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering* 21, 2 (1995), 126–137.
[78] E. Stensrud, T. Foss, B. Kitchenham, and I. Myrtveit. 2003. A further empirical investigation of the relationship of MRE and project size. *ESE* 8, 2 (2003), 139–161.
[79] R. Storn and K. Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over cont. spaces. *JoGO* 11, 4 (1997), 341–359.
[80] M. Tan, L. Tan, and S. Dara. 2015. Online Defect Prediction for Imbalanced Data. In *ICSE*.
[81] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. 2016. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *38th ICSE*. https://doi.org/10.1145/2884781.2884857
[82] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. 2018. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2794977
[83] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *KDD'13*. ACM, 847–855.
[84] F. Walkerden and R. Jeffery. 1999. An empirical study of analogy-based software effort estimation. *ESE* 4, 2 (1999), 135–158.
[85] Jianfeng Wen, Shixian Li, Zhiyong Lin, Yong Hu, and Changqin Huang. 2012. Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology* 54, 1 (2012), 41–59.
[86] P. A. Whigham, C. A. Owen, and S. G. Macdonell. 2015. A Baseline Model for Software Effort Estimation. *TOSEM* 24, 3, Article 20 (May 2015), 11 pages. https://doi.org/10.1145/2738037
[87] Gerhard Wittig and Gavin Finnie. 1997. Estimating software development effort with connectionist models. *Information and Software Technology* 39, 7 (1997), 469–476.
[88] D. H. Wolpert. 1996. The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation* 8, 7 (1996), 1341–1390. https://doi.org/10.1162/neco.1996.8.7.1341
[89] Q. Zhang and H. Li. 2007. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation* 11, 6 (Dec 2007), 712–731. https://doi.org/10.1109/TEVC.2007.892759