Sequential Model Optimization for Software Process Control

Tianpei Xia, Jianfeng Chen, Rui Shu, Tim Menzies
txia4@ncsu.edu,jchen37@ncsu.edu,rshu@ncsu.edu,timm@ieee.org
North Carolina State University
Raleigh, United States

ABSTRACT

Many methods have been proposed to estimate how much effort is required to build and maintain software. Much of that research assumes a "classic" waterfall-based approach rather than contemporary agile projects. Also, much of that work tries to recommend a single method– an approach that makes the dubious assumption that one method can handle the diversity of software project data.

To address these drawbacks, we apply a configuration technique called "ROME" (Rapid Optimizing Methods for Estimation), which uses sequential model-based optimization to find what combination of techniques works best for a particular data set. In this paper, we test this method using data from 1161 classic waterfall projects and 446 contemporary agile projects (from Github).

We find that ROME achieves better performance (in terms of magnitude of relative error and standardized accuracy) than existing state-of-the-art methods for both classic and contemporary problems. From this work, we conclude that it is not best to recommend *one* method for estimation. Rather, it is better to search through a wide range of different methods to find what works best for local data.

To the best of our knowledge, this is the largest effort estimation experiment yet attempted and the only one to test its methods on classic and contemporary projects.

KEYWORDS

Effort Estimation, COCOMO, Hyperparameter Tuning, Regression Trees, Sequential Model Optimization

1 INTRODUCTION

Estimating development effort can be difficult [26], and incorrect estimates can harm the outcome of software projects [34, 35, 58, 61]. This is true for both classic waterfall projects and contemporary agile projects. In the case of large government waterfall projects, it is required that the proposed budget is double-checked by some estimation model [37]. In the case of agile projects (where resources are adjusted as the work progresses), when developers are forced to build their software using too few resources, then the first thing that is usually jettisoned is the software quality task [36]. When monitoring for "project health", the managers of large open source distributions will shun such distressed projects (so that software will not get widely used [62]).

Much of the prior work on effort estimation has focuses on classic waterfall projects [1, 7, 56, 57] (where the estimate is required before the project commences). There are many challenges with adapting classic waterfall estimation to contemporary agile projects:

- Firstly, the goal of estimation has to change.
- Secondly, the way we select estimation models has to mature.

Regarding *the goal of estimation*, in classic waterfall estimation, the goal is to get the budget right, before any work starts. However, when estimating contemporary agile projects, the goal is different. Some agile projects are fully staffed by large fluctuating volunteer groups working in their spare time. For those groups, delivering software is less a matter of project management as it is the enthusiasm of their user community for their product. But there are other kinds of contemporary agile projects that do need good estimation methods. Increasingly, commercial companies spend developer salary to maintain and improve agile open source projects. Braiek et al. document contemporary agile open source data mining products that are developed by commercial companies as a way to attract (and hold) more customers on their platforms [9]. Bird et al. report the surprising finding that, certain high profile contemporary agile projects are *not* built by a diverse open source community from around the globe. Instead, that software is mostly updated at two commercial sites during normal office hours [5]. Krishna et al. report that IBM asked for help to adjust, on a month-to-month basis, the staffing allocations for their suite of contemporary agile open source tools (which IBM maintains for its client base) [30]. That is, whereas classic waterfall projects need estimates of for future work, the managers of these agile projects need estimates to know if their current staff allocation is sufficient for the tasks at hand [48].

Regarding *how we select estimation models*, we note that as software engineering gets more diverse, it becomes less and less likely that any single estimation model will work across all those projects. So instead of recommending a particular estimation model:

> *To find what works best for local data, we need ways to survey a wide range of different estimation models.*

For this surveying task, we recommend a new approach called "ROME" (Rapid Optimizing Methods for Estimation), which uses sequential model-based optimization to explore possible configurations for an effort estimator. In that process, the results from exploring a few configurations are used to guess results across the remaining configurations. The configuration that yields the best guess (lowest error) is then actually applied, after which ROME updates its knowledge of what is a good configuration.

To evaluate ROME, we ask these research questions:

RQ1: Is effort estimation effective for classic waterfall and contemporary agile projects? Effort estimation needs to be effective to use in real-world software tasks. According to Sarro et al, industrial competitive predictions of project effort usually lie within 0.3 and 0.4 of the actual value [52]. We provide evidence that the performance of our method in classic waterfall and contemporary agile data sets lies within the currently claimed industrial human-expert-based thresholds, thereby demonstrating that:

Submitted to ICSE'2020, August 23, 2019.

> ***Lesson1:*** Effort estimation is effective on both classic waterfall projects and contemporary agile projects.

RQ2: Does ROME have better performance than existing estimation methods? To answer this question, we study 1161 classic waterfall projects and 446 contemporary agile projects (from Github). ROME's performance is compared to some standard effort estimators as well as two recent prominent systems: Whigham et al.'s ATML tool from TOSEM'15 [63] as well as Sarro et al's LP4EE tool from TOMSE'18 [51]. We find that:

> ***Lesson2:*** ROME generated best estimates in most cases.

Here, we measured "best" using the measures that are standard in the field; i.e. MacDonell"s and Shepperd's standardized accuracy measure [55] and the MRE measure used by other researchers [52].

RQ3: ***When we have new effort data sets, what configurations to use for effort estimation tasks?*** The tool we call ROME is a combination of sequential model optimization and CART. For pragmatic reasons, practitioners prefer a simpler rig. Hence we are often asked if the optimizer is required or if, usually, certain configurations generally work well across all data sets. To answer this question, we counted what configurations were selected in the experiments of this paper. In those counts, we saw:

> ***Lesson3:*** There is no clear pattern in what configurations are needed. Hence, model optimization needs to be repeated for each new data set.

RQ4: ***When we apply ROME on effort data sets, can it help us to find the most important features of the data?*** One feature of ROME is that if a feature is not informative, it will be dropped in the generated estimation model. Hence, when we say "most important", we really mean the "mostly used in our methods". Looking across our results, we find that certain size features are always used, but always in combination with a wide variety of other features. Hence:

> ***Lesson4:*** There are no "best" set of effort estimation features since each project uses these features in a different way.

Overall the contributions of this paper are:

- Our results clearly deprecate the use of off-the-shelf estimation tools. Based on the **RQ3** and **RQ4** results, practitioners should use tools like ROME to find the features/modeling options that work best for their local data.
- To the best of our knowledge, this is the largest effort estimation experiment yet reported (we use data from 1161 classic waterfall projects and 446 contemporary agile projects).
- Using that data, this paper makes a clear demonstration that effort estimation works well for classic waterfall projects as well as contemporary agile projects. In terms of the practicality of effort estimation research, this is a landmark result since it means that decades of research into effort estimation of classic waterfall projects can now be applied to contemporary agile software systems.
- We offer a new benchmark in effort estimation.
- We offer an open source version of ROME[1].

[1]https://github.com/arennax/effort_rome

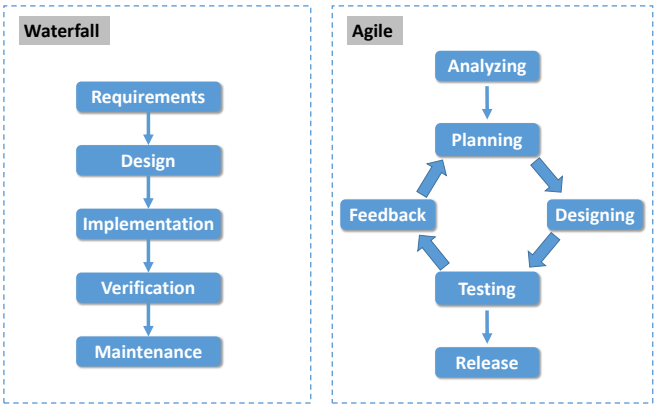

Figure 1: Waterfall vs. Agile in Software Development

The last one is more of a system contribution than a research contribution. Nevertheless, in terms of support the reproduction and extension of our results, this contribution is useful.

The rest of this paper is structured as follows. The next section discusses the history and different methods for effort estimation tasks. This is followed by a description of our experimental data, methods and the results. After that, a discussion section explores open issues with this work.

2 BACKGROUND

Software effort estimation is the procedure to provide approximate advice on how much human effort is required to plan, design and develop a software project. Usually, this human effort is expressed in terms of hours, days or months of human work. Since software development is a highly dynamic and fluid process, any estimate can only be approximate. Still, doing estimation is necessary since it is important to allocate resources properly in software projects to avoid waste. In some cases, improper allocation of funding can cause a considerable waste of resource and time [12, 21, 23, 49].

Much effort estimation work assumes a classic waterfall model [1, 7, 56, 57], first documented by Royce et al. in 1970 [50]. In this approach, project teams move to the next phase of development or testing if the previous step successfully completes. Estimation happens before the coding started. Further, once the funds are allocated, there is little opportunity to change that allocation.

Currently, the dominating software development style is agile model (first documented by Edmonds et al. in 1974 [14]). Agile uses continuous iteration of development and testing. Unlike the Waterfall model, development and testing activities are concurrent. This allows more communication between customers, developers, managers, and testers. Figure 1 contrasts these two models.

Having said that the agile style dominates, we also hasten to add that waterfall projects still exist and still needs effort estimation. This is particularly true in the case of large government or military software contracts, especially when their funding comes from legislation. For such projects, funds have to be allocated before the work starts. Also, as said in the introduction, for such large government waterfall projects, it is often required that the proposed budget is double-checked by some estimation model [37]. For these reasons:

Effort estimation methods need to support both *classic waterfall projects* and *contemporary agile projects.*

Effort estimation in software development can be categorized into human-based and algorithm-based methods [28, 53]. In this paper we focus on algorithm-based methods since they are preferred when estimates have to be audited or debated (these methods are explicit and available for inspection). To understand the range of possible estimates, we can run the algorithm as many times as necessary, which may not be applicable by using human-based methods. Algorithm-based methods can have comparable performance to human-based ones. Jørgensen et al. indicates that even very strong advocates of human-based methods acknowledge that algorithm-based methods are useful for learning the uncertainty about particular estimates [25].

Algorithm-based methods have been widely explored in the past few decades including classic model like COCOMO and more recent proposals like ATLM [63] and LP4EE [51].

2.1 COCOMO

COCOMO (the COnstructive COst MOdel) is a procedural cost estimate model for software projects proposed by Boehm et al. based on LOC (number of Lines of Code). It is often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time and quality. In late 1970s, Boehm was able to gather 63 project data points that could be published and to extend the model to include alternative development modes that covered other types of software such as business data processing. The resulting model was called the Constructive Cost Model, or COCOMO, and was published along with the data in the book Software Engineering Economics [8]. In this first version model (COCOMO-I), project attributes were scored using just a few coarse-grained values (very low, low, nominal, high, very high). These attributes are *effort multipliers* where a off-nominal value changes the estimate by some number greater or smaller than one. In COCOMO-I, all attributes (except KLOC) effect effort linearly.

Boehm created a consortium for industrial organizations after COCOMO was released. It collected information on 161 projects from commercial, aerospace, government, and non-profit organizations. Based on an analysis of those 161 projects, new attributes called *scale factors* were added to the original model, which had an *exponential impact* on effort. Using the new data, Boehm et al. developed COCOMO-II model that map the project descriptors (very low, low, etc.) into the specific values [7]:

$$effort = a \prod_i EM_i * KLOC^{b+0.01 \sum_j SF_j} \quad (1)$$

Inside this equation, a, b are the *local calibration* parameters (with default values of 2.94 and 0.91). *EM* stands for effort multipliers, and *SF* are scale factors. Boehm offers a simple linear time *local calibration* procedure [7] to update these defaults using the local training data. The calculated *effort* measures "development months" where one month is 152 hours of work (and includes development and management hours). For details about COCOMO attributes, see tiny.cc/ccm_attr.

2.2 Beyond COCOMO

For modern software development, it is necessary to develop new technique and make changes to improve COCOMO-style estimation. Robles et al. report that more companies are turning to open source software projects (e.g. Agile software projects on Github), other than traditional waterfall style projects for their new business strategy [48]. For old parametric estimating models like COCOMO, Shepperd et al. found it is difficult to determine some of their features for the estimations [53]. COCOMO measured software size by using LOC (line of code), but this feature is not available during the coding procedure, and it is difficult to make comparisons between different programming languages that may take varying numbers of statements to perform a given function. Jeffery et al. indicated that parametric model like COCOMO need to be calibrated to be used effectively in their study [24], which is another evidence that old parametric estimating models like COCOMO may not be appropriate for newer tasks.

2.2.1 ATLM. Automatically Transformed Linear Model (ATLM) is a multiple linear regression model proposed by Whigham et al. [63]. It calculates the effort as:

$$effort = \beta_0 + \sum_i \beta_i \times a_i + \varepsilon_i$$

where a_i is explanatory attribute and ε_i is error to the actual value. The prediction weight β_i is determined using least square error estimation [43]. 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" [22] 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.

2.2.2 LP4EE. Linear Programming for Effort Estimation (LP4EE) is a newly developed method by Sarro et al. [51], it aims to achieve the best outcome from a mathematical model with a linear objective function 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 minimizes the Sum of Absolute Residual (SAR), when a new project is presented to the model, LP4EE predicts the effort as

$$effort = a_1 * x_1 + a_2 * x_2 + \ldots + a_n * x_n$$

where x_i is the value of a given project feature and a_i is the corresponding coefficient evaluated by linear programming. Sarro et al. propose LP4EE 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[51].

2.2.3 Machine Learning-based Effort Estimators. Many machine learning algorithms have been used for software effort estimation. Random Forest [10] and Support Vector Regression [11] are such instances of regression methods. Random Forest (RF) is an ensemble learning method for regression (and classification) tasks that builds a set of trees when training the model. To make the final prediction , it uses the mode of the classes (classification) or mean prediction (regression) of the individual trees. Support Vector Regression (SVR) uses kernel functions to project the data onto a new hyperspace where complex non-linear patterns can be simply represented. Another learning approach is to use a $K = 5$ nearest-neighbor method [56]. For each test instance, KNN then selects k similar analogies out of a training set. The resultant prediction is the the mean of the class value of those k neighbors.

Some algorithm-based estimators use regression trees such as CART [32]. CART is a tree learner that divides a data set, 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. Note that we choose the tuning range by using advice from Fu et al. [19].

Table 1: CART's parameters.

Parameter	Type	Default	Tuning Range	Description
max_feature	numerical	None	[0.01, 1]	Number of features to consider when looking for the best split
max_depth	numerical	None	[1, 12]	The maximum depth of the decision tree
min_sample_split	numerical	2	[0, 20]	Minimum samples required to split internal nodes
min_sample_leaf	numerical	1	[1, 12]	Minimum samples required to be at a leaf node

Before moving on from CART, we note a detail that will become important when we discuss our third research question. Note that *decreasing* max_depth and *increasing* min_sample_leaf will result in smaller trees. In such smaller trees, few features will appear; specifically, on those features that most minimize the standard deviation of the target class. In the experimental rig described below, many times, we will generate trees using different settings to Table 1. By counting the the number of times a feature appears in these trees, we can infer what features are the most important to effort estimation.

2.2.4 Hyperparameter Optimization. Hyperparameters control the algorithm policies of the learners. Choosing appropriate hyperparameters plays a critical role in the performance of machine learning models. Tuning hyperparameters is the process of searching the most optimal hyperparameter options for machine learning models [4, 17]. Some popular methods to tune the hyperparameters are grid search and differential evolution.

Grid search [3] is a technique that using brute force of all combinations for hyperparameters. Although the Grid search method is a simple algorithm to use, it suffers if data have high dimensional space called the "curse of dimensionality". Previous work has shown that grid search might also miss important optimizations [20] or run needlessly slowly since, often, only a few of the tuning parameters really matter [2].

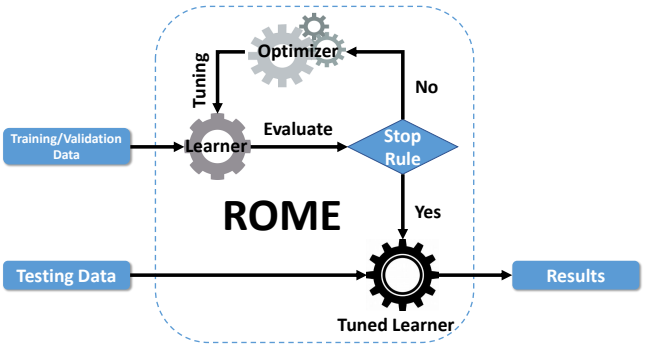

Figure 2: ROME's architecture

Differential evolution (DE) [60]. 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, we replace the old tuning x_k with y_k where $y_k = a_k + f \times (b_k - c_k)$ 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).

Bayesian optimization [45] works by assuming the unknown function was sampled from a Gaussian Process and maintains a posterior distribution for this function as observation are made. However, it might not be well-suited for optimization over continuous domains with large number of dimensions [18].

2.2.5 ROME. Standard hyperparameter optimization with DE or Baysian optimization can be a tedious and time consuming task [20]. Utilizing sequential model optimization, FLASH terminates after just a few dozen executions of different learner control parameters. ROME uses FLASH [42] to tune CART [32].

As shown in Figure 2, ROME has a learning layer and a optimizing layer. When training data arrives, the estimator in the learning layer is being trained, and the optimizer in optimizing layer provides better hyperparameters to the learner to help improve the performance of estimators. Such trained learner will be evaluated on the validation data afterwards. Once some stopping criteria is met, the generated learner is then passed to the test data for final testing.

When we design ROME, we want it to be as flexible as possible. It was simple to "pop the top" and replace the optimizing layer with another optimizer. In this paper, ROME uses FLASH [42] as the optimizer. Since the result from that initial study were promising, we paused further experimentation to record those results. In future work, we will try other optimizers.

FLASH comes from research into software configuration. One of the new insights that leads to this paper was that "configuration" is a synonym for "hyperparameter optimziation". Hence,

Table 2: Some data from the NASA10 data set (one row per project). For a definition of the terms in row1 ("prec", "flex", "resl" etc.) see tiny.cc/ccm_attr. As to the different columns, scale factors change effort exponentially while effort multipliers have a linear impact on effort. Any effort multiplier with a value of "3" is a *nominal* value; i.e. it multiplies the effort by a multiple of 1.0. Effort multipliers above and below "3" can each effect project effort by a multiple ranging from 0.7 to 1.74.

prec	flex	resl	team	pmat	rely	cplx	data	ruse	time	stor	pvol	acap	pcap	pcon	aexp	plex	ltex	tool	sced	site	docu	kloc	months
2	2	2	3	3	4	5	4	3	5	6	4	4	4	3	4	3	3	1	3	4	4	77	1830
2	2	2	3	3	5	5	2	3	5	6	2	4	3	3	2	1	2	2	3	4	4	24	648
2	2	2	3	3	4	5	3	3	5	5	4	3	3	3	3	2	2	1	3	4	4	23	492
2	2	3	3	2	4	4	3	2	3	3	4	3	3	3	3	3	4	2	3	5	3	146	3292
2	3	3	5	3	3	4	3	2	4	4	2	5	5	4	5	1	5	3	3	6	3	113	1080

scale factors | **effort multipliers** | **size** | **effort**

Table 3: Descriptive Statistics of the classic effort data sets. Terms in *red* are removed from this study, for reasons discussed in the text.

	feature	min	max	mean	std
kemerer	Langu.	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

hyperparameter-optimization-via-configuration tools has not previously been explored in the literature. Also, prior to this paper, such optimizers have not been used for effort estimation.

FLASH is a sequential model-based optimizer [3] (also known in the machine learning literature as an *active learner* [13] or, in the statistics literature as *optimal experimental design* [44]). No matter whatever the name is, the idea behind it is the same: reflect on the model built so far to find the next best example to evaluate. To tune a learning algorithm, FLASH explores N possible tunings as follows:

(1) Set the evaluation budget b. Based on prior work [42], we used $b = 200$.
(2) Run the learning algorithm with $n = 20$ to randomly select tunings.
(3) Build an *archive* of n examples holding pairs of parameter settings and their resulting performance scores.
(4) Using that archive, learn a *surrogate* to predicts performance. As per the methods of Nair et al. [42], our surrogates come from CART [32].
(5) Use the surrogate to guess M performance scores where $M < N$ and $M \gg n$ parameter settings. Note that this step is very fast because all required is to run M vectors downwards some very small CART trees.
(6) use a *selection function* to select the most "interesting" setting. We use the setting whose prediction has the smallest predicted error.
(7) Collect performance scores by evaluating "interesting" using the data miners. Set $b = b - 1$.
(8) Add "interesting" to archive. If $b > 0$, goto step 4.
(9) Else, halt.

In summary, given what we already know about the tunings (represented in a CART tree), FLASH finds the potentially best tunings (in Step 6); then evaluate the performance (in Step 7); then update the model with the results of that evaluation.

3 EMPIRICAL STUDY

3.1 Data

To evaluate the proposed ROME framework comprehensively, we test it out on both COCOMO-style data and non COCOMO-style

Table 4: Descriptive Statistics of the Github data sets, for details of each feature, see tiny.cc/condatadetail. Terms in *red* are removed from this study, for reasons discussed in the text.

	feature	min	max	mean	std
java_init	*LOC*	5	31	14.3	7.5
	EI	39	450	186.6	136.8
	EO	100	2307	999.1	589.6
	AFP	97	2284	993.9	597.4
	APEX	39	450	186.6	136.8
	LPEX	100	2307	999.1	589.6
	FILES	97	2284	993.9	597.4
	Effort	23	1107	219.2	263.1
webshop_init	*LOC*	5	31	14.3	7.5
	EI	39	450	186.6	136.8
	EO	100	2307	999.1	589.6
	AFP	97	2284	993.9	597.4
	APEX	39	450	186.6	136.8
	LPEX	100	2307	999.1	589.6
	FILES	97	2284	993.9	597.4
	Effort	23	1107	219.2	263.1

	feature	min	max	mean	std
java_incre	*LOC*	5	31	14.3	7.5
	EI	39	450	186.6	136.8
	EO	100	2307	999.1	589.6
	AFP	97	2284	993.9	597.4
	APEX	39	450	186.6	136.8
	LPEX	100	2307	999.1	589.6
	FILES	97	2284	993.9	597.4
	Effort	23	1107	219.2	263.1
webshop_incre	*LOC*	5	31	14.3	7.5
	EI	39	450	186.6	136.8
	EO	100	2307	999.1	589.6
	AFP	97	2284	993.9	597.4
	APEX	39	450	186.6	136.8
	LPEX	100	2307	999.1	589.6
	FILES	97	2284	993.9	597.4
	Effort	23	1107	219.2	263.1

	feature	min	max	mean	std
java_final	*LOC*	5	31	14.3	7.5
	EI	39	450	186.6	136.8
	EO	100	2307	999.1	589.6
	AFP	97	2284	993.9	597.4
	APEX	39	450	186.6	136.8
	LPEX	100	2307	999.1	589.6
	FILES	97	2284	993.9	597.4
	Effort	23	1107	219.2	263.1
webshop_final	*LOC*	5	31	14.3	7.5
	EI	39	450	186.6	136.8
	EO	100	2307	999.1	589.6
	AFP	97	2284	993.9	597.4
	APEX	39	450	186.6	136.8
	LPEX	100	2307	999.1	589.6
	FILES	97	2284	993.9	597.4
	Effort	23	1107	219.2	263.1

data. For COCOMO-style data, we include 216 projects from the SEACRAFT repository[2]; In Table 2, we list a sample of our data. This data set has been widely used to evaluate effort estimation methods for COCOMO-sytle data, which serves the same purpose to compare our proposed framework with the COCOMO-II procedure.

To test how ROME performs on non COCOMO data, we use 945 classic effort projects from the SEACRAFT, plus data collected from 446 projects on Github by Qi et al. [47] from Github, separately. See Table 3 and Table 4.

Note that some features of these non COCOMO style data sets are not used in our experiment because they are (1) naturally irrelevant to their effort values (e.g., ID, Syear), (2) unavailable at the prediction phase (e.g., duration, LOC), (3) highly correlated or overlap to each other (e.g., raw function point and adjusted function points). A data cleaning process is applied to solve this issue. Those removed features are highlighted as italic in Table 3 and Table 4.

3.2 Experimental Rig

In our experiments, we used a *M*N-way* cross-validation to split training and testing data for the estimators. That is, in 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. [41], we use $N = 3$ and $M = 20$ for our effort data sets.

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

In this experiment, we do not tune ATLM or LP4EE since they were designed to be used "off-the-shelf" (Whigham et al. [63] declare that one of ATLM's most important features is that it does not need tuning). We also do not tune SVR and RF since we treat them as baseline algorithm-based methods in our benchmarks (i.e. use default settings in scikit-learn for these algorithms). Here, we add KNN and CART with default settings, since these methods often appear in effort estimation literature [28, 37, 51, 52]. As to COCOMO-II, we applied Boehm's local calibration procedure [7] on the training data to adjust the (a, b) parameters of Equation 1. Lastly, we compared the performance of our optimizer FLASH with that of Differential Evolution [60]. Using advice from Storn and Fu et al. [19, 60], for DE we use $\{np, g, cr, generations\} = \{20, 0.75, 0.3, 10\}$.

[2]http://tiny.cc/seacraft

3.3 Performance Metrics

The results from each test set are evaluated in terms magnitude of the relative error (MRE) and Standardized Accuracy (SA). 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 actual effort:

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

MRE is criticized by some researchers as it is biased towards error underestimations [16, 27, 29, 46, 54, 59]. Nevertheless, we use it here since there exists known baselines for human performance in effort estimation expressed in terms of MRE [39].

Because of the issues with MRE, some researchers prefer the use of other (more standardized) measures, such as Standardized Accuracy (SA) [31, 55]. SA is based on Mean Absolute Error (MAE), which is defined in terms of

$$MAE = \frac{1}{N}\sum_{i=1}^{n} |RealEffort_i - EstimatedEffort_i|$$

where N is the number of projects used for evaluating the performance. 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 [55]. 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 [55].

Note that for MRE values, *smaller* are *better* and for SA values, *larger* are *better*. We use these since there are advocates for both in the literature. For example, Shepperd and MacDonell argue convincingly for the use of SA [55] (as well as for the use of effect size tests in effort estimation). Also in 2016, MRE was used by Sarro et al. [52] to argue their estimators were competitive with human estimates (which Molokken et al. [40] says lies within 30% and 40% of the true value).

Table 5: MRE (Magnitude of the Relative Error), lower values are better. For each row, the gray cells show the results that are statistically significantly better than anything else on that row (as judged by a Scott-Knot bootstrap test plus an A12 effect size test). If multiple treatments tied for "best", then there will be multiple gray cells in a row.

Dataset		Scikit-Learn				Tuned		New methods		COCOMO
		KNN	SVR	CART	RF	CART_DE	ROME	ATLM	LP4EE	COCOMO-II
classic	kemerer	0.56	0.59	0.55	0.50	0.32	0.37	0.76	0.54	N/A
	albrecht	0.45	0.56	0.53	0.46	0.32	0.33	1.40	0.44	N/A
	isbsg10	0.73	0.72	0.74	0.78	0.59	0.62	1.27	0.75	N/A
	finnish	0.64	0.74	0.57	0.57	0.48	0.42	0.87	0.63	N/A
	miyazaki	0.47	0.37	0.47	0.46	0.32	0.32	0.37	0.33	N/A
	maxwell	0.56	0.56	0.52	0.51	0.38	0.36	2.82	0.51	N/A
	desharnais	0.50	0.48	0.49	0.46	0.35	0.35	0.54	0.38	N/A
	kitchenham	0.39	0.60	0.49	0.43	0.38	0.34	1.06	0.38	N/A
	china	0.64	0.71	0.71	0.69	0.64	0.61	0.48	0.45	N/A
cocomo	cocomo10	0.67	0.86	0.33	0.30	0.30	0.28	2.49	0.32	0.60
	cocomo81	0.93	0.89	0.77	0.76	0.65	0.64	3.37	0.65	0.49
	nasa93	0.70	0.84	0.42	0.41	0.42	0.40	0.90	0.38	0.61
contemporary	java_init	0.52	0.65	0.33	0.32	0.28	0.28	0.67	0.31	N/A
	java_incre	0.62	0.72	0.42	0.41	0.32	0.28	0.45	0.33	N/A
	java_final	0.50	0.64	0.31	0.31	0.28	0.24	0.53	0.31	N/A
	webshop_init	0.41	0.43	0.50	0.40	0.27	0.28	0.67	0.25	N/A
	webshop_incre	0.44	0.44	0.44	0.40	0.28	0.28	0.53	0.38	N/A
	webshop_final	0.46	0.45	0.52	0.44	0.28	0.28	0.66	0.41	N/A

Table 6: SA (Standard Accuracy), higher values are better. Same format at Table 5; i.e. best results on are shown in gray.

Dataset		Scikit-Learn				Tuned		New methods		COCOMO
		KNN	SVR	CART	RF	CART_DE	ROME	ATLM	LP4EE	COCOMO-II
classic	kemerer	0.38	0.28	0.42	0.41	0.55	0.43	0.30	0.40	N/A
	albrecht	0.51	0.30	0.41	0.49	0.59	0.65	0.34	0.47	N/A
	isbsg10	0.28	0.25	0.20	0.22	0.33	0.30	0.30	0.22	N/A
	finnish	0.40	0.24	0.42	0.44	0.49	0.54	0.41	0.39	N/A
	miyazaki	0.45	0.41	0.41	0.46	0.53	0.53	0.50	0.52	N/A
	maxwell	0.39	0.30	0.37	0.44	0.51	0.55	-1.07	0.52	N/A
	desharnais	0.44	0.43	0.39	0.46	0.53	0.53	0.37	0.48	N/A
	kitchenham	0.47	0.32	0.34	0.41	0.40	0.44	-0.03	0.52	N/A
	china	0.28	0.21	0.12	0.21	0.27	0.30	0.12	0.32	N/A
cocomo	cocomo10	0.22	0.14	0.52	0.59	0.59	0.61	-0.13	0.29	0.30
	cocomo81	0.10	0.05	0.18	0.15	0.27	0.25	-1.14	0.20	0.27
	nasa93	0.08	0.14	0.36	0.37	0.36	0.41	0.34	0.41	0.30
contemporary	java_init	0.37	0.30	0.48	0.51	0.59	0.57	0.33	0.58	N/A
	java_incre	0.35	0.25	0.53	0.49	0.54	0.63	0.37	0.61	N/A
	java_final	0.43	0.30	0.57	0.57	0.62	0.67	0.41	0.62	N/A
	webshop_init	0.51	0.48	0.45	0.49	0.59	0.59	0.36	0.58	N/A
	webshop_incre	0.53	0.43	0.56	0.57	0.64	0.67	0.49	0.65	N/A
	webshop_final	0.47	0.44	0.49	0.52	0.61	0.61	0.44	0.58	N/A

3.4 Statistical Methods

From the cross-valuations, we report the *median* value, which is the 50th percentile of the test scores seen in the *M*N results*. For each data set, 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 [38].

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, we use a conjunction of *both* a 95% bootstrap significance test [15] *and* a A12 test for a non-small effect size difference in the distributions [37]. These tests were used since their non-parametric nature avoids issues with non-Gaussian distributions.

4 RESULTS

In this section, we present the experimental results. To answer the questions raised in Section 3, we conducted our experiments in the following sections:

- Compare performance of ROME with other methods on COCOMO-style data, classic effort data and Agile data sets collected from Github.
- Look into the internal structure of ROME and count the feature node in the tree it built.

RQ1: Is effort estimation effective for classic waterfall and contemporary agile projects?

Table 7: How often is each treatment seen to be best in Table 5 and Table 6.

Rank	Method	Win Times
1	CART_FLASH (ROME)	33/36
2	CART_DE	24/36
3	LP4EE	15/36
4	ATLM	3/36
5	RF	2/36
5	KNN	2/36
6	CART	1/36
7	SVR	0/36

To find if effort estimation method is effective, we ran ROME on both classic waterfall data sets and contemporary agile data sets. The performance value, in terms of MRE, is shown in Table 5. Recall that Sarro et al. argued that effective software projects have predictions of effort lie 0.3 and 0.4 of the actual value [52]. As can be observed, ROME obtained MRE value less than 0.40 in 15 out of all 18 cases. Also, in terms of applicability to contemporary methods, it is significant to note that all the MREs seen in the contemporary projects are under 0.30. That is, with these results, we can recommend ROME to the current practice, especially for the current contemporary agile projects. Overall:

> ***Lesson1:*** Effort estimation is effective on both classic waterfall projects and contemporary agile projects.

In terms of the practicality of effort estimation research, this is a landmark result since it means that decades of research into effort estimation of classic waterfall projects can now be applied to contemporary agile software systems.

RQ2: Does ROME have better performance than existing estimation methods?

To answer this question, we ran ROME and the other baseline methods *LP4EE*, *ATLM*, *KNN*, *SVR*, *CART*, *RF*, on classic waterfall data sets and contemporary agile data sets. MRE and SA scores for all our methods are shown in Table 5 and Table 6. Note the COCOMO-II is only applied to the COCOMO data sets (since the other data sets do not have the features needed by COCOMO).

In Table 5 and Table 6, each row shows results from a different data set. For each row, the gray cells show the results that are statistically significantly better than anything else on that row (as judged by a Scott-Knot bootstrap test plus an A12 effect size test). If multiple treatments tied for "best", then there will be multiple gray cells in a row. In those tables, better methods have more gray cells. Table 7 tallies the gray cells counts for all methods.

From the tallies of Table 7, we conclude that KNN, SVR, CART (untuned), RF and ATLM most often perform worse than anything else. While LP4EE does best for standard accuracy, in terms of MRE, it is not competitive against the tuned methods (CART, tuned by DE or FLASH). As to DE tuning CART, it performs better than LP4EE, but not as good as ROME (CART tuned by FLASH). In summary:

> ***Lesson2:*** ROME generated best estimates in most cases.

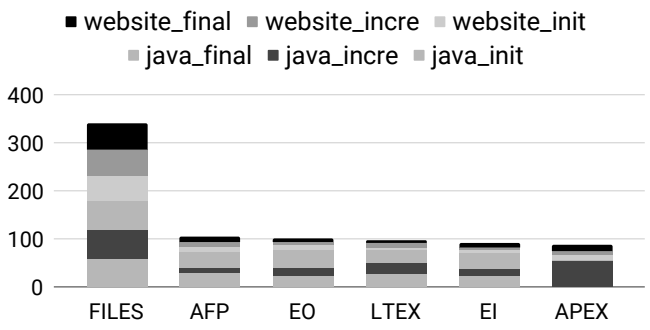

Figure 3: Selected features on Github data sets (for SA)

RQ3: When we have new effort data sets, what configurations to use for effort estimation tasks?

When we discuss this work with our 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. If the hyperparameter tunings for effort estimators 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 promising result since, in future when new data arrives, the complexities of tuning in ROME framework 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 our *M*N-way* cross validations (this table uses results from FLASH tuning CART since, as shown below, this usually leads to best MRE results). Note that in those results it is not true that across most data sets there is a setting that is usually selected (though min_samples_leaf less than 3 is often a popular setting). Accordingly, from Table 8, we concludes that there is much variations of the best tunings.

This finding is quite aligned with Fu et al. [19], where for software defect predictors, no best tunings for all tasks. Therefore, we always prefer to have a fast hyperparameter tuning technique to quickly find the best tuning for the current tasks. Our ROME framework is such of tool to use.

Since there are no "best" default settings for all, based on the results of Table 7, for similar effort estimation tasks, we say:

> ***Lesson3:*** There is no clear pattern in what configurations are needed. Hence, model optimization needs to be repeated for each new data set.

RQ4: When we apply ROME on effort data sets, can it help us to find the most important features of the data?

When CART's tuning parameters were described in §2.2.3, it was observed that when CART is run multiple times (with different hyperparameters) then it can be used to gauge the value of using a particular feature.

Figure 3 and Figure 4 show counts of how often a feature appeared in the trees found by ROME from the above experiments. Here, we only show data from the classic COCOMO and contemporary Github projects since the classic non-COCOMO data sets all use different features.

Table 8: Tunings discovered by hyperparameter selections (CART+FLASH, MRE results). 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
cocomo10	23	38	18	21	42	45	11	02	85	11	04	00	79	12	06	03
cocomo81	26	33	18	23	52	22	18	08	73	25	02	00	78	17	04	01
nasa93	31	27	28	24	47	29	18	06	55	21	11	13	53	27	14	06
java_init	13	14	57	26	36	32	24	08	64	19	12	05	71	21	07	01
java_incre	21	13	36	30	29	39	22	10	42	26	19	13	83	12	02	03
java_final	19	16	32	33	24	45	25	06	44	21	18	17	72	11	12	05
webshop_init	33	21	26	20	57	37	05	01	38	36	13	13	41	27	23	09
webshop_incre	26	16	27	31	31	30	23	16	41	22	23	16	44	35	17	04
webshop_final	42	16	22	20	33	22	17	28	59	27	09	05	62	31	04	03

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

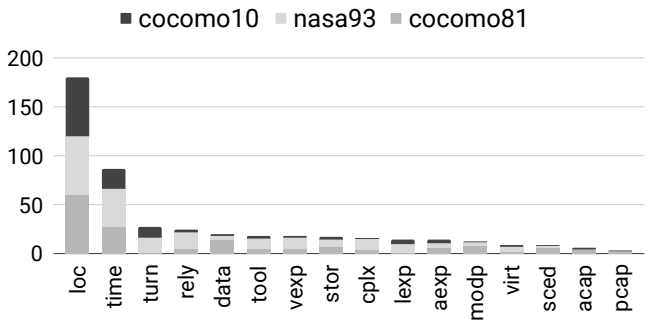

Figure 4: Selected feature on COCOMO data sets (for SA)

In Figure 3, the maximum number of times a feature can appear is 360 times (6 data sets, 3 way cross-validation, 20 repeats). One size attribute (FILES) appears very frequently but it is not often picked just by itself (we know this from the max_depth results of Table 8 where more often than not, CART used trees that held more than three features). But as to what other features were combined with FILES, that is clear. Looking at the AFP, EO, LTEX, EI, APEX results of Figure 3, we see that every other feature got used, sometimes.

A similar pattern appears in Figure 4. In this figure, the maximum number of times a feature can appear is 180 (3 data sets, 3 way cross-validation, 20 repeats). Once again, a size attribute (LOC) appears very frequently. But just as before, we see that every other feature got used, sometimes. Hence we say:

> ***Lesson4:*** There are no "best" set of effort estimation features since each project uses these features in a different way

As mentioned in the introduction, the results from **RQ3** and **RQ4** clearly deprecate the use of off-the-shelf estimation tools. Practitioners should use tools like ROME to find the features/modeling options that work best for their local data.

5 THREATS TO VALIDITY

Internal Bias: Many of our methods contain stochastic random operators. To reduce the bias from random operators, we repeated our 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 positive 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 we assert that this study suffers much less parameter bias than other studies.

Sampling Bias: While we tested ROME on both old COCOMO-Style data sets, classic effort data sets and newly collected open source data sets, it would be inappropriate to conclude that ROME tuning always perform better than others methods for other data sets. As researchers, what we can do to mitigate this problem is to carefully document our methods, publish our tools as open source software packages, and support the research community as they try to repeat/improve/refute our results on a broader set of data.

Another sampling bias comes from our choice of effort estimation technologies. Here, we compared ROME against technologies that are often seen in the effort estimation literature. We also took care to include in our comparisons two new and prominent methods recently published in TOSEM. But even with all that, this study has not explored all the effort estimation methods seen in the recent literature. To some extent, that was because no single paper can explore all algorithms. But also, sometimes we choose not to explore certain algorithms since they are out-of-scope for this study. For example, apart from LP4EE, Sarro et al. also offer another estimation method based on genetic algorithms called CoGEE [52]. That tool optimizes for multiple goals so it would not be a fair comparison to the tools used here (in defense of that decision, we note that the authors do not compare LP4EE to CoGEE in their TOSEM'18 paper).

6 CONCLUSIONS AND FUTURE WORK

Effort estimation methods need to support both classic waterfall projects and contemporary agile projects. For something as complex as the effort estimation of modern software projects, no single method works best. Instead, best results come from trying out a large number of candidate methods.

Sequential model-based optimization is an effective way to explore a range of configuration options for effort estimation. Our sequential optimizer came from research into software configuration. One of the new insights that leads to this paper was that "configuration" is a synonym for "hyperparameter optimziation". Hence, hyperparameter-optimization-via-configuration tools has not previously been explored in the literature. Also, prior to this paper, such optimizers have not been used for effort estimation.

When this optimizer was applied to 1161 classic waterfall projects and 446 contemporary agile projects we found that:

- **RQ1:** we could successfully apply the same optimization method to classic and contemporary projects. This is a significant result since it means that decades of effort estimation research can now be applied to contemporary agile systems.
- **RQ2:** those optimizations yielded better estimates than other methods studied here.
- **RQ3, RQ4:** different data sets need different hyperparameter optimizations and use different features. This means that we should deprecate the use of off-the-shelf estimation tools. Practitioners should use tools like ROME to find the features/modeling options that work best for their local data.

To the best of our knowledge, this is the largest effort estimation experiment yet reported.

As to future work, there is much to do. Clearly, we need to try other learners (e.g. neural nets, Bayesian learners or gradient boosting tree) and other optimizers (e.g. SMAC [33] or vZ [6]).

Also, now that we can use Github data for effort estimation, it is time to scale this analysis to the large number of projects available at that source. In the study of this paper, our **RQ3, RQ4** results found no stability in the features used or hyperparameter options selected. We conjecture that such stable conclusions may exist– if we look at much more project data.

More generally, in the study of effort estimation, most prior work only focus on comparisons of new estimation methods, but very less studies comparing latest technique with old classic models (e.g. COCOMO). Given the results of this paper, it is now important to validate newly proposed methods against different type of effort project data sets (e.g. Waterfall and Agile). Further, if we are mining current Github projects, we might be able to use the methods of this paper to go beyond mere effort estimation to look more predict better for other measures of project health (e.g. number of new contributors each month).

REFERENCES

[1] Oddur Benediktsson, Darren Dalcher, Karl Reed, and Mark Woodman. 2003. COCOMO-based effort estimation for iterative and incremental software development. *Software Quality Journal* 11, 4 (2003), 265–281.

[2] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13, 1 (Feb. 2012), 281–305.

[3] 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.

[4] Andre Biedenkapp, Katharina Eggensperger, Thomas Elsken, Stefan Falkner, Matthias Feurer, Matilde Gargiani, Frank Hutter, Aaron Klein, Marius Lindauer, Ilya Loshchilov, et al. 2018. Hyperparameter Optimization. *Artificial Intelligence* 1 (2018), 35.

[5] Christian Bird and Nachiappan Nagappan. 2012. Who? where? what? examining distributed development in two large open source projects. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 237–246.

[6] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *vZ*-an optimizing SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 194–199.

[7] Barry Boehm, Chris Abts, A Winsor Brown, Sunita Chulani, Bradford K Clark, Ellis Horowitz, Ray Madachy, Donald J Reifer, and Bert Steece. 2000. Cost estimation with COCOMO II. *ed: Upper Saddle River, NJ: Prentice-Hall* (2000).

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

[9] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The Open-closed Principle of Modern Machine Learning Frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 353–363. https://doi.org/10.1145/3196398.3196445

[10] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[11] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 27.

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

[13] S. Das, W. Wong, T. Dietterich, A. Fern, and A. Emmott. 2016. Incorporating Expert Feedback into Active Anomaly Discovery. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 853–858. https://doi.org/10.1109/ICDM.2016.0102

[14] Ernest A Edmonds. 1974. A process for the development of software for non-technical users as an adaptive system. *General Systems* 19 (1974), 215–218.

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

[16] 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.

[17] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. 2017. Forward and reverse gradient-based hyperparameter optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 1165–1173.

[18] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

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

[20] 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).

[21] S. Germano and A. Hufford. 2016. Finish Line to Close 25% of Stores and Replace CEO Glenn Lyon. https://www.wsj.com/articles/finish-line-to-close-25-of-stores-swaps-ceo-1452171033. (2016).

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

[23] V. Hazrati. 2011. IT Projects: 400% Over-Budget and only 25% of Benefits Realized. https://www.infoq.com/news/2011/10/risky-it-projects. (2011).

[24] D Ross Jeffery and G Low. 1990. Calibrating estimation tools for software development. *Software Engineering Journal* 5, 4 (1990), 215–221.

[25] M. Jørgensen and T. M. Gruschke. 2009. The impact of lessons-learned sessions on effort estimation and uncertainty assessments. *TSE* 35, 3 (2009), 368–383.

[26] C. F. Kemerer. 1987. An empirical validation of software cost estimation models. *CACM* 30, 5 (1987), 416–429.

[27] 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.

[28] 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.

[29] M. Korte and D. Port. 2008. Confidence in software cost estimation results based on MMRE and PRED. In *PROMISE'08*. 63–70.

[30] Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Tim Menzies. 2018. What is the connection between issues, bugs, and enhancements?: Lessons learned from 800+ software projects. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 306–315.

[31] W. B. Langdon, J. Dolado, F. Sarro, and M. Harman. 2016. Exact mean absolute error of baseline predictor, MARP0. *IST* 73 (2016), 16–18.

[32] R. Olshen C. Stone L.Breiman, J. Friedman. 1984. *Classification and Regression Trees*. Wadsworth.

[33] Marius Lindauer and Frank Hutter. 2017. Warmstarting of Model-based Algorithm Configuration. *CoRR* abs/1709.04636 (2017). arXiv:1709.04636 http://arxiv.org/abs/1709.04636

[34] S. McConnell. 2006. *Software estimation: demystifying the black art*. Microsoft press.

[35] Emilia Mendes and Nile Mosley. 2002. Further investigation into the use of CBR and stepwise regression to predict development effort for web hypermedia

applications. In *ESEM'02*. IEEE, 79–90.
[36] Tim Menzies, Markland Benson, Ken Costello, Christina Moats, Melissa Northey, and Julian Richardson. 2008. Learning better IV&V practices. *Innovations in Systems and Software Engineering* 4, 2 (01 Jun 2008), 169–183. https://doi.org/10.1007/s11334-008-0046-3
[37] 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
[38] 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
[39] K. Molokken and M. Jorgensen. 2003. A review of software surveys on software effort estimation. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* 223–230. https://doi.org/10.1109/ISESE.2003.1237981
[40] Kjetil Molokken and Magne Jorgensen. 2003. A review of software surveys on software effort estimation. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on.* IEEE, 223–230.
[41] 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*.
[42] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel. 2018. Finding Faster Configurations using FLASH. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2870895
[43] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. 1996. *Applied linear statistical models*. Vol. 4. Irwin Chicago.
[44] Fredrik Olsson. 2009. A literature survey of active machine learning in the context of natural language processing. (2009).
[45] Martin Pelikan. 1999. A simple implementation of the Bayesian optimization algorithm (BOA) in C++(version 1.0). *Illigal Report* 99011 (1999).
[46] 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.
[47] Fumin Qi, Xiao-Yuan Jing, Xiaoke Zhu, Xiaoyuan Xie, Baowen Xu, and Shi Ying. 2017. Software effort estimation based on open source projects: Case study of Github. *Information and Software Technology* 92 (2017), 145–157.
[48] Gregorio Robles, Jesús M González-Barahona, Carlos Cervigón, Andrea Capiluppi, and Daniel Izquierdo-Cortázar. 2014. Estimating development effort in free/open source software projects by mining software repositories: a case study of openstack. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 222–231.
[49] K. Roman. 2016. Federal government's Canada.ca project 'off the rails' https://www.cbc.ca/news/politics/canadaca-federal-website-delays-1.3893254. (2016).
[50] W Royce. 1970. The software lifecycle model (Waterfall Model). In *Proc. WESTCON*, Vol. 314.
[51] Federica Sarro and Alessio Petrozziello. 2018. Linear Programming as a Baseline for Software Effort Estimation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2018).
[52] F. Sarro, A. Petrozziello, and M. Harman. 2016. Multi-objective software effort estimation. In *ICSE*. ACM, 619–630.
[53] M. Shepperd. 2007. Software project economics: a roadmap. In *2007 Future of Software Engineering*. IEEE Computer Society, 304–315.
[54] M. Shepperd, M. Cartwright, and G. Kadoda. 2000. On building prediction systems for software engineers. *EMSE* 5, 3 (2000), 175–182.
[55] M. Shepperd and S. MacDonell. 2012. Evaluating prediction systems in software project estimation. *IST* 54, 8 (2012), 820–827.
[56] M. Shepperd and C. Schofield. 1997. Estimating software project effort using analogies. *TSE* 23, 11 (1997), 736–743.
[57] Alaa F Sheta. 2006. Estimation of the COCOMO model parameters using genetic algorithms for NASA software projects. *Journal of Computer Science* 2, 2 (2006), 118–123.
[58] I. Sommerville. 2010. *Software engineering*. Addison-Wesley.
[59] 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.
[60] 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.
[61] A. Trendowicz and R. Jeffery. 2014. Software project effort estimation. *Foundations and Best Practice Guidelines for Success, Constructive Cost Model–COCOMO pags* (2014), 277–293.
[62] Dindin Wahyudin, Khabib Mustofa, Alexander Schatten, Stefan Biffl, and A Min Tjoa. 2007. Monitoring the âĂIJhealthâĂİ status of open source web-engineering projects. *International Journal of Web Information Systems* 3, 1/2 (2007), 116–139.
[63] 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