BiLO-CPDP: Bi-Level Programming for Automated Model Discovery in Cross-Project Defect Prediction

Ke Li[♮], Zilin Xiang[♯], Tao Chen[§], Kay Chen Tan[♣*]
[♯]College of Computer Science and Engineering, UESTC, Chengdu, 611731, China
[♮]Department of Computer Science, University of Exeter, Exeter, EX4 4QF, UK
[§]Department of Computer Science, Loughborough University, Loughborough, LE11 3TU, UK
[♣]Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong SAR
k.li@exeter.ac.uk,ziling.xiang@hotmail.com,t.t.chen@lboro.ac.uk,kaytan@cityu.edu.hk

ABSTRACT

Cross-Project Defect Prediction (CPDP), which borrows data from similar projects by combining a transfer learner with a classifier, have emerged as a promising way to predict software defects when the available data about the target project is insufficient. However, developing such a model is challenge because it is difficult to determine the right combination of transfer learner and classifier along with their optimal hyper-parameter settings. In this paper, we propose a tool, dubbed `BiLO-CPDP`, which is the first of its kind to formulate the automated CPDP model discovery from the perspective of bi-level programming. In particular, the bi-level programming proceeds the optimization with two nested levels in a hierarchical manner. Specifically, the upper-level optimization routine is designed to search for the right combination of transfer learner and classifier while the nested lower-level optimization routine aims to optimize the corresponding hyper-parameter settings. To evaluate `BiLO-CPDP`, we conduct experiments on 20 projects to compare it with a total of 21 existing CPDP techniques, along with its single-level optimization variant and `Auto-Sklearn`, a state-of-the-art automated machine learning tool. Empirical results show that `BiLO-CPDP` champions better prediction performance than all other 21 existing CPDP techniques on 70% of the projects, while being overwhelmingly superior to `Auto-Sklearn` and its single-level optimization variant on all cases. Furthermore, the unique bi-level formalization in `BiLO-CPDP` also permits to allocate more budget to the upper-level, which significantly boosts the performance.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; **Software defect analysis**.

*K. Li, Z. Xiang and T. Chen contributed equally to this research.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416617

KEYWORDS

Cross-project defect prediction, transfer learning, classification techniques, automated parameter optimization, configurable software and toolComparison under Different Levels of Environmental Changes

ACM Reference Format:
Ke Li, Zilin Xiang, Tao Chen, and Kay Chen Tan. 2020. BiLO-CPDP: Bi-Level Programming for Automated Model Discovery in Cross-Project Defect Prediction. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416617

1 INTRODUCTION

Software defects are errors in code and its logic that cause a software product to malfunction or to produce incorrect/unexpected results. Given that software systems become increasingly ubiquitous in our modern society, software defects are highly likely to result in disastrous consequences to businesses and daily lives. For example, the latest *Annual Software Fail Watch* report from Tricentis[1] shows that, globally, software defects/failures affected over 3.7 billion people and caused $1.7 trillion in lost revenue.

One of the key reasons behind the prevalent defects in modern software systems is their increasingly soaring size and complexity. Due to the limited resource for software quality assurance and the intrinsic dependency among a large number of software modules, it is expensive, if not impossible, to rely on human efforts (e.g., code review) to thoroughly inspect software defects. Instead, it is more pragmatic to predict the defect-prone software modules to which software engineers are suggested to focus their limited software quality assurance resource. To this end, machine learning algorithms have been widely used to automate the process of defect prediction.

As discussed in [52], one of the keys to the success of defect prediction models is the amount of data available for model training. In practice, however, it is unfortunately not uncommon that such data is scarce or even unavailable. This can be attributed to the small size of the company and/or the targeted software project is the first of its kind. Cross project defect prediction (CPDP), which aims to predict defects in the a software project by leveraging experience (e.g., training data or hyper-parameters of trained defect prediction models) from other existing ones, has therefore become extremely appealing [15]. Unfortunately, partially due to the difference of the

[1]https://www.tricentis.com/resources/software-fail-watch-5th-edition/

data distribution between the source and the target projects, the performance of vanilla CPDP is not as promising as it was supposed to be [36]. Transfer learning, which is able to transfer knowledge across different domains, has shown to be able to overcome the aforementioned challenges (e.g., data scarcity and data distribution discrepancy) and has gradually become the main driving force for CPDP [29]. Generally speaking, the basic idea is to equip a machine learning classifier with a transfer learner that enables its ability to learn from other projects in model building.

There is *No Free Lunch* in defect prediction given that machine learning enabled defect prediction models often come with configurable and adaptable parameters (87% prevalent classifiers are with at least one parameter [43, 44]). The prediction accuracy on various software projects largely depends on the parameter settings of those defect prediction models [25, 26]. Furthermore, it becomes more complicated in CPDP because: 1) the configurable parameters is augmented by the transfer learner (85% widely used CPDP techniques require at least one parameter to setup in the transfer learner) thus lead to an enlarged search space; 2) there exist complex yet unknown interactions among the parameters of the classifier and those of the transfer learner (that is to say parameter optimization over either the classifier or the transfer learner alone may not lead to the overall optimal performance); and 3) the optimal selection of the combination of classifier and transfer learner is as important as parameter optimization but is unfortunately ignored in the current literature (most, if not all, CPDP models are designed with an *ad-hoc* combination of transfer learner and classifier, the performance of which is reported to be far from optimal [21]). Although there exist some prior works considering the hyper-parameter optimization for CPDP models [31, 34], they only consider the hyper-parameters associated with the classifier. As investigated in a latest empirical study [21], this practice is far from truly optimizing the performance of the underlying CPDP model while the settings of hyper-parameters of the transfer learner are more decisive.

Bearing the above considerations in mind, we propose a new tool, dubbed `BiLO-CPDP`, to automate the model discovery for CPDP tasks. It provides an unified perspective for the combinatorial selection of classifier and transfer learner, as well as their hyper-parameter optimization within the mathematical framework of bi-level programming, where two levels of nested optimization problems are formulated: the upper-level optimization problem is solved subject to the optimality of a lower-level optimization problem. Specifically, the upper-level optimization problem aims to identify the optimal combination of transfer learner and classifier from a given portfolio; while the lower-level optimization problem is dedicated to searching for the optimal parameter setting associated with the corresponding transfer learner and classifier. Note that a combination of transfer learner and classifier is not considered to be feasible for comparison unless the corresponding parameters have been optimized. In `BiLO-CPDP`, the upper-level optimization is formulated as a combinatorial optimization problem which is solved by the Tabu search [10] while the lower-level optimization is modeled as an expensive optimization problem with a limited budget to be solved by Tree-structured Parzen Estimator (TPE) [4], a state-of-the-art Bayesian optimization algorithm.

To evaluate the the effectiveness of `BiLO-CPDP` for automated model discovery in CPDP, we conduct experiments to compare it with 21 existing CPDP techniques, its single-level variant and `Auto-Sklearn` [8] — a state-of-the-art automated machine learning (AutoML) tool— over 20 distinct projects. The results fully demonstrate the overwhelming superiority of `BiLO-CPDP` over the others with statistical significance and a large effect size on all projects.

In summary, the key contributions of this paper are as follows:

- To the best of our knowledge, `BiLO-CPDP` is the first of its kind for automating CPDP from the perspective of bi-level programming. Given that `BiLO-CPDP` is not only able to automatically search the optimal combination of transfer learner and classifier, but also can set their appropriate hyper-parameter settings, it paves a new avenue for automated model discovery in CPDP.
- Through extensive experiments with 21 existing CPDP techniques, we show that `BiLO-CPDP` is the best on 14 out of 20 projects, and second to only one existing technique for another five. This fully demonstrates the effectiveness and importance brought by automatically choosing the appropriate transfer learner and classifier associated with their optimal hyper-parameter settings for CPDP.
- In terms of optimization problem formulation, on all projects, we show that the bi-level programming formulated in `BiLO-CPDP` is statistically better than hybridizing both combinatorial selection and parameter optimization as a single-level global optimization problem, which is perhaps a more conservative solution as used in, e.g., `Auto-Sklearn` [8].
- Interestingly, from our experimental results, we disclose that choosing the best combination of classifier and transfer learner (upper level) is more important than fully optimizing their parameters (lower level). Henceforth, given the limited resource for software quality assurance, it is beneficial to allocate more search budget to the upper-level optimization.

In the rest of this paper, Section 2 gives the background about bi-level programming. Section 3 delineates the algorithmic implementation of `BiLO-CPDP` step by step. The experimental setup is introduced in Section 4 and the results are analyzed in Section 5. Thereafter, Section 6 and 7 reviews the related works and discusses the threats to validity, respectively. Finally, Section 8 concludes this paper and threads some lights on future directions.

2 BI-LEVEL PROGRAMMING

Bi-level programming is a mathematical program within which one optimization problem is nested within another in a hierarchical manner [42]. It is ubiquitous in many real-world optimization and public/private sector decision-making problems where the realized outcome of any solution or decision taken by the upper-level authority (a.k.a. leader) to optimize their objectives is affected by the response of lower-level entities (a.k.a. follower), who seek to optimize their own outcomes. This is in principle similar to the Stackelberg games [47] in which a leader first makes its move and a follower maximizes the corresponding gain by taking the leader's move into account. It is interesting to note that the two levels of optimization problems are asymmetric in bi-level programming.

That is to say, the upper-level leader has the entire picture of optimization problems at both levels whereas the lower-level follower usually takes the decisions from the leader and then optimizes its own strategies.

The bi-level programming formulated in BiLO-CPDP can be mathematically defined as:

$$\begin{array}{ll} \underset{\mathbf{x}^u \in \Lambda^d \times \mathbb{R}^n, \mathbf{x}^l \in \mathbb{R}^n}{\text{maximize}} & F(\mathbf{x}^u, \mathbf{x}^{l*}) \\ \text{subject to} & \mathbf{x}^{l*} \in \text{argmax}\{f_{\mathbf{x}^u}(\mathbf{x}^l)\} \end{array}, \tag{1}$$

where $\mathbf{x}^u \in \Lambda^d \times \mathbb{R}^n$ and $\mathbf{x}^l \in \mathbb{R}^n$ denote the upper- and lower-level variables[2] while $F : \Lambda^d \times \mathbb{R}^n \rightarrow \mathbb{R}$ and $f_{\mathbf{x}^u} : \mathbb{R}^n \rightarrow \mathbb{R}$ are the upper-level and lower-level objective functions, respectively (details can be found in Section 3). A bi-level programming that involves nested optimization/decision-making tasks at both levels. For any given combination $\mathbf{x}^u$, there exists a $(\mathbf{x}^u, \mathbf{x}^{l*})$ pair where $\mathbf{x}^{l*}$ is an optimal (or near-optimal) response to $\mathbf{x}^u$ represents a feasible solution to the upper-level optimization problem given that it also satisfies the constraints therein.

3 BI-LEVEL PROGRAMMING FOR AUTOMATED CPDP MODEL DISCOVERY

The CPDP model building process consists of two intertwined parts: 1) transfer learning that augments data from different domains by selecting relevant instances or assigning appropriate weights to different instances; and 2) defect prediction model building based on adapted data. As reported in a latest research [21], the performance of a CPDP model largely depend on the combination of transfer learner and classifier along with their hyper-parameter settings. In light of this, the BiLO-CPDP proposed in this work was specifically designed to address such a problem. Through automatically discovering the best combination of transfer learner and classifier as well as their optimal hyper-parameter settings, BiLO-CPDP serves as an automatic tool that provides a *de nova* CPDP model discovery. In this section, we will delineate the architecture of BiLO-CPDP and the algorithmic details of its optimization routines at both levels.

3.1 Overview of BiLO-CPDP

The overall architecture of BiLO-CPDP is illustrated in Fig. 1 which consists of three key phases, i.e., *data pre-processing*, *optimization* and *performance validation*.

(1) **Data Pre-processing:** Given a raw dataset with $N > 1$ projects, software engineers are asked to specify which one is the target domain that serves as the target domain data while the remaining $N - 1$ projects are then used as the source domain data. In particular, all source domain data are used in the model training while a part of the target domain data is used as the hold-out set for the testing purpose. As the default in BiLO-CPDP, we use 10% of the target domain data for testing while the remaining 90% is for training. This is because some transfer learners considered in this work do need data from the target domain in training, e.g., MCWs [33]. For other transfer learners that can be trained independently to the target domain, we use all data for testing.

[2] $\Lambda^d \times \mathbb{R}^n$ means that the problem is a discrete combinatorial problem.

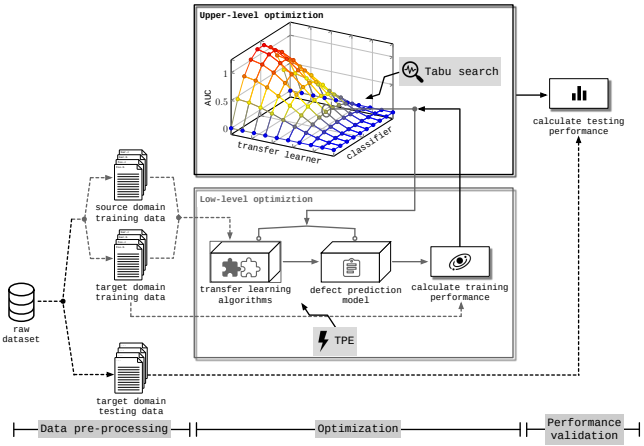

Figure 1: The overall architecture of BiLO-CPDP.

(2) **Optimization:** BiLO-CPDP models CPDP as a bi-level programming that not only identifies the most competitive combination of transfer leaner and classifier for the underlying CPDP task (tackled by the upper-level routine), but also equips the chosen CPDP model with the appropriate hyper-parameter settings (carried out by the lower-level routine). Since the resources for software quality assurance are often limited, the entire optimization process would inevitably be constrained under a computational budget of running time. In this regard, the unique bi-level programming formulated in BiLO-CPDP can in fact provide a fine-grained and flexible allocation of the budget between upper- and lower-level, whose effects will be investigated as part of the experimental evaluation in Section 5.4. The CPDP model, which has the best combination with its optimal hyper-parameter settings in terms of the training accuracy, is returned in the end. Note that due to the lack of data samples, using training accuracy in the parameter optimization of transfer learner is not uncommon and has shown promising results for CPDP [21].

(3) **Performance Validation:** After the optimization phase, as an optional module in BiLO-CPDP, the generalization of the built CPDP model can be validated and tested by using the hold-out set from the target domain data, which is unknown during training stage. In practice, this will be the new project that one wishes to predict defects for. In BiLO-CPDP, the area under the receiver operating characteristic (ROC) curve, i.e., AUC [51], is applied as the performance metric to measure the effectiveness of a model. Formally, AUC is defined as:

$$AUC = \frac{\sum_{t^- \in \mathcal{D}^-} \sum_{t^+ \in \mathcal{D}^+} \mathbf{1}\left[Pred\left(t^-\right) < Pred\left(t^+\right)\right]}{|\mathcal{D}^-| \cdot |\mathcal{D}^+|}, \tag{2}$$

where $Pred(t)$ is the probability that sample t is predicted to be a positive sample, and $\mathbf{1}\left[f\left(t^-\right) < f\left(t^+\right)\right]$ is an indicator function which returns 1 if $f(t^-) < f(t^+)$ otherwise it returns 0. $\mathcal{D}^-$ is the set of negative samples, and $\mathcal{D}^+$ is the set of positive samples. Apart from the fact that AUC has been widely for software defect prediction [21], it has two distinctive characteristics: 1) different from other prevalent

metrics like precision and recall, AUC does not depend on a particular threshold [51], which is difficult to tweak in order to carry out an unbiased assessment; and 2) it is not sensitive to imbalanced data which is not uncommon in software defect prediction [22]. The larger the AUC value is, the better prediction accuracy the model achieves. In particular, the AUC value ranges between 0 and 1 where 0 indicates the worst performance, 0.5 corresponds a randomly guessed performance and 1 represents the best performance. Note that AUC is also the metric used in *optimization* phase to evaluate and compare training accuracy.

3.2 Upper-Level Optimization

Tables 1 and 2 respectively list the transfer learners and the classifiers considered in our work, which form the portfolios. Note that all transfer learners considered in BiLO-CPDP have been used in either the defect prediction or CPDP literature while the classifiers come from scikit-learn[3], the state-of-the-art machine learning Python library. In addition, the corresponding hyper-parameters associated with those transfer learners and classifiers along with their value ranges are also provided in the corresponding tables. Any combination of a transfer learner and a classifier comes up with a CPDP model. The ultimate goal of the upper-level optimization is to search for the best combination out of all possible alternatives (208 in this work) for the underlying CPDP task. In particular, for each candidate combination of transfer learner and classifier, their corresponding hyper-parameter settings are optimized via a lower-level optimization routine which will be explained in Section 3.3.

At the upper-level in BiLO-CPDP, the search of the best combination of transfer learner and classifier is solved as a combinatorial optimization problem as specified below.

- **Search space:** For the upper level, the search space consists of all the valid combinations of transfer learners and classifier picked up from the given portfolios, i.e., those listed in Tables 1 and 2. In practice, such portfolios can be amended and specified by the software engineers based on their preferences/requirements.
- **Objective function:** Recall from the equation (1), the objective function for the upper level $F(\mathbf{x}^u, \mathbf{x}^{l*})$ takes a combination from the portfolio ($\mathbf{x}^u$) and the optimized hyper-parameter of such combination ($\mathbf{x}^{l*}$) as inputs. It then outputs the corresponding training AUC obtained by training the CPDP model for comparison. Note that $\mathbf{x}^{l*}$ is initially unknown for a given $\mathbf{x}^u$ at the upper-level before running a lower-level optimization routine. Therefore, the objective function at upper-level optimization is constrained and determined by the lower-level optimization.
- **Optimization algorithm:** For the upper-level optimization in BiLO-CPDP, we use Tabu search [10] to serve as the optimizer, which is also the entry point of the optimization phase. In particular, we use Tabu search in this work because:
  - Our problem is expensive and thus it is unrealistic for an exact search to reach the optimal solution. Metaheuristic such as Tabu search, which does not guarantee optimum but can often produce near-optimal result, is more practical and acceptable.
  - Unlike other metaheuristics, Tabu search employs local search to speed up its convergence [10].
  - Tabu search permits a better chance to escape from local optima than other local search methods [10].

  As shown in in Algorithm 1 and Algorithm 2, Tabu search carries out a neighborhood search where the neighborhood of the current solution is restricted by the search history of previously visited solutions and is stored in the form of a *tabu list* (lines 5 and 6 in Algorithm 1 and lines 5 to 9 in Algorithm 2). If all neighbors are *tabu*, it is acceptable to take a move that worsen the value of the objective function (lines 3 and 4 in Algorithm 2). This is what enables Tabu search to escape from local optima, which is highly likely to cause issues with a traditional gradient decent method. According to a provided selection criteria, Tabu search only keep a record of some previously visited states.

Algorithm 1: runTabuSearch: Upper-level optimization that tunes the combination of transfer learner and classifier.

Input: Portfolio of transfer learners $\mathcal{T}$ and classifiers C
Output: Optimal CPDP model $\mathbf{x}^u_{\text{opt}}$ and its optimal parameter settings $\mathbf{x}^{l*}_{\text{opt}}$

1 Randomly initialize a valid combination of transfer learner and classifier $\mathbf{x}^u \leftarrow (t, c)$; /* $t \in \mathcal{T}$ and $c \in C$ are a candidate transfer learner and classifier */
2 $\ell_t \leftarrow \emptyset$; /* ℓ_t is the tabu list */
3 **while** *The overall time budget is not exhausted* **do**
4 $\quad[(\mathbf{x}^u, \mathbf{x}^{l*}), F(\mathbf{x}^u, \mathbf{x}^{l*})] \leftarrow$ searchCandidate$(\mathbf{x}^u, \ell_t)$;
5 $\quad$**if** $\mathbf{x}^u \notin \ell_t$ **then**
6 $\quad\quad \ell_t \leftarrow \ell_t \bigcup \{\mathbf{x}^u\}$;
7 **return** $(\mathbf{x}^u_{\text{opt}}, \mathbf{x}^{l*}_{\text{opt}}) \leftarrow \text{argmax}_{\mathbf{x}^u \in \ell_t}\{F(\mathbf{x}^u, \mathbf{x}^{l*})\}$;

3.3 Lower-Level Optimization

As introduced in Section 3.1, the main purpose of the lower-level optimization is to optimize the hyper-parameters associated with the chosen combination of transfer learner and classifier. Specifically, this level in BiLO-CPDP is modeled and tackled as below.

- **Search space:** At this level, the search space is the configuration space of the corresponding parameters for the transfer learner and classifier picked up from the upper-level routine. Indeed, as can be seen from Tables 1 and 2, such a configuration space might be different depending on the chosen combination of transfer learner and classifier.
- **Objective function:** Recall from the equation (1), when a combination of transfer learner and classifier is picked up from the upper-level routine, the objective function for the lower-level $f(\mathbf{x}^l)$ takes the configuration of the corresponding hyper-parameters as the inputs ($\mathbf{x}^l$) and outputs the training AUC for the CPDP model. The AUC collected from the result of the low-level routine is finally used as the objective value at the upper-level routine to steer the optimization.

[3] https://scikit-learn.org/stable/

Table 1: Overview of 13 selected transfer learners ([N], [R] and [C] denote integer, real and categorical value, respectively).

Algorithm	Parameter	Range	Algorithm	Parameter	Range	Algorithm	Parameter	Range
NNfilter [45]	k [N] metric [C]	[1, 100] Euc, Man, Che, Min, Mah	CDE_SMOTE [23]	k [N] metric [C]	[1, 100] Euc, Man, Che, Min, Mah	FSS_bagging [11]	topN [N] threshold [R] ratio [R]	[1, 15] [0.3, 0.7] [0.1, 0.5]
TCA+ [29]	kernel [C] dime [N] lamb [R] gama [R]	primal, rbf, linear, sam [5, max(N_s, N_t)] $[10E-7, 100]$ $[10E-6, 100]$	GIS [16]	prob [R] chrm_size [R] pop_size [N] num_parts [N] num_gens [N] mcount [N]	[0.02, 0.1] [0.02, 0.1] [2, 30] [2, 6] [5, 20] [3, 10]	CLIFE_MORPH [32]	n [N] alpha [R] beta [R] per [R]	[1, 100] [0.05, 0.2]] [0.2, 0.4] [0.6, 0.9]
						HISNN [38]	minham [N]	[1, N_s]
MCWs [33]	k [N] sigma [R] lambda [R]	[2, N_s] [0.01, 10] $[10E-7, 100]$	FeSCH [30]	nt [N] strategy [C]	[1, N_s] SFD, LDF, FCR	UM [50]	p [R] qua_T [C]	[0.01, 0.1] cli , cohen
TD [12]	strategy [C] k [N]	NN, EM [1, N_s]	VCB [37]	m [N] lambda [R]	[2, 30] [0.5, 1.5]	PCAmining [28]	dime [N]	[5, max(N_s, N_t)

For full specification of all the parameters, please visit our repository: https://github.com/COLA-Laboratory/ase2020

Table 2: Overview of 16 selected classifiers ([N], [R] and [C] denote integer, real and categorical value, respectively).

Algorithm	Parameter	Range	Algorithm	Parameter	Range	Algorithm	Parameter	Range
Extra Trees Classifier (EXs)	max_e [N] criterion [C] min_s_l [N] splitter [C] min_a_p [N]	[10, 100] gini, entropy [1, 20] random, best [2, N_s/10]	Extra Tree Classifier (EXtree)	max_e [N] criterion [C] min_s_l [N] splitter [C] min_a_p [N]	[10, 100] gini, entropy [1, 20] random, best [2, N_s/10]	Decision Tree (DT)	max_e [N] criterion [C] min_s_l [N] splitter [C] min_a_p [N]	[10, 100] gini, entropy [1, 20] auto, sqrt, log2 [2, N_s/10]
Random Forest (RF)	m_stim [N] criterion [C] splitter [C] min_s_l [N] min_a_p [N]	[10, 100] gini, entropy auto, sqrt, log2 [1, 20] [2, N_s/10]	Support Vector Machine (SVM)	C [R] kernel [C] degree [N] coef0 [R] gamma [R]	[0.001, 10] rbf, lin, poly, sig [1, 5] [0, 10] [0.01, 100]	Multilayer Perceptron (MLP)	active [C] hid_l_s [N] solver [C] iter [N]	iden, log, tanh, relu [50, 200] lbfgs, sgd, adam [100, 250]
Passive Aggressive Classifier (PAC)	C [R] fit_int [C] tol [R] loss [C]	[0.001, 100] true, false $[10E-6, 0.1]$ hinge, s_hinge	Perceptron	penalty [C] alpha [R] fit_int [C] tol [R]	L1, L2 $[10E-6, 0.1]$ true, false $[10E-6, 0.1]$	Naive Bayes (NB)	NBType [C] alpha [R] norm [C]	gauss, multi, comp [0, 10] ture, false
Ridge	alpha [R] fit_int [C] tol [R]	$[10E-5, 1000]$ ture, false $[10E-6, 0.1]$	Bagging	n_est [N] max_s [R] max_f [R]	[10, 200] [0.7, 1.0] [0.7, 1.0]	Logistic Regression (LR)	penalty [C] fit_int [C] tol [R]	L1, L2 ture, false $[10E-6, 0.1]$
KNearest Neighbor(KNN)	n_neigh [N] p [N]	[1, 50] [1, 5]	Radius Neighbors Classifier (RNC)	radius [R] weight [C]	[0, 10000] uni, dist	Nearest Centroid Classifier (NCC)	metric [C] shrink_t [R]	Euc, Man, Che, Min, Mah [0, 10]
adaBoost	n_est [N] rate [R]	[10, 1000] [0.01, 10]						

For full specification of all the parameters, please visit our repository: https://github.com/COLA-Laboratory/ase2020

- **Optimization algorithm:** It is not uncommon that the training and evaluation of a CPDP model is computationally demanding and time consuming. To this end, in BiLO-CPDP, we apply the Tree-structured Parzen Estimator (TPE) [4] — a state-of-the-art Bayesian optimization algorithm for hyper-parameter optimization of machine learning algorithms — as the optimizer for the lower-level optimization, due primarily to the following reasons:
  - TPE copes with a wide range of variables well, including integer, real, and categorical ones, which fits precisely with our need [4].
  - Recent work on CPDP [21] and from the machine learning community [7] have reported the outstanding performance of TPE for expensive configuration problems.

As the pseudo-code shown in Algorithm 3, the TPE algorithm first uses a space-filling technique to sample a set of hyper-parameters' values from the given configuration space Θ_c of transfer learner and classifier, which would then be trained for collecting the training AUC performance (line 1). All these constitute the initial dataset $\mathcal{D}$. During the main while-loop, a relatively cheap surrogate model of the expensive physical model training and the AUC evaluation is built based on all sampled data in $\mathcal{D}$ (line 3). Thereafter, a promising hyper-parameter configuration trial $\mathbf{x}^{l_c}$ is identified by optimizing the acquisition function (i.e., expected improvement) following a classic Bayesian optimization rigour. The AUC of $\mathbf{x}^{l_c}$ is thereafter evaluated and used to augment $\mathcal{D}$ (lines 4 to 6). At the end, the best hyper-parameter setting $\mathbf{x}^{l*}$

Algorithm 2: searchCandidate($\mathbf{x}^u, \ell_t$): Search the next combination candidate from the neighborhood of $\mathbf{x}^u$

Input: Candidate CPDP model $\mathbf{x}^u$ and newest *tabu list* ℓ_t
Output: The best CPDP model within $\mathbf{x}^u$'s neighbor and its optimal parameter settings $\mathbf{x}_{\mathtt{opt}}$, the performance of this CPDP model $f_{\mathtt{opt}}$, i.e., the value of $F(\mathbf{x}^u, \mathbf{x}^{l*})$

1 $\delta \leftarrow$ Get the neighbors of $\mathbf{x}^u$;
2 $\Theta_{\mathbf{x}^u} \leftarrow$ Get the configuration space of the transfer learner and the classifier specified by $\mathbf{x}^u$;
3 $[\mathbf{x}^{l\prime}, f_{\mathbf{x}^u}(\mathbf{x}^{l\prime})] \leftarrow$ runTPE($\mathbf{x}^u, \Theta_{\mathbf{x}^u}$);
4 $\mathbf{x}_{\mathtt{opt}} \leftarrow (\mathbf{x}^u, \mathbf{x}^{l\prime})$, $f_{\mathtt{opt}} \leftarrow f_{\mathbf{x}^u}(\mathbf{x}^{l\prime})$;
5 **foreach** $\mathbf{x}^c \in \delta \wedge \mathbf{x}^c \notin \ell_t$ **do**
6 $\Theta_{\mathbf{x}^c} \leftarrow$ Get the configuration space of the transfer learner and the classifier specified by $\mathbf{x}^c$;
7 $[\mathbf{x}^{l\prime}, f_{\mathbf{x}^c}(\mathbf{x}^{l\prime})] \leftarrow$ runTPE($\mathbf{x}^c, \Theta_{\mathbf{x}^c}$);
8 **if** $f_{\mathbf{x}^c}(\mathbf{x}^{l\prime}) > f_{\mathtt{opt}}$ **then**
9 $\mathbf{x}_{\mathtt{opt}} \leftarrow (\mathbf{x}^c, \mathbf{x}^{l\prime})$, $f_{\mathtt{opt}} \leftarrow f_{\mathbf{x}^c}(\mathbf{x}^{l\prime})$;
10 **return** $\mathbf{x}_{\mathtt{opt}}, f_{\mathtt{opt}}$;

Algorithm 3: runTPE($\mathbf{x}^u, \Theta_{\mathbf{x}^u}$): Lower-level optimization for identifying the optimal hyper-parameters.

Input: Combination of transfer learner and classifier $\mathbf{x}^u$, configuration space Θ_c
Output: Optimized hyper-parameters $\mathbf{x}^{l*}$ and its objective function $f_{\mathbf{x}^u}(\mathbf{x}^{l*})$

1 $\mathcal{D} \leftarrow$ Use space-filling to sample a set of hyper-parameters from Θ_c and evaluate their objective functions;
2 **while** *The lower-level time budget is not exhausted* **do**
3 Use Tree Parzen to build a surrogate model based on $\mathcal{D}$;
4 $\mathbf{x}^{l_c} \leftarrow$ Best configuration based on the AUC predicted by the acquisition function over the surrogate model;
5 $f_{\mathbf{x}^u}(\mathbf{x}^{l_c}) \leftarrow$ Evaluate the objective function of $\mathbf{x}^{l_c}$ by physically training the CPDP model;
6 $\mathcal{D} = \mathcal{D} \bigcup \{(\mathbf{x}^{l_c}, f_{\mathbf{x}^u}(\mathbf{x}^{l_c}))\}$;
7 **return** $(\mathbf{x}^{l*}, f_{\mathbf{x}^u}(\mathbf{x}^{l*})) \leftarrow \mathrm{argmax}_{(\mathbf{x}^{l_c}, f_{\mathbf{x}^u}(\mathbf{x}^{l_c})) \in \mathcal{D}} \{f_{\mathbf{x}^u}(\mathbf{x}^{l_c})\}$;

in $\mathcal{D}$ along with its AUC performance $f_{\mathbf{x}^u}(\mathbf{x}^{l*})$ are returned to the upper-level optimization routine (line 7).

4 EXPERIMENTAL SETUP

This section introduces our experiment setups[4].

4.1 Dataset

In our experiments, the dataset of software projects is collected according to the following three *inclusion* criteria:

(1) To promote the reproducibility and practicality of our experiments, we only consider projects hosted in public repositories and are related to non-academic software.
(2) To mitigate potential conclusion bias, projects are required to cover different corpora and domains.
(3) To ensure the credibility of experiments, we focus on projects that have already been used in the CPDP literature.

Note that a project is temporarily selected if it meets all above three criteria. To further refine our dataset composition, we apply the following two *exclusion* criteria to rule out inappropriate projects.

(1) It is not uncommon that the projects are evolved with more than one version during their lifetime. Since different versions of the same project are highly likely to share many similarities, they may simplify the transfer learning. In this case, only the latest version of the project is kept.
(2) To promote the robustness of experiments, projects with repeated or missing data are ruled out from our consideration.

Based on the above inclusion criteria, we select five publicly available datasets, i.e., `JURECZKO`, `NASA`, `SOFTLAB`, `AEEEM`, `ReLink`. Note that all these datasets have been reviewed and discussed in many recent survey in the CPDP literature [13–15, 51]. Thereafter, `SOFTLAB` is further ruled out from our consideration according to the above exclusion criteria. In addition, `NASA` is also not considered in our experiments since its data quality is relatively poor as reported in [41]. At the end, the dataset considered in our experiments consist of 20 open source projects with 10,952 instances. Its characteristics are summarized as follows:

- `AEEEM` [6]: This dataset contains 5 open source projects with 5,371 instances. In particular, each instance has 61 metrics with two different types, including static and process metrics like the entropy of code changes and source code chorn.
- `ReLink` [49]: This dataset consists of 3 open source projects with 649 instances. In particular, each instance comes with 26 static metrics. Note that the defect labels are further manually verified after being generated from source code management system commit comments.
- `JURECZKO` [19]: This dataset originally consists of 92 released software collected from a mix of open sourced, proprietary and academic projects. With respect to the first inclusion criterion, those proprietary and academic projects are not considered. Moreover, since the projects in `JURECZKO` have been updated more than once, according to the first exclusion criterion, only the latest version of a project is considered in our experiments. Ultimately, we choose 12 open source projects with 4,932 instances from `JURECZKO`.

4.2 Experimental Procedure

Our experimental procedure follows the three-phases workflow of `BiLO-CPDP` introduced in Section 3.1. Here we explain the corresponding settings for each phase.

- In the *data pre-processing* phase for all peer CPDP techniques, all projects in this work will be used as target domain data in a round-robin manner, forming 20 different CPDP tasks. This aims to mitigate the potential bias in conclusion.
- In the *optimization* phase, each CPDP task is allocated with an overall time budget of one hour (i.e., 3,600 seconds, as suggested by Feurer et al. [8]) while setting each lower-level exploitation as 20 seconds in `BiLO-CPDP`. When applicable,

[4] All source code and data of this work can be publicly accessed via our repository: https://github.com/COLA-Laboratory/ase2020

Table 3: Scott-Knott test on BiLO-CPDP and existing CPDP techniques over 30 runs. (the larger rank, the better; gray=the best)

CPDP Technique	Apache	EQ	JDT	LC	ML	PDE	Safe	Tomcat	Zxing	ant	camel	ivy	jEdit	log4j	lucene	poi	synapse	velocity	xalan	xerces
NNfilter-NB	**6**	5	4	**8**	**5**	3	**8**	**14**	4	6	**7**	9	3	8	**8**	6	2	3	8	7
UM-NB	2	7	6	5	4	3	1	9	4	10	**7**	11	5	9	7	**8**	**6**	8	10	8
UM-LR	5	3	6	7	4	3	6	12	4	9	5	10	7	10	5	4	4	3	5	7
CLIFE-NB	3	4	4	3	1	3	6	9	1	8	4	6	8	7	4	6	4	5	4	4
CLIFE-KNN	3	3	**7**	4	2	3	1	12	4	7	3	6	7	6	4	4	4	4	6	4
FeSCH-RF	3	3	4	2	3	3	5	10	1	6	2	8	8	4	4	5	4	5	4	6
GIS-NB	1	3	1	3	1	1	1	9	4	10	5	5	8	7	4	4	4	6	2	5
FeSCH-LR	4	3	4	1	2	3	5	2	1	7	3	7	5	5	4	4	5	7	7	6
CLIFE-SVM	4	3	4	4	3	2	1	7	4	2	3	8	6	6	4	7	3	3	8	3
TD-RF	3	3	6	3	2	3	5	6	3	5	3	2	8	3	4	4	3	5	6	3
TD-LR	2	1	4	3	2	3	5	9	1	4	3	6	8	6	4	4	3	6	9	3
TD-MLP	4	3	4	3	1	3	1	7	4	7	3	8	7	4	2	5	3	5	9	3
TD-DT	4	6	6	3	2	3	4	8	1	6	3	7	6	5	1	3	3	5	2	3
FeSCH-DT	4	3	3	3	2	3	5	8	2	7	3	5	2	8	1	2	3	2	5	3
VCB-SVM	4	2	**7**	3	2	3	2	2	1	2	2	2	5	3	4	2	1	1	1	2
CDE_SMOTE-RF	3	4	4	5	2	3	1	1	4	1	1	1	1	3	2	1	1	1	1	1
CDE_SMOTE-KNN	4	3	4	3	1	3	5	1	4	1	1	1	1	4	6	1	3	2	**12**	1
FSS_bagging-RF	2	**8**	5	3	1	3	1	4	4	3	2	2	6	1	4	3	3	6	2	2
FSS_bagging-NB	2	3	2	4	3	2	3	11	4	6	3	3	8	1	2	2	3	2	3	3
FSS_bagging-LR	3	3	5	3	3	3	6	5	1	4	2	8	9	2	3	4	4	4	2	5
HISNN-NB	4	1	1	1	1	1	6	3	4	4	3	4	4	3	2	2	3	3	2	2
BiLO-CPDP	**6**	**8**	**7**	6	**5**	**4**	7	13	**5**	**11**	6	**12**	**10**	**11**	7	**8**	**6**	**9**	11	**9**

The raw AUC values can be found in our repository: https://github.com/COLA-Laboratory/ase2020

the same budget is given to other state-of-the-art peer CPDP techniques that permit hyper-parameter optimization in the comparison, e.g., Auto-sklearn [8]. We apply the TPE algorithm implementation integrated in Hyperopt[5], a popular Python library for hyper-parameter tuning in machine learning [5], for the lower-level routine of BiLO-CPDP.

- In the *performance validation* phase, AUC is used as the performance metric. Due to the stochastic nature of BiLO-CPDP and some peer CPDP techniques considered, each technique is independently repeated 30 times for a given CPDP task and the mean AUC values are recorded for comparison.

4.3 Ranking, Statistical Test and Effect Size

In our experiments, we use the following three statistical measures to interpret the statistical significance of our comparative results.

- **Scott-Knott test:** Instead of merely comparing the raw AUC values, we apply the Scott-Knott test to rank the performance of different peer techniques over 30 runs on each project, as recommended by Mittas and Angelis [27]. In a nutshell, the Scott-Knott test uses a statistical test and effect size to divide the performance of peer techniques into several clusters. In particular, the performance of peer techniques within the same cluster are statistically insignificant, i.e., their overall AUC values are statistically equivalent. Note that the clustering process terminates until no split can be made. Finally, each cluster can be assigned a rank according to the mean AUC values achieved by the peer techniques within the cluster. In particular, since a greater AUC is preferred, the larger the rank is, the better performance of the technique achieves.
- **Wilcoxon signed-rank test:** We apply the Wilcoxon signed-rank test [48] with a significant level $p = 0.05$ [3] to investigate the statistical significance of the comparisons. It is a non-parametric statistical test that makes little assumption about the underlying distribution of the data and has been recommended in software engineering research [3].
- **A_{12} effect size:** To ensure the resulted differences are not generated from a trivial effect, we apply A_{12} [46] as the effect size measure to evaluate the probability that one technique is better than another. According to Vargha and Delaney [46], when comparing BiLO-CDPD with another peer technique in our experiments, $A_{12} = 0.5$ means they are equivalent. $A_{12} > 0.5$ denotes that BiLO-CDPD is better for more than 50% of the times. In particular, $0.56 \leq A_{12} < 0.64$ indicates a small effect size while $0.64 \leq A_{12} < 0.71$ and $A_{12} \geq 0.71$ mean a medium and a large effect size, respectively.

Note that both Wilcoxon signed-rank test and A_{12} are also used in the Scott-Knott test for generating the clusters.

4.4 Research Questions

We seek to answer the following four research questions (RQs) through our experimental evaluation:

- **RQ1:** Is BiLO-CPDP able to automatically configure a CPDP model having better performance than the existing CPDP techniques under their reported settings?

[5]http://hyperopt.github.io/hyperopt/

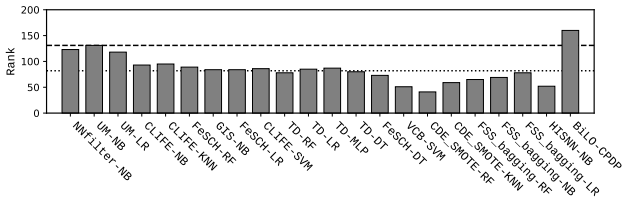

Figure 2: Total ranks achieved by BiLO-CPDP (the right most one) and the 21 peer techniques (the larger rank, the better; the dashed line and dotted line denote the best and average result over the 21 peer techniques, respectively).

- **RQ2:** How is the performance of BiLO-CPDP comparing with Auto- Sklearn, a state-of-the-art AutoML tool?
- **RQ3:** Is the bi-level programming in BiLO-CPDP beneficial?
- **RQ4:** Given a limited computational budget, which level in BiLO-CPDP is more important and deserves more budget?

5 RESULTS AND DISCUSSIONS

In this section, we present and discuss the results of our empirical experiments and address the RQs posed in Section 4.4.

5.1 Comparison with Existing CPDP Work

5.1.1 Method. In order to answer RQ1, we use the transfer learners and classifiers collected in Tables 1 and 2 to constitute 21 peer CPDP techniques in comparison with BiLO-CPDP. Note that although there are only 13 transfer learners listed in Table 1, some of them are combined with more than one classifier to constitute different CPDP models used in the literature (e.g., TD is combined with classifiers RF, LR, MLP and DT that constitute four different CPDP models in [12]). For the parameter settings, we use the tuned values as reported in the corresponding work.

5.1.2 Results and Analysis. From the experimental results on the Scott-Knott test shown in Table 3, it is clear to see that BiLO-CPDP is the best on 14 out of 20 (70%) projects, second only to one other on five cases. In contrast, most of the other peer CPDP techniques, albeit hand crafted by domain experts, are not as competitive as BiLO-CPDP. In particular, NNfilter-NB is the most outstanding peer technique that is the best on only 7 out 20 (35%) projects while the other peer techniques rarely take the best rank across all 20 projects. Noteworthily, the performance of NNfilter-NB ties with BiLO-CPDP in two of its best results. In terms of the total ranks achieved over all projects, as shown in Fig. 2, we can observe the clear superiority of BiLO-CPDP which is at least 50% better than the other 21 peer techniques. Furthermore, we notice that the superior performance of BiLO-CPDP is consistent across all 20 projects in view of its top three ranked positions achieved in all projects. In contrast, the performance of existing CPDP techniques exhibit clear variations depending on the underlying target projects.

Table 4: Mean AUC (standard deviation) for BiLO-CPDP and Auto-Sklearn over 30 runs (gray=better; bold=p<.05).

Project	BiLO-CPDP	Auto-sklearn	p-value
poi	**8.1703E-1 (4.38E-3)**	6.5262E-1 (6.98E-3)	1.71E-6
synapse	**7.1999E-1 (7.72E-3)**	6.0183E-1 (3.68E-3)	1.65E-6
Zxing	**6.3949E-1 (5.37E-3)**	6.2615E-1 (1.45E-6)	1.91E-6
ant	**8.0006E-1 (8.26E-3)**	7.4530E-1 (7.63E-3)	1.71E-6
log4j	**8.4196E-1 (1.52E-2)**	6.0965E-1 (1.19E-2)	1.57E-6
Safe	**7.9923E-1 (2.09E-2)**	6.7513E-1 (6.15E-3)	1.64E-6
ivy	**8.0657E-1 (3.51E-3)**	7.2407E-1 (9.64E-4)	1.19E-6
PDE	**6.8539E-1 (2.57E-3)**	5.9781E-1 (2.22E-16)	1.62E-6
camel	**6.2228E-1 (4.01E-3)**	5.9006E-1 (1.11E-16)	1.62E-6
lucene	**7.1065E-1 (8.13E-3)**	6.4408E-1 (4.87E-6)	1.37E-6
JDT	**7.3705E-1 (1.09E-2)**	6.7517E-1 (1.11E-16)	1.66E-6
jEdit	**8.5207E-1 (3.77E-2)**	7.1589E-1 (5.70E-3)	1.68E-6
EQ	**7.1714E-1 (1.34E-2)**	6.0201E-1 (3.33E-16)	1.73E-6
velocity	**7.0220E-1 (8.40E-3)**	6.0896E-1 (4.50E-2)	1.61E-6
Tomcat	**7.7295E-1 (1.40E-3)**	7.3892E-1 (1.32E-2)	1.45E-7
Apache	7.4808E-1 (8.27E-3)	7.4787E-1 (3.33E-16)	6.58E-1
ML	**6.4966E-1 (1.58E-3)**	6.1708E-1 (2.22E-16)	1.73E-6
xerces	**7.1552E-1 (1.03E-2)**	5.9892E-1 (6.03E-3)	1.71E-6
LC	**7.0859E-1 (1.89E-2)**	6.2476E-1 (1.11E-16)	1.73E-6
xalan	**7.6250E-1 (7.67E-3)**	6.7732E-1 (2.31E-2)	1.71E-6

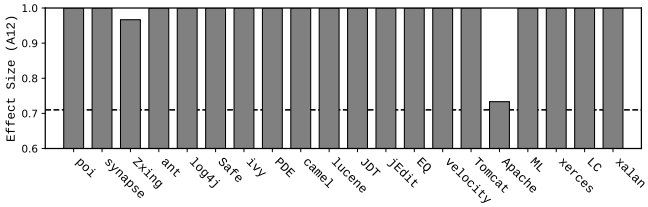

Figure 3: A_{12} result between BiLO-CPDP and Auto-Sklearn over 30 runs ($A_{12} > 0.5$ means BiLO-CPDP is better).

Response to RQ1: BiLO-CPDP *is generally better than the other 21 existing CPDP techniques over all 20 projects. Unlike others that were hand-crafted by domain experts to certain extents,* BiLO-CPDP *builds an effective CPDP model in a completely automated manner, leading to highly competitive performance over different projects.*

5.2 Comparison with Auto-Sklearn

5.2.1 Method. In principle, BiLO-CPDP is an AutoML tool that automatically searches for the right combination of transfer learner and classifier and their optimized hyper-parameter settings for a given CPDP task. To validate its competitiveness from the perspective of AutoML, we compare the performance of BiLO-CPDP with Auto-Sklearn[6] [8], a state-of-the-art and readily available AutoML tool that can also optimize the combination and its parameters.

5.2.2 Results and Analysis. From the comparison results of AUC values shown in Table 4, we clearly see the overwhelmingly superior performance of BiLO-CPDP versus Auto-Sklearn where the AUC values obtained by BiLO-CPDP are all better than those of Auto-Sklearn. In particular, all those better results, except on

[6]https://automl.github.io/auto-sklearn/master/

Apache, are statistically significant, according to the p values shown in the last column of Table 4. Furthermore, as shown in Fig. 3, all A_{12} values suggest a large effect size. In particular, we see an overwhelming $A_{12} = 1$, except only on Zxing and Apache. These indicate that the improvements on the AUC results brought by BiLO-CPDP over that of the Auto-Sklearn are significantly large in general.

The results are caused by the fact that Auto-Sklearn does not have a bi-level structure, hence it encodes all transfer learner and classifier combinations along with their corresponding hyper-parameter settings into an integrated solution at a single-level, which is solved by the SMAC algorithm [17]. During its optimization process, a combination of transfer learner and classifier is selected first. Thereafter, the variables corresponding to the hyper-parameters of the chosen transfer learner and the classifier become active while the remaining variables are set to be dummy. By this means, the total number of variables considered in Auto-Sklean goes up to 93, resulting a unnecessarily much larger search space comparing with BiLO-CPDP. Given the limited budget, Auto-Sklearn therefore ends up with a less effective exploration of both useful combinations of transfer learners and classifiers and their hyper-parameter settings.

Response to RQ2: *Comparing with the state-of-the-art AutoML tool* Auto-Sklearn, BiLO-CPDP *achieves significantly better results given a limited computational budget.*

5.3 Comparison with Single-Level Variant

5.3.1 Method. It is conservative to curious about the usefulness brought by this bi-level programming formulation and why not simply formulating a single-level problem that consists of both combination and parameters. The comparison with Auto-Sklearn, which is at a single-level, partially validates this concern, but the results can be biased by the fact that it uses a different optimization algorithm. To fully evaluate the effectiveness of bi-level programming, we develop a single-level variant of BiLO-CPDP, dubbed SLO-CPDP, which differs from BiLO-CPDP only on the solution representation.

Specifically, SLO-CPDP is similar to Auto-Sklearn in the sense that they both work on single-level optimization — the transfer learner and classifier, together with their hyper-parameters, are encoded as a single solution representation. However, the difference is that SLO-CPDP exploits the TPE algorithm as the Bayesian optimizer, which is identical to BiLO-CPDP. Auto-Sklearn, in contrast, uses the classic SMAC algorithm that leverages Random Forest to build the surrogate model.

5.3.2 Results and Analysis. From the AUC values shown in Table 5, we observe a rather superior performance achieved by BiLO-CPDP over SLO-CPDP. Specifically, BiLO-CPDP again obtains a better AUC value on all 20 projects. In particular, all better results are with statistical significance (p<.05), as shown in the last column of Table 5. Furthermore, from Fig. 4, we find that the differences between the AUC values achieved by BiLO-CPDP and SLO-CPDP are with a large effect size. Given such a result, we can infer that the ineffectiveness of SLO-CPDP can be attributed to the enlarged search space caused by the unwise coupling of transfer learner and classifier combination along with their parameters at a single-level.

Table 5: Mean AUC (standard deviation) for BiLO-CPDP and SLO-CPDP over 30 runs (gray=better; bold=p<.05).

Project	BiLO-CPDP	SLO-CPDP	p-value
poi	**8.1703E-1 (4.38E-3)**	5.7493E-1 (2.71E-1)	1.92E-6
synapse	**7.1999E-1 (7.72E-3)**	5.0601E-1 (2.33E-1)	1.73E-6
Zxing	**6.3949E-1 (5.37E-3)**	5.4209E-1 (1.46E-1)	1.92E-6
ant	**8.0006E-1 (8.26E-3)**	6.3099E-1 (1.76E-1)	1.73E-6
log4j	**8.4196E-1 (1.52E-2)**	6.2807E-1 (2.49E-1)	2.60E-6
Safe	**7.9923E-1 (2.09E-2)**	7.4254E-1 (3.96E-2)	5.74E-5
ivy	**8.0657E-1 (3.51E-3)**	6.3528E-1 (1.77E-1)	1.73E-6
PDE	**6.8539E-1 (2.57E-3)**	5.0874E-1 (2.52E-1)	1.73E-6
camel	**6.2228E-1 (4.01E-3)**	4.7330E-1 (2.15E-1)	5.22E-6
lucene	**7.1065E-1 (8.13E-3)**	5.8568E-1 (2.05E-1)	1.15E-4
JDT	**7.3705E-1 (1.09E-2)**	6.2603E-1 (1.81E-1)	1.36E-5
jEdit	**8.5207E-1 (3.77E-2)**	5.5203E-1 (2.50E-1)	1.73E-6
EQ	**7.1714E-1 (1.34E-2)**	5.4016E-1 (2.21E-1)	2.88E-6
velocity	**7.0220E-1 (8.40E-3)**	5.3123E-1 (2.11E-1)	1.73E-6
Tomcat	**7.7295E-1 (1.40E-3)**	5.6400E-1 (2.40E-1)	1.92E-6
Apache	**7.4808E-1 (8.27E-3)**	5.2924E-1 (1.23E-1)	1.01E-6
ML	**6.4966E-1 (1.58E-3)**	5.8287E-1 (1.53E-1)	7.16E-4
xerces	**7.1552E-1 (1.03E-2)**	6.3384E-1 (1.29E-1)	6.87E-5
LC	**7.0859E-1 (1.89E-2)**	5.3199E-1 (2.38E-1)	1.02E-5
xalan	**7.6250E-1 (7.67E-3)**	5.9866E-1 (2.21E-1)	5.79E-5

Response to RQ3: *The bi-level programming in* BiLO-CPDP *considerably contributes to its effectiveness. In contrast to the single-level where the combination and parameters are formulated in a "flat" way, bi-level programming significantly reduces the search space and steer the search in a hierarchical manner, leading to better performance under a limited budget.*

5.4 Impact of Budget for the Two Levels

5.4.1 Method. In practice, it is not uncommon that the resource for defect prediction, particular the time budget, is limited. In our experiments, the total budget allocated to BiLO-CPDP is one hour in total, following the best practice in the AutoML community [8]. However, the unique bi-level programming formulated in BiLO-CPDP allows us a flexible control over the budget allocated to the two levels, hence it is interested to know how their budget allocations may impact the performance. To this end, within the one hour total budget, we set two budget allocation strategies: one with high budget to the upper-level, dubbed BiLO-CPDP(h), that allows 20 seconds for each low-level optimization, leaving more resources for exploring the combinations at the upper-level. Note that 20 seconds are very short for some model training thus is counter-intuitive. This is also the default setting in BiLO-CPDP we used for other experiments. Another one preserves high budget to the low-level, dubbed BiLO-CPDP(l), in which the lower-level optimization is allocated with 100 training and AUC evaluations. This allows a low-level routine to consume at least 80 seconds (the smallest amount of time required to complete 100 evaluations among all combinations) for more sufficient exploration of the hyper-parameters.

5.4.2 Results and Analysis. As shown in Table 6, we can see that BiLO-CPDP(h) is overwhelmingly superior to BiLO- CPDP(l) where it obtains better AUC values on all 20 projects. In addition, from

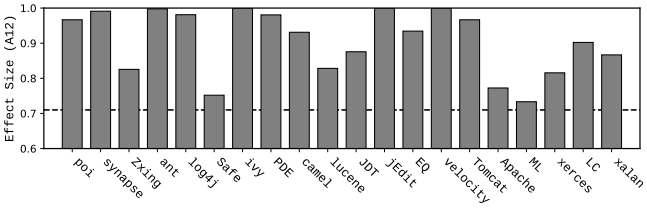

Figure 4: A_{12} result between BiLO-CPDP and SLO-CPDP over 30 runs ($A_{12} > 0.5$ means BiLO-CPDP is better).

Table 6: Mean AUC (standard deviation) for BiLO-CPDP(h) and BiLO-CPDP(l) over 30 runs (gray=better; bold=p<.05).

Project	BiLO-CPDP(h)	BiLO-CPDP(l)	p-value
poi	**8.1703E-1 (4.38E-3)**	5.9447E-1 (2.74E-1)	3.85E-6
synapse	**7.1999E-1 (7.72E-3)**	6.1897E-1 (1.26E-1)	1.73E-6
Zxing	**6.3949E-1 (5.37E-3)**	6.1108E-1 (2.10E-2)	8.46E-6
ant	**8.0006E-1 (8.26E-3)**	4.6150E-1 (3.31E-1)	1.91E-6
log4j	**8.4196E-1 (1.52E-2)**	5.1593E-1 (2.91E-1)	2.46E-6
Safe	**7.9923E-1 (2.09E-2)**	7.1771E-1 (1.38E-1)	2.87E-6
ivy	**8.0657E-1 (3.51E-3)**	5.9659E-1 (3.04E-1)	5.23E-6
PDE	**6.8539E-1 (2.57E-3)**	4.3027E-1 (3.05E-1)	1.71E-6
camel	**6.2228E-1 (4.01E-3)**	3.9054E-1 (2.79E-1)	7.90E-6
lucene	**7.1065E-1 (8.13E-3)**	5.3341E-1 (2.69E-1)	1.12E-5
JDT	**7.3705E-1 (1.09E-2)**	3.7420E-1 (3.51E-1)	3.85E-6
jEdit	**8.5207E-1 (3.77E-2)**	5.0339E-1 (3.30E-1)	1.72E-6
EQ	**7.1714E-1 (1.34E-2)**	4.2035E-1 (3.22E-1)	8.37E-6
velocity	**7.0220E-1 (8.40E-3)**	5.0660E-1 (2.55E-1)	1.92E-6
Tomcat	**7.7295E-1 (1.40E-3)**	5.6762E-1 (2.87E-1)	2.50E-6
Apache	**7.4808E-1 (8.27E-3)**	7.1257E-1 (2.62E-2)	7.22E-6
ML	**6.4966E-1 (1.58E-3)**	3.7510E-1 (3.07E-1)	5.70E-6
xerces	**7.1552E-1 (1.03E-2)**	4.8507E-1 (2.72E-1)	3.87E-6
LC	**7.0859E-1 (1.89E-2)**	4.9327E-1 (3.00E-1)	1.23E-4
xalan	**7.6250E-1 (7.67E-3)**	4.9144E-1 (3.05E-1)	1.91E-6

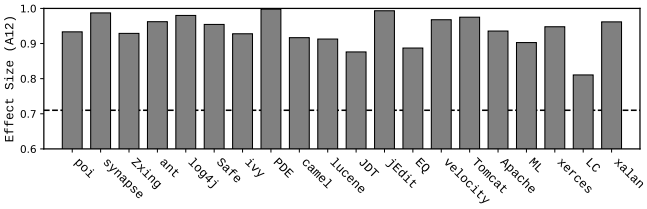

Figure 5: A_{12} result between BiLO-CPDP(h) and BiLO-CPDP(l) over 30 runs ($A_{12} > 0.5$ means BiLO-CPDP(h) is better).

the comparison results of A_{12} shown in Fig. 5, we can see that the differences between AUC values achieved by the two budget allocation strategies are categorized to have a large effect size.

The performance differences are due to the fact that the CPDP model training can be rather time-consuming and unfavorable, especially given a limited budget. Therefore, for BiLO-CPDP(l), once a combination of transfer learner and classifier is selected at the upper-level routine, its initiative to favor better exploration of the hyper-parameters at the lower-level routine can easily consume a significant amount of the budget (the median computational time is around 300 seconds according to our offline statistics). This has caused the combinatorial space of transfer learners and classifiers to become severely under-explored. In contrast, by strictly restricting the budget at the lower-level optimization routine, BiLO-CPDP(h) suffers from a limited exploration of the hyper-parameter space, but permitting a sufficient chance to explore many combinations of transfer learners and classifiers. From the results, it evidences that exploring the combination space is more important than using the hyper-parameter space under a limited budget.

Response to RQ4: *Given a limited budget, it is recommended to allocate more expenditure to the upper-level optimization routine in* BiLO-CPDP. *By this means, more combinations of transfer learner and classifier can be investigated even without fully optimized hyper-parameters, which is more beneficial to performance.*

6 RELATED WORK

In the past decades, machine learning classifiers have become the core techniques for defect prediction, in which the success can be greatly affected by the setting of the classifiers' hyper-parameters [20]. This is a challenging issue, as Jiang et al. [18] pointed out that simply using the default values are dreadful, causing severely bad performance of the prediction. The automated parameter optimization for defect predictors is therefore crucial. Indeed, a large scale empirical study by Tantithamthavorn et al. [43, 44] found that well-tuned hyper-parameters can significantly boost the performance of the classifiers in defect prediction. Fu et al. [9] even suggest that such optimization should become a standard practice in every single Software Engineering task. In light of this, Agrawal and Menzies [2] have applied Differential Evolution to tune SMOTE, a pre-processor for handling data imbalance, for predicting software defects. Their work focus on within-project defect prediction though. Similarly, DODGE [1] is a recent tool that optimizes the parameters of data pre-processor and classifier. Although they aim for within-project case, the combination of pre-processor and classifier can be resemble to our CPDP task. However, their optimization assumes conservative hybridization of all the parameters and the combinations as a single-level optimization problem.

The importance of automated parameter optimization remains stand in the context of CPDP, where the problem become even more complex as the parameters of transfer learners also come into play. Qu et al. [35] have shown that the parameter settings of classifiers for CPDP are even more important. A few automated optimizers exist for CPDP, for example, Öztürk [31] and Qu et al. [34] examine various different optimization algorithms to tune CPDP models. Nevertheless, they focus only on the parameter tuning whilst ignore the combination of transfer learner and classifier during optimization. Indeed, Li et al. [21] further demonstrate that the parameter interactions between transfer learner and classifier, as well as their combination, also play an integral role to the prediction performance. Auto-Sklearn [8], which is a widely-used generic tool to tune arbitrary machine learning algorithms, is also highly potential for CPDP tuning. However, again, its design has restricted that the combination of transfer learner and classifier along with their parameters need to be tuned as a single optimization problem, which worsen its performance compared with BiLO-CPDP, as we have shown in Section 5.

Although the potentials of bi-level programming have been explored for other Software Engineering problems, e.g., code smell detection [39] and test case generation [40], to the best of our knowledge, its adoption has never been reported in the context of CPDP. Our work is therefore unique to all aforementioned techniques in the sense that:

- BiLO-CPDP is the first of its kind to formulate bi-level programming for the parameter optimization of CPDP.
- BiLO-CPDP automatically optimizes not only the parameters, but also discover the possible combination of transfer learner and classifier from a given portfolio.
- We show that exploring the combination of transfer learner and classifier is more important than the their parameters tuning, the former should thus deserve more computational budget. In this regard, the bi-level programming formulated in BiLO-CPDP provides better flexibility to achieve such a requirement of fine-grained budget allocation.

7 THREATS TO VALIDITY

Similar to many empirical studies in software engineering, our work is subject to threats to validity.

Construct threats can be raised from the experiment uncertainty caused by the learning and optimization. To mitigate this, we have repeated 30 runs for each techniques and compare the techniques using Scott-Knott test [27], supported by Wilcoxon signed-rank test [48] and A_{12} effect size metric [46]. Therefore, whenever we report *"A is better than B"*, we imply that A is indeed statistically better with large effect size. The single metric AUC may also subject to such a threat. However, AUC was chosen mainly due to its parameter-free nature and high reliability as reported in the machine learning community [24].

Internal threats can be related to the parameter setting, which in our case the key parameter is the time budget for optimization. Indeed, a different budget may affect the result, and therefore we have set a total budget following the state-of-the-practice suggested in the AutoML community [8], which is reasonable given the required runs. We have also investigated the relative importance of budget allocation between the upper- and lower-level in BiLO-CPDP.

External threats are concerned with whether the findings are generailzable to other projects. To mitigate such, as discussed in Section 4, our 20 projects cover a wide spectrum of the real-world cases with diverse characteristics, each of which was used as the target domain data to be predicted using the other 19 ones as sources.

8 CONCLUSION

The choice of combination of transfer learner and classifier along with their hyper-parameter settings have a significant impact to the performance of CPDP model. In this paper, we propose BiLO-CPDP, a tool that is able to automatically develop a high-performance CPDP model for the given CPDP task. Specifically, BiLO-CPDP, for the first time, formulates the automated CPDP model discovery problem from a bi-level programming perspective. In particular, the upper-level optimization routine searches for the right combination of transfer learner and classifier while the lower-level optimization routine optimizes the corresponding hyper-parameters associated with the chosen combination. Furthermore, the hierarchical optimization paradigm allows a more flexible control of the computational budget at both levels. From our empirical study, we have shown that BiLO-CPDP

- automatically develops a better CPDP model comparing to 21 state-of-the-art CPDP techniques with hand-crafted combination and reported parameter settings.
- overwhelmingly outperforms Auto-Sklearn, a state-of-the-art AutoML tool, and the single-level optimization variant of BiLO-CPDP.
- allows software engineers to set more search budget for the upper-level, which significantly boosts the performance.

BiLO-CPDP showcases the importance of automatically optimizing the combination of transfer learners and classifiers, together with their parameters. This paves a new way to enable more intelligent parameter optimization and adaptation for CPDP model building. In future, we seek to consider multiple objectives within the bi-level programming and to investigate more precise effects of allocating budget between the two levels. We also plan to further distinguish between the parameters for transfer learner and classifier at the low-level, as it has been shown that the parameter tuning of the former is more important than the latter [21].

REFERENCES

[1] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies. 2019. How to "DODGE" Complex Software Analytics. *IEEE Transactions on Software Engineering* (2019), 1–1.

[2] Amritanshu Agrawal and Tim Menzies. 2018. Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction. In *ICSE'18: Proc. of the 40th International Conference on Software Engineering*. ACM, 1050–1061.

[3] Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE'11: Proc. of the 33rd International Conference on Software Engineering*. ACM, 1–10.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *NIPS'11: Proc. of the 25th Annual Conference on Neural Information Processing Systems*. 2546–2554.

[5] James Bergstra, Daniel Yamins, and David D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *ICML'13: Proc. of the 30th International Conference on Machine Learning*, Vol. 28. 115–123.

[6] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*. 31–41.

[7] Matthias Feurer and Frank Hutter. 2019. Hyperparameter Optimization. In *Automated Machine Learning - Methods, Systems, Challenges*. 3–33.

[8] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *NIPS'15: Proc. of the 2015 Annual Conference on Neural Information Processing Systems*. 2962–2970.

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

[10] Fred Glover and Manuel Laguna. 1998. *Tabu Search*. Vol. 1–3. Springer US, 2093–2229.

[11] Zhimin He, Fayola Peters, Tim Menzies, and Ye Yang. 2013. Learning from open-source projects: An empirical study on defect prediction. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 45–54.

[12] Steffen Herbold. 2013. Training data selection for cross-project defect prediction. In *ESEM'13: Proc. of the 9th International Conference on Predictive Models in Software Engineering*. 1–10.

[13] Steffen Herbold. 2017. A systematic mapping study on cross-project defect prediction. *CoRR* abs/1705.06429 (2017).

[14] Steffen Herbold, Alexander Trautsch, and Jens Grabowski. 2018. A Comparative Study to Benchmark Cross-Project Defect Prediction Approaches. *IEEE Trans. Software Eng.* 44, 9 (2018), 811–833.

[15] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2019. A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction. *IEEE Trans. Software Eng.* 45, 2 (2019), 111–147.

[16] Seyedrebvar Hosseini, Burak Turhan, and Mika Mäntylä. 2018. A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. *Information and Software Technology* 95 (2018), 296–312.

[17] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION5: Proc. of the 5th International Conference Learning and Intelligent Optimization (Lecture Notes in Computer Science)*, Vol. 6683. Springer, 507–523.

[18] Yue Jiang, Bojan Cukic, and Tim Menzies. 2008. Can data transformation help in the detection of fault-prone modules?. In *DEFECTS*. ACM, 16–20.

[19] Marian Jureczko and Lech Madeyski. 2010. Towards identifying software project clusters with regard to defect prediction. In *PROMISE'10: Proc. of the 6th International Conference on Predictive Models in Software Engineering*. 9.

[20] Akif Günes Koru and Hongfang Liu. 2005. An investigation of the effect of module size on defect prediction using static measures. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[21] Ke Li, Zilin Xiang, Tao Chen, Shuo Wang, and Kay Chen Tan. 2020. Understanding the Automated Parameter Optimization on Transfer Learning for CPDP: An Empirical Study. In *ICSE'20: Proc. of the 42th International Conference on Software Engineering*. accepted for publication.

[22] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. 2018. Progress on approaches to software defect prediction. *IET Software* 12, 3 (2018), 161–175.

[23] Nachai Limsettho, Kwabena Ebo Bennin, Jacky W Keung, Hideaki Hata, and Kenichi Matsumoto. 2018. Cross project defect prediction using class distribution estimation and oversampling. *Information and Software Technology* 100 (2018), 87–102.

[24] Charles X. Ling, Jin Huang, and Harry Zhang. 2003. AUC: a Statistically Consistent and more Discriminating Measure than Accuracy. In *IJCAI'03: Proc. of the 8th International Joint Conference on Artificial Intelligence*. 519–526.

[25] Thilo Mende. 2010. Replication of defect prediction studies: problems, pitfalls and recommendations. In *PROMISE'10: Proc. of the 6th International Conference on Predictive Models in Software Engineering*. 5.

[26] Thilo Mende and Rainer Koschke. 2009. Revisiting the evaluation of defect prediction models. In *PROMISE'09: Proc. of the 5th International Workshop on Predictive Models in Software Engineering*. 7.

[27] Nikolaos Mittas and Lefteris Angelis. 2013. Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm. *IEEE Trans. Software Eng.* 39, 4 (2013), 537–551.

[28] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. 452–461.

[29] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *ICSE'13: Proc. of the 35th International Conference on Software Engineering*. 382–391.

[30] Chao Ni, Wang-Shu Liu, Xiang Chen, Qing Gu, Dao-Xu Chen, and Qi-Guo Huang. 2017. A cluster based feature selection method for cross-project software defect prediction. *Journal of Computer Science and Technology* 32, 6 (2017), 1090–1107.

[31] Muhammed Maruf Öztürk. 2019. The impact of parameter optimization of ensemble learning on defect prediction. *The Computer Science Journal of Moldova* 27, 1 (2019), 85–128.

[32] Fayola Peters, Tim Menzies, Liang Gong, and Hongyu Zhang. 2013. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1054–1068.

[33] Shaojian Qiu, Lu Lu, and Siyu Jiang. 2018. Multiple-components weights model for cross-project software defect prediction. *IET Software* 12, 4 (2018), 345–355.

[34] Yubin Qu, Xiang Chen, Yingquan Zhao, and Xiaolin Ju. 2018. Impact of Hyper Parameter Optimization for Cross-Project Software Defect Prediction. *International Journal of Performability Engineering* 14, 6 (2018), 1291–1299.

[35] Yubin Qu, Xiang Chen, Yingquan Zhao, and Xiaolin Ju. 2018. Impact of Hyper Parameter Optimization for Cross-Project Software Defect Prediction. *International Journal of Performability Engineering* 14, 6 (2018).

[36] Foyzur Rahman, Daryl Posnett, and Premkumar T. Devanbu. 2012. Recalling the "imprecision" of cross-project defect prediction. In *FSE'12: Proc. of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 61.

[37] Duksan Ryu, Okjoo Choi, and Jongmoon Baik. 2016. Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empirical Software Engineering* 21, 1 (2016), 43–71.

[38] Duksan Ryu, Jong-In Jang, and Jongmoon Baik. 2015. A hybrid instance selection using nearest-neighbor for cross-project defect prediction. *Journal of Computer Science and Technology* 30, 5 (2015), 969–980.

[39] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. 2014. Code-smell detection as a bilevel problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 1 (2014), 1–44.

[40] Dilan Sahin, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. 2015. Model transformation testing: a bi-level search-based software engineering approach. *Journal of Software: Evolution and Process* 27, 11 (2015), 821–837.

[41] Martin J. Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data Quality: Some Comments on the NASA Software Defect Datasets. *IEEE Trans. Software Eng.* 39, 9 (2013), 1208–1215.

[42] Ankur Sinha, Pekka Malo, and Kalyanmoy Deb. 2018. A Review on Bilevel Optimization: From Classical to Evolutionary Approaches and Applications. *IEEE Trans. Evolutionary Computation* 22, 2 (2018), 276–295.

[43] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering*. 321–332.

[44] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2019. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering* 45, 7 (2019), 683–711.

[45] Burak Turhan, Tim Menzies, Ayse Basar Bener, and Justin S. Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14, 5 (2009), 540–578. https://doi.org/10.1007/s10664-008-9103-7

[46] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong.

[47] Heinrich Von Stackelberg. 2010. *Market structure and equilibrium*. Springer Science & Business Media.

[48] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods.

[49] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. ReLink: recovering links between bugs and changes. In *ESEC/FSE'11: Proc. of 19th ACM SIGSOFT Symposium on the Foundations ofSoftware Engineering and 13th European Software Engineering Conference*. 15–25.

[50] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 182–191.

[51] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. 2018. How Far We Have Progressed in the Journey? An Examination of Cross-Project Defect Prediction. *ACM Trans. Softw. Eng. Methodol.* 27, 1 (2018), 1:1–1:51.

[52] Thomas Zimmermann, Nachiappan Nagappan, Harald C. Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *ESEC/FSE'09: Proc. of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 91–100.