International Journal of Innovative Computing, Information and Control
Volume **15**, Number **2**, April **2019**

ICIC International ©2019 ISSN 1349-4198
pp. **569–589**

MODEL-BASED SOFTWARE EFFORT ESTIMATION – A ROBUST COMPARISON OF 14 ALGORITHMS WIDELY USED IN THE DATA SCIENCE COMMUNITY

Passakorn Phannachitta[1] and Kenichi Matsumoto[2]

[1]College of Arts, Media and Technology
Chiang Mai University
239 Suthep, Muang, Chiang Mai 50200, Thailand
passakorn.p@cmu.ac.th

[2]Graduate School of Science and Technology
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, Japan
matumoto@is.naist.jp

Received June 2018; revised October 2018

Abstract. *The emergence of the data science discipline has facilitated the development of novel and advanced machine-learning algorithms for tackling tasks related to data analytics. For example, ensemble learning and deep learning have frequently achieved promising results in many recent data-science competitions, such as those hosted by Kaggle. However, these algorithms have not yet been thoroughly assessed on their performance when applied to software effort estimation. In this study, an assessment framework known as a stable-ranking-indication method is adopted to compare 14 machine-learning algorithms widely adopted in the data science communities. The comparisons were carried out over 13 industrial datasets, subject to six robust and independent performance metrics, and supported by the Brunner statistical test method. The results of this study proved to be stable because similar machine-learning algorithms achieved similar performance results; particularly, random forest and bagging performed the best among the compared algorithms. The results further offered evidence that demonstrated how to build an effective stacked ensemble. In other words, the optimal approach to maximizing the overall expected performance of the stacked ensemble can be derived through a balanced trade-off between maximizing the expected accuracy by selecting only the solo algorithms that are most likely to perform outstandingly on the dataset, and maximizing the level of diversity of the algorithms. Precisely, the stack combining bagging, random forests, analogy-based estimation, adaBoost, the gradient boosting machine, and ordinary least squares regression was shown to be the optimal stack for the software effort estimation datasets.*

Keywords: Software effort estimation, Data science, Kaggle, Robust statistics, Empirical software engineering

1. **Introduction.**

Software effort estimation is one of the most vital parts of the software development process. It highly influences the success of a software development project to the extent that inaccurate estimation may hinder effective project planning and prevent a software project from being completed within the schedule or budget [1, 2].

Recently, the emergence of data science competition communities has accelerated advancement in the development of machine-learning algorithms. For example, an increased number of studies publishing the winning solutions for data science competitions hosted by the well-known Kaggle [3] can be widely observed. These studies include, for example,

DOI: 10.24507/ijicic.15.02.569

the work by Abril and Sugiyama [4], and the studies mentioned in the review by Goldbloom [5]. However, the machine-learning algorithms widely recognized as the essence of the competition winners, such as the gradient boosting machine (*GBM*) and deep learning (*Deep*) [6], have not yet been extensively examined in the literature on software effort estimation.

The principle aim of this study is to determine a stable performance ranking of different machine-learning algorithms that appear in the data science competition communities, and to determine whether these algorithms will also offer outstanding performance for software effort estimation. The motivational observation is that the Classification and Regression Trees (*CART*) have long been the standard benchmark datasets for benchmarking software effort estimators [7, 8, 9]. However, *CART* do not seem to be the algorithm of choice of the data science community. This study improves the statistically significant testing components of a robust evaluation framework known as the stable-ranking-indication method proposed by [10], which is the widely accepted evaluation framework for software effort estimators, to compare 14 machine-learning algorithms, classified into six groups, over 13 industrial software effort estimation datasets. In short, the significance of the present study is as follows.

- To the researchers' knowledge, this study offers the largest performance comparison results of the more recently developed machine-learning algorithms when utilized as software effort estimators. The selected algorithms are not only those most frequently appearing on the current leaderboard of many machine-learning competitions, but also those that are straightforwardly accessible to any practitioners through the well-known libraries, such as Scikit-learn [11] and Keras [12].
- The results obtained by the experiment are stable and show similar behaviors to algorithms whose theoretical concepts are similar. Furthermore, all the experimental conditions used are provided in a great detail to maximize replicability.
- The present study appears to be the first to offer concrete evidence showing that combining multiple solo-algorithms into a stacked ensemble of multi-algorithms by maximizing both the potential performance of each individual algorithm and to maximizing the level of diversity of the selected individual algorithms has great potential to offer a more accurate estimation for software effort as compared with other different stacking approaches.
- The results in the present study are generally in good agreement with those of the theoretical works in the data science community; however, they appear to be contrary to earlier studies in software effort estimation. Precisely, *CART* did not appear to be an outstanding algorithm despite being known as the standard benchmark and the state-of-the-art algorithm for the software effort estimation community. On the contrary, many ensemble-based techniques, such as random forest (*RF*), bootstrap aggregation (*Bagging*), and GBM, which have been part of the winning solutions of many recent machine-learning competitions, have shown a significantly better performance over software effort estimation datasets.

The details of the evaluation and analysis are organized in this paper as follows. Section 2 provides the essential background and related works. Section 3 explains the 14 machine-learning algorithms along with their implementation in the study. Section 4 elaborates the use of the stable-ranking-indication method throughout this study. Section 5 presents the results and findings. Section 6 further discusses the results and validity of threats of the study. Finally, the conclusion is provided in Section 7.

2. Background.

2.1. Model-based software effort estimation. Software effort estimation is the process of estimating the amount of effort necessary to complete a particular software development project [13, 14]. Software project managers require an accurate and reliable estimation that can exploit information clear enough for them to ensure that the project will be completed on time and under budget [1, 2].

According to the surveys on the techniques used in software effort estimation [15, 16], research studies are more likely to encourage the adoption of model-based techniques in real-world applications. The key reasons are that model-based techniques do not rely on human intuition, which is difficult to justify and difficult to transfer between human individuals. More importantly, model-based techniques are theoretically explicable and replicable. To date, several variations of model-based techniques have been made available; those that appear most commonly in the literature are linear models, decision trees, and instance-based algorithms [2, 10, 13]. The methodology behind these techniques is to attempt to generate the estimated effort value to be the closest possible to the actual effort. This estimation is based on the value of one or multiple explanatory variables that may describe the actual effort. For example, the ordinary least squares regression assumes the linear relationship between the effort variable (y) and the explanatory variables (X) to be in the following form:

$$\hat{y}(\beta, X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n, \tag{1}$$

where $\hat{y}$ is the estimated effort value of y, β_i is the intercept adjusted by the model for which the difference between $\hat{y}$ and y is minimized, and X_i is a vector of the software project feature, which is treated as an explanatory variable.

2.2. Nomination of the best algorithm for software effort estimation. Ranking instability appears as a main problem in research studies in software effort estimation because different types of estimation algorithms are suggested for practical applications. This problem consequently hinders the transfer of knowledge between the research community and the industries, since conflicting results can make the practitioners confront difficulties in deciding which performance results should be followed. Shepperd and Kadoda [17] are among the first who presented the problem to software effort estimation communities. They undertook a simulation study to investigate the conflicting results and suggested that the conflict in the results is mainly subject to the differences in the dataset characteristics, such as distribution, as used in different earlier studies.

In 2004, the tera-PROMISE repository was founded by Menzies and Sayyad [18]. The repository soon became one of the most widely used repositories by software engineering researchers. With a great variety of software effort estimation datasets gathered from various sources available in the repository, researchers have been able to revisit the ranking instability problem. For example, Keung et al. [10] adopted an assessment framework to target the stable and precise ranking of software effort estimators. The assessment is based on the total number of times an estimator is statistically significantly outperformed by all the other estimators of the comparison set. In Keung et al. [10], nine machine-learning algorithms commonly appearing in the software effort estimation literature together with ten data preprocessors were compared subject to even error measures. Supported by the Wilcoxon rank-sum test [19], *CART* was nominated to be the most suitable algorithm for software effort estimation datasets. Subsequently, *CART* has generally appeared as the standard benchmark algorithm in prominent literature on software effort estimation, such as in several studies by Kocaguneli et al. [7, 8, 9].

2.3. Data science and data science competitions.

Data science is an emerging field of scientific study focusing on exercising multidisciplinary expertise to support and guide the extraction and synthesis of knowledge from data. An expert data scientist team can deliver an organization a massive competitive advantage by generating business value from the data that are already flowing throughout the organization. For example, market analysis and customer segmentation are the data analytics tasks mostly carried out to effectively market to each target customer type [20]. Overall, application of data science in industries can be used to increase the enterprise's net profit by means of maximizing the revenue, and minimizing the possible loss. This, therefore, makes the profession of a data scientist a rising career option, attracting significant attention from the industry, which is demonstrated by the fact that Glassdoor recently nominated the vocation of a data scientist as one of the best jobs in America based on competitiveness of salary and job satisfaction [21].

Given the fact that data science is relatively new to the industry where the skill and the experience of a data scientist are strongly associated with fruitful outcomes, searching for data scientists specialized in particular business functions may not be simple. In recent days, data science competitions have become more and more commonplace in companies during the hiring of skilled and experienced data scientists. The current most popular competition is Kaggle [5], which is where many big technology companies like Facebook host competitions to tackle difficult problems for million-dollar prizes. This highly competitive environment fosters competitions in many different positive ways, such as by facilitating advancements in machine-learning techniques by motivating individuals with expertise to improve their competencies. In the authors' view, these competitions have influenced the acceleration of advancements in machine learning much faster than any other kinds of activities have in the past. Since the performance of machine-learning techniques are what determines estimation performance, there is no reason not to attempt to transfer the lessons learned from the competitions to the software effort estimation community so as to investigate the practices which are dominating the competitions' leaderboards.

3. Fourteen Machine-Learning Algorithms.

At the time of writing, there are voluminous machine-learning algorithms listed on the public Wikipedia page of the official Kaggle competition web site [3]. In this study, a comparison of 14 algorithms applicable to numerical estimation problems are carried out. Other algorithms were excluded because they are mainly for classification and clustering, which are inapplicable to the tasks related to software effort estimation. Implementation of the selected algorithms is made to be available in widely-recognized libraries known as Scikit-learn [11] and the Keras python library [12]. These 14 algorithms can be classified into 6 groups, as given below.

3.1. Five generalized linear models.

Ordinary least squares regression (*OLS*) is one of the simplest machine-learning algorithms for numerical estimation problems. It is highly sensitive to random error and generally produces high variance, one of the main sources of estimation errors leading to overfitting. In a study by Keung et al. [10], *OLS* ranked the worst in their experimental setup.

Ridge regression (*Ridge*) applies a regularization technique to alleviating the problem of overfitting due to the high variance generally produced by *OLS*. Generally, the potential reason for such high variance is because the resultant model develops a high regression coefficient when it attempts to fit every single data point in the dataset. To address such problems, *Ridge* imposes a penalty on the size of the coefficients; this makes the resultant model become somewhat biased, but the huge reduction in the variance makes *Ridge* generally more accurate than the simple *OLS*.

Least absolute shrinkage and selection operator regression (*LASSO*) applies regularization to avoiding the problem of overfitting with the main difference from *Ridge* being that *LASSO* attempts to perform feature selection to eliminate highly dependent features, ultimately leading to reduction in error. *LASSO* is suitable for problems in which the datasets contain large amounts of features that may be dependent where access to feature reduction techniques is limited.

Elastic-Net regression (*ElasticNet*) attempts to combine the strengths of *Ridge* and *LASSO*, by adopting a regularization technique that applies both feature selection and the assigning of penalty on the size of the regression coefficients.

Least-angle regression (*LARS*) applies feature selection based on the correlation coefficient over iterations. Particularly, in an iteration, *LARS* finds the *OLS* model with the strongest correlation coefficient with the target variables. *LARS* is suitable for datasets having the number of features excessively higher than the number of instances.

3.2. **One decision tree.** *CART* generates a binary decision tree that best explains the target variable. Based on information gain, *CART* recursively partitions the training set into mutually exclusive instance subsets where the degree of variability, measured by the Gini index [10], within each subset is minimized. The recurrent partitioning will be terminated when no further gain can be achieved. Keung et al. [10] nominated *CART* as the best software effort estimation algorithm in their proposal of the stable-ranking-indication method.

3.3. **One support vector machine.** *Support vector regression* (*SVR*) attempts to find a certain function that best fits the dataset based on optimization. If there is no explicit linear relationship between the descriptive and the target variables, *SVR* will utilize a kernel function to map these variables into a higher dimension and finds a hyperplane that can simply fit the dataset. This therefore transforms a simple regression problem into optimization problem that focuses on finding the optimal hyperplane and minimizing the error residual. In addition, the use of kernel functions enables *SVR* to handle non-linear regression, making it a powerful technique for handling data with an arbitrary shape.

3.4. **One instance-based learning.** *Analogy-based estimation* (*ABE*) applies the k-nearest neighbor algorithm to estimating the target variable. Applying *ABE* in software effort estimation, the estimated effort value for a new software project can be defined as the total amount of effort used in past similar software projects [22]. Despite its simplicity, *ABE* is a favorite technique for both research communities and the industry [23] mostly because it usually delivers excellent accuracy as well as being intuitively easy to understand and use.

3.5. **Two neural networks.** *Artificial neural networks* (*ANN*) simulates the human brain mechanism to solve machine-learning problems. It is composed of interconnected nodes lying in different layers to form a network. Generally, *ANN* consists of an input layer, an output layer, and one hidden layer. In the case of numerical estimation problems, the task for *ANN* is to learn the optimal weight values assigned to each edge in the network in order to produce the estimated target variable with minimal error residual.

Deep can be considered an *ANN* with more than one hidden layer. This, thus, forms a much more complex network. The distinctive features inherent in such complex networks on implementation are not only performance and flexibility in handling non-linear problems but also a complete automated feature selection, particularly useful for massive and unstructured data. The theoretical work of *Deep* has been available for decades [6]; however, the bottlenecks in the technologies to handle such a computationally expensive network had hindered the workforce as regards its empirical support. More recently, the establishment of massively parallel chips housed in graphic processing units [24] has caused a significant leap in the availability of computation, ultimately making *Deep* accessible by consumer markets.

3.6. **Four ensemble learning models.**

Bagging builds an ensemble model by randomly generating instance subsets (with replacements) from the original training datasets, then applying a learning model, e.g., a regression tree for general cases, to all the instance subsets. The final estimated value is the mean value aggregated from all the estimated values produced. Theoretically [25], any ensemble method can potentially reduce the error due to undesirable model variance when the aggregated results are based on independent sampling.

RF is a particular kind of *Bagging* especially tuned for minimizing the error residual produced by regression trees. The extension from *Bagging* is that *RF* also performs feature reduction along with a subsetting of instances. With sound theoretical support and with its performance being recognized for a wide range of problems, *RF* is one of the most popular techniques for both research communities and practitioners [4].

Adaptive boosting (*AdaBoost*) first builds a regression tree over the entire training dataset. Then, it iteratively builds a weak estimator focusing on more difficult sub tasks and adds them to the stronger estimator built in the earlier iterations. The method used in building weak learners is based on assigning weights on the less accurately estimated instances higher than those of the accurately estimated. In the succeeding iteration, the model will focus more on the instances with high weight values. However, *AdaBoost* may not be able to offer high accuracy when the dataset is highly contaminated with noise [26, 27] because the adaptation toward lower accuracy instances can instead result in amplifying the erroneous data instances.

GBM is also based on the boosting principle with the main difference from *AdaBoost* being that the error residual produced by a strong learner is also fed to the training of its succeeding weak learner as an additional input. It thus enables the potential improvement of the model by adding the error-correcting components to the iterative workflow. In other words, a major improvement from the other ensemble methods is that *GBM* directly utilizes the value of the estimation error in an attempt to reduce it. In addition, in order to optimize the error correction process, *GBM* makes use of the gradient descent technique [11] as an optimization method guiding the error reduction based on boosting to move toward the resultant model with minimal overall error. Chollet [6] noted that since 2017, *GBM* has been dominating the Kaggle leaderboards for the problems in which structured data are available. Otherwise, *Deep* is more suitable.

A summary of these algorithms is shown in Table 1, along with the libraries used for their implementation in this study's experiments.

4. **Evaluation Methodologies.**

The experimental framework is mainly based on the researchers' improvement of the statistical methods used in Keung et al.'s stable ranking indication method [10]. A detailed explanation of the framework is provided throughout this section. Let the term *estimator* denote the combination of a data normalization technique, a variable selection method, and a machine-learning algorithm. This study is concerned with data normalization and variable selection in the experiment because they often have a significant impact on the estimation performance of any machine-learning task [7, 23].

Table 1. Learning algorithms and their hyperparameter configurations

Abbr.	Algorithm	Library used in the study [11, 12]
1 OLS	Ordinary least squares regression	Sklearn.linear_model.LinearRegression
2 Ridge	Ridge regression	Sklearn.linear_model.Ridge
3 LASSO	Least absolute shrinkage and selection operator regression	Sklearn.linear_model.Lasso
4 ElasticNet	Elastic-Net regression	Sklearn.linear_model.ElasticNet
5 LARS	Least angle regression	Sklearn.linear_model.Lars
6 CART	Classification and regression tree	Sklearn.tree.DecisionTreeRegressor
7 SVR	Support vector regression	Sklearn.svm.SVR
8 ABE	Analogy-based estimation	Sklearn.neighbors.KNeighborsRegressor
9 ANN	Artificial neural networks	Sklearn.neural_network.MLPRegressor
10 Deep	Deep neural networks	Keras.wrappers.scikit_learn.KerasRegressor
11 Bagging	Bootstrap aggregating	Sklearn.ensemble.BaggingRegressor
12 RF	Random forest	Sklearn.ensemble.RandomForestRegressor
13 AdaBoost	Adaptive boosting	Sklearn.ensemble.AdaBoostRegressor
14 GBM	Gradient boosting machine	Sklearn.ensemble.GradientBoostingRegressor

All the implementations of the machine-learning algorithms, data normalization techniques, and feature selection methods were developed using Python version 3.6, Scikit-learn library version 0.19.0, and Keras version 2.1.6. The experiments were carried out on an Intel Core i5 laptop with 8 GB of main memory. For the components related to the statistical test, a robust statistical test library proposed by Wilcox [19] was used. It was only available as an R package named *WRS* at the time of writing.

4.1. **Datasets.** Thirteen datasets were selected, as listed in Table 2, for the performance comparison of the algorithms. They are all available in the tera-Promise data repository (accessible through [18]). It is a large repository storing numerous software engineering datasets. As demonstrated in Table 2, Albrecht contains IBM software projects completed in the 70s [28]. Cocomo-sdr contains 12 projects from various companies in Turkey [29]. Cocomo81-e and Cocomo81-non-e are homogenized from the well-known COCOMO81 datasets [30], where the projects were completed in the embedded mode and the non-embedded mode of the COCOMO principle, respectively. Desharnais contains software projects developed in Canada in the late 90s [22]. The researchers homogenized the Desharnais dataset into two datasets based on the computer languages used in their development. Desharnais-cobol and Desharnais-4GL are those that were developed in Cobol and 4GL languages, respectively. The Finnish dataset contains project cases from different companies in Finland [31]. Kemerer contains large business applications [32]. Maxwell contains projects from commercial banks in Finland [33]. Miyazaki94 contains projects developed in the COBOL language [34]. Nasa93-c1, Nasa93-c2, and Nasa93-c5 are decomposed from the Nasa93 dataset [35] by using the development center variables. The postfixes c-1, c-2, and c-5 indicate the centers at which each project was developed.

To maximize the generalization ability of empirical results, Kitchenham et al. [36] suggested that the distributions of all the experimental datasets should be required to be totally independent. Following the techniques discussed in Kitchenham et al., Figure

Table 2. Experimental datasets

	Datasets	#Features	#Instances	The development effort		
				Min	Median	Max
1	Albrecht	7	24	0.5	11.5	105.2
2	Cocomo-sdr	23	12	1.0	3.5	22.0
3	Cocomo81-e	17	28	9.0	354.0	11400.0
4	Cocomo81-non-e	17	35	5.9	50.0	6400.0
5	Desharnais-cobol	8	67	805.0	3913.5	23940.0
6	Desharnais-4GL	8	10	546.0	1123.5	5880.0
7	Finnish	4	38	460.0	5430.0	26670.0
8	Kemerer	5	15	23.2	130.3	1107.3
9	Maxwell	24	62	583.0	5189.5	63694.0
10	Miyazaki94	7	48	5.6	38.1	1586.0
11	Nasa93-c1	20	12	24.0	66.0	360.0
12	Nasa93-c2	20	37	8.4	82.0	1350.0
13	Nasa93-c5	20	39	72.0	571.4	8211.0

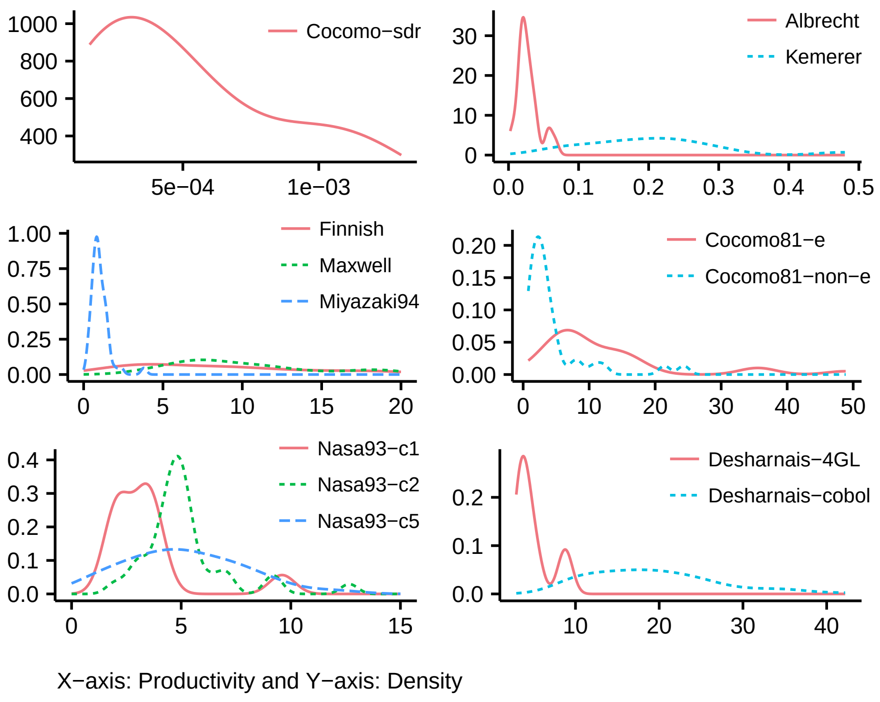

Figure 1. The kernel density plots of the selected datasets showing their high level of independence in software productivity, i.e., effort/software size

1 illustrates the kernel density plots of the distributions of the development productivity calculated by dividing the effort variable with the software size variable [23] for all the datasets given in Table 2. Overall, the kernel density plots confirm that our 13 selected datasets were independent.

4.2. **Training/test sample generation.** The leave-one-out approach [13] is the selected sampling technique for the present study. It is a technique that makes one instance become the test instance of a model built from the remaining instances. This technique will be repeated until all the instances are tested. The leave-one-out approach is opted over other popular alternatives, such as 10-fold cross validation [37], based on the suggestion from the study by Kocaguneli and Menzies [13] that the leave-one-out approach could be concluded as the most suitable sampling technique for software effort estimation studies because it is more robust when experimenting on small- and medium-sized datasets, i.e., containing less than 100 instances.

4.3. **Data preprocess.** For each experiment, nine pairs of data preprocessors were initially generated based on three data normalization techniques and three feature subset selection methods. The single best preprocessor for a certain training instance set is determined by using the widely used grid search technique [38]. Precisely, a process pipeline that first examines a normalization technique, followed by a variable selection method, and finally the learning algorithm, was implemented. The grid search was applied to these pipelines to determining the one that best fits the training instance being examined. This pipeline is then applied to the test instance to estimating the effort.

The three candidate normalization techniques are as follows. *Apply nothing* means to leave all data values unadjusted. *Interval* 0-1 normalizes each feature value x_i of a continuous feature x by subtracting the minimum value of x of x_i, and dividing the product by the difference between the maximum value of x and the minimum value of x. This technique is mostly used in the literature on software effort estimation, such as [8, 10], mainly to ensure equal influence of all the features. *Logarithmic transformation* converts all the continuous features to a natural logarithmic scale. This technique is a very simple procedure that approximates a normal distribution [39], enabling a wide range of statistical methods that require normal distribution, such as correlation analysis, to correctly capture the characteristic of the datasets.

For the choices of variable selection methods, the three candidate methods are as follows. *Select all features* means to select all the explanatory variables of the dataset. *Recursive feature elimination* iteratively removes variables from the dataset until the termination criteria are met, e.g., the default configuration of the Scikit-learn implementation is to eliminate half of the variables from the initial set. For each iteration, *Recursive feature elimination* fits models, e.g., *OLS*, over all the possible variable subsets with the size of the current subset minus one. Then, it ranks the accuracy score, e.g., the mean squared error, of all the models and eliminates the variable associated with the model with the least accuracy. *Principal component analysis* transforms the dataset by mapping its data to a new lower-dimensional space without dropping any important information. This transformation is done by decomposing the entire dataset into orthogonal components that explain the maximum variance.

4.4. **Hyperparameter optimization.** Hyperparameter optimization is an influential factor toward the performance of many empirical software engineering tasks, such as software defect prediction [40]. Table 3 presents a set of hyperparameter configurations that the researchers' attempted to optimize for each of the 14 machine learning algorithms. Particularly, the selection of the optimal hyperparameter is included in the same process pipeline, such that the grid search has determined the three-tuple (data normalization technique, feature subset selection method, and the best hyper-parameter set) for every single training instance set examined in the experiments of the study.

4.5. **Performance metrics.** The performance metrics mostly used in the software effort estimation literature are the quantified levels of estimation errors [2, 10, 41]. Regarding the stable-ranking-identification method, it is recommended that the largest possible number

Table 3. Learning algorithms and their set of their hyperparameters for optimization. α is the regularization strength used in linear models. ρ is the parameter used in scaling the regularizers exclusive for *ElasticNet*. C is the penalty parameter of the error term used in training an SVR model. N is the number of instances in a dataset.

Algorithm	Hyperparameter set
OLS	–
Ridge	$\alpha = \{0.01, 0.05, 0.1, 0.5, 1\}$
LASSO	$\alpha = \{0.01, 0.05, 0.1, 0.5, 1\}$
ElasticNet	$\alpha = \{0.01, 0.05, 0.1, 0.5, 1\}$, $\rho = \{0, 0.2, 0.4, 0.6, 0.8, 1\}$
LARS	$\alpha = \{0.01, 0.05, 0.1, 0.5, 1\}$
CART	minimum samples split = $\{2, 3, \ldots, \lceil\sqrt{N}\rceil\}$
SVR	kernel = {linear, rbf}, C = $\{1, 10, 100, 1000\}$
ABE	number of analogs = $\{1, 3, 5\}$, distance = {Manhattan, Euclidean}
ANN	hidden layer sizes = $\{2, 4, 8, \ldots, \lceil\log_2(N)\rceil\}$, activation = relu, solver = adam
Deep	hidden layer sizes = $\{2, 4, 8, \ldots, \lceil\log_2(N)\rceil\}$, number of hidden layers = $\{2, 4\}$, batch size = N, epochs = 20, activation = relu, solver = adam
Bagging	number of estimators = $\{2, 4, 8, \ldots, 256\}$, max samples = $\{0.5, 1.0\}$
RF	number of estimators = $\{2, 4, 8, \ldots, 256\}$, minimum samples split = $\{2, 3, \ldots, \lceil\sqrt{N}\rceil\}$, max depths = $\lceil\sqrt{N}\rceil$
AdaBoost	number of estimators = $\{2, 4, 8, \ldots, 256\}$, learning rate = $\{0.001, 0.01, 0.1\}$
GBM	number of estimators = $\{2, 4, 8, \ldots, 256\}$, minimum samples split = $\{2, 3, \ldots, \lceil\sqrt{N}\rceil\}$, max depths = $\lceil\sqrt{N}\rceil$, learning rate = $\{0.001, 0.01, 0.1\}$

of performance metrics be used to maximize the credibility of the performance conclusion. The procedure used in selecting the performance metrics is as given below.

1) A literature review on software effort estimation was carried out over 3 years of publications appearing in the IEEE Xplore digital library [42].
2) The surveys suggested ten metrics; namely, the *Mean Magnitude of Relative Error (MMRE)*, the *Prediction at level 25 (Pred(25))*, the *Mean Magnitude of Error Relative to the Estimate (MMER)*, the *Mean Absolute Error (MAE)*, the *Mean of the Balanced Relative Error (MBRE)*, the *Inverted Mean of the Balanced Relative Error (MIBRE)*, the *Standard Deviation (SD)*, the *Relative Standard Deviation (RSD)*, the *Logarithmic Standard Deviation (LSD)*, and the *Standardized Accuracy (SA)* which have all appeared in more than one publication [1, 14, 23, 43, 44].
3) *MMRE* and *Pred(25)* were removed from the list since they have been widely criticized as being biased metrics [14, 41].
4) The redundancy analysis was carried out by applying the *redun* function of the R package, Hmisc [45], over the estimation results produced by the ten performance metrics.
5) Several metrics were very likely to be redundant; thus, only six independent metrics were selected and applied throughout the experiments of the present study. The six metrics and their formulas are depicted as follows:

$$MAE = mean\left(\forall i\ |y_i - \hat{y}_i|\right), \tag{2}$$

where y_i and $\hat{y}_i$ denote the actual effort and the estimated effort for a project$_i$, that is, for the i^{th} instance of the dataset, respectively.

$$RSD = \sqrt{\frac{\sum_{i=1}^{N}\left(\frac{y_i - \hat{y}_i}{ss_i}\right)^2}{N-1}}, \tag{3}$$

where N is the total number of instances of a dataset, and ss_i denotes the explanatory variable describing the software size of an instance i. For the dataset where the adjusted function points (*afp* [23]) are available, ss_i will be *afp*. Otherwise ss_i will denote one single variable that best describes the software size, such as line of codes.

$$LSD = \sqrt{\frac{\sum_{i=1}^{N}\left(e_i - \left(-\frac{s^2}{2}\right)\right)^2}{N-1}}, \tag{4}$$

where e_i is given by $\ln(y_i) - \ln(\hat{y}_i)$, and s^2 is the variance of e_i. As noted in the study proposing *LSD* [41], the mean and the variance of errors of a model used by the logarithm are equal to $s^2/2$ and s^2, respectively, if they exhibit normal distribution.

$$MBRE = mean\left(\forall i \frac{|y_i - \hat{y}_i|}{\min(y_i, \hat{y}_i)}\right) \tag{5}$$

$$MMER = mean\left(\forall i \frac{|y_i - \hat{y}_i|}{\hat{y}_i}\right) \tag{6}$$

$$SA = \left(1 - \frac{MAE_{P_j}}{MAE_{random}}\right), \tag{7}$$

where MAE_{P_j} is the *MAE* of the estimator; that is, the pipeline, P_j is being evaluated, and MAE_{random} is the *MAE* of a large number, for example, 1000 runs [43], of random guesses. *SA* shows how much better P_j is than to random guessing the effort by simply picking an effort value from the dataset.

4.6. **Pairwise comparison.**

The overall performance of an effort estimator is determined by the number of times it was statistically significantly outperformed by the others, where, 0 times indicates the ideal performance result. This counting method is known as *win-tie-loss* statistics, mainly used in recent and prominent literature on empirical software engineering, such as [7, 8, 40, 43]. For the testing of statistical significance, Kitchenham et al. [36] recommended the Brunner test [46] as the most suitable method for datasets having a size and distribution similar to the common software effort estimation datasets.

In the experiment, a counter named *losses* was associated with an estimator. When the Brunner test suggests that the performance between any pair of estimators being evaluated is significantly different, that is, the p-value is less than 0.05, the *losses* counter associated with the estimator with less accuracy will undergo an increment. Following Keung et al. [10], the conclusion on the performance of the various estimators can be stably drawn from this type of analysis if and only if the estimators based on similar algorithms are ranked closely to each other.

5. **Results.**

5.1. **The performance comparison.**

Table 4 shows the ranking results of the 14 estimators. Each column of the table sorts the estimators with regard to their total *losses*. Given that 14 estimators were compared using six performance metrics, the maximum *losses* an estimator can achieve for each dataset is 13 × 6 = 78. In these results, the

Table 4. Comparison results sorted by the number of *losses* in ascending order. Lower *losses* indicate better performance. Entries highlighted in lightgray represent these estimators with total *losses* lower than 5% of the maximum possible *losses*, which is 13 × 6 × 0.05 = 3.9.

Rank	Albrecht	Cocomo-sdr	Cocomo81-e	Cocomo81-non-e	Desharnais-Cobol
1	ABE(0)	ABE(0)	ANN(0)	GBM(0)	ANN(0)
2	AdaBoost(0)	Bagging(0)	Bagging(1)	Bagging(3)	Deep(0)
3	Bagging(0)	ElasticNet(0)	RF(1)	AdaBoost(4)	ElasticNet(0)
4	CART(0)	OLS(0)	ABE(3)	ABE(6)	GBM(0)
5	GBM(0)	RF(0)	CART(9)	RF(6)	LARS(0)
6	OLS(0)	Ridge(0)	Deep(10)	CART(10)	LASSO(0)
7	RF(0)	SVR(0)	SVR(10)	SVR(13)	OLS(0)
8	Ridge(0)	CART(1)	ElasticNet(12)	Ridge(20)	RF(0)
9	SVR(0)	AdaBoost(2)	GBM(13)	OLS(22)	Ridge(0)
10	ANN(6)	GBM(2)	Ridge(14)	LASSO(23)	SVR(0)
11	ElasticNet(10)	LARS(2)	AdaBoost(18)	ElasticNet(24)	Bagging(2)
12	LASSO(11)	LASSO(2)	OLS(22)	LARS(34)	ABE(8)
13	LARS(12)	ANN(5)	LASSO(27)	ANN(40)	CART(19)
14	Deep(27)	Deep(40)	LARS(33)	Deep(46)	AdaBoost(41)
Rank	Desharnais-4GL	Finnish	Kemerer	Maxwell	Miyazaki94
1	ABE(0)	ABE(0)	ElasticNet(0)	ABE(0)	AdaBoost(0)
2	Bagging(0)	ANN(0)	LASSO(0)	ANN(0)	GBM(0)
3	RF(1)	Bagging(0)	Ridge(0)	Deep(0)	RF(0)
4	CART(3)	GBM(1)	OLS(2)	RF(0)	Bagging(1)
5	GBM(4)	RF(1)	CART(3)	SVR(5)	ABE(2)
6	AdaBoost(5)	LARS(2)	AdaBoost(4)	AdaBoost(6)	OLS(2)
7	SVR(8)	LASSO(2)	SVR(6)	GBM(6)	Ridge(5)
8	LARS(10)	OLS(2)	LARS(10)	Bagging(7)	LASSO(6)
9	LASSO(10)	Ridge(2)	RF(12)	LARS(10)	ElasticNet(9)
10	OLS(10)	AdaBoost(3)	GBM(27)	LASSO(10)	CART(10)
11	ANN(13)	ElasticNet(4)	Bagging(30)	OLS(10)	LARS(13)
12	Ridge(16)	Deep(7)	Deep(30)	Ridge(10)	ANN(26)
13	ElasticNet(17)	CART(8)	ABE(34)	ElasticNet(13)	SVR(30)
14	Deep(52)	SVR(19)	ANN(60)	CART(40)	Deep(54)
Rank	Nasa93-c1	Nasa93-c2	Nasa93-c5		
1	ABE(0)	ABE(1)	Bagging(0)		
2	AdaBoost(0)	AdaBoost(2)	RF(0)		
3	CART(0)	ElasticNet(2)	SVR(0)		
4	OLS(0)	SVR(5)	ABE(1)		
5	RF(0)	Bagging(6)	GBM(1)		
6	GBM(1)	LARS(7)	AdaBoost(2)		
7	LARS(1)	ANN(8)	LASSO(3)		
8	LASSO(1)	RF(8)	Ridge(3)		
9	Bagging(2)	LASSO(9)	LARS(4)		
10	ElasticNet(4)	GBM(17)	OLS(4)		
11	Ridge(4)	Ridge(18)	ElasticNet(5)		
12	ANN(23)	CART(20)	CART(6)		
13	SVR(36)	OLS(23)	Deep(7)		
14	Deep(64)	Deep(56)	ANN(9)		

performance of the estimators was not highly consistent over the selected datasets, the ensemble-learning algorithms, and *ABE* were observed to perform better than the others. *Deep* was observed to perform to worst. Another thing to observe in these results is that tie-ranks largely appeared in a way that a simple rank aggregation, for example, the mean rank, may not be fully appropriate for illustrating the performance of the estimation. For example, *Bagging* was ranked #11 in the Desharnais-Cobol datasets; however, its total *losses* were only 2/78 times. Subsequently, for each dataset, we defined that the estimators which achieved the total *losses* less than 5% of the maximum possible losses (i.e., $78 \times 0.05 = 3.9$) as being the estimators performing equally highly. These estimators are highlighted in gray in Table 4.

Table 5 summarizes the results of Table 4 by summing the number of datasets for which the estimators performed highly or poorly. Three bands of performance were defined: *losses* $\in [0\%, 5\%)$ represent the estimator performing the best for their corresponding datasets. For the other bands, *losses* $\in [5\%, 15\%)$ and *losses* $\in [15\%, \infty)$ represent the moderate performers and the poor performers of their corresponding datasets, respectively. The estimators with the higher number of datasets that fall in band *losses* $\in [0\%, 5\%)$ are the overall best performing estimators in terms of both accuracy and consistency (i.e., rarely outperformed by any other estimator across multiple datasets).

Table 5. The overall *losses* over 13 datasets. Entries are sorted by the number of datasets for which an algorithm achieved total *losses* less than 5% of the maximum possible *losses* (i.e., 3.9) in the experiment.

Rank	Algorithm	losses $\in [0\%, 5\%)$	losses $\in [5\%, 15\%)$	losses $\in [15\%, \infty)$
1	RF	10/13	2/12	1/13
2	Bagging	10/13	2/12	1/13
3	ABE	10/13	2/12	1/13
4	GBM	8/13	2/13	3/13
5	AdaBoost	7/13	4/13	2/13
6	OLS	7/13	3/13	3/13
7	CART	6/13	5/13	2/13
8	LASSO	6/13	5/13	2/13
9	Ridge	6/13	3/13	4/13
10	ElasticNet	4/13	5/13	4/13
11	LARS	4/13	5/13	4/13
12	SVR	4/13	5/13	4/13
13	ANN	4/13	3/13	6/13
14	Deep	2/13	3/13	8/13

In addition to the results of Table 4, Table 5 clearly shows that the estimators can be consistently divided into three bands based on their overall performance. That is, *RF*, *Bagging*, and *ABE*, performed the best overall, as shown by the highest number of datasets they achieved for *losses* less than 5%. These algorithms performed poorly, that is, with *losses* higher than 15%, only on the Kemerer dataset. On the other end of the table, *ANN* and *Deep* were the poor performers overall. Their *losses* were found to exceed 5% for the highest number of datasets tested. As for the remaining estimators, in most cases, their *losses* were lower than 15%; however, they achieved less than 5% *losses* for significantly fewer times than *RF*, *Bagging*, and *ABE*. This finding aligns well with Keung et al. [10] that the largest proportion of algorithms being compared with the stable ranking indication method are those having mediocre performance with non-stable ranking results. Note that in Table 5, *RF* and *Bagging* were ranked higher than *ABE* because the two algorithms had total *losses*, calculated over all datasets, less than that of *ABE*.

Research studies in data science, such as Abril and Sugiyama [4], have suggested that learning algorithms based on ensemble learning are more suitable for structured datasets, where *Deep* appears to be more suitable for unstructured large datasets. Thus, the results of this study are in good agreement with the research findings of the data science community.

In summary, the assessment framework can offer a conclusive stable ranking such that estimators having similar algorithms, such as *RF* and *Bagging*, rank closely together. It can be seen that a stable conclusion regarding performance of the selected algorithms can be drawn and ranked as follows:

$$\begin{gathered}\text{Ensemble learning based on Bagging}\\ \geq \\ \text{Instance-based learning}\\ > \\ \text{Ensemble learning based on Boosting}\\ > \\ \text{General linear models} \approx \text{Decision tree} \approx \text{SVM}\\ > \\ \text{Neural networks}\end{gathered}$$

5.2. **On the value of stacked ensembles.** Kocaguneli et al. [7], based on empirical experiments extended from Keung et al. [10], observed that a combination of multiple estimators using the stacking technique, e.g., simply taking the average of the estimated target variables from many estimators, can generally improve accuracy compared to using a single estimator. Kocaguneli et al. suggested an effective scheme to generate an algorithm stack based on the selection of algorithms that perform the best overall from the analysis of the estimation performance in terms of rank changes over a large set of pairwise comparisons with regard to large sets of algorithms and datasets [7].

However, Kocaguneli et al.'s proposed stacking schemes do not consider the diversity between the combined algorithms, though diversity is the essential theoretical component of the ensemble in machine learning [47]. Elsewhere, many studies on ensemble, such as several studies by Brown et al. [25, 48, 49] and Hogarth [50], have demonstrated that an attempt to maximize the level of diversity could result in gaining much promising accuracy. To observe the trade-off between (1) selecting the solo estimators in order to maximize the expected performance of each individual estimator, and (2) selecting the solo estimators in order to maximize the diversity of the selected algorithms, four stacking schemes were compared with *RF* and *CART*. The four stacking schemes are as given below.

- *RF+Bagging+ABE* denotes the stack that maximizes the overall expected accuracy of each individual estimator. They are the highest performed estimators in this study's experiments, as demonstrated in the results of Table 4.
- *Ensembles+ABE* denotes the stack of the top 5 estimators given in Table 4. This stack somewhat trades off the overall expected accuracy with the diversity by including *AdaBoost* and *GBM* in the stack.
- *Ensembles+ABE+OLS* denotes the stack of the top 6 estimators, presented in Table 4. This stack further trades off the overall expected accuracy of each individual

estimator with the diversity by including *OLS* in the stack. Note that, only *OLS* was included because it performed within the 5% *losses* threshold for more than half of the datasets.

- *RF+ABE+OLS+CART+SVR+ANN* denotes the stack that maximizes the diversity of the estimators, for instance, all of these estimators are based on different algorithms.

Table 6 shows the results of the comparison among the four stacking schemes, *RF* and *CART* using the same approach that generated the results presented in Table 4 and Table 5. The most noteworthy result, as shown in Table 6, is that neither the maximization of the overall expected accuracy of each individual estimator nor the level of diversity yielded the best performance from the comparisons. Notwithstanding, a trade-off between the two objectives did. To the best of the researchers' knowledge, this finding is novel for the software effort estimation community and should be a good direction in the future to search for an optimum way to generate an effective effort estimator stack.

Another noteworthy fact that can be observed from Table 6 is that *Ensembles+ABE+OLS* is the best performer and that it has never exhibited *losses* exceeding 15%. In comparing the two best stacking schemes of Table 6, *Ensembles+ABE+OLS* considered the level of diversity more than *Ensembles+ABE* because it added *OLS* to the stack.

Table 6. Comparison of the four stacked ensemble schemes with *RF* and *CART*, using the approach used in generating the results of Table 4 and Table 5.

	Algorithm stacks	losses $\in$ $[0\%, 5\%)$	losses $\in$ $[5\%, 15\%)$	losses $\in$ $[15\%, \infty)$
1	Ensembles+ABE+OLS	11/13	2/13	0/13
2	Ensembles+ABE	11/13	1/13	1/13
3	RF+ABE+OLS+CART+SVR+ANN	10/13	3/13	0/13
4	RF+Bagging+ABE	5/13	5/13	3/13
5	RF	4/13	6/13	3/13
6	CART	3/13	3/13	7/13

Figure 2 illustrates the raw performance spectrums of *Ensembles+ABE+OLS*, *RF*, and *CART* based on the six performance metrics used throughout the study. Plots in this figure sort all the metrics over 13 datasets in the ascending order. The lower values indicate the better performance of *MAE*, *RSD*, *LSD*, *MBRE*, and *MMER*. In contrast, the higher values indicate the better performance of *SA*. In these plots, clearly the performance spectrums of *Ensembles+ABE+OLS* were demonstrated to be better than the other two solo-estimators for all six metrics.

In sum, stacking ensembles significantly outperformed the solo estimators in this experiment. These results align with the findings of a study of Kocaguneli et al. [7] and confirm its superiority. Furthermore, similar to Kocaguneli et al., this study found that the performance of the process of stacking ensembles relies on optimizing the set of solo estimators, where the highest expectable accuracy of stacking ensembles is likely to be obtained when the trade-off between selecting the solo estimators with the highest expected performance in the case of each individual, and maximizing the diversity of the selected algorithms is optimum.

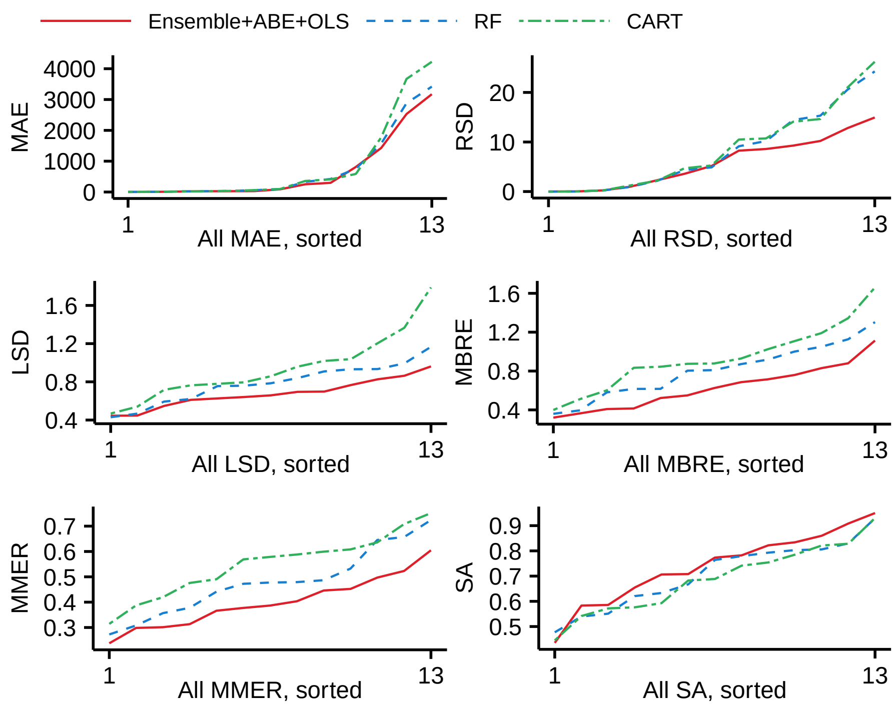

FIGURE 2. The spectrums of the six performance metrics comparing *Ensembles+ABE+OLS*, *RF*, and *CART*. Lower *MAE*, *RSD*, *LSD*, *MBRE*, and *MMER* indicate better performance. Higher *SA* indicates better performance.

6. **Discussion.**

6.1. **Findings contrary to existing studies.** The finding most contrary to earlier studies in software effort estimation is that *CART* does not appear to be an outstanding algorithm. It is the authors' belief that the findings of the present study are well substantiated. The main differences between the setup of this study and that of the study which nominated *CART* as the best non-ensemble algorithm are as follows.

- Different statistical significance test methods were used in this study. Several years after the publication of Keung et al. [10], Kitchenham et al. [36] corroborated theoretical and empirical works and recommended that the Brunner's test appears to be the superior test method in comparison to the Wilcoxon rank-sum test widely used in software effort estimation studies in the past.
- In one of the authors' past studies [51], it was observed that many performance metrics used in the literature on software effort estimation are redundant. For example, even though *SD* and *RSD* are two of the few metrics used, Foss et al. [41] proved that they are less vulnerable to bias than many other metrics. The test of redundancy using the *redun* R package suggested that only one of the two should be used. Subsequently, six metrics were used in the experiments, and the results in Figure 2 show that all the selected six metrics were not redundant.
- Kitchenham et al. [36] also suggested applying the Kernel density estimation to all the datasets used in empirical studies in order to confirm that they are independent in the distributions. Kernel density estimation was suggested over other common alternatives such as box plots because of its superior ability to capture and visualize the existence of outliers and extreme values. The level of independence is important

to justify the generalization ability of the findings obtained from the datasets. Figure 1 confirms that experimental datasets are totally independent.

Aside from the results of *CART*, *OLS* is the other algorithm whose performance significantly differed from that of earlier studies. The main reasons for *OLS* performing much better in the present study's experiments is the use of the grid search hyperparameter optimization. In the experiments of this study, the optimal choices with regard to data normalization method and feature selection techniques are exhaustively searched along with the optimal hyperparameter set. That is, *OLS* discussed in the present study can be considered as the optimal choice from all the linear models preprocessed with a majority of the data preprocessors examined in the study of Keung et al. [10]. This may confirm the importance of the hyperparameter optimization process in software analytics, such as software effort estimation. Even though the hyperparameter optimization has been widely studied in other areas such as defect prediction, there was only one single recent study by Xia et al. [44] that has explored its importance in software effort estimation.

6.2. **Findings consistent with previous studies.** The findings showing the superiority of ensemble-learning algorithms are in broad agreement with the studies in both software effort estimation and other domains. Abril and Sugiyama [4] adopted an estimator based on *RF* to approach a Kaggle competition in stock prediction and won it. In the literature on software effort estimation, the comparison of performance between the stacked ensemble and the non-stacked ensemble has not been extensively explored. That is, there seems to be no single software effort estimation study extensively exploring the potential performance of the stacked ensemble that combines multiple ensemble-based solo algorithms, such as a combination of *RF*, *Bagging*, and *AdaBoost*. Even though Minku and Yao [52] showed that *Bagging* performed better than other ensemble-learning algorithms on software effort datasets, they did not further investigate whether *Bagging* could offer better performance when combined with other algorithms as a stacked ensemble. Therefore, the present study appears to be the first that offers concrete evidence showing that combining multiple ensemble algorithms and non-ensemble algorithms into a stacked ensemble has great potential to offer a more accurate estimation for software effort.

For the results concerning the relatively poor performance of *ANN* and *Deep*, we hypothesized that significant overfitting had occurred. Chollet [6] noted that the worst overfitting was more likely to be introduced to some deep learning-based estimators when the datasets are small and have a clear structure. As for empirical evidence, Nassif et al. [53] compared multiple software effort estimators based on neural networks and deep learning algorithms and noted that all the estimators that they compared tended to generate an inaccurate estimation for nearly all of their experimental datasets.

Another result that aligns well with the studies in software effort estimation is the outstanding performance of *ABE*. From Table 4 and Table 5, it is evident that *ABE* was the only non-ensemble algorithm to achieve consistent and high rank. Even if *ABE* is based on the simple k-nearest neighbors principle which has not often shown outstanding performance elsewhere [54], the performance of *ABE* has long been recognized in the software effort estimation community [22, 23, 55]. The authors believe that its essential hypothesis, stating similar software projects with similar characteristics are likely to require similar effort to complete their development, is consistent, making the k-nearest neighbors perform highly and distinguishably for software effort estimation.

6.3. **Threats to validity.** Use of hyperparameter optimization via the grid search technique is this study's strategy as far as internal validity is concerned. For external validity, the high level of independence confirmed by the kernel density estimations is the approach that was opted to handle the possible external validity threats. Finally, the researchers attempted to maximize the construct validity by pruning a list of commonly used metrics to six robust and non-redundant metrics by carrying out a statistical redundancy test. It is important to note that recent studies in software effort estimation were often criticized for their inadequate use of biased performance metrics such as *MMRE* [41]. In contrast to this, the authors believe that the strategies that were selected in this study to approach the validity threats will promote a reliability of the results and the findings of the study.

7. **Conclusion.** The present study extensively explores whether the machine-learning algorithms that performed outstandingly in recent active data-science competitions will also perform equally well for software effort estimation. Particularly, 14 machine learning algorithms including the increasingly popular: the gradient boosting machine and deep learning, were compared over 13 benchmark-standard software effort estimation datasets [18], and all of them were confirmed as genuinely independent. Based on the stable ranking evaluation method that is most widely used in the literature on software effort estimation, the following are the key findings of this study.

- Ensemble learning algorithms based on the principle of Bootstrap aggregating, for example, *Bagging* and *RF*, performed the best overall over the 13 datasets. This finding is different from that in other areas, such as computational physics [56], where an ensemble based on the Boosting principle generally offers higher performance.
- *ABE* appeared to be the highest-performing non-ensemble learning algorithm in the comparisons. This finding aligns well with many studies in software effort estimation such as the studies by Kocaguneli et al. [8] and Shepperd and Schofield [22].
- Combining multiple effort estimators into a stacked ensemble, for instance, to simply average the estimated effort values, consistently offered higher accuracy than any of the 14 estimators compared in the study. Particularly, the stacked ensemble that offered the best overall accuracy in this study took the average of the estimated effort values produced by *Bagging*, *RF*, *ABE*, *AdaBoost*, *GBM*, and *OLS*. Each of these algorithms had its hyperparameters and data preprocessors optimized by using the grid search technique [38].
- The researchers confirm the findings of Xia et al. [44] that hyperparameter configurations strongly affect the performance offered by machine learning-based software effort estimators.
- The stable-ranking-identification method [10] is recommended for comparing multiple machine-learning algorithms. The most important lessons for the use of the method are as follows. (1) The number of datasets should be large, and all of the datasets must have significantly different distributions. Kernel density estimation [36] is recommended by the authors to examine the distribution. (2) The number of performance metrics should be large, and none of the metrics should be overlapped. A redundancy test [45] is recommended by the authors to observe this. (3) The statistical test method should be sufficiently robust. For small datasets, such as the software effort estimation datasets which generally contain less than 100 instances, the Brunner test [46] is recommended by the authors.

For future research in software effort estimation that adopts machine-learning algorithms, the authors propose that finding the optimal combination scheme for generating the optimal stacked ensemble should be undertaken. The key results of the present study hinted strongly that better selection schemes for combining the estimators should consider both the overall expected accuracy of each individual estimator as well as the level of diversity of the estimators to be combined. However, to be able to directly optimize the two objectives, further investigation is required to find a more precise way to quantify overall expected accuracy and level of diversity.

REFERENCES

[1] L. L. Minku and X. Yao, Which models of the past are relevant to the present? A software effort estimation approach to exploiting useful past models, *Automat. Softw. Eng.*, vol.24, no.3, pp.499-542, 2017.

[2] M. K. S. Ganesh and K. Thanushkodi, An efficient software cost estimation technique using fuzzy logic with the aid of optimization algorithm, *International Journal of Innovative Computing, Information and Control*, vol.11, no.2, pp.587-597, 2015.

[3] *Kaggle: Your Home for Data Science*, https://www.kaggle.com, 2018.

[4] I. M. de Abril and M. Sugiyama, Winning the Kaggle algorithmic trading challenge with the composition of many models and feature engineering, *IEICE T. Inf. Syst.*, vol.96, no.3, pp.742-745, 2013.

[5] A. Goldbloom, Data prediction competitions – Far more than just a bit of fun, *Proc. of IEEE Int. Conf. on Data Mining Workshops*, pp.1385-1386, 2010.

[6] F. Chollet, *Deep Learning with Python*, Manning Publications Co., 2017.

[7] E. Kocaguneli, T. Menzies and J. Keung, On the value of ensemble effort estimation, *IEEE T. Software Eng.*, vol.38, no.6, pp.1403-1416, 2012.

[8] E. Kocaguneli, T. Menzies, J. Keung, D. Cok and R. Madachy, Active learning and effort estimation: Finding the essential content of software effort estimation data, *IEEE T. Software Eng.*, vol.39, no.8, pp.1040-1053, 2013.

[9] E. Kocaguneli, T. Menzies, J. Hihn and B. H. Kang, Size doesn't matter? On the value of software size features for effort estimation, *Proc. of the 8th Int. Conf. on Predictive Models in Software Engineering*, pp.89-98, 2012.

[10] J. Keung, E. Kocaguneli and T. Menzies, Finding conclusion stability for selecting the best effort predictor in software effort estimation, *Automat. Softw. Eng.*, vol.20, no.4, pp.543-567, 2013.

[11] *Scikit-Learn: Machine Learning in Python*, http://scikit-learn.org/, 2018.

[12] *Keras: The Python Deep Learning Library*, https://keras.io/, 2018.

[13] E. Kocaguneli and T. Menzies, Software effort models should be assessed via leave-one-out validation, *J. Syst. Software*, vol.86, no.7, pp.1879-1890, 2013.

[14] F. Sarro, A. Petrozziello and M. Harman, Multi-objective software effort estimation, *Proc. of the 38th Int. Conf. on Software Engineering*, pp.619-630, 2016.

[15] M. Jorgensen and M. Shepperd, A systematic review of software development cost estimation studies, *IEEE T. Software Eng.*, vol.33, no.1, pp.33-53, 2007.

[16] K. Molokken and M. Jorgensen, A review of software surveys on software effort estimation, *Proc. of Int. Symp. on Empirical Software Engineering*, pp.223-230, 2003.

[17] M. Shepperd and G. Kadoda, Comparing software prediction techniques using simulation, *IEEE T. Software Eng.*, vol.27, no.11, pp.1014-1022, 2001.

[18] *Tera-PROMISE: One of the Largest Repositories of SE Research Data*, http://openscience.us/repo/, 2015.

[19] R. Wilcox, *Modern Statistics for the Social and Behavioral Sciences: A Practical Introduction*, CRC Press, 2011.

[20] K. S. Mathad, S. Chittal, S. Sharma, S. Mulik, K. Rajhansh and B. Tech, Share market analysis and prediction system using machine learning, *International Journal of Engineering Science and Computing*, vol.7, no.6, pp.12795-12800, 2017.

[21] L. Columbus, *Data Scientist Is the Best Job In America According Glassdoor's 2018 Rankings*, https://www.forbes.com/sites/louiscolumbus/2018/01/29/data-scientist-is-the-best-job-in-america-according-glassdoors-2018-rankings, 2018.

[22] M. Shepperd and C. Schofield, Estimating software project effort using analogies, *IEEE T. Software Eng.*, vol.23, no.11, pp.736-743, 1997.

[23] P. Phannachitta, J. Keung, A. Monden and K. Matsumoto, A stability assessment of solution adaptation techniques for analogy-based software effort estimation, *Empir. Softw. Eng.*, vol.22, no.1, pp.474-504, 2017.

[24] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang and V. Volkov, Parallel computing experiences with CUDA, *IEEE Micro*, vol.28, no.4, pp.13-27, 2008.

[25] G. Brown, J. L. Wyatt and P. Tiňo, Managing diversity in regression ensembles, *J. Mach. Learn. Res.*, vol.6, pp.1621-1650, 2005.

[26] E. Bauer and R. Kohavi, An empirical comparison of voting classification algorithms: Bagging, boosting, and variants, *Machine Learning*, vol.36, nos.1-2, pp.105-139, 1999.

[27] H. Zhao and S. Ram, Constrained cascade generalization of decision trees, *IEEE T. Knowl. Data. En.*, vol.16, no.6, pp.727-739, 2004.

[28] A. J. Albrecht and J. E. Gaffney, Software function, source lines of code, and development effort prediction: A software science validation, *IEEE T. Software Eng.*, vol.9, no.6, pp.639-648, 1983.

[29] A. Bakır, B. Turhan and A. B. Bener, A new perspective on data homogeneity in software cost estimation: A study in the embedded systems domain, *Software Qual. J.*, vol.18, no.1, pp.57-80, 2010.

[30] K. Srinivasan and D. Fisher, Machine learning approaches to estimating software development effort, *IEEE T. Software Eng.*, vol.21, no.2, pp.126-137, 1995.

[31] B. Kitchenham and K. Känsälä, Inter-item correlations among function points, *Proc. of the 15th Int. Conf. on Software Engineering*, pp.477-480, 1993.

[32] C. F. Kemerer, An empirical validation of software cost estimation models, *Commun. ACM*, vol.30, no.5, pp.416-429, 1987.

[33] K. Maxwell, *Applied Statistics for Software Managers*, Prentice-Hall, Englewood Cliffs, NJ, 2002.

[34] Y. Miyazaki, M. Terakado, K. Ozaki and H. Nozaki, Robust regression for developing software estimation models, *J. Syst. Software*, vol.27, no.1, pp.3-16, 1994.

[35] T. Menzies, D. Port, Z. Chen, J. Hihn and S. Stukes, Validation methods for calibrating software effort models, *Proc. of the 27th Int. Conf. on Software Engineering*, pp.587-595, 2005.

[36] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs and A. Pohthong, Robust statistical methods for empirical software engineering, *Empir. Softw. Eng.*, vol.22, no.2, pp.579-630, 2017.

[37] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence*, pp.1137-1143, 1995.

[38] V. Vo, J. Luo and B. Vo, A turning points method for stream time series prediction, *International Journal of Innovative Computing, Information and Control*, vol.9, no.10, pp.3965-3980, 2013.

[39] B. Kitchenham and E. Mendes, Software productivity measurement using multiple size measures, *IEEE T. Software Eng.*, vol.30, no.12, pp.1023-1035, 2004.

[40] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden and S. Mensah, Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, *IEEE T. Software Eng.*, 2017.

[41] T. Foss, E. Stensrud, B. Kitchenham and I. Myrtveit, A simulation study of the model evaluation criterion MMRE, *IEEE T. Software Eng.*, vol.29, no.11, pp.985-995, 2003.

[42] M. Wilde, IEEE Xplore digital library, *The Charleston Advisor*, vol.17, no.4, pp.24-30, 2016.

[43] E. Kocaguneli, T. Menzies and E. Mendes, Transfer learning in effort estimation, *Empir. Softw. Eng.*, vol.20, no.3, pp.813-843, 2015.

[44] T. Xia, R. Krishna, J. Chen, G. Mathew, X. Shen and T. Menzies, Hyperparameter optimization for effort estimation, *arXiv:1805.00336*, 2018.

[45] *Hmisc: Harrell Miscellaneous*, https://cran.r-project.org/package=Hmisc, 2018.

[46] E. Brunner, U. Munzel and M. L. Puri, The multivariate nonparametric Behrens-Fisher problem, *J. Stat. Plan. and Inf.*, vol.108, no.1, pp.37-53, 2002.

[47] X. Zhu, P. Zhang, X. Lin and Y. Shi, Active learning from stream data using optimal weight classifier ensemble, *IEEE T. Syst. Man. Cy. B.*, vol.40, no.6, pp.1607-1621, 2010.

[48] G. Brown, J. Wyatt, R. Harris and X. Yao, Diversity creation methods: A survey and categorisation, *Inform. Fusion*, vol.6, no.1, pp.5-20, 2005.

[49] G. Brown and L. I. Kuncheva, "Good" and "bad" diversity in majority vote ensembles, *Int. Workshop on Multiple Classifier Systems*, pp.124-133, 2010.

[50] R. M. Hogarth, A note on aggregating opinions, *Organ. Behav. Hum. Perf.*, vol.21, no.1, pp.40-46, 1978.

[51] P. Phannachitta, J. Keung, K. E. Bennin, A. Monden and K. Matsumoto, Filter-INC: Handling effort-inconsistency in software effort estimation datasets, *Proc. of the 23rd Asia-Pacific Software Engineering Conference*, pp.185-192, 2016.

[52] L. L. Minku and X. Yao, Ensembles and locality: Insight on improving software effort estimation, *Inform. Software Tech.*, vol.55, no.8, pp.1512-1528, 2013.

[53] A. B. Nassif, M. Azzeh, L. F. Capretz and D. Ho, Neural network models for software development effort estimation: A comparative study, *Neural Comput. Appl.*, vol.27, no.8, pp.2369-2381, 2016.
[54] J. Wei, X. Qi and M. Wang, Collaborative representation classifier based on k nearest neighbors for classification, *J. Softw. Eng.*, vol.9, no.1, pp.96-104, 2015.
[55] P. Phannachitta, Robust comparison of similarity measures in analogy based software effort estimation, *Proc. of the 11th Int. Conf. on Software, Knowledge, Information Management and Applications*, pp.1-7, 2017.
[56] T. Chen and T. He, Higgs boson discovery with boosted trees, *NIPS Workshop on High-Energy Physics and Machine Learning*, pp.69-80, 2015.