Published in IEEE Transactions on Software Engineering, October 2016, Volume 42, Issue 10, Pages 977-998.
http://doi.org/10.1109/TSE.2016.2543218

HYDRA: Massively Compositional Model for Cross-Project Defect Prediction

Xin Xia, *Member, IEEE*, David Lo, *Member, IEEE*, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang

Abstract—Most software defect prediction approaches are trained and applied on data from the same project. However, often a new project does not have enough training data. Cross-project defect prediction, which uses data from other projects to predict defects in a particular project, provides a new perspective to defect prediction. In this work, we propose a HYbrid moDel Reconstruction Approach (HYDRA) for cross-project defect prediction, which includes two phases: genetic algorithm (GA) phase and ensemble learning (EL) phase. These two phases create a massive composition of classifiers. To examine the benefits of *HYDRA*, we perform experiments on 29 datasets from the PROMISE repository which contains a total of 11,196 instances (i.e., Java classes) labeled as defective or clean. We experiment with logistic regression as the underlying classification algorithm of *HYDRA*. We compare our approach with the most recently proposed cross-project defect prediction approaches: TCA+ by Nam et al., Peters filter by Peters et al., GP by Liu et al., MO by Canfora et al., and CODEP by Panichella et al. Our results show that *HYDRA* achieves an average F1-score of 0.544. On average, across the 29 datasets, these results correspond to an improvement in the F1-scores of 26.22 , 34.99, 47.43, 28.61, and 30.14 percent over TCA+, Peters filter, GP, MO, and CODEP, respectively. In addition, HYDRA on average can discover 33 percent of all bugs if developers inspect the top 20 percent lines of code, which improves the best baseline approach (TCA+) by 44.41 percent. We also find that HYDRA improves the F1-score of Zero-R which predict all the instances to be defective by 5.42 percent, but improves Zero-R by 58.65 percent when inspecting the top 20 percent lines of code. In practice, Zero-R can be hard to use since it simply predicts all of the instances to be defective, and thus developers have to inspect all of the instances to find the defective ones. Moreover, we notice the improvement of HYDRA over other baseline approaches in terms of F1-score and when inspecting the top 20 percent lines of code are substantial, and in most cases the improvements are significant and have large effect sizes across the 29 datasets.

Index Terms—Cross-project defect prediction, transfer learning, genetic algorithm, ensemble learning

1 Introduction

Software defect prediction can help in allocating test resources by predicting defect-prone classes, files, or modules prior to the testing phase [52]. A number of defect prediction approaches have been proposed which leverage machine learning techniques to build a prediction model from historical data stored in software repositories [10], [20], [25], [34], [35], [57]. These approaches typically use various features, e.g., process metrics, previous-defect metrics, source code metrics, etc., to characterize a class/file/module and employ a classification algorithm to predict if a class/file/module is defective or not. Most defect prediction approaches are trained and applied on classes/files/modules from the same project. These *within-project* defect prediction approaches require sufficient training (historical) data from a project.

- *X. Xia and X. Wang are with the College of Computer Science and Technology, Zhejiang University Hangzhou, Zhejiang 310000, China. E-mail: {xxkidd, wangxinyu}@zju.edu.cn.*
- *D. Lo is with the School of Information Systems, Singapore Management University, Singapore 17890. E-mail: davidlo@smu.edu.sg.*
- *S. Jialin Pan is with the School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: sinnopan@ntu.edu.sg.*
- *N. Nagappan is with the Testing, Verification and Measurement Research, Microsoft Research, Redmond, WA 98052. E-mail: nachin@microsoft.com.*

However, in practice, it is rare that sufficient training data is available for a new project, but there is plenty of data from other projects. For example, the PROMISE repository [33] provides many publicly released defect prediction datasets. Cross-project defect prediction, which uses training data from other projects (aka. *source projects*) to predict defective instances (i.e., classes/files/ modules) in a particular project of interest (aka. *target project*), provides a new perspective to defect prediction [9], [28], [36], [41], [42], [58]. In this paper, we refer to defect prediction approaches that are trained and applied on instances from the same project as *within-project defect prediction approaches*. On the other hand, we refer to approaches that also use training data from other projects as *cross-project defect prediction approaches*.

Cross-project defect prediction is a challenging task since a prediction model that is trained on one or a set of projects might not generalize well to other projects [58]. The challenge is how to create a model to better capture generalizable properties of defective instances that will work for the target project, and *(fully or partly) ignore non-generalizable properties* that do not hold for the target project. In the machine learning literature, to overcome the difference in data distributions between domains, transfer learning [13], [15], [39], [40] which extracts common knowledge from the one domain and transfers it to another domain, has been proposed. Cross-project defect prediction can be viewed as a specific case of transfer learning, which extracts knowledge from a set of source projects and transfers it to a target project.

In this paper, we propose our HYbrid moDel Reconstruction Approach (*HYDRA*) which addresses the above challenge by iteratively learning new classifiers and compositions of classifiers to collectively better capture generalizable properties in every new iteration. Rather than learning only one or a few classifiers, *HYDRA* tunes a two-layer hierarchical composition of a massive number of classifiers. The tuning process is done in many iterations, with the help of Genetic Algorithm (GA) and Ensemble Learning (EL), which gradually *steers* the composite model to better capture generalizable properties; this is done by learning new classifiers and new compositions of classifiers, and by assigning weights to these classifiers, compositions of classifiers, and training instances. Our approach is different from the existing studies on cross-defect prediction which only build one classifier [9], [28], [36], [42] or unify a few classifiers [41].

HYDRA considers the setting where there are numerous labeled data from multiple source projects, however there is only a limited amount of labeled data (e.g., 5 percent of the data are labeled) from a target project. This limited amount of labeled data from a target project is referred to as *training target data*. *HYDRA* includes two phases: genetic algorithm (GA) phase and ensemble learning (EL) phase. In the GA phase, we first build a classifier for each source project data merged with the training target data, and another classifier for the training target data alone. Next, we build a GA classifier by assigning different weights to the multiple classifiers using genetic algorithm. Genetic algorithm will search for the best weights which optimize F1-score [19] on the training target data. The goal is to reduce training error to approximate the generalization error since there are not sufficient instances in training target data to be divided into training and validation sets [46]. In the EL phase, we iterate the GA phase many times. For each iteration, we build a GA classifier, and assign a weight to the GA classifier according to its prediction error rate on the training target data; also, we increase the weights of instances in source projects and the training target data if they are wrongly classified by the GA classifier built in the previous iteration. At the end of the GA and EL phases, we have a massive composition of classifiers and we use it to predict defective instances in the target project.

We evaluate our approach against seven existing approaches [9], [15], [28], [36], [41], [42], [58] using 29 datasets from the PROMISE data repository which contains a total of 11,196 instances. Our results show that *HYDRA* achieves the best performance. *HYDRA* achieves an average F1-scores of 0.544. On average, across the 29 datasets, our approach improves the F1-scores of Zimmermann et al.'s approach [58] by 40.21 percent, of TCA+ [36] by 26.22 percent, of Peters filter [42] by 34.99 percent, of GP [28] by 47.43 percent, of MO [9] by 28.61 percent, and of CODEP [41] by 30.14 percent, respectively. We also compare our approach with *TransferBoost* [15] which is recently proposed in the machine learning literature by Eaton and desJardins, and the results show that our approach improves *TransferBoost* by 39.49 percent. In addition, HYDRA on average can discover 33 percent of all bugs if developers only inspect the top 20 percent lines of code, which improves the best baseline approach (TCA+) by 44.41 percent. We address the following research questions:

RQ1: How effective is HYDRA? How much improvement can it achieve over the baseline approaches?

On average across the 29 projects, the average F1 and PofB20 scores for HYDRA are 0.544 and 33.0 percent, which improves the baseline approaches by a substantial margin.

RQ2: Can HYDRA outperform conventional within-project defect prediction?

On average across the 29 datasets, HYDRA outperforms the within-project defect prediction with 5 percent labeled data in terms of F1-score and PofB20 by 19.46, and 62.40 percent, respectively. Moreover, HYDRA achieves similar results as within-project defect prediction with 90 percent labeled data.

RQ3: Do different percentages of labeled instances from a target project affect the performance of HYDRA?

We notice that for small number of instances, such as 1-3 percent of the total number of instances, the F1-score is low. With more labeled instances from the target projects, the performance is improved. Also the average percentages of bugs detected when inspecting 20% of code is relatively stable, and it varies from 31.5-35.5 percent.

RQ4: How much time does it take for HYDRA to run?

We find that the model building and prediction time for HYDRA are reasonable. On average, HYDRA needs 1.5 minutes to train a model, and 1.7 seconds to predict the labels of instances in the testing set using the model.

The main contributions of this paper are:

1) We propose a novel cross-project defect prediction approach named *HYDRA*, which utilizes the advantages of genetic algorithm (GA) and ensemble learning (EL) to build and iteratively tune a massively compositional model.
2) We evaluate our approach and those proposed by Zimmermann et al., Nam et al., Peters et al., Liu et al., Canfora et al., Panichella et al., and Eaton and desJardins on 29 datasets containing a total of 11,196 instances. The experiment results show that our approach can achieve a substantial improvement over these baseline approaches.

The remainder of the paper is organized as follows. We describe the motivation of building a compositional model and the high-level architecture of *HYDRA* in Section 2. We elaborate *HYDRA* in Section 3. We present our experiments in Section 4. We discuss the other settings of HYDRA, and threats to validity in Section 5. We briefly review related work in Section 6. We conclude this work and point out potential future directions in Section 7.

2 Motivation and Architecture

In this section, we present the motivation of building a compositional model, followed by the architecture of *HYDRA*.

2.1 Why Compositional Model?

If a single model built from one source project, using a state-of-the-art defect prediction approach, can perform very well across a wide-variety of target projects, there is no need for a compositional model. To validate the need for a compositional model, we investigate how models learned from different source projects affect the performance of a state-of-the-art cross-project defect prediction approach.

TABLE 1
Experiment Results Using TCA+

Source⇒Target	F1	Source⇒Target	F1
log4j⇒ant	0.300	ant⇒lucene	0.536
lucene⇒ant	**0.359**	log4j⇒lucene	0.503
poi⇒ant	0.275	poi⇒lucene	0.547
synapse⇒ant	0.256	synapse⇒lucene	**0.556**
ant⇒log4j	0.372	ant⇒poi	0.563
lucene⇒log4j	**0.378**	log4j⇒poi	0.578
poi⇒log4j	0.339	lucene⇒poi	0.541
synapse⇒log4j	0.343	synapse⇒poi	**0.594**
ant⇒synapse	0.208	lucene⇒synapse	0.202
log4j⇒synapse	**0.301**	poi⇒synapse	0.215

The best F1-scores for each target project are in bold.

To do this, we evaluate the performance of TCA+ [36] on five projects from the PROMISE repository: ant-1.4 (ant), log4j-1.0 (log4j), lucene-2.0 (lucene), poi-1.5 (poi), and synapse-1.0 (synapse). The details of the five projects can be found in Table 3.[1] We first identify all combinations of source-target project pairs. For example, if we choose the target project as ant, we select the remaining 4 projects as the source projects, i.e., log4j⇒ant, lucene⇒ant, poi⇒ant, and synapse⇒ant. We choose logistic regression [7] implemented in WEKA [18] as the underlying machine learning classifier, and we measure the performance of TCA+ using F1-score.[2] F1-score is a harmonic mean of precision and recall. Table 1 presents the F1-scores for the five datasets by using TCA+ with logistic regression as the underlying machine learning classifier.

We notice that for a specific target project, if we choose different source projects to perform transfer learning (using TCA+), the performance would be different. This phenomenon is referred to as source component shift in the literature [51]. For example, for the target project synapse, if we choose log4j as the source project, the F1-score is 0.301 using TCA+ with logistic regression. However, if we choose lucene as the source project, the F1-score is 0.202.

Due to the phenomenon of source component shift, if we poorly choose a source project, then this source project may inhibit learning (aka. *negative transfer* [15], [40]) resulting in poor prediction performance. Fortunately, for cross-project defect prediction, we have many source projects which are well labeled. Thus, it would be interesting to investigate a technique that can use all source projects to do cross-project defect prediction, and reduce the effect of *source component shift*. To achieve this goal, we build a massively compositional model using our proposed approach *HYDRA*.

2.2 Overall Architecture

Fig. 1 presents the overall architecture of HYDRA. HYDRA contains two steps: model building step and prediction step. In the model building step, our goal is to build a cross-project prediction model learned from instances in multiple source projects and the training target data (i.e., 5 percent instances from the target project that are labeled as defective or clean). In the prediction step, we apply this model to predict if a new class/file/module in the target project has defects or not.

1. The details of Table 3 is on Page 8 to just before Section 3.3.
2. For the details of F1-score, please see Section 4.2.1.

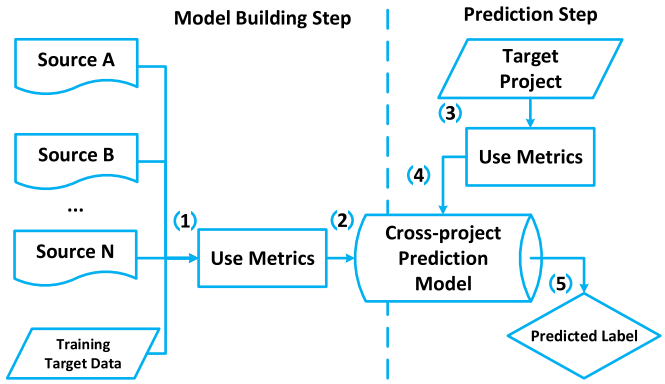

Fig. 1. Overall architecture of HYDRA.

Our framework takes as inputs instances from various source projects with known labels (i.e., *defective* or *clean*), and a small number of labeled instances from the target project (i.e., 5 percent of the instances). Next, it uses various metrics from instances in the various source projects and the training target data (Step 1). Various types of metrics can be used, e.g., process metrics, previous-defect metrics, source code metrics, and entropy-of-change metrics [14], [26], [38], [43]. Table 2 shows the metrics that were used by Jureczko and Madeyski in their defect prediction work [24] and are also used in this work. Notice that we use the same metrics from the source projects and the target project. Next, our framework builds a cross-project prediction model based on the metrics from the various source projects and the training target data (Step 2). The model is a machine learning classifier which assigns labels (in our case: *defective* or *clean*) to an instance (in our case: a class/file/module) based on its metrics.

Fig. 2 illustrates the model built in the model building step of *HYDRA*, which contains two phases: genetic algorithm (GA) phase and ensemble learning (EL) phase. In the GA phase, for each source project S_i and training target data T_t, we build a classifier M_i, and in total we build $(N+1)$ classifiers. Next, *HYDRA* uses genetic algorithm (GA) to search for the best composition of these classifiers; we refer to the composite classifier as a GA classifier. In the EL phase, we build multiple GA classifiers, by running the GA phase multiple times, and compose these GA classifiers according to their training error rate.

After the model is constructed, in the prediction step, it is then used to predict whether an unlabeled instance in the target project would have defects or not. For each of such instances, we first extract the same metrics as those extracted in the model building step (Step 3). We then input the values of these metrics into the model (Step 4). The model will output the prediction result which is one of the following labels: *defective* or *clean* (Step 5).

3 Proposed Approach

We have N source projects $\{S_1, S_2, \ldots, S_N\}$ and a target project T. Each source project contains many instances, and an instance corresponds to a class/file/module (depending on the granularity considered). Each instance has two parts: a set of metrics x and a label y which corresponds to the defect information ($y = 1$ represents *defective*, $y = 0$

TABLE 2
Metrics for Defect Prediction Used by Jureczko and Madeyski [24]

Metrics	Description
wmc	the number of methods used in a given class [11]
dit	the maximum distance from a given class to the root of an inheritance tree [11]
noc	the number of children of a given class in an inheritance tree [11]
cbo	the number of classes that are coupled to a given class [11]
rfc	the number of distinct methods invoked by code in a given class [11]
lcom	the number of method pairs in a class that do not share access to any class attributes [11]
lcom3	another type of lcom metric proposed by Henderson-Sellers [21]
npm	the number of public methods in a given class [5]
loc	the number of lines of code in a given class [5]
dam	the ratio of the number of private/protected attributes to the total number of attributes in a given class [5]
moa	the number of attributes in a given class which are of user-defined types [5]
mfa	the number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class [5]
cam	the ratio of the sum of the number of different parameter types of every method in a given class to the product of the number of methods in the given class and the number of different method parameter types in the whole class [5]
ic	the number of parent classes that a given class is coupled to [49]
cbm	the total number of new or overwritten methods that all inherited methods in a given class are coupled to [49]
amc	the average size of methods in a given class [49]
ca	afferent coupling, which measures the number of classes that depends upon a given class [30]
ce	efferent coupling, which measures the number of classes that a given class depends upon [30]
max_cc	the maximum McCabe's cyclomatic complexity (CC) score [31] of methods in a given class
avg_cc	the arithmetic mean of the McCabe's cyclomatic complexity (CC) scores [31] of methods in a given class

represents *clean*)[3]. An instance is defective if it has one or more bugs. For unlabeled instances in the target project T, the goal of *HYDRA* is to predict their defect information by using the model trained using instances in the source projects $\{S_1, S_2, \ldots, S_N\}$ and a small number of instances from the target project whose labels are known (aka. training target data) T_t. In the following paragraphs, we present the GA phase and EL phase, respectively.

3. Note the datasets provided by Jureczko and Madeyski [24] contain the bug count information, in this paper, we remove the bug count information. For each instance, if its bug count is more than 1, we set the label of the instances as y = 1 (defective); else we set y = 0 (clean). In this paper, we use the defective/clean labels instead of absolute bug counts since all of the papers that present the baseline approaches (i.e., [9], [15], [28], [36], [41], [42], [58]) use the same setting, and we follow them.

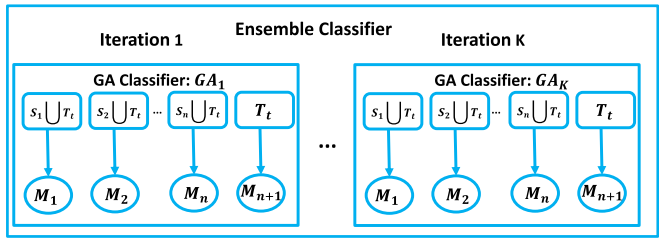

Fig. 2. Model built using HYDRA.

3.1 The GA Phase

In the GA phase, we build a total of $(N+1)$ classifiers: for a source project $S_i, \{1 \leq i \leq N\}$, we merge it with T_t (following the approach in [15]), i.e., $S_i \bigcup T_t$, and build a classifier M_i on the merged data; for the training target data T_t, we build the $(N+1)$th classifier M_{N+1} by training on T_t. We measure the performance of M_i by computing its F1-score on the training target data T_t. By default, we use logistic regression as the underlying classifier to build the $(N+1)$ classifiers. In the defect prediction literature, F1-score is one of the most important metrics which measures how good a defect prediction approach is [29], [36], [42], [58]. Due to the phenomenon of *source component shift*, some classifiers have better performance (i.e., F1-score) than the others. Intuitively, those classifiers who obtain better performance are supposed to be associated with a higher weight.

The output of the GA phase is a heuristically near optimal composite model (i.e., a GA classifier) which assigns different weights to the $(N+1)$ classifiers. In the following paragraphs, we first define the GA classifier and the search space of the potential composition of the $(N+1)$ classifiers: $M_1, M_2, \ldots, M_{N+1}$. Next, we present a detailed procedure to learn the GA classifier.

3.1.1 GA Classifier

A GA classifier is a weighted composition of $(N+1)$ classifiers. Given an instance j, a classifier M_i will output the likelihood of the instance j to be defective, denoted as $Score_i(j)$, whose value ranges from 0 to 1.A GA classifier will predict whether the instance j is defective or not by comparing the weighted sum of the $(N+1)$ classifiers with their likelihood scores on the instance j as weights against a user-predefined threshold score. Definition 1 provides a mathematical definition of the GA classifier.

Definition 1. (GA Classifier) *Consider N source projects $\{S_1, S_2, \ldots, S_N\}$, and a training target data T_t. We build $(N+1)$ classifiers from the source projects and the training target data. A GA classifier composes these $(N+1)$ classifiers and assigns a label to an instance j as follows:*

$$Label(j) = \begin{cases} 1 \text{ (i.e., defective)}, & if\ Comp(j) \geq threshold \\ 0 \text{ (i.e., clean)}, & Otherwise, \end{cases}$$

where,

$$Comp(j) = \frac{\sum_{i=1}^{N+1} \alpha_i \times Score_i(j)}{LOC(j)}. \quad (1)$$

In the above equation, $Score_i(j)$ is the likelihood score outputted by the ith classifier for instance j, α_1 to α_{N+1} are the weights of the $(N+1)$ classifiers, threshold is the boundary

used to decide whether an instance is defective or not, and $LOC(j)$ is the number of lines of codes for instance j. Instance j would be classified as defective (i.e., $y = 1$) if its composite score $Comp(j)$ is larger than or equal to $threshold$; otherwise it is classified as clean. Note that α_1 to α_{N+1} and $threshold$ are the parameters of a GA classifier. Thus, We denote a GA classifier as ($\sum_{i=1}^{N+1} \alpha_i M_i$, $threshold$) where each M_i is a classifier, α_i is the weight of M_i, and $threshold$ is the defect boundary.

The search space of all possible compositions corresponds to the various assignments of values to the weights $\{\alpha_1, \alpha_2, \ldots, \alpha_{N+1}\}$, and the defect boundary $threshold$. Each weight is a real number from zero to one and $threshold$ is a real number from zero to $N + 1$.

We include LOC in Equation (1) to maximize the number of buggy instances found given a budget (e.g., inspecting only 20 percent of the number of LOC). If two instances have equal likelihood to be buggy and one of them has a higher LOC, to find as many bugs as possible within the budget, we need to pick the instance with the lower LOC.

Notice in the GA phase, we use training target data T_t to build a classifier M_{N+1}. Since the number of instances in T_t is small, and we do not have a separate validation set, we evaluate the error rate of M_{N+1} by using the same set T_t. We find that M_{N+1} does not yield the best error rate on T_t, since the number of instances in T_t is small and M_{N+1} does not get enough training. Thus, during the GA phase, the weights of the other classifiers $\{\alpha_1, \alpha_2, \ldots, \alpha_N\}$ are not zeroes.

3.1.2 Detailed Procedure

To learn the weights and the threshold, we employ genetic algorithm. Genetic algorithm is a well-known search algorithm which models solutions in a search space as *chromosomes*. In our setting, a solution is a set of values for the weights and the threshold of a GA classifier. A chromosome contains a set of *genes* where a gene corresponds to a part of a solution (e.g., a value of a weight, in our setting). Genetic algorithm starts with a random selection of chromosomes, referred to as the initial *population*. It then evolves the population by generating subsequent *generations*, where each generation is a population of chromosomes. GA evolves the population by three operations: (1) selection operator, which selects *parent* chromosomes according to their fitness scores; (2) crossover operator, where the selected parents exchange their genes with a given probability; (3) mutation operator, where the genes of new chromosomes would be modified according to a given probability. More details about GA can be found in [17], [48].

We use a simple GA [17], [48] implemented in jgap [32] in this paper. Chromosomes are represented as an array of $(N + 2)$ doubles whose values—the first $(N + 1)$ doubles represent the weights $\{\alpha_1, \alpha_2, \ldots, \alpha_{N+1}\}$, and the last double represents the threshold whose value ranges from zero to $N + 1$. We use the Roulette wheel selection procedure [17], [48] as the selection operator. It assigns a higher probability to a chromosome with a higher fitness score to be selected. Fitness score measures the quality of a solution in a search space. We set the fitness score as the F1-score of the GA classifier on the training target data T_t, i.e., after we choose the weights and the threshold, we use the composite model (i.e., GA classifier) to predict the label of instances in T_t and compute the resulting F1-score. For the crossover operator, we use the single point crossover operator. It processes pairs of chromosomes and for each pair, with a certain probability, it randomly picks a gene (i.e., a double value) from a parent chromosome and swaps that gene and the subsequent ones with corresponding genes from the other parent chromosome. For the mutation operator, we use random mutation. For each gene in the first N genes, with a certain probability, it randomly swaps the gene with another double value in the range of zero to one. And for the $(N + 1)$th gene, with a certain probability, it randomly swaps the gene with another double value in the range of zero to $(N + 1)$.

Algorithm 1. The GA Phase of *HYDRA*

1: **GAPhase(**$\{S_1, S_2, \ldots, S_N\}$**,** T_t**,** $PopSize$**,** $MaxGen$**)**
2: **Input:**
3: $\{S_1, S_2, \ldots, S_N\}$: Source Projects
4: T_t: Training target data
5: $PopSize$: Number of chromosomes in a population. One chromosome is represented by an array of $(N + 2)$ doubles.
6: $MaxGen$: Maximum number of generations
7: **Output:** Composite GA Classifier ($\sum_{i=1}^{N+1} \alpha_i M_i$, $threshold$).
8: **Method:**
9: **for all** $S_i \subseteq \{S_1, S_2, \ldots, S_N\}$ **do**
10: Build a classifier M_i by using $S_i \bigcup T_t$;
11: **end for**
12: Build a classifier M_{N+1} by using T_t;
13: Let P = Initial population with $PopSize$ members;
14: Evaluate P and record the best solution (i.e., the solution with the maximum F1-score on T_t) found so far;
15: Let $curGen$ = 0, and set $P^{'} = P$;
16: **while** $curGen < MaxGen$ **do**
17: Let $P^{'} = select(P^{'})$;
18: $P^{'} = crossover(P^{'})$;
19: $P^{'} = mutation(P^{'})$;
20: Evaluate $P^{'}$ and record the best solution so far;
21: $curGen$ = $curGen$ + 1;
22: **end while**
23: **Output** ($\sum_{i=1}^{N+1} \alpha_i M_i$, $threshold$) which achieves the highest F1-score.

Algorithm 1 presents the detailed steps to train a GA classifier. For each source project S_i, we first build a classifier M_i based on instances in S_i and T_t (Lines 9-11). Similarly, we build a classifier M_{N+1} using the training target data T_t alone (Line 12). Then, we create an initial population (i.e., P) containing $PopSize$ chromosomes (i.e., solutions) that are created in a random manner. That is, for each chromosome, the first $(N + 1)$ doubles (i.e., $\{\alpha_1, \alpha_2, \ldots, \alpha_{N+1}\}$) are initialized by randomly selecting a double from 0 to 1, and the $(N + 2)$th double (i.e., the $threshold$) is initialized by randomly selecting a double from 0 to $N + 1$ (Line 13). And we record the best solution (i.e., the solution with the maximum F1-score on T_t) among the solutions in P (Line 14). Remember that each solution in P is a set of weights $\{\alpha_1, \alpha_2, \ldots, \alpha_{N+1}\}$ and a threshold. Next, we evolve the population in $MaxGen$ iterations; for each iteration, we perform the selection, crossover, and mutation operations on the current population, and record the best solution found so far (Lines 16 to 22). The algorithm returns the $\alpha_1, \alpha_2, \ldots, \alpha_N$ and

threshold values which maximize the F1-score on T_t (i.e., the best solution among solutions in the initial population and the populations generated in the $MaxGen$ generations).

3.2 The EL Phase

In the EL phase, we iterate the GA phase a number of times to learn a composition of GA classifiers. To do this, we adapt AdaBoost [16], which is one of the most famous and widely used ensemble learning algorithms. AdaBoost proceeds in a number of iterations and generates one classifier in each iteration. In each iteration, the classifier built is tweaked such that instances that get misclassified by previous classifiers get a higher weight and thus are deemed to be more important to be classified correctly. AdaBoost can be used with any underlying/base classification algorithms. In the EL phase, we follow the principle of AdaBoost to generate multiple GA classifiers. However, there are several differences between our EL phase and AdaBoost: (1) AdaBoost is not for transfer learning—it is designed for traditional supervised learning, while our approach is for transfer learning. (2) To adapt AdaBoost for transfer learning, we modify the way in which Adaboost [16] assigns weights to instances and evaluates the effectiveness of a classifier. Different from AdaBoost, where instances comes from one domain, for our setting, we have instances from source projects and those from training target data. Our EL phase adjusts the weights of instances from source projects differently from those from training target data. During the iterations, the focus of our EL phase is to minimize errors on the prediction of instances in the training target data, while AdaBoost tries to minimize prediction errors of all training instances.

The details of the EL phase is as follows. For each iteration k, we build a composite GA classifier GA_k using instances in $\{S_1, S_2, \ldots, S_N\}$ and T_t. Next, we assign different weights to the data instances in $\{S_1, S_2, \ldots, S_N\}$ and T_t. For data instances that GA_k predicts correctly, we assign lower weights to them, and for data instances which GA_k predicts wrongly, we assign higher weights to them. Also, we assign weights to instances in training target data differently from those in source projects since our goal is to minimize errors on instances in the training target data.[4] In the next iteration $k+1$, since different data instances have different weights, GA_{k+1} will prioritize data instances with higher weights. The underlying classifiers (i.e., logistic regression), which are parts of the GA classifier, are able to process weighted instances in the training data and will prioritize those with higher weights.

Notice that in the EL phase, we create an ensemble of multiple GA classifiers. We choose this design rather than using only the best performing GA classifier to prevent overfitting [19], i.e., the model that fits best on the training data may not show good performance when it is applied to the testing data.

Fig. 3 presents an example of the EL phase of *HYDRA*. We have instances from two source projects (circles) and the training target data (squares). The size of the circles and squares represents their weights. We show 2 iterations of the EL phase in the figure. For each iteration, we train a classifier (the solid line) according to the instances in these projects. In iteration 1,

4. More details on this are presented later in Algorithm 2.

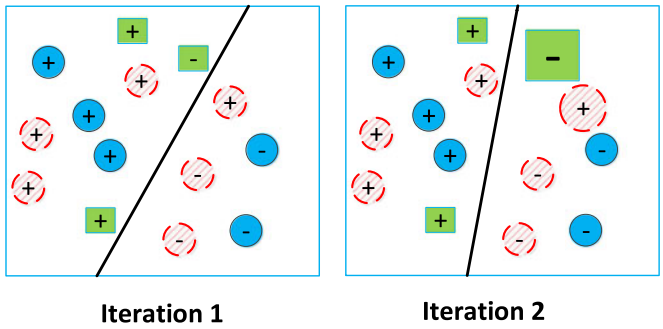

Fig. 3. An example of the EL phase of HYDRA. We have instances from two source projects (the blue and red circles), and instances from a training target data (the squares). "+" and "-" represent clean and defective labels respectively. The solid lines in the figures represent how the classifiers predict clean and defective instances.

the classifier wrongly predicts the "-" instance in the training target data, and wrongly predicts one of the "+" instances in one of the source projects. Thus, higher weights are assigned to these two wrongly predicted instances in the next iteration. In iteration 2, since the weights for the "-" instance in the training target data and "+" instance in the source project are increased, the classifier is biased to predicting the right labels of these two instances. However, the classifier in iteration 2 still predicts the wrong label for the "+" instance. Thus in the next iterations, the EL phase will further increase the weight of the "+" instance whose label is wrongly predicted.

After we reassign weights to the data instances, we also assign a weight to GA_k according to its prediction results (i.e., error rate ϵ_k) on instances in the training target data T_t. The error rate of GA_k, i.e., ϵ_k, is computed based on instances in T_t which are wrongly labeled by GA_k. Considering that each instance in T_t has a weight $w^i_{T_t}$, the cost of misclassification on different instances are different. We thus compute the error rate as follows:

$$\epsilon_k = \frac{\sum_{i=1}^{|T_t|} w^i_{T_t} |GA_k(x^i_{T_t}) - y^i_{T_t}|}{\sum_{j=1}^{|T_t|} w^j_{T_t}}. \quad (2)$$

In the above equation, $\{x^i_{T_t}, y^i_{T_t}\}$ denotes the ith instance in the training target data T_t. Recall that, an instance consists of a set of metrics (e.g., $x^i_{T_t}$) and a defect information label (e.g., $y^i_{T_t}$). $GA_k(x)$ denotes the predicted label for an unlabeled instance, with a set of metrics x using the classifier GA_k. For example, consider three instances with weights 0.4, 0.5, and 0.6, and labels 1, 1, and 0. After we run the GA_k classifier, the predicted labels are 1, 0, and 1. Then, the error rate for GA_k would be:

$$\epsilon(k) = \frac{0.4 * |1-1| + 0.5 * |0-1| + 0.6 * |1-0|}{0.4 + 0.5 + 0.6} = 0.73.$$

Notice that we use a different optimization criteria in the EL phase (i.e., error rate) and GA phase (F1-score). In the EL phase, we follow the principle of AdaBoost [16], and AdaBoost also uses error rate as the optimization criteria. In the GA phase, since our GA classifier combines a number of classifiers, if we set the fitness function as the error rate (i.e., minimize the error rate), due to the imbalance distribution of defective and clean instances in the source projects, the

GA classifier is likely to be prone to predict all of the instances to be of the majority class.

At the end of the K iterations, we have a total of K GA classifiers, and each GA classifier has a weight. We refer to the combination of the K classifiers as an *ensemble classifier*. For a new instance in the target project, we input it into the ensemble classifier, and the ensemble classifier will output a predicted label.

Algorithm 2 presents the detailed steps of the EL phase of *HYDRA*. In the algorithm, the ith source project is denoted as S_i and the jth instance of S_i is denoted as $\{x^j_{S_i}, y^j_{S_i}\}$ where $x^j_{S_i}$ is the set of metrics of the jth instance and $y^j_{S_i}$ is its defect information (i.e., defective or clean). Moreover, we denote the weight of the jth instance in the ith source project as $w^j_{S_i}$, and the weight of the jth instance in the training target data T_t as $w^j_{T_t}$. We use the instances in the source projects and training target data as proxies to the unlabeled data in the target project. The EL phase builds multiple models that can predict the labels of these proxies in varying amount of accuracies, and then ensemble these strong and weak classifiers together.

Algorithm 2. The EL Phase of *HYDRA*

1: **ELPhase(**$\{S_1, S_2, \ldots, S_N\}$**,** T_t**,** K**)**
2: **Input:**
3: $\{S_1, S_2, \ldots, S_N\}$: Source projects
4: T_t: Training target data
5: K: Maximum number of iterations
6: **Output:** Ensemble Classifier $\sum_{k=1}^{K} \beta_k \cdot GA_k$.
7: **Method:**
8: Compute the number of instances: $n_s = \sum_{i=1}^{N} |S_i|$, and $n = n_s + |T_t|$;
9: Set $\beta_s = \frac{1}{2}\ln(1 + \sqrt{2\ln\frac{n_s}{K}})$;
10: Initialize the weights of instances in $\{S_1, S_2, \ldots, S_N\}$, and T_t. We set the weights equally, i.e., $w^j_{S_i} = \frac{1}{n}$, and $w^j_{T_t} = \frac{1}{n}$;
11: **for all** iteration k from 1 to K **do**
12: Normalize the weights in $\{S_1, S_2, \ldots, S_N\}$, and T_t such that the summation of all the weight equals to 1;
13: Input $\{S_1, S_2, \ldots, S_N\}$, and T_t into the GA phase (i.e., Algorithm 1) to get a GA classifier GA_k;
14: Let ϵ_k denote the error rate of GA_k on T_t according to Equation (2):
15: If $\epsilon_k > \frac{1}{2}$, **Break**;
16: Set $\beta_k = \frac{\epsilon_k}{1-\epsilon_k}$, with $\epsilon_k \leq \frac{1}{2}$;
17: Reassign the weights in $\{S_1, S_2, \ldots, S_N\}$, and T_t:

$$w^j_{S_i} = w^j_{S_i} \exp^{-\beta_s |GA_k(x^j_{S_i}) - y^j_{S_i}|},\ 1 \leq i \leq N,\ 1 \leq j \leq |S_i|$$

$$w^j_{T_t} = w^j_{T_t} \exp^{-\beta_k |GA_k(x^j_{T_t}) - y^j_{T_t}|},\ 1 \leq j \leq |T_t|$$

18: **end for**
19: **Output** Ensemble Classifier $\sum_{k=1}^{K} \beta_k GA_k$.

The approach first computes the number of instances in source projects (n_s) and the total number of labeled instances (n) (Line 8). Then, it initializes the β_s which will be used to reassign weights of instances in source projects (Line 9). Our approach initializes the β_s following the approach in [13]. β_s (often referred to as the learning rate in the literature [13]) is set to be inversely proportional to K and proportional to n_s. It is set inversely proportional to K such that the values of the weights are adjusted less abruptly if more iterations are available. With more iterations, we can learn to optimize the weights at a slower pace and this may increase accuracy. β_s is proportional to n_s since the lower is the number of instances in the source projects (n_s), the less able is the algorithm in learning a good model, and thus the learning should be set at a slower pace.

Next, it initializes the weights of the instances in $\{S_1, S_2, \ldots, S_N\}$, and T_t (Line 10). After these initializations, we iterate the GA phase up to K times to get the ensemble classifier. For each iteration k, we first normalize the weights of all instances following AdaBoost [16] (Line 12), and then input the instances in source projects and training target data into the GA phase presented in Algorithm 1 to get the GA classifier GA_k (Line 13). For iteration k, we compute the error rate by running GA_k on instances in T_t (Line 14). The value of the error rate is from 0 to 1, where 0 means that all the instances are correctly classified, and 1 means that all the instances are wrongly classified. If the error rate is more than 0.5, it means that the performance of GA_k is even lower than random guess; when this happen, we terminate the EL phase, discard classifier GA_k, and use all of the previous GA classifiers (i.e., all GA_i, where $i < k$) to form the ensemble classifier (Line 15). If the error rate is less than or equal to 0.5, our approach calculates weight β_k for GA_k and it also reassigns the weights of instances in the source projects and training target data, respectively (Lines 16 and 17). Notice that the reassignments of weights of instances in the source projects and training target data are done differently.

Note that the formula in Line 17 follows the weight reassignment strategy of AdaBoost [16] and Dai et al.'s work [13]. In Line 17, the value in the "$||$" denotes the difference in the predicted and actual value. The larger the difference is, the more the weight should be adjusted. Different from prior approaches, we use a different learning rate for instances in source projects and instances in training target data. Thus, the weight of instances in the source projects are changed at a different rate than instances in the target project. We do this to put more importance to instances in the training target data than those in the source projects. At the end of the EL phase, we get the final ensemble classifier $\sum_{k=1}^{K} \beta_k GA_k$. To help make the error rate ϵ_k closer to the minimum error rate, we set a large value to the maximum number of iterations K. In this paper, by default, we set K as 100.

Notice that in the EL phase, if the error rate ϵ_k is larger than or equal to 1/2 in the first iteration, our HYDRA effectively only runs the GA phase, and returns one GA classifier. Notice that it does not necessarily mean our HYDRA fails to perform well in this case, and there are various reasons that ϵ_k is larger than or equal to 1/2 in the first iteration. For example, the class distributions on the training target data could be severely imbalance, or the number of instances in the training target data could be too small. Moreover from our empirical evaluation, we find even if we only use one GA classifier, the performance of HYDRA is much better than the baseline approaches.[5]

5. For more details, please refer to Section 5.1.

TABLE 3
Statistics of the Datasets

Dataset	LOC	#I/#D/%D	Dataset	LOC	#I/#D/%D
ant-1.3	37,699	125/20/16%	redaktor	59,280	176/27/15.3%
ant-1.4	54,195	178/40/22.5%	synapse-1.0	28,806	157/16/10.2%
ant-1.5	87,047	293/32/10.9%	synapse-1.1	42,302	222/60/27.0%
ant-1.6	113,246	351/92/26.2%	synapse-1.2	53,500	256/86/33.6%
ant-1.7	208,653	745/166/22.3%	tomcat	300,674	858/77/9.0%
log4j-1.0	21,549	135/34/25.2%	velocity-1.4	51,713	196/147/75.0%
log4j-1.1	19,938	109/37/33.9%	velocity-1.6	57,012	229/78/34.1%
log4j-1.2	38,191	205/189/92.2%	xalan-2.4	225,088	723/110/15.2%
lucene-2.0	50,596	195/91/46.7%	xalan-2.5	304,860	803/387/48.2%
lucene-2.2	63,571	247/144/58.3%	xalan-2.6	411,737	885/411/46.4%
lucene-2.4	102,859	340/203/59.7%	xalan-2.7	428,555	909/898/98.8%
poi-1.5	55,428	237/141/59.5%	xerces-1.2	159,254	440/71/16.1%
poi-2.0	93,171	314/37/11.8%	xerces-1.3	167,095	453/69/15.2%
poi-2.5	119,731	385/248/64.4%	xerces-1.4	141,180	588/437/74.3%
poi-3.0	129,327	442/281/63.6%	**Total**	3,626,257	11,196/4,629/41.3%

LOC denotes the total number of LOC. #I denotes the number of instances. #D denotes the number of defective instances. %D denotes the proportion of defective instances.

3.3 Complexity Analysis

Notice our HYDRA can employ different underlying classifiers, and we denote the time complexity for the underlying classifier as U. In the GA phase, we denote the population size as P, number of generations as G, the number of classifiers as $(N+1)$ (N refers to the number of source projects), and the length of the chromosomes is $(N+2)$. Then the time complexity for the GA phase is $\mathcal{O}(GA) = \mathcal{O}(N \times U + P \times G \times N)$. In the EL phase, if we denote the number of iteration as T, then the time complexity for the EL phase is $\mathcal{O}(T \times GA)$. Thus, the time complexity for HYDRA is $\mathcal{O}(T \times (N \times U + P \times G \times N))$.

4 Experiments

In this section, we evaluate the performance of *HYDRA*. The experimental environment is a Windows 7, 64-bit, Intel Xeon 2.53 GHz server with 24 GB RAM.

4.1 Experiment Setup

We evaluate HYDRA using defect datasets originally collected by Jureczko and Madeyski [24] from the PROMISE data repository [8] which consists of 29 releases from 10 different open-source projects. Each instance in the 29 datasets corresponds to a Java class and consists of two parts: 20 static code metrics and a label (*defective* or *clean*). Table 3 presents the statistics of the 29 datasets.

By default, we randomly select 5 percent of the instances in a target project to construct a training target data (i.e., T_t). For the EL phase, we iterate 100 times (i.e., $K = 100$) to reduce overfitting [16], [56]. Since our approach involves a degree of randomness (i.e., we randomly select 5% of the target instances), following past studies, e.g., [3], we run *HYDRA* multiple times (i.e., 50 times) and record the average F1-score across the multiple runs.

We compare *HYDRA* with prior cross-project defect prediction approaches including: BASIC [58][6], TCA+ [36], Peters filter [42], GP [28], MO [9], and CODEP [41]. For TCA+ and TransferBoost, we use the source code provided by the authors. We re-implement GP on top of Leyan,[7] which is a java implementation of genetic programming algorithm. We re-implement MO on top of MOEA framework,[8] which is an open source Java library for multi-objective evolutionary algorithms. Notice in multi-objective learning, there would be a set of solutions which satisfy Pareto optimal [9], we evaluate each of the solution on the testing set, and record the best F1-score and cost effectiveness (PofB20) scores. For BASIC, TCA+, Peters filter, and CODEP, these approaches do not involve any randomness, i.e., the results would be the same no matter how many times they are run. For GP [28] and MO [9], since they use evolutionary algorithms, we run the algorithms 10 times. Aside from the above approaches, we also compare our approach with a state-of-the-art transfer learning approach named *TransferBoost* [15]. Since *TransferBoost* also involves randomization (i.e., it needs some labeled instances from the target project), we also run it 50 times, and compute its average performance. The above approaches use an underlying standard classifier. In this paper, we choose logistic regression as this underlying classifier. We use the same logistic regression implementation (i.e., LIBLINEAR) and parameters settings as those used by Nam et al. [36],—i.e., we use the options "-S 0 (use logistic regression) and "-B -1 (use no bias term) of LIBLINEAR.

The parameters of the genetic algorithm used by HYDRA and MO are as follows:

- *Population size:* we set a moderate population size with $PopSize = 500$.
- *Number of generations:* we set the maximum number of generations $MaxGen = 200$.
- *Crossover Operator:* we use a single point crossover operator with probability $p_c = 0.35$.
- *Mutation Operator:* we use a random mutation operator with probability $p_m = 0.08$.

To simulate the practical usage of our approach and follow the setting used in previous studies [36], [42], [43], [45],

6. We refer to Zimmermann et al.'s approach as BASIC in this section

7. http://www.leyan.org/Genetic+Programming
8. http://www.moeaframework.org/

[52], when we consider a release of a project as a target project, we choose releases of other projects as the source projects. For example, if we choose ant-1.5 as the target project, we use all releases of other projects (i.e., log4j, lucene, poi, redaktor, synapse, tomcat, velocity, xalan, and xerces) as the source projects, and exclude other releases from the same project (i.e., ant). For *HYDRA* and *TransferBoost*, we take all instances from the source projects and 5 percent of the instances in the target project (with their labels), to predict the labels of the remaining 95 percent of the instances in the target project. For the other approaches, to ensure that we use the same test set to evaluate all approaches for a fair comparison, we remove the same 5 percent of the instances in the target project, and predict the labels of the same remaining 95 percent of the instances in the target project. Also, for some baseline approaches, such as BASIC and TCA+, we adapt them so that they can benefit from all datasets (rather than only one dataset, which is the setting used in the original paper) so that the setting is similar to that of our approach and other baselines.

4.2 Evaluation Metrics

We use two evaluation metrics: F1-score and cost effectiveness. F1-score is useful when there are sufficient resources to inspect all of the predicted buggy changes. Cost effectiveness is useful when there are limited resources to inspect a limited amount of code due to a hectic schedule of development.

4.2.1 F1-Score

There are four possible outcomes for an instance in a target project: An instance can be classified as defective when it is truly defective (true positive, TP); it can be classified as defective when it is actually clean (false positive, FP); it can be classified as clean when it is actually defective (false negative, FN); or it can be classified as clean and it is truly clean (true negative, TN). Based on these possible outcomes, precision, recall and F1-score are defined as:

Precision: the proportion of instances that are correctly labeled as defective among those labeled as defective,

$$P = TP/(TP + FP). \tag{3}$$

Recall: the proportion of defective instances that are correctly labeled,

$$R = TP/(TP + FN). \tag{4}$$

F1-Score: a summary measure that combines both precision and recall—it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision),

$$F = (2 \times P \times R)/(P + R). \tag{5}$$

There is a trade-off between precision and recall. The trade-off causes difficulties to compare the performance of several prediction models by using only precision or recall [19]. For this reason, we compare the prediction results using F1-score, which is a harmonic mean of precision and recall. This follows the setting used in many defect prediction studies [25], [36] and other software analytics studies [37], [50], [55].

4.2.2 Cost Effectiveness

Cost effectiveness is a widely used evaluation metric for defect prediction [4], [23], [43], [44], [45], which evaluates prediction performance given a cost limit. In our setting, the cost is the lines of code to inspect, and the benefit is the number of bugs detected. We use the same cost effectiveness setup as the one used by Jiang et al. [23]. They measure the percentage of bugs that a developer can identify by inspecting the top 20 percent lines of code. They refer to this number as *PofB20*.

To compute PofB20 we sort instances in the test data based on the confidence levels that a defect prediction technique outputs for each of them. An instance with a higher confidence level is deemed to be more likely to be buggy by the defect prediction technique. We then simulate a developer that inspects these potentially buggy instances one at a time. As the instances are inspected one at a time, we accumulate the number of lines of code that are inspected and the number of bugs identified. We stop the process when 20 percent of the lines of code have been inspected and output the percentage of bugs that are identified. This number is the PofB20 score. A higher cost effectiveness score represents that a developer can detect more bugs when inspecting a limited number of LOC.

In our HYDRA, suppose we have n GA classifier. For a new instance new, each GA classifier GA_k will compute a composite score that indicates the likelihood that new is buggy, i.e., $Comp_k(new)$. Then, the final confidence score that HYDRA outputs for new can be computed as $\sum_{k=1}^{n} \beta_k \times Comp_k(new)$. In this paper, for each instance in the test set, we get its confidence score. Next, we rank the instances based on their confidence scores to compute the PofB20 score.

4.3 Research Questions

RQ1 How effective is HYDRA? How much improvement can it achieve over the baseline approaches?

In this RQ, we investigate the extent *HYDRA* advances the state-of-the-art approaches. To answer this research question, we compare *HYDRA* with BASIC, TCA+, Peters filter, GP, MO, CODEP, and *TransferBoost*. We compute F1-scores and cost effectiveness (NofB20) to evaluate the performance of these five approaches on the 29 datasets from the PROMISE repository. For each dataset, by default, we run HYDRA and the baseline approaches 50 times. To check if the differences in the performance of HYDRA and the baseline approaches are statistically significant, for the each dataset, we apply the Wilcoxon signed-rank test [54] at 95 percent significance level on two 50 paired data which corresponds to the F1-scores and PofB20 scores of two competing approaches respectively. Since we run the test many times (twice for each dataset), we also use Bonferroni correction [1] to counteract the results of multiple comparisons.

We also use Cliff's delta (δ) [12], which is a non-parametric effect size measure that quantifies the amount of difference between two approaches. In our context, we use Cliff's delta to compare HYDRA with the baseline approaches. The delta values range from -1 to 1, where $\delta = -1 \; or \; 1$ indicates the absence of overlap between two approaches (i.e., all values of one group are higher than the values of the

TABLE 4
Cliff's Delta and the Effectiveness Level [12]

Cliff's Delta ($\|\delta\|$)	Effectiveness Level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$\|\delta\| \geq 0.474$	Large

other group, and vice versa), while $\delta = 0$ indicates the two approaches are completely overlapping. Table 4 describes the meaning of different Cliff's delta values and their corresponding interpretation [12].

RQ2 *Can HYDRA outperform conventional within-project defect prediction?*

As we use some labeled training data from a target project (i.e., training target data), we also investigate whether *HYDRA* could achieve better performance than conventional within-project prediction using some data from the target project. In within-project prediction, some labeled training data from a target project are input to a base classifier and the resultant model is used to label the other data from the target project. Moreover, previous studies show that the performance of these *within-project* defect prediction approaches would be improved if there are sufficient training data from a project [58]. Thus, we are also interested in whether our approach, which leverages defect data from other projects, could achieve similar result as within-project prediction when a sufficient number of within-project training data is available.

Considering the above goals, we investigate two settings. First, since by default HYDRA requires 5 percent of the instances from the target project to be labeled, we investigate the performance of conventional within-project prediction using the same 5 percent data. In this setting, we use the same test set as the one we use to evaluate HYDRA. Second, we randomly select 90 percent of the instances from the target project, and build a classifier to predict the label of the remaining 10 percent of the instances. With 90 percent of the instances labeled, it is likely that conventional within-project prediction can learn a good model to predict the remaining 10 percent of the instances. Notice that for within-project setting, the class distributions in the training set and test set are the same as the class distribution in the original dataset, i.e., we keep the ratio of defective and clean instances in the training set and test set the same as the original dataset.

RQ3 Do different percentages of labeled instances from a target project affect the performance of HYDRA?

HYDRA requires a small number of labeled data from the target project (i.e., training target data). We investigate whether different numbers of instances in the training target data affect the performance of *HYDRA*. By default, the number of instances in the training target data is set to be 5 percent of the total number of instances in the target project. To answer this question, we vary the number of instances from 1-15 percent of the total number of instances in the target project. Notice our HYDRA cannot work if we do not include any data from the target project, since HYDRA adjusts its parameters according to the prediction results from the small number of instances in the target project. Additionally, we also investigate the effectiveness of HYDRA when a fixed budget is specified, i.e., an absolute number of instances are selected from a target project.

RQ4 *How much time does it take for HYDRA to run?*

HYDRA builds a GA classifier by composing many off-the-shelf classifiers. In the EL phase, multiple GA classifiers are combined. Building these many classifiers requires substantial computational time. Thus, in this research question, we investigate the time efficiency of *HYDRA*. We run *HYDRA* 10 times and report the average model training and application time. Model training time refers to the time to convert a training data into *HYDRA* ensemble learner. Model application time refers to the time for *HYDRA* ensemble learner to predict the label of an instance. We compare the model training and application time of *HYDRA* with those of other approaches.[9]

4.4 RQ1: HYDRA versus Other Algorithms

Tables 5 and 6 presents the F1-scores and cost effectiveness (PofB20) of *HYDRA* compared with those of BASIC, TCA+, Peters filter, GP, MO, CODEP and *TransferBoost*. The F1-scores of *HYDRA* vary from 0.190−0.991. Across the 29 datasets, the average F1-scores of *HYDRA* is 0.544. From Table 5, the improvements of our approach over the baselines are substantial. On average across the 29 datasets, *HYDRA* outperforms BASIC, TCA+, Peters filter, GP, MO, CODEP, and TransferBoost by 40.21, 26.22, 34.99, 47,43, 28.61, 30.14, and 39.49 percent, respectively.

The PofB20 scores of *HYDRA* vary from 12.9-67.5 percent. Across the 29 datasets, the average PofB20 score of *HYDRA* is 33.0 percent. From Tables 6, the improvements of our approach over the baselines are substantial. On average across the 29 datasets, *HYDRA* outperforms BASIC, TCA+, Peters filter, GP, MO, CODEP, and TransferBoost by 54.75, 44.41, 49.40, 71.25, 72.98, 77.80, and 62.34 percent, respectively.

Among the seven baseline approaches, TCA+ achieves the best performance; here, we compare HYDRA with TCA+ with different percentages of LOC that are inspected. We record the average cost effectiveness scores across the 29 datasets. Fig. 4 presents the cost effectiveness graphs for HYDRA compared with TCA+. We notice that HYDRA is better than TCA+ for a wide range of percentages of LOC to inspect.

Tables 7 and 8 present the *p*-values and Cliff's delta when we compare HYDRA with the baseline approaches in terms of F1 and PofB20 scores. Notice in our study, we use Bonferroni correction to counteract the results of multiple comparisons, thus the p-values are adjusted. And we consider that HYDRA statistically significantly improves a baseline approach at the confidence level of 95 percent if the adjusted *p*-value is less than 0.05. We notice in most of the cases, HYDRA shows significant improvement over the baseline approaches with large effect size.

Tables 9 and 10 present the number of datasets where HYDRA performs statistically significantly better than the baseline approaches (+), performs more or less equally well with the best performing baseline approaches (=), and

9. Notice for TCA+ and Peters filter, there would be a preprocessing of the data; we record the training time as the sum of the preprocessing time and the underlying machine learning classifier training time. For the approaches that involve randomization, we run the approaches 10 times.

TABLE 5
F1-Scores Our Approach HYDRA Compared with BASIC, TCA+, Peters Filter, GP, MO, CODEP, and TransferBoost, Respectively

Datasets	HYDRA	Basic	TCA+	Peters Filter	GP	MO	CODEP	TransferBoost.
ant-1.3	0.396 ± 0.010	0.507 ± 0.021	0.276 ± 0.022	0.365 ± 0.030	0.438 ± 0.011	0.255 ± 0.014	0.458 ± 0.009	**0.522** ± 0.018
ant-1.4	0.329 ± 0.012	0.318 ± 0.016	0.345 ± 0.023	0.379 ± 0.016	0.349 ± 0.013	**0.380** ± 0.015	0.372 ± 0.010	0.324 ± 0.012
ant-1.5	0.347 ± 0.014	**0.435** ± 0.016	0.213 ± 0.012	0.296 ± 0.011	0.380 ± 0.021	0.202 ± 0.013	0.345 ± 0.012	0.416 ± 0.013
ant-1.6	0.602 ± 0.015	0.612 ± 0.017	0.389 ± 0.011	0.526 ± 0.010	0.609 ± 0.015	0.374 ± 0.013	0.614 ± 0.009	**0.618** ± 0.016
ant-1.7	0.468 ± 0.010	**0.565** ± 0.016	0.325 ± 0.011	0.486 ± 0.012	0.557 ± 0.015	0.333 ± 0.012	0.516 ± 0.010	0.563 ± 0.021
log4j-1.0	0.413 ± 0.002	0.526 ± 0.015	0.342 ± 0.010	0.442 ± 0.016	0.444 ± 0.016	0.353 ± 0.011	**0.536** ± 0.011	0.507 ± 0.026
log4j-1.1	0.538 ± 0.008	0.551 ± 0.009	0.493 ± 0.014	0.604 ± 0.015	0.449 ± 0.010	0.413 ± 0.010	**0.625** ± 0.011	0.547 ± 0.021
log4j-1.2	**0.914** ± 0.014	0.319 ± 0.011	0.695 ± 0.010	0.412 ± 0.016	0.183 ± 0.012	0.757 ± 0.009	0.286 ± 0.011	0.333 ± 0.020
lucene-2.0	**0.648** ± 0.010	0.366 ± 0.012	0.535 ± 0.016	0.446 ± 0.016	0.365 ± 0.014	0.574 ± 0.012	0.441 ± 0.010	0.377 ± 0.012
lucene-2.2	**0.657** ± 0.010	0.299 ± 0.013	0.555 ± 0.017	0.288 ± 0.011	0.319 ± 0.016	0.526 ± 0.015	0.436 ± 0.015	0.283 ± 0.015
lucene-2.4	**0.691** ± 0.012	0.366 ± 0.015	0.582 ± 0.019	0.327 ± 0.011	0.410 ± 0.010	0.602 ± 0.023	0.428 ± 0.018	0.358 ± 0.016
poi-1.5	**0.742** ± 0.003	0.318 ± 0.015	0.550 ± 0.011	0.518 ± 0.017	0.279 ± 0.012	0.606 ± 0.031	0.572 ± 0.011	0.322 ± 0.012
poi-2.0	**0.283** ± 0.003	0.265 ± 0.016	0.224 ± 0.012	0.162 ± 0.015	0.282 ± 0.013	0.196 ± 0.010	0.213 ± 0.018	0.262 ± 0.014
poi-2.5	**0.780** ± 0.002	0.326 ± 0.011	0.601 ± 0.014	0.720 ± 0.008	0.311 ± 0.010	0.333 ± 0.019	0.586 ± 0.023	0.332 ± 0.011
poi-3.0	**0.807** ± 0.005	0.314 ± 0.016	0.608 ± 0.009	0.684 ± 0.006	0.393 ± 0.014	0.495 ± 0.010	0.475 ± 0.031	0.315 ± 0.010
redaktor	0.295 ± 0.006	0.490 ± 0.011	0.287 ± 0.004	0.300 ± 0.002	0.256 ± 0.019	0.296 ± 0.010	0.336 ± 0.024	**0.510** ± 0.010
synapse-1.0	0.252 ± 0.012	**0.438** ± 0.012	0.212 ± 0.014	0.275 ± 0.003	0.413 ± 0.023	0.190 ± 0.011	0.255 ± 0.026	0.426 ± 0.009
synapse-1.1	0.494 ± 0.011	0.341 ± 0.014	0.416 ± 0.011	**0.539** ± 0.005	0.444 ± 0.025	0.400 ± 0.012	0.393 ± 0.020	0.347 ± 0.008
synapse-1.2	**0.529** ± 0.011	0.448 ± 0.013	0.435 ± 0.009	0.495 ± 0.012	0.507 ± 0.010	0.399 ± 0.011	0.500 ± 0.018	0.445 ± 0.006
tomcat	0.190 ± 0.012	0.372 ± 0.014	0.162 ± 0.006	0.263 ± 0.015	**0.406** ± 0.019	0.000 ± 0.015	0.368 ± 0.012	0.387 ± 0.004
velocity-1.4	0.793 ± 0.014	0.133 ± 0.015	0.655 ± 0.015	0.314 ± 0.016	0.198 ± 0.008	**0.807** ± 0.016	0.185 ± 0.011	0.149 ± 0.008
velocity-1.6	**0.503** ± 0.011	0.303 ± 0.016	0.444 ± 0.011	0.297 ± 0.017	0.257 ± 0.007	0.357 ± 0.011	0.354 ± 0.011	0.321 ± 0.012
xalan-2.4	0.315 ± 0.010	0.353 ± 0.023	0.245 ± 0.018	0.356 ± 0.011	0.371 ± 0.014	0.242 ± 0.017	**0.399** ± 0.010	0.354 ± 0.011
xalan-2.5	**0.593** ± 0.012	0.433 ± 0.015	0.546 ± 0.010	0.394 ± 0.011	0.400 ± 0.016	0.518 ± 0.011	0.455 ± 0.010	0.434 ± 0.011
xalan-2.6	**0.656** ± 0.010	0.458 ± 0.016	0.514 ± 0.010	0.446 ± 0.010	0.377 ± 0.016	0.388 ± 0.011	0.482 ± 0.009	0.457 ± 0.014
xalan-2.7	**0.991** ± 0.007	0.458 ± 0.011	0.693 ± 0.023	0.445 ± 0.012	0.360 ± 0.014	0.980 ± 0.017	0.484 ± 0.008	0.467 ± 0.015
xerces-1.2	0.240 ± 0.012	0.248 ± 0.016	0.232 ± 0.011	0.186 ± 0.013	0.254 ± 0.010	**0.263** ± 0.008	0.241 ± 0.010	0.237 ± 0.011
xerces-1.3	0.417 ± 0.014	0.421 ± 0.010	0.255 ± 0.010	**0.487** ± 0.016	0.400 ± 0.011	0.254 ± 0.019	0.342 ± 0.011	0.424 ± 0.013
xerces-1.4	**0.903** ± 0.012	0.267 ± 0.014	0.666 ± 0.012	0.269 ± 0.005	0.294 ± 0.012	0.765 ± 0.011	0.402 ± 0.012	0.267 ± 0.011
Average.	**0.544 ± 0.223**	**0.388 ± 0.110**	**0.431 ± 0.166**	**0.403 ± 0.136**	**0.369 ± 0.099**	**0.423 ± 0.215**	**0.417 ± 0.116**	**0.390 ± 0.108**

The results are in the form of mean ± standard deviation. The last column show the average F1-scores. The best F1-scores are in bold.

performs statistically significantly worse than the baseline approaches (-) in terms of F1 and PofB20 scores. We use Bonferroni correction to adjust the multiple p-values computed by Wilcoxon signed-rank test. All the significance is at the confidence level of 95 percent. From these tables, our HYDRA improves the baselines statistically significantly in most of the time. For example, when compared HYDRA with TCA+, we notice HYDRA statistically significantly improves TCA+ in 27 and 20 datasets in terms of F1 and PofB20 scores, while TCA+ statistically significantly improves HYDRA only on one and seven datasets in terms of F1 and PofB20 scores.

4.5 RQ2: HYDRA versus within-Project

Table 12 presents the F1-scores and cost effectiveness (PofB20) of *HYDRA* compared with those of within-project prediction under two settings. From Table 12, the improvement of our approach over within-project prediction with 5 percent labeled data is substantial. On average across the 29 datasets, HYDRA outperforms the within-project classifier with 5 percent labeled data in terms of F1-socre and PofB20 by 19.46, and 62.40 percent, respectively. Moreover, *HYDRA* achieves similar results as the within-project prediction with 90 percent labeled data. The average F1-scores of within-project prediction with 90 percent data is 0.557; it is 0.544 for *HYDRA*. Note that *HYDRA* only requires 5 percent labeled data from the target project. Also, we notice HYDRA outperforms the PofB20 of within-project prediction with 90 percent labeled data by 25.74 percent.

To investigate whether the improvement of HYDRA over within-project prediction with 5 percent labeled data is significant. For each dataset, we apply the Wilcoxon signed-rank test on the paired data which correspond to the F1-scores and PofB20 to test whether the improvement of HYDRA over the baseline approaches are significant. We also use Bonferroni correction to counteract the results of multiple comparisons. Table 11 presents the number of datasets where HYDRA performs statistically significantly better than within-project prediction with 5 percent labeled data (+), performs more or less equally well with within-project prediction with 5 percent labeled data (=), and performs statistically significantly worse than within-project prediction with 5% labeled data (-) in terms of F1 and PofB20. We notice in most of the cases, HYDRA improves the within-project prediction with 5 percent labeled data statistically significantly.

We also notice that including the instances from the source project is better than a prediction model built on limited data. By analyzing the models constructed in our experiments, we find that typically the best weights for models learned from source project $\{\alpha_1, \ldots, \alpha_N\}$ are not all zeroes.

4.6 RQ3: Effect of Varying the Number of Instances in the Training Target Project

Fig. 5 presents the average F1-scores across the 29 datasets with various number of instances from the target projects. We notice that for small number of instances, such as 1-3 percent of the total number of instances, the F1-score is

TABLE 6
Cost Effectiveness (PofB20) of Our Approach HYDRA Compared with BASIC, TCA+, Peters Filter, GP, MO, CODEP, and TransferBoost, Respectively

Datasets	HYDRA	Basic	TCA+	Peters Filter	GP	MO	CODEP	TransferBoost.
ant-1.3	20.0% ± 1.0%	**35.1%** ± 1.1%	20.1% ± 2.2%	25.2% ± 1.3%	20.0% ± 1.0%	35.0% ± 2.2%	25.5% ± 1.6%	31.6% ± 2.5%
ant-1.4	**46.8%** ± 1.2%	17.2% ± 1.5%	32.2% ± 1.5%	21.5% ± 1.4%	14.9% ± 1.6%	19.1% ± 1.2%	6.5% ± 1.2%	25.5% ± 1.0%
ant-1.5	**28.6%** ± 1.0%	14.5% ± 2.2%	17.6% ± 1.6%	20.1% ± 1.6%	22.9% ± 1.6%	14.3% ± 1.4%	22.6% ± 1.0%	14.3% ± 1.0%
ant-1.6	14.4% ± 1.0%	23.6% ± 1.5%	20.5% ± 1.0%	25.4% ± 2.2%	25.5% ± 1.8%	9.2% ± 1.5%	**30.4%** ± 1.0%	22.5% ± 1.2%
ant-1.7	24.8% ± 0.8%	29.1% ± 1.6%	24.3% ± 1.0%	30.0% ± 1.6%	27.8% ± 1.9%	**32.8%** ± 2.4%	27.6% ± 1.2%	27.3% ± 1.0%
log4j-1.0	19.7% ± 1.1%	36.0% ± 2.3%	37.7% ± 1.2%	**42.4%** ± 1.8%	27.9% ± 1.2%	29.5% ± 0.8%	31.2% ± 1.1%	28.3% ± 1.3%
log4j-1.1	16.5% ± 1.2%	**34.5%** ± 1.2%	31.4% ± 1.0%	30.4% ± 1.8%	24.4% ± 1.1%	33.7% ± 1.6%	33.6% ± 1.1%	16.7% ± 1.6%
log4j-1.2	**54.8%** ± 1.4%	11.6% ± 1.4%	17.1% ± 1.2%	15.8% ± 1.5%	6.4% ± 1.0%	19.7% ± 1.1%	6.7% ± 1.7%	14.2% ± 0.6%
lucene-2.0	**35.4%** ± 1.0%	29.9% ± 2.2%	19.4% ± 1.8%	34.3% ± 1.3%	34.0% ± 1.6%	3.4% ± 1.1%	20.8% ± 2.5%	25.7% ± 1.2%
lucene-2.2	**38.5%** ± 1.0%	26.6% ± 2.6%	16.0% ± 1.5%	19.9% ± 1.6%	25.2% ± 2.3%	33.7% ± 1.0%	24.5% ± 1.4%	27.8% ± 1.6%
lucene-2.4	**39.5%** ± 1.0%	17.6% ± 1.0%	17.5% ± 2.3%	15.8% ± 1.1%	20.6% ± 1.2%	13.8% ± 1.0%	12.5% ± 1.6%	13.7% ± 1.6%
poi-1.5	**35.6%** ± 1.0%	20.4% ± 1.2%	23.2% ± 1.6%	22.3% ± 1.1%	20.2% ± 1.1%	15.2% ± 1.0%	14.9% ± 1.6%	13.1% ± 1.7%
poi-2.0	15.8% ± 1.2%	20.6% ± 1.4%	30.9% ± 1.4%	23.3% ± 1.6%	17.9% ± 1.8%	**38.5%** ± 1.2%	15.4% ± 1.3%	11.1% ± 1.2%
poi-2.5	**49.6%** ± 0.4%	11.5% ± 1.6%	12.9% ± 1.3%	22.5% ± 1.0%	7.4% ± 1.5%	9.9% ± 1.4%	6.2% ± 1.2%	10.7% ± 1.4%
poi-3.0	**42.4%** ± 0.3%	10.6% ± 1.9%	24.5% ± 1.0%	16.6% ± 1.0%	13.8% ± 1.2%	12.4% ± 2.4%	12.8% ± 2.1%	17.7% ± 1.6%
redaktor	**50.0%** ± 0.2%	17.9% ± 0.3%	17.8% ± 1.2%	17.9% ± 1.3%	17.9% ± 1.4%	14.3% ± 1.5%	17.6% ± 1.2%	48.1% ± 1.7%
synapse-1.0	23.8% ± 1.1%	28.8% ± 1.3%	0.2% ± 1.4%	9.6% ± 1.5%	33.3% ± 1.5%	4.8% ± 1.6%	**38.2%** ± 1.3%	35.0% ± 0.4%
synapse-1.1	**33.0%** ± 0.8%	27.3% ± 1.3%	25.5% ± 2.1%	24.2% ± 2.1%	27.3% ± 1.6%	18.2% ± 2.5%	26.1% ± 1.3%	24.7% ± 0.4%
synapse-1.2	24.8% ± 1.3%	19.5% ± 1.4%	18.6% ± 1.0%	19.3% ± 3.2%	14.5% ± 1.6%	13.8% ± 1.0%	**24.9%** ± 1.0%	21.6% ± 0.2%
tomcat	21.9% ± 1.5%	26.3% ± 1.0%	22.8% ± 0.8%	21.1% ± 1.2%	21.9% ± 1.4%	15.8% ± 0.6%	**26.4%** ± 0.5%	25.7% ± 0.7%
velocity-1.4	**67.5%** ± 0.5%	5.4% ± 1.4%	36.7% ± 0.6%	21.9% ± 1.0%	3.3% ± 1.5%	29.5% ± 0.4%	0.2% ± 1.4%	8.0% ± 1.5%
velocity-1.6	**46.3%** ± 0.3%	17.4% ± 2.1%	35.8% ± 0.5%	23.6% ± 0.5%	17.4% ± 2.6%	7.9% ± 0.5%	14.7% ± 0.2%	13.0% ± 2.6%
xalan-2.4	12.9% ± 0.2%	28.2% ± 1.5%	21.8% ± 1.3%	26.9% ± 1.2%	25.6% ± 1.1%	**36.5%** ± 1.4%	17.9% ± 0.6%	17.3% ± 2.5%
xalan-2.5	**38.5%** ± 0.4%	20.3% ± 1.6%	26.4% ± 1.4%	16.6% ± 0.5%	14.3% ± 1.1%	27.9% ± 1.5%	16.5% ± 2.4%	16.0% ± 1.0%
xalan-2.6	**31.5%** ± 1.2%	23.7% ± 1.5%	18.6% ± 1.1%	20.3% ± 1.5%	20.6% ± 1.0%	25.1% ± 1.1%	20.6% ± 1.5%	28.1% ± 1.2%
xalan-2.7	**51.4%** ± 1.0%	18.3% ± 1.0%	15.4% ± 1.0%	15.4% ± 1.6%	12.4% ± 1.0%	18.2% ± 1.5%	13.8% ± 1.6%	19.9% ± 1.1%
xerces-1.2	15.8% ± 1.0%	12.5% ± 0.5%	**18.3%** ± 1.0%	10.1% ± 1.4%	10.3% ± 1.0%	5.2% ± 1.6%	6.0% ± 1.3%	7.5% ± 1.1%
xerces-1.3	13.0% ± 1.4%	22.8% ± 1.0%	**34.2%** ± 1.5%	32.1% ± 1.0%	14.0% ± 1.6%	2.1% ± 1.2%	12.4% ± 2.4%	10.2% ± 3.0%
xerces-1.4	**44.7%** ± 1.3%	13.4% ± 1.1%	26.5% ± 1.0%	17.5% ± 1.1%	17.1% ± 1.4%	14.1% ± 1.1%	12.6% ± 1.5%	14.2% ± 1.1%
Average.	**33.0% ± 14.4%**	**21.4% ± 7.9%**	**22.9% ± 8.3%**	**22.1% ± 7.1%**	**19.3% ± 7.6%**	**19.1% ± 10.9%**	**18.6% ± 9.3%**	**20.3% ± 9.2%**

The results are in the form of mean ± standard deviation. The last column show the average PofB20 scores. The best PofB20 scores are in bold.

low. The average F1-scores vary from 0.484-0.558. With more instances selected from the target projects, the performance is improved. For example, when we choose 15 percent of the total number of instances, the average F1-score is 0.557. Moreover, we notice that the F1-scores increase, when the size of the training target data increases from 1 to 5 percent, is substantially larger than the F1-scores increase, when the size of the training target data increases from 5 to 15 percent. We also notice that the F1-scores of HYDRA are stable when the size of the training target data increases from 10 to 15 percent. This indicates that at 10 percent, HYDRA already has sufficient training data from the target project. Adding more training data from the target project has little impact on the performance of HYDRA.

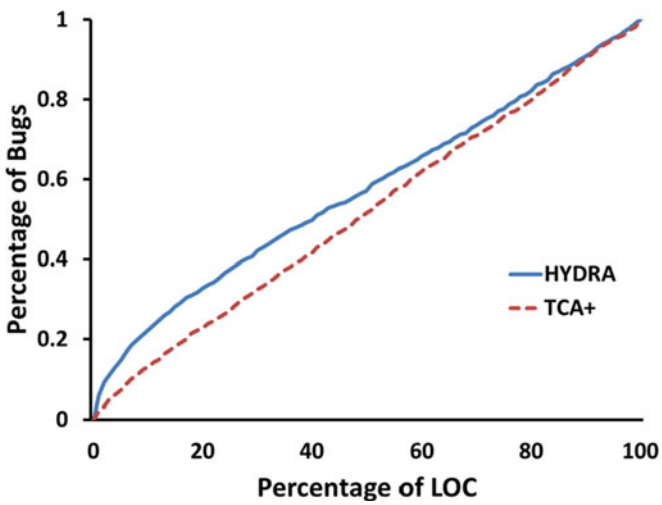

Fig. 4. Cost effectiveness graph for HYDRA and TCA+.

Fig. 6 presents the average percentages of bugs detected when inspecting 20 percent of the lines of code across 29 datasets with various number of instances from the target projects. We notice that the average percentages of bugs detected is relatively stable, and it varies from 31.5-35.5 percent.

Also, the average percentages of bugs detected do not increase with the size of the training target data increases. For example, the average percentages of bugs detected is 32.7 percent when 5 percent data are selected from the target project, while the number is 31.8 percent when 6 percent data are from the target project.

When considering both Figs. 5 and 6, we find that the F1-scores will increase when the size of the training target data is increased from 1 to 15 percent, while the PofB20 scores are relatively stable. The results seem to indicate that much training data is needed for precise classification of instances as buggy or not. The more training data is available, the more precise is the classification. On the other hand, the results seem to indicate that even a small amount of training data suffices to rank instances such that many of the buggy ones are listed in the first 20 percent of the code. Additional training data does not improve this ranking further when the top 20 percent of the code is inspected.

TABLE 7
P-Value and Cliff's Delta (δ) for HYDRA Compared with the Baseline Approaches in Terms of F1-Score

Datasets	H. vs. Basic		H. vs. TCA+		H. vs. Peters Filter		H. vs. GP		H. vs. MO		H. vs. CODEP		H. vs. Tr.Bo.	
	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ
ant-1.3	$1.2e^{-9}$	-0.65	$6.2e^{-12}$	0.52	0.0014	0.43	0.0003	-0.31	$5.3e^{-15}$	0.87	$1.1e^{-9}$	-0.56	$5.8e^{-13}$	-0.86
ant-1.4	0.02	0.08	0.0004	-0.15	0.0005	-0.46	0.072	-0.11	$1.5e^{-9}$	-0.84	$2.4e^{-7}$	-0.76	0.123	0.04
ant-1.5	$1.9e^{-15}$	0.83	$4.3e^{-11}$	0.66	$1.4e^{-6}$	0.55	0.0001	-0.35	$1.5e^{-6}$	0.86	0.234	0.04	$1.8e^{-15}$	-0.91
ant-1.6	0.014	-0.10	$5.5e^{-15}$	0.94	$1.3e^{-7}$	0.66	0.234	-0.02	$2.2e^{-15}$	0.96	0.224	-0.03	0.0012	-0.12
ant-1.7	$1.4e^{-14}$	-0.62	$4.6e^{-13}$	0.73	0.055	-0.12	$3.6e^{-8}$	-0.66	$1.8e^{-15}$	0.78	$1.3e^{-13}$	-0.78	$2.4e^{-14}$	-0.77
log4j-1.0	$5.8e^{-13}$	-0.69	$3.4e^{-9}$	0.68	$1.2e^{-12}$	-0.44	$2.4e^{-13}$	0.45	$1.8e^{-9}$	0.56	$2.4e^{-14}$	-0.82	$2.3e^{-13}$	-0.71
log4j-1.1	0.05	-0.14	0.0014	0.45	$1.1e^{-13}$	-0.65	$1.2e^{-9}$	0.87	$4.3e^{-11}$	0.68	$5.4e^{-13}$	-0.78	0.128	-0.04
log4j-1.2	$2.2e^{-13}$	1.00	$3.4e^{-15}$	0.96	$2.2e^{-13}$	1.00	$3.5e^{-16}$	1.00	$1.1e^{-13}$	0.91	$2.2e^{-13}$	1.00	$2.8e^{-13}$	1.00
lucene-2.0	$3.3e^{-11}$	1.00	$3.2e^{-13}$	0.77	$2.6e^{-13}$	0.93	$4.5e^{-15}$	1.00	$1.2e^{-6}$	0.81	$2.4e^{-10}$	0.88	$2.1e^{-10}$	0.98
lucene-2.2	$2.2e^{-16}$	1.00	$4.2e^{-13}$	0.86	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$5.4e^{-10}$	0.78	$2.4e^{-11}$	0.89	$2.2e^{-16}$	1.00
lucene-2.4	$2.2e^{-16}$	1.00	$3.2e^{-15}$	0.92	$2.2e^{-16}$	1.00	$5.4e^{-13}$	0.92	$2.3e^{-6}$	0.68	$2.9e^{-13}$	0.92	$2.2e^{-16}$	1.00
poi-1.5	$2.2e^{-16}$	1.00	$5.6e^{-13}$	0.89	$3.4e^{-14}$	0.91	$2.2e^{-16}$	1.00	$3.6e^{-8}$	0.74	$3.4e^{-14}$	0.95	$2.2e^{-16}$	1.00
poi-2.0	$1.4e^{-6}$	0.51	$4.8e^{-13}$	0.61	$2.2e^{-16}$	1.00	0.346	0.00	$5.5e^{-8}$	0.84	$4.5e^{-10}$	0.75	0.045	0.12
poi-2.5	$2.2e^{-16}$	1.00	$5.4e^{-13}$	0.91	$4.3e^{-10}$	0.72	$5.1e^{-13}$	1.00	$3.8e^{-16}$	1.00	$3.4e^{-13}$	0.95	$2.2e^{-16}$	1.00
poi-3.0	$2.2e^{-16}$	1.00	$2.4e^{-15}$	0.95	$1.3e^{-9}$	0.71	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.5e^{-15}$	1.00	$2.2e^{-16}$	1.00
redaktor	$2.2e^{-16}$	-1.00	0.432	0.08	0.312	-0.01	$1.4e^{-10}$	0.65	0.112	0.02	0.0003	-0.55	$2.2e^{-16}$	-1.00
synapse-1.0	$2.5e^{-11}$	-0.98	$1.5e^{-13}$	0.51	0.0003	-0.35	$1.4e^{-13}$	-0.64	$6.6e^{-10}$	0.65	0.129	-0.01	$1.8e^{-13}$	-0.86
synapse-1.1	$2.4e^{-13}$	0.65	$3.2e^{-12}$	0.68	$2.1e^{-7}$	-0.54	$1.8e^{-13}$	0.58	$2.5e^{-15}$	0.72	$3.5e^{-12}$	0.78	$3.2e^{-14}$	0.81
synapse-1.2	$3.9e^{-15}$	0.85	$5.4e^{-15}$	0.82	0.0001	0.55	0.0001	0.52	$3.2e^{-12}$	1.00	0.0004	0.42	$2.4e^{-8}$	0.65
tomcat	$2.2e^{-15}$	-0.97	$1.9e^{-7}$	0.53	$1.8e^{-15}$	-0.76	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	1.00	$1.8e^{-15}$	-1.00	$2.2e^{-16}$	-1.00
velocity-1.4	$2.2e^{-16}$	1.00	$6.9e^{-12}$	0.90	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	0.0001	-0.14	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
velocity-1.6	$2.2e^{-16}$	1.00	$3.5e^{-16}$	0.98	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
xalan-2.4	0.0001	-0.45	$1.8e^{-12}$	0.56	$1.2e^{-5}$	-0.32	$1.7e^{-12}$	-0.54	$1.8e^{-13}$	0.67	$2.4e^{-12}$	-0.72	0.0001	-0.45
xalan-2.5	$2.4e^{-12}$	0.95	$1.8e^{-10}$	0.82	$2.2e^{-15}$	1.00	$2.4e^{-11}$	1.00	$1.9e^{-12}$	0.95	$6.4e^{-12}$	0.91	$2.9e^{-15}$	0.90
xalan-2.6	$2.2e^{-13}$	1.00	$3.5e^{-13}$	0.98	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.8e^{-15}$	1.00	$3.2e^{-15}$	0.96	$2.5e^{-15}$	0.96
xalan-2.7	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
xerces-1.2	0.2490	-0.02	0.123	0.05	$1.8e^{-12}$	0.62	0.112	0.12	0.0001	-0.31	0.4561	0.00	0.3210	-0.01
xerces-1.3	0.1345	-0.03	$9.2e^{-12}$	0.78	$1.8e^{-12}$	-0.82	0.1720	0.13	$3.5e^{-13}$	0.92	$1.2e^{-5}$	0.91	0.1023	-0.04
xerces-1.4	$2.2e^{-16}$	1.00	$5.4e^{-12}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-12}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00

Table 13 presents the F1-scores and cost effectiveness (PofB20) of HYDRA when there are only 10, 20, 30, 40, 50, and 60 instances in the training target project. We randomly select these 10, 20, 30, 40, 50, and 60 instances, and repeat the process 10 times and record the average F1 and cost effectiveness scores. The F1-scores and PofB20 vary from 0.529-0.580, and 30.6-35.1 percent, respectively. We notice the performance of HYDRA increases when the number of instances in the training target project increases from 10 to 50, and decreases when the number of instances from 50 to 60. Even with a smaller number of instances in the training target project, the performance of HYDRA is better than the other baseline approaches. For example, when we label only 20 instances in the target project, HYDRA could achieve F1-score and PofB20 up to 0.559, and 35.1 percent, respectively.

4.7 RQ4: Time Efficiency of HYDRA

Table 14 presents the average model training and application time across the 29 datasets. Due to the space limitation, we do not list the time for each individual datasets. From Table 14, we notice that the model training and application time of *HYDRA* is reasonable, e.g., on average, we need about 1.5 minutes to train a model, and 1.7 seconds to predict the labels of instances in the testing set using the model. Note that the model does not need to be updated all the time and it can be used to label many instances. *HYDRA* model training time is longer than those of BASIC, Peters filter, and *TransferBoost* but shorter than that of TCA+, GP, and CODEP. The training time for TCA+ is long (i.e., nearly 1 hour to train a model) because TCA+ needs to perform many matrix operations. *HYDRA* model application time is longer than those of the other approaches but we believe it is still acceptable (it can label thousands of instances in seconds).

5 Discussions

5.1 Impact of the Number of Iterations in the EL Phase

By default, we set the number of iterations of HYDRA as 100. Here, we also investigate other numbers of iterations, i.e., we increase the number of iterations from 1, 10, 20, 30, ... , 200. Fig. 7 presents the average F1 and PofB20 scores of HYDRA with different number of iterations. We notice that the performance of HYDRA increases when we increase the number of iterations from 1 to 100, and it is stable when we increase the number of iterations from 100 to 200. Thus, in practice, we suggest users to set the number of iteration as 100, since more iterations translates to higher runtime cost. Moreover, when we set the number of iterations as 1, it means we only have the GA phase, and the F1 and PofB20 scores of HYDRA are 0.501 and 0.28, respectively. These scores are much less than those of HYDRA with enough iterations. Thus, combining the GA and EL phases improves the performance of HYDRA.

TABLE 8
P-Value and Cliff's Delta (δ) for HYDRA Compared with the Baseline Approaches in Terms of PofB20

Datasets	H. versus Basic		H. versus TCA+		H. versus Peters Filter		H. versus GP		H. versus MO		H. versus CODEP		H. versus Tr. Bo.	
	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ
ant-1.3	$2.5e^{-12}$	-1.00	0.3122	0.00	$1.8e^{-11}$	-0.61	0.5033	0.00	$5.6e^{-13}$	-1.00	$1.8e^{-11}$	-0.62	$2.2e^{-16}$	-1.00
ant-1.4	$2.8e^{-16}$	1.00	$2.2e^{-16}$	1.00	$3.9e^{-12}$	0.92	$2.2e^{-16}$	1.00	$5.8e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.8e^{-16}$	1.00
ant-1.5	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$3.3e^{-13}$	0.99	$4.2e^{-16}$	0.90	$2.2e^{-16}$	1.00	$2.8e^{-10}$	0.88	$2.2e^{-16}$	1.00
ant-1.6	$8.1e^{-12}$	-0.92	$5.1e^{-12}$	-0.72	$2.3e^{-15}$	-0.99	$2.4e^{-12}$	-1.00	$5.4e^{-11}$	0.82	$2.2e^{-16}$	-1.00	$1.7e^{-12}$	-0.81
ant-1.7	$1.4e^{-14}$	-0.62	$4.6e^{-13}$	0.73	0.055	-0.12	$3.6e^{-8}$	-0.66	$1.8e^{-15}$	0.78	$1.3e^{-13}$	-0.78	$2.4e^{-14}$	-0.77
log4j-1.0	$2.4e^{-14}$	-1.00	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	$3.2e^{-12}$	-0.82	$2.4e^{-15}$	0.84	$1.8e^{-15}$	-0.92	0.0001	-0.65
log4j-1.1	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	$2.9e^{-15}$	-1.00	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	0.114	0.00
log4j-1.2	$2.2e^{-16}$	1.00	$3.8e^{-15}$	1.00	$2.2e^{-16}$	1.00	$5.1e^{-16}$	1.00	$1.9e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.8e^{-16}$	1.00
lucene-2.0	$5.4e^{-15}$	0.72	$2.2e^{-16}$	1.00	0.2104	0.04	0.0089	0.07	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$1.9e^{-12}$	1.00
lucene-2.2	$3.6e^{-13}$	0.87	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.8e^{-12}$	0.91	$1.3e^{-6}$	0.72	$2.9e^{-11}$	0.98	$3.6e^{-12}$	0.92
lucene-2.4	$2.2e^{-16}$	1.00	$2.5e^{-15}$	1.00	$2.2e^{-16}$	1.00	$5.9e^{-16}$	1.00	$1.9e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.8e^{-16}$	1.00
poi-1.5	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
poi-2.0	$2.5e^{-8}$	-0.51	$2.2e^{-16}$	-1.00	$1.8e^{-8}$	-0.82	0.0011	-0.12	$2.2e^{-16}$	-1.00	0.6012	0.00	$1.5e^{-11}$	0.52
poi-2.5	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
poi-3.0	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
redaktor	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
synapse-1.0	$1.4e^{-6}$	-0.62	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$1.2e^{-10}$	-0.73	$6.4e^{-14}$	1.00	$2.2e^{-16}$	-1.00	$2.5e^{-15}$	-0.92
synapse-1.1	$5.6e^{-13}$	0.58	$2.4e^{-12}$	0.68	$2.2e^{-11}$	0.72	$1.4e^{-13}$	0.65	$2.2e^{-15}$	1.00	$3.1e^{-12}$	0.72	$4.4e^{-14}$	0.85
synapse-1.2	$3.1e^{-15}$	0.56	$6.2e^{-14}$	0.58	$4.3e^{-12}$	0.55	$2.4e^{-13}$	1.00	$2.2e^{-16}$	1.00	0.431	-0.01	$2.6e^{-8}$	0.35
tomcat	$2.5e^{-8}$	-0.62	0.031	-0.08	0.2031	0.02	0.6012	0.00	$2.4e^{-12}$	0.92	$1.8e^{-12}$	-0.92	$3.4e^{-16}$	-0.82
velocity-1.4	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
velocity-1.6	$2.2e^{-16}$	1.00	$6.5e^{-14}$	0.99	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
xalan-2.4	$2.2e^{-16}$	-1.00	$1.8e^{-12}$	-0.98	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	$1.6e^{-15}$	-0.82	$2.5e^{-10}$	-0.78
xalan-2.5	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
xalan-2.6	$6.4e^{-13}$	0.82	$2.2e^{-16}$	1.00	$2.6e^{-12}$	0.91	$2.4e^{-15}$	1.00	$8.2e^{-9}$	0.72	$2.2e^{-16}$	1.00	0.0002	0.55
xalan-2.7	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
xerces-1.2	0.0002	-0.21	$1.4e^{-7}$	-0.66	$1.6e^{-12}$	0.65	$8.1e^{-12}$	0.69	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00
xerces-1.3	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	$2.2e^{-16}$	-1.00	0.301	-0.08	$2.2e^{-16}$	1.00	0.112	0.03	0.0001	0.32
xerces-1.4	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00	$2.2e^{-16}$	1.00

5.2 Longitudinal Data Setup

In the default setting, we randomly select 5 percent of the target instances for training (we refer to it as HYDRA with random selection setup). This default setup may use future data to build a model. This setting may overestimate the performance of HYDRA since in practice one will have no access to future data. To assess the severity of this issue, in this section, we evaluate the performance of HYDRA following a longitudinal data setup. In the longitudinal data setup, we sort the instances in the dataset in temporal order, i.e., the instances are sorted according to the time they are added into project with older instances being listed first. We then select the first 5 percent instances to construct the training target data. Since only 15 datasets provide the temporal order information, we only evaluate HYDRA following the longitudinal data setup with these 15 datasets.

Table 15 presents the F1 and PofB20 scores for HYDRA following the longitudinal data setup. On average across the 15 datasets, HYDRA with longitudinal data setup achieves an average F1 and PofB20 score of 0.571 and 31.8 percent, respectively. Notice the average F1 and PofB20 scores of HYDRA with the random selection setup for the 15 datasets are 0.570 and 31.5 percent respectively, and the difference is small. Moreover, for each dataset, we also apply Wilcoxon signed-rank test with Bonferroni correctionto evaluate whether there is a significant difference in the performance of HYDRA

TABLE 9
Number of Datasets Where HYDRA Statistically Significantly Improves over a Baseline Approach (+), Performs More or Less Equally Well with a Baseline Approach (=), and Statistically Significantly Loses with a Baseline Approach (-) in Terms of F1

HYDRA versus Baselines	+	=	-
HYDRA versus Basic	17	4	8
HYDRA versus TCA+	27	1	1
HYDRA versus Peters Filter	19	2	8
HYDRA versus GP	19	4	6
HYDRA versus MO	25	1	3
HYDRA versus CODEP	17	4	8
HYDRA versus TransferBoost	15	5	9

TABLE 10
Number of Datasets Where HYDRA Statistically Significantly Improves over a Baseline Approach (+), Performs More or Less Equally Well with a Baseline Approach (=), and Statistically Significantly Loses with a Baseline Approach (-) in Terms of PofB20

HYDRA versus Baselines	+	=	-
HYDRA versus Basic	18	0	11
HYDRA versus TCA+	20	2	7
HYDRA versus Peters Filter	22	0	7
HYDRA versus GP	18	4	7
HYDRA versus MO	25	0	4
HYDRA versus CODEP	18	3	8
HYDRA versus TransferBoost	21	1	7

TABLE 11
Number of Datasets Where HYDRA Shows Statistical Improvement over Within-Project Defect Prediction with 5 Percent Labeled Data (+), Indifferent with the Within-Project Defect Prediction with 5 Percent Labeled Data (=), and Within-Project Defect Prediction with 5 Percent Labeled Data Shows Statistical Improvement over HYDRA (-) in Terms of F1 and PofB20

Evaluation Metrics	+	=	-
F1	27	2	0
PofB20	22	3	4

following the longitudinal data setup and the random selection setup. We find that for all of the 15 datasets, there is no significant difference at the confidence level of 95 percent. Thus, the threat to validity caused by the random selection setup is not-significant; the performance of HYDRA when it is evaluated following the longitudinal data setup or random selection setup is more or less the same.

5.3 Impact of Different Number of Repeated Runs

Notice that in our experiment setup, to deal with randomness in the approaches, by default, we run all approaches many times, and report the average F1 and PofB20 scores across the multiple runs. Here, we would like to investigate whether the performance of these approaches will be substantially varied if we run HYDRA with different number of repeated runs. We run HYDRA and the baseline approaches 10-100 times, and Figs. 8 and 9 present average F1 and PofB20 scores for HYDRA compared with the baseline approaches with different numbers of repeated runs. We notice that the performance of HYDRA is stable for different numbers of repeated runs, and the F1 and PofB20 scores vary from 0.539–0.549, and 0.328-0.335, respectively. Thus, different numbers of repeated runs have limited impact on the performance HYDRA. Also, compared with the baseline approaches, we notice that HYDRA outperforms them for every number of repeated runs.

5.4 Fixed Inspection Budget

We use PofB20 as the default cost effectiveness evaluation metric following [4], [23], [43], [44], [45]. Unfortunately, for a project with a large number of LOC, inspecting 20 percent of the LOC is impractical. For example, suppose a system has 1M LOC, inspecting 20 percent of the LOC means that developers need to inspect 200K LOC, which will take a lot of time and resources. In this section, rather than using PofB20, we investigate the cost effectiveness of HYDRA and TCA+ given a fixed inspection budget, i.e., an absolute number of LOC to inspect. We experiment with the following fixed budget: 500, 1,000, 2,000, and 5,000 LOC. We compare HYDRA with TCA+ since we find TCA+ achieves the best performance among the baseline approaches.

TABLE 12
F1-Scores and PofB20 of Our Approach (HYDRA) Compared with Those of Within-Project Prediction (5, 90 Percent)

Datasets	F1-Score			PofB20		
	HYDRA	5%	90%	HYDRA	5%	90%
ant-1.3	**0.396 ± 0.010**	0.204 ± 0.013	0.254 ± 0.010	**20.0% ± 1.0%**	10.5% ± 1.1%	0.0% ± 1.4%
ant-1.4	**0.329 ± 0.012**	0.191 ± 0.011	0.181 ± 0.006	**46.8% ± 1.2%**	21.3% ± 1.2%	0.0% ± 1.5%
ant-1.5	**0.347 ± 0.014**	0.208 ± 0.010	0.199 ± 0.017	28.6% ± 1.0%	14.3% ± 1.0%	**99.0% ± 1.0%**
ant-1.6	**0.602 ± 0.015**	0.469 ± 0.014	0.569 ± 0.010	14.4% ± 1.0%	15.2% ± 1.0%	**30.6% ± 1.2%**
ant-1.7	**0.468 ± 0.010**	0.393 ± 0.010	0.441 ± 0.010	24.8% ± 0.8%	24.4% ± 1.0%	**27.3% ± 1.3%**
log4j-1.0	0.413 ± 0.002	0.322 ± 0.014	**0.424 ± 0.003**	19.7% ± 1.1%	10.0% ± 1.0%	**98.0% ± 1.2%**
log4j-1.1	0.538 ± 0.008	0.265 ± 0.021	0.534 ± 0.005	16.5% ± 1.2%	**17.9% ± 0.6%**	3.0% ± 1.0%
log4j-1.2	0.914 ± 0.014	0.901 ± 0.014	**0.944 ± 0.012**	**54.8% ± 1.4%**	20.2% ± 0.7%	16.7% ± 1.0%
lucene-2.0	0.648 ± 0.010	0.434 ± 0.010	**0.661 ± 0.012**	**35.4% ± 1.0%**	13.8% ± 1.2%	27.3% ± 1.1%
lucene-2.2	0.657 ± 0.010	0.540 ± 0.016	**0.678 ± 0.013**	38.5% ± 1.0%	16.1% ± 1.3%	**43.5% ± 1.4%**
lucene-2.4	0.691 ± 0.012	0.663 ± 0.019	**0.759 ± 0.011**	**39.5% ± 1.0%**	23.0% ± 1.1%	3.6% ± 1.4%
poi-1.5	0.742 ± 0.003	0.660 ± 0.012	**0.774 ± 0.016**	35.6% ± 1.0%	29.7% ± 1.7%	**53.6% ± 2.4%**
poi-2.0	**0.283 ± 0.003**	0.163 ± 0.004	0.141 ± 0.017	15.8% ± 1.2%	**27.8% ± 1.4%**	0.0% ± 0.0%
poi-2.5	0.780 ± 0.002	0.736 ± 0.006	**0.847 ± 0.015**	**49.6% ± 0.4%**	22.0% ± 0.2%	37.8% ± 1.2%
poi-3.0	**0.807 ± 0.005**	0.781 ± 0.005	0.805 ± 0.014	**42.4% ± 0.3%**	23.2% ± 1.1%	19.6% ± 1.3%
redaktor	0.295 ± 0.006	0.174 ± 0.012	**0.457 ± 0.011**	50.0% ± 0.2%	**63.0% ± 0.6%**	0.0% ± 0.0%
synapse-1.0	0.252 ± 0.012	0.110 ± 0.011	**0.537 ± 0.013**	23.8% ± 1.1%	25.0% ± 1.1%	**50.0% ± 2.2%**
synapse-1.1	0.494 ± 0.011	0.377 ± 0.011	**0.511 ± 0.011**	33.0% ± 0.8%	19.4% ± 1.0%	**33.3% ± 2.4%**
synapse-1.2	0.529 ± 0.011	0.371 ± 0.012	**0.551 ± 0.014**	**24.8% ± 1.3%**	15.8% ± 1.4%	6.3% ± 2.0%
tomcat	0.190 ± 0.012	0.200 ± 0.015	**0.416 ± 0.015**	21.9% ± 1.5%	9.7% ± 1.2%	18.2% ± 1.5%
velocity-1.4	0.793 ± 0.014	0.822 ± 0.016	**0.897 ± 0.021**	**67.5% ± 0.5%**	40.3% ± 1.2%	29.0% ± 1.5%
velocity-1.6	0.503 ± 0.011	0.387 ± 0.023	**0.576 ± 0.014**	**46.3% ± 0.3%**	12.4% ± 1.1%	26.3% ± 1.2%
xalan-2.4	**0.315 ± 0.010**	0.216 ± 0.013	0.276 ± 0.004	12.9% ± 0.2%	**22.0% ± 0.5%**	10.5% ± 0.4%
xalan-2.5	**0.593 ± 0.012**	0.562 ± 0.022	0.553 ± 0.015	**38.5% ± 0.4%**	18.0% ± 1.0%	24.6% ± 1.1%
xalan-2.6	**0.656 ± 0.010**	0.641 ± 0.015	**0.704 ± 0.015**	**31.5% ± 1.2%**	19.3% ± 1.2%	22.4% ± 1.2%
xalan-2.7	0.991 ± 0.007	0.981 ± 0.011	**0.992 ± 0.013**	**51.4% ± 1.0%**	17.6% ± 1.0%	24.8% ± 1.5%
xerces-1.2	0.240 ± 0.012	0.191 ± 0.012	0.109 ± 0.014	**15.8% ± 1.0%**	14.2% ± 1.3%	0.0% ± 1.5%
xerces-1.3	0.417 ± 0.014	0.369 ± 0.011	**0.456 ± 0.015**	13.0% ± 1.4%	3.2% ± 1.0%	**40.7% ± 1.1%**
xerces-1.4	0.903 ± 0.012	0.883 ± 0.014	**0.911 ± 0.013**	**44.7% ± 1.3%**	20.2% ± 1.5%	15.3% ± 1.3%
Average.	**0.544 ± 0.223**	**0.456 ± 0.258**	**0.557 ± 0.250**	**33.0% ± 14.6%**	**20.3% ± 10.9%**	**26.2% ± 25.7%**

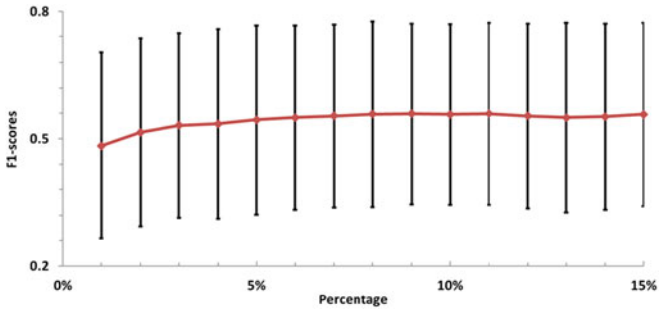

Fig. 5. Avg. F1 for different percentages of training target data instances. The error bars indicate one standard deviation above and below the average. The error bars are wide since we merge results from 29 different datasets.

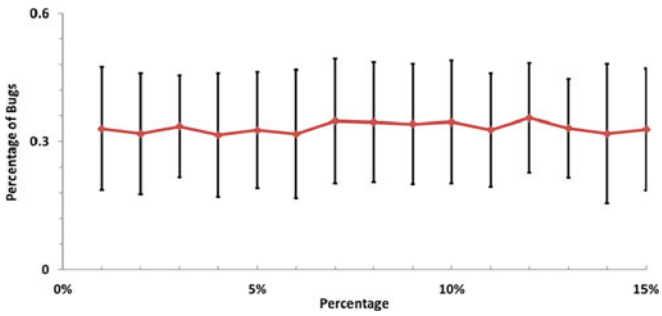

Fig. 6. Avg. PofB20 for different percentages of training target data instances. The error bars indicate one standard deviation above and below the average. The error bars are wide since we merge results from 29 different datasets.

TABLE 13
F1-Score and Cost Effectiveness (PofB20) of HYDRA with Different Number of Instances in the Training Target Project (10 Instances to 60 Instances)

# Instances	F1-score	PofB20
10	0.529 ± 0.221	30.6% ± 12.8%
20	0.559 ± 0.223	35.1% ± 13.1%
30	0.540 ± 0.232	33.5% ± 14.6%
40	0.565 ± 0.231	34.8% ± 14.1%
50	0.580 ± 0.226	34.8% ± 13.5%
60	0.551 ± 0.228	33.2% ± 13.8%

TABLE 14
Avg. Model Training (Train) and Application (Appl.) Time

Algorithms	Train	Appl.
HYDRA	88.599 s	1.748 s
BASIC	0.543 s	0.007 s
TCA+	3882.063 s	0.008 s
Peters	1.230 s	0.010 s
GP	139.697 s	4.203 s
MO	0.614 s	0.001 s
CODEP	850.764 s	0.016 s
TransferBoost	39.144 s	0.134 s

Table 16 presents the cost effectiveness of HYDRA compared with TCA+ with different numbers of LOCs to inspect. From the table, we notice that our HYDRA still improves TCA+ by a substantial margin. On average across 29 datasets, HYDRA can detect 2.0 to 18.6 percent bugs when developers inspect 500 to 5,000 LOC. Recall that the average LOC in our collected 29 datasets is 125,043; thus inspecting 500 and 5,000 LOC means we only investigate 0.4 and 4 percent LOC in a project. In practice, we believe inspecting 4 percent LOC in a project is affordable for a project team, and our approach can detect 18.6 percent of the

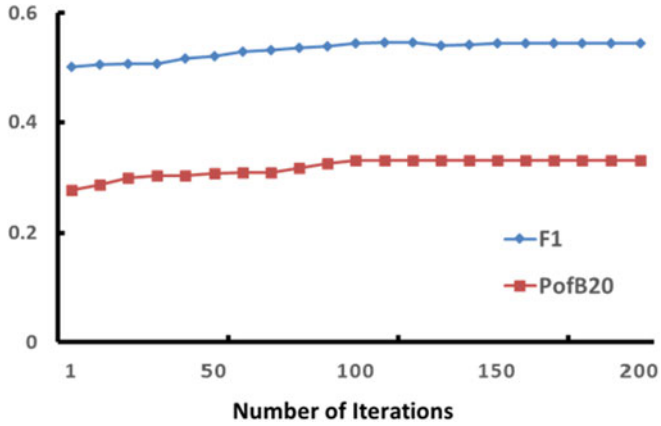

Fig. 7. Average F1 and PofB20 scores for HYDRA with different number of iterations in the EL phase.

TABLE 15
F1 and PofB20 Scores for HYDRA Following the Longitudinal Data Setup

Datasets	F1	PofB20
ant-1.3	0.403 ± 0.012	21.0% ± 1.0%
ant-1.4	0.317 ± 0.008	45.3% ± 1.1%
ant-1.5	0.357 ± 0.010	29.3% ± 1.3%
ant-1.6	0.593 ± 0.011	14.0% ± 1.0%
ant-1.7	0.476 ± 0.015	26.8% ± 1.0%
log4j-1.0	0.406 ± 0.018	20.1% ± 0.6%
log4j-1.1	0.543 ± 0.003	17.1% ± 1.0%
log4j-1.2	0.911 ± 0.004	55.0% ± 1.2%
lucene-2.0	0.650 ± 0.012	35.7% ± 1.0%
lucene-2.2	0.661 ± 0.011	38.7% ± 1.1%
lucene-2.4	0.694 ± 0.013	39.6% ± 1.3%
xalan-2.4	0.320 ± 0.012	13.1% ± 1.0%
xalan-2.5	0.596 ± 0.010	38.7% ± 1.1%
xalan-2.6	0.661 ± 0.010	32.0% ± 1.0%
xalan-2.7	0.984 ± 0.010	50.8% ± 1.5%
Average.	**0.571 ± 0.201**	**31.8% ± 13.1%**

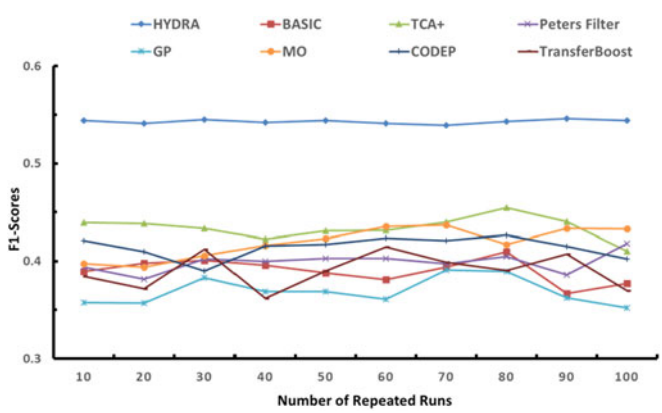

Fig. 8. Average F1-scores for HYDRA compared with the baseline approaches with different number of repeated runs.

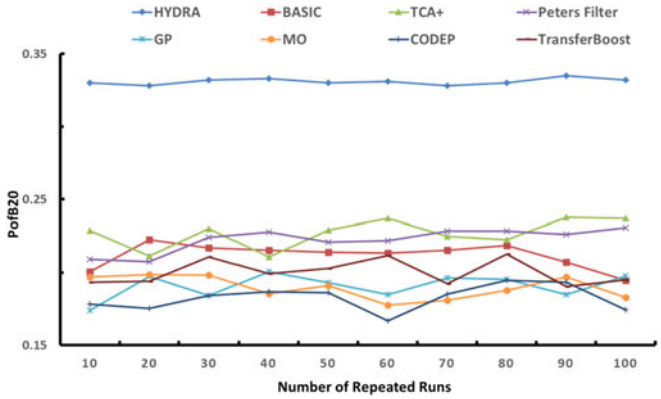

Fig. 9. Average PofB20 scores for HYDRA compared with the baseline approaches with different number of repeated runs.

bugs after such inspection which shows the potential benefit of our proposed approach.

We also compare our HYDRA with a model which simulates developer intuition. We model the developer intuition as follows. For each project, we have multiple versions, e.g., we have five versions of ant, which include ant-1.3, ant-1.4, ant-1.5, ant-1.6, and ant-1.7. We analyze the older versions of a project to find files which are defective in many older versions, and use these to predict files that are most likely to be defective in the latest version of the project in our dataset. To do so, for each file in the latest version, we count the number of older versions in which the file was defective C denoted as $DefectCount$. This model then recommends the top $x\%$ defective files according to their $DefectCount$ scores. In practice, developers may follow this intuition to inspect code and find bugs, i.e., if file A is defective many times in the previous versions, it is likely to be defective in the latest version too. In this paper, we vary $x\%$ from 1–5 percent, and we choose ant-1.7, log4j-1.2, lucene-2.4, poi-3.0, synapse-1.2, velocity-1.6, xalan-2.7, and xerces-1.4 as the latest versions.

Table 17 presents the number of LOC (#LOC) to inspect and percentage of defects (%Defect) detected when the top 1–5 percent of the files recommended by the developer intuition model are inspected. Comparing the results in Tables 16 and 17, we notice our HYDRA could detect more defects than the developer intuition model by inspecting less LOC. For example, for ant-1.7, the developer intuition model can detect 3.61 percent of the defects by inspecting the top 1 percent defective files, but this translates to 13,614 LOC. On the other hand, our Hydra can detect 14.3 percent of the defects by inspecting 5,000 LOC. Our approach can thus detect many more defects by inspecting much less LOC.

TABLE 16
Cost Effectiveness of HYDRA Compared with TCA+ with Different Numbers of LOCs to Inspect (Number of LOC $\in \{500, 1{,}000, 2{,}000, 5{,}000\}$)

LOC	500		1,000		2,000		5,000	
	HYDRA	TCA+	HYDRA	TCA+	HYDRA	TCA+	HYDRA	TCA+
ant-1-3	2.1%	1.3%	3.7%	3.7%	10.6%	9.5%	12.0%	11.4%
ant-1-4	3.5%	1.5%	6.0%	4.1%	17.3%	10.6%	39.7%	23.5%
ant-1-5	1.3%	0.5%	2.3%	1.4%	6.6%	3.5%	15.1%	7.9%
ant-1-6	0.5%	0.5%	0.9%	1.3%	2.5%	3.3%	5.8%	7.3%
ant-1-7	0.5%	0.3%	0.8%	0.8%	2.4%	2.1%	14.3%	13.1%
log4j-1-0	3.7%	4.4%	6.4%	12.2%	18.3%	31.5%	11.1%	25.0%
log4j-1-1	3.3%	3.9%	5.8%	11.0%	16.6%	28.3%	14.4%	15.0%
log4j-1-2	5.7%	1.1%	10.0%	3.1%	28.7%	8.1%	52.1%	11.2%
lucene-2-0	2.8%	1.0%	4.9%	2.7%	14.0%	6.9%	32.2%	15.3%
lucene-2-2	2.4%	0.6%	4.2%	1.8%	12.1%	4.5%	27.9%	10.1%
lucene-2-4	1.5%	0.4%	2.7%	1.2%	7.7%	3.1%	17.7%	6.8%
poi-1-5	2.6%	1.0%	4.5%	2.9%	12.8%	7.5%	29.5%	16.7%
poi-2-0	0.7%	0.8%	1.2%	2.3%	3.4%	6.0%	7.8%	13.2%
poi-2-5	1.7%	0.3%	2.9%	0.8%	8.3%	1.9%	19.1%	4.3%
poi-3-0	1.3%	0.5%	2.3%	1.3%	6.6%	3.4%	15.1%	7.6%
redaktor	3.4%	0.8%	5.9%	2.1%	16.9%	5.4%	38.8%	12.1%
synapse-1-0	3.3%	0.0%	5.8%	0.0%	16.5%	0.0%	15.8%	0.0%
synapse-1-1	3.1%	1.5%	5.5%	4.2%	15.6%	10.8%	18.3%	15.2%
synapse-1-2	1.9%	0.9%	3.2%	2.4%	9.3%	6.3%	21.3%	13.9%
tomcat	0.3%	0.2%	0.5%	0.5%	1.5%	1.4%	13.4%	13.3%
velocity-1-4	5.2%	1.8%	9.1%	5.0%	26.1%	12.8%	20.1%	28.4%
velocity-1-6	3.2%	1.6%	5.7%	4.4%	16.2%	11.3%	37.4%	25.1%
xalan-2-4	0.2%	0.2%	0.4%	0.7%	1.1%	1.7%	12.6%	13.9%
xalan-2-5	0.5%	0.2%	0.9%	0.6%	2.5%	1.6%	15.8%	13.4%
xalan-2-6	0.3%	0.1%	0.5%	0.3%	1.5%	0.8%	3.5%	12.8%
xalan-2-7	0.5%	0.1%	0.8%	0.2%	2.4%	0.6%	5.5%	3.4%
xerces-1-2	0.4%	0.3%	0.7%	0.8%	2.0%	2.0%	4.6%	4.5%
xerces-1-3	0.3%	0.5%	0.5%	1.5%	1.6%	3.7%	3.6%	8.3%
xerces-1-4	1.3%	0.5%	2.2%	1.3%	6.3%	3.4%	14.6%	7.5%
Average.	**2.0%**	**0.9%**	**3.5%**	**2.6%**	**9.9%**	**6.6%**	**18.6%**	**12.4%**

TABLE 17
Number of LOC (#LOC) to Inspect and Percentage of Defects (%Defect) Detected the Top 1-5 Percent of Files Recommended by the Developer Intuition Model Are Inspected

Projects	1%		2%		3%		4%		5%	
	#LOC	%Defect	#LOC	%Defect	#LOC	%Defect	#LOC	%Defect	#LOC	%Defect
ant-1.7	13,614	3.61%	20,676	6.63%	27,622	11.45%	32,193	15.06%	38,970	19.28%
log4j-1.2	1,012	1.06%	1,924	2.12%	2,755	3.17%	4,101	4.23%	4,515	5.29%
lucene-2.4	11,745	1.48%	15,409	2.96%	20,802	4.93%	23,852	5.91%	27,288	7.88%
poi-3.0	10,379	1.42%	13,570	2.85%	16,948	4.63%	21,189	6.05%	24,733	6.76%
synapse-1.2	1,464	2.33%	2,491	3.49%	3,082	5.81%	5,846	8.14%	6,629	10.47%
velocity-1.6	752	2.56%	2,388	5.13%	2,822	7.69%	4,717	8.97%	5,298	8.97%
xalan-2.7	23,256	1.00%	35,295	2.00%	45,889	3.01%	50,317	4.01%	63,670	5.01%
xerces-1.4	4,444	0.46%	5,804	1.60%	6,089	2.29%	6,524	2.97%	7,673	3.43%

5.5 HYDRA versus Zero-R and Random Prediction

We also compare HYDRA with Zero-R and random prediction. Zero-R is a constant classifier which simply predicts every instance to be defective. In random prediction, we randomly predict an instance to be defective or clean according to the ratio of defective instances to total instances. The precision for random prediction is the percentage of defective instances in the data set. Since random prediction is a random classifier with two possible outcomes (e.g., defective or clean), its recall is 0.5. Table 18 presents the F1-scores of HYDRA compared with Zero-R and random prediction. On average across the 29 datasets, Zero-R and random prediction achieve the F1-scores to 0.516 and 0.395. HYDRA improves the average F1-scores of Zero-R and random prediction by 5.42 and 38.23 percent.

Notice for Zero-R, all of the instances are classified to be defective, i.e., all the instances have equal confidence scores. To compute the PofB20 score for Zero-R, we randomly select the instances until the total number of the selected instances is less than 20 percent of the total number of LOC in the project. We repeat the process 50 times, and compute the average PofB20 scores. Table 19 presents the PofB20 of our approach (HYDRA) compared with that of Zero-R. From the table, we note that on average across the 29

TABLE 18
F1-Scores of Our Approach (HYDRA) Compared with Zero-R, and Random Prediction (Random)

Datasets	HYDRA	Zero-R	Random.
ant-1.3	**0.396 ± 0.010**	0.276	0.242
ant-1.4	0.329 ± 0.012	**0.367**	0.310
ant-1.5	**0.347 ± 0.014**	0.197	0.179
ant-1.6	**0.602 ± 0.015**	0.415	0.344
ant-1.7	**0.468 ± 0.010**	0.364	0.308
log4j-1.0	**0.413 ± 0.002**	0.402	0.335
log4j-1.1	**0.538 ± 0.008**	0.507	0.404
log4j-1.2	0.914 ± 0.014	**0.959**	0.648
lucene-2.0	**0.648 ± 0.010**	0.636	0.483
lucene-2.2	0.657 ± 0.010	**0.737**	0.538
lucene-2.4	0.691 ± 0.012	**0.748**	0.544
poi-1.5	0.742 ± 0.003	**0.746**	0.543
poi-2.0	**0.283** ± 0.003	0.211	0.191
poi-2.5	0.780 ± 0.002	**0.784**	0.563
poi-3.0	**0.807 ± 0.005**	0.777	0.560
redaktor	**0.295 ± 0.006**	0.266	0.235
synapse-1.0	**0.252 ± 0.012**	0.185	0.169
synapse-1.1	**0.494 ± 0.011**	0.426	0.351
synapse-1.2	**0.529 ± 0.011**	0.503	0.402
tomcat	**0.190 ± 0.012**	0.165	0.152
velocity-1.4	0.793 ± 0.014	**0.857**	0.600
velocity-1.6	0.503 ± 0.011	**0.508**	0.405
xalan-2.4	**0.315 ± 0.010**	0.264	0.233
xalan-2.5	0.593 ± 0.012	**0.650**	0.491
xalan-2.6	**0.656** ± 0.010	0.634	0.482
xalan-2.7	0.991 ± 0.007	**0.994**	0.664
xerces-1.2	0.240 ± 0.012	**0.278**	0.244
xerces-1.3	**0.417 ± 0.014**	0.264	0.234
xerces-1.4	**0.903** ± 0.012	0.853	0.598
Average.	**0.544 ± 0.223**	**0.516**	**0.395**

TABLE 19
PofB20 of Our Approach (HYDRA) Compared with Zero-R

Datasets	HYDRA	Zero-R
ant-1.3	20.0% ± 1.0%	**20.5%** ± 6.9%
ant-1.4	**46.8%** ± 1.2%	21.5% ± 3.7%
ant-1.5	**28.6%** ± 1.0%	19.7% ± 5.3%
ant-1.6	14.4% ± 1.0%	**20.3%** ± 2.7%
ant-1.7	**24.8%** ± 0.8%	20.5% ± 3.4%
log4j-1.0	19.7% ± 1.1%	**23.5%** ± 3.8%
log4j-1.1	16.5% ± 1.2%	**21.1%** ± 6.4%
log4j-1.2	**54.8%** ± 1.4%	19.0% ± 4.2%
lucene-2.0	**35.4%** ± 1.0%	21.1% ± 6.7%
lucene-2.2	**38.5%** ± 1.0%	22.6% ± 4.8%
lucene-2.4	**39.5%** ± 1.0%	23.1% ± 6.1%
poi-1.5	**35.6%** ± 1.0%	22.1% ± 3.3%
poi-2.0	15.8% ± 1.2%	**17.6%** ± 6.9%
poi-2.5	**49.6%** ± 0.4%	22.6% ± 4.1%
poi-3.0	**42.4%** ± 0.3%	20.7% ± 3.7%
redaktor	**50.0%** ± 0.2%	23.0% ± 6.1%
synapse-1.0	**23.8%** ± 1.1%	20.6% ± 8.9%
synapse-1.1	**33.0%** ± 0.8%	18.0% ± 2.3%
synapse-1.2	**24.8%** ± 1.3%	19.3% ± 3.4%
tomcat	21.9% ± 1.5%	**22.1%** ± 4.6%
velocity-1.4	**67.5%** ± 0.5%	20.7% ± 12.1%
velocity-1.6	**46.3%** ± 0.3%	21.7% ± 7.3%
xalan-2.4	12.9% ± 0.2%	**21.6%** ± 3.3%
xalan-2.5	**38.5%** ± 0.4%	19.1% ± 3.8%
xalan-2.6	**31.5%** ± 1.2%	19.8% ± 1.0%
xalan-2.7	**51.4%** ± 1.0%	20.0% ± 2.5%
xerces-1.2	15.8% ± 1.0%	**19.4%** ± 3.7%
xerces-1.3	13.0% ± 1.4%	**22.8%** ± 6.4%
xerces-1.4	**44.7%** ± 1.3%	18.0% ± 4.1%
Average.	**33.0%**	20.8%

TABLE 20
Number of Datasets Where HYDRA Statistically Significantly Improves over Zero-R and Random Prediction (+), Performs More or Less Equally Well with Zero-R and Random Prediction (=), and Statistically Significantly Loses with Zero-R and Random Prediction (-) in Terms of F1

HYDRA versus Baselines	+	=	-
HYDRA versus Zero-R	14	7	8
HYDRA versus Random	28	0	1

datasets, Zero-R only achieves a PofB20 score to 20.8 percent, and HYDRA improves it by 58.7 percent in terms of PofB20.

We apply the Wilcoxon signed-rank test on the 29 datasets to test whether the improvement of HYDRA over the Zero-R and random prediction are significant. We also use Bonferroni correction to counteract the results of multiple comparisons. Table 20 presents the number of datasets where HYDRA statistically significantly improves over Zero-R and random prediction (+), performs more or less equally well with Zero-R and random prediction (=), and statistically significantly loses with Zero-R and random prediction (-) in terms of F1. We notice in most of the cases, HYDRA shows statistically significantly improvement over Zero-R and random prediction. For example, comparing HYDRA and Zero-R, we notice HYDRA statistically significantly improves over Zero-R in 14 datasets, while Zero-R statistically significantly improves over HYDRA in 8 datasets.

Table 21 presents the number of datasets where HYDRA statistically significantly improves over Zero-R (+), performs more or less equally well with Zero-R (=), and statistically significantly loses from Zero-R (-) in terms of PofB20. We notice in most of the datasets (i.e., 20 datasets), HYDRA achieves statistically significant improvements over Zero-R, while Zero-R statistically significantly improves over HYDRA in seven datasets.

5.6 Incorporating Labeled Instances

HYDRA uses a limited number of labeled instances in the target project. In the transfer learning literature, it and *TransferBoost* belong to the family of *inductive or supervised transfer learning approaches* [40]. Different from these two, the other baselines use all of the unlabeled data from the target project, and do not need any labeled data from the target project. They belong to the family of *feature-based transductive or unsupervised transfer learning approaches* [40]. For *inductive transfer learning approaches*, some effort is needed to label a small number of data in the target projects. However, for many real projects, often there are a limited number of training instances from defects that testers and users reported, c.f., [2].

TABLE 21
Number of Datasets Where HYDRA Statistically Significantly Improves over Zero-R (+), Performs More or Less Equally Well with Zero-R (=), and Statistically Significantly Loses with Zero-R (-) in Terms of PofB20

HYDRA versus Baselines	+	=	-
HYDRA versus Zero-R	20	2	7

TABLE 22
HYDRA Compared with Basic*, TCA+*, Peters*, GP*, MO*, and CODEP*

Approaches	F1-score	PofB20
HYDRA	0.544 ± 0.223	$33.0\% \pm 14.6\%$
BASIC*	0.385 ± 0.114	$20.7\% \pm 7.2\%$
TCA+*	0.431 ± 0.163	$24.1\% \pm 8.5\%$
Peters*	0.421 ± 0.140	$22.8\% \pm 7.2\%$
GP*	0.379 ± 0.102	$21.8\% \pm 7.8\%$
MO*	0.433 ± 0.214	$21.5\% \pm 11.2\%$
CODEP*	0.419 ± 0.115	$22.3\% \pm 9.7\%$

To investigate whether the 5 percent labeled data in the target projects affect the performance of the other baselines, we separate instances belonging to each of the target projects into two sets: the same 5 percent labeled data that are used by *HYDRA*, and the remaining 95 percent of the data. We incorporate the 5 percent labeled data into the training dataset of the other baselines, to confirm whether the performance of these two approaches would be different. Table 22 presents the F1-scores and PofB20 of *HYDRA* compared with those of BASIC, TCA+, Peters filter, GP, MO, and CODEP with 5 percent labeled data from the target projects (referred to as BASIC*, TCA+*, Peters*, GP*, MO*, and CODEP* respectively). We notice that for BASIC*, its average F1-score is slightly decreased. For TCA+*, its average F1-score is the same. For others (i.e., Peters*, GP*, MO*, and CODEP*), their F1-scores are slightly increased. Still their F1-scores are lower than those of *HYDRA*. Also, the PofB20 scores for the baseline approaches are slightly increased, however still they are lower than those of HYDRA.

5.7 Threats to Validity

Threats to internal validity relates to errors and the replication of the baseline approaches. We have double checked our experiments and datasets, still there could be errors that we did not notice. Also, all of our datasets are from PROMISE repository, still there can be some quality problems among the datasets.

Threats to external validity relates to the generalizability of our results. We have analyzed 29 defect datasets from 10 different open-source software projects, which contain a total of 11,196 instances. In the future, we plan to reduce this threat further by analyzing even more defect data. One potential threat to validity is the quality of our defect datasets. All of our datasets are obtained from the PROMISE repository, which were used in many past studies. Still, there could be quality issues in these datasets. A related threat to validity corresponds to the single source of data (i.e., PROMISE repository). Furthermore, all of the 29 datasets are from open source projects. In the future, we plan to reduce this threat by performing experiments on additional datasets beyond those in the PROMISE repository especially those that are extracted from commercial software projects.

Threats to construct validity refers to the suitability of our evaluation measures. We use F1-score and PofB20, and one or both of them have been used in past studies to evaluate defect prediction tool's effectiveness [23], [36], [42], [43]. Another threat to validity relates to our conclusion. In this paper, we run Wilcoxon signed-rank test to investigate

whether the improvements of HYDRA over the baseline approaches are significant. To counteract the bias due to multiple comparisons, we employ Bonferroni correction. Both Wilcoxon signed-rank test and Bonferroni correction are classical statistical methods.

6 Related Work

There have been a number of studies on defect prediction [14], [23], [25], [29], [36], [38], [42], [47], [52], [52], [57], [58]. Most of these studies predict defect by leveraging machine learning techniques and are evaluated in within-project defection prediction setting [14], [23], [25], [38], [47], [57]. In this setting, defect prediction approaches are trained and applied on classes/files/modules from the same project. Koru et al. perform an empirical study on two commercial systems, and find that smaller modules will be proportionally more defect-prone compared to larger ones [27]. Bettenburg et al. use an algorithm called MARS which is a global model that has local consideration to improve the performance of defect prediction [6]. Kim et al. propose the change classification problem, and use support vector machines (SVM) to classify a change to be buggy or clean [25]. However, in practice, it is rare that sufficient training data is available for a new project, but there are plenty of data from other projects.

To address the limitation of within-project defect prediction, recently, a number of cross-project defect prediction approaches have been proposed. Turhan et al. employ a k-nearest neighbor approach to select instances from source projects to be used as training data; for every unlabeled instance in a target project, they select 10 nearest instances from source projects [52]. Ma et al. propose transfer naive Bayes (TNB) which addresses the difference in the data distribution between source and target projects by weighting training instances [29]. Similar to the work by Turhan et al., Peters et al. also use a nearest neighbor approach to select instances from source projects; however, a different instance selection mechanism is employed [42]. Nam et al. extends TCA, which transforms data from source and target projects to a latent space where the two data sets are close to each other [36]. They propose TCA+ which extends TCA with some data pre-processing options and a heuristic to decide the best pre-processing option to use. Liu et al. propose a genetic programming based approach (GP) which constructs a classification model in the form of a tree considering defect data from multiple software repositories [28]. Canfora et al. construct a classification model (MO) by using multi-objective genetic algorithm for cross-project defect prediction [9]. Panichella et al. propose an approach named CODEP that uses a classification model to combine results of six classification algorithms (i.e., logistic regression, RBF network, multi-layer perceptron, etc.) for cross-project defect prediction [41]. Turhan et al. perform an empirical study on the effectiveness of the combination of within and cross (i.e. mixed) project data for binary defect prediction [53].

In the machine learning community, there have been a number of studies on transfer learning [13], [15], [39], [40]. The previous studies on transfer learning can be classified into two categories: (1) a small amount of labeled data are available in the target task[10] [13], [15], i.e., supervised transfer learning; or (2) only some unlabeled data are available in the target task [22], [39], i.e., unsupervised transfer learning. TransferBoost [15] is one the state-of-the-art multi-source supervised transfer learning algorithms, where multiple source domains (i.e. source projects in our context) are available to learn an adaptive prediction model for a target domain (i.e. target project). It also builds a model following the AdaBoost framework, but our *HYDRA* is different from *TransferBoost* in several aspects: (1) In each iteration, *HYDRA* builds multiple classifiers for each source project and training target data, and leverages GA to search for a semi-optimal composition of these classifiers; on the other hand, *TransferBoost* builds one classifier from all instances in source projects and training target data; (2) The strategy to assign weights to the instances in the source projects and training target data is different. We have demonstrated that *HYDRA* outperforms *TransferBoost*.

7 Conclusion and Future Work

In this paper, we propose a new cross-project defect prediction approach named *HYDRA*. *HYDRA* includes two phases: genetic algorithm (GA) phase and ensemble learning (EL) phase. In the GA phase, *HYDRA* first builds a classifier for each source project and the target project. Next, *HYDRA* builds a composite classifier, referred to as a GA classifier, by assigning different weights, learned using genetic algorithm, to each classifier. In the EL phase, *HYDRA* iterates the GA phase many times to create many GA classifiers. In each iteration, *HYDRA* builds a GA classifier, and assigns a weight to the GA classifier according to its prediction error rate in the training data. In the end, we have a massive composition of classifiers which is used to predict defective instances in the target project. We evaluate our approach on 29 datasets from 10 different open-source software projects. The results show that *HYDRA* achieves an average F1-scores of 0.544. On average, across the 29 datasets, these results correspond to an improvement in the F1-scores of 26.22, 34.99, 47.43, 28.61, 30.14, and 39.49 percent over TCA+, Peters filter, GP, MO, and CODEP, and *TransferBoost*, respectively. In addition, HYDRA, on average, can discover 33 percent of all bugs if developers inspect the top 20 percent lines of code, which improves the best baseline approach (TCA+) by 44.41 percent. Notice HYDRA only improves the F1-score of Zero-R which predict all the instances to be defective by 5.42 percent, but it improves the PofB20 of Zero-R by 58.65 percent. Although the improvement of F1-score is relatively small compared with PofB20, in practice, Zero-R is hard to use since it simply predicts all of the instances to be defective, and thus developers have to inspect all of the instances to find the defective ones. Moreover, we notice the improvement of HYDRA over other baseline approaches in terms of F1-score and when inspecting the top 20 percent lines of code are substantial, and in most cases the improvements are significant and have large effect sizes across the 29 datasets.

In the future, we plan to evaluate *HYDRA* with datasets from more software projects, and develop a better technique

10. In our setting, a target task is a target software project.

which can improve the prediction performance further. We also plan to extend this work to predict the number of bugs in each instance (instead of only predicting defective/clean labels) by leveraging the bug count information.

Acknowledgments

We thank Jaechang Nam and Sunghun Kim for providing us the source code of TCA+, and Eric Eaton for providing us the source code of *TransferBoost*. Xinyu Wang is the corresponding author. This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, NSFC Program (No.61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2015BAH17F01. The source code and datasets of HYDRA can be downloaded from: https://github.com/xinxia1986/TSE-Code_HYDRA.

References

[1] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons" in *Encyclopedia of Measurement and Statistics*, N. J. Salkind, Ed. Newbury Park, CA, USA: Sage, 2007, Available: http://www. utdallas.edu/ herve/abdi-bonferroni2007-pretty. pdf

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proc. OOPSLA Workshop Eclipse Technol. eXchange*, 2005, pp. 35–39.

[3] A. Arcuri and L. C. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Software Eng.*, 2011, pp. 1–10.

[4] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom Java software," in *Proc. 18th IEEE Int. Symp. Software Reliability*, 2007, pp. 215–224.

[5] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[6] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *Proc. 9th IEEE Work. Conf. Mining Softw. Repositories*, 2012, pp. 60–69.

[7] C. M. Bishop and N. M. Nasrabadi, *Pattern Recognition and Machine Learning*, vol. 1. New York, NY, USA: Springer, 2006.

[8] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of empirical software engineering data," Dept. Comput. Sci., North Carolina State University, 2015, http://openscience.us/repo

[9] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proc. IEEE 6th Int. Conf. Softw. Testing, Verification Validation*, 2013, pp. 252–261

[10] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Inf. Sci.*, vol. 179, no. 8, pp. 1040–1058, 2009.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[12] N. Cliff. *Ordinal Methods for Behavioral Data Analysis*. Psychology Press, New York, USA, 2014.

[13] W. Dai, Q. Yang, G.-R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proc. 24th Int. Conf. Mach. Learning*, 2007, pp. 193–200.

[14] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Working Conf. Mining Softw. Repositories*, 2010, pp. 31–41.

[15] E. Eaton and M. des Jardins, "Selective transfer between learning tasks using task-based boosting," in *Proc. 25th AAAI Conf. Artif. Intell.*, 2011, pp. 337–342.

[16] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *Proc. 2nd Eur. Conf. Comput. Learning Theory*, 1995, pp. 23–37.

[17] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Mach. Learning*, vol. 3, no. 2, pp. 95–99, 1988.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *ACM SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[19] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. Burlington, MA, USA: Morgan Kaufmann, 2006.

[20] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.

[21] B. Henderson-Sellers, *Object-Oriented Metrics, Measures of Complexity*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1996.

[22] J. Jiang and C. Zhai, "Instance weighting for domain adaptation in NLP," in *Proc. 45th Annu. Meeting Assoc. Comput. Linguistics*, 2007, pp. 22.

[23] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th Int. Conf. Autom. Softw. Eng.*, 2013, pp. 279–289.

[24] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, pp. 9.

[25] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar./Apr. 2008.

[26] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 481–490.

[27] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Softw. Eng.*, vol. 13, no. 5, pp. 473–498, 2008.

[28] Y. Liu, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 852–864, Nov./Dec. 2010.

[29] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Inf. Softw. Technol.*, vol. 54, no. 3, pp. 248–256, 2012.

[30] R. Martin, "OO design quality metrics—An analysis of dependencies," in *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, 1994, pp. 151–170.

[31] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[32] K. Meffert, N. Rotstan, C. Knowles, and U. Sangiorgi. (2011). Jgap-java genetic algorithms and genetic programming package [Online]. Available: http://jgap.sourceforge.net/

[33] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of empirical software engineering data," Dept. Comput. Sci., North Carolina State University, 2015, http://openscience.us/repo

[34] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[35] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proc. 4th Int. Workshop Predictor Models Softw. Eng.*, 2008, pp. 47–54.

[36] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. Int. Conf. Soft. Eng.*, 2013, pp. 382–391.

[37] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 63.

[38] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," in *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, Dec. 1996.

[39] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE Trans. Neural Netw.*, vol. 22, no. 2, pp. 199–210, Feb. 2011.

[40] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.

[41] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'nion fait la force," in *Proc. Softw. Evolution Week—IEEE Conf. Softw. Maintenance, Reengineering Reverse Eng.*, 2014, pp. 164–173.

[42] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Proc 10th Int. Workshop Mining Softw. Repositories*, 2013, pp. 409–418.

[43] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. Int. Conf. Softw. Eng*, 2013, pp. 432–441.

[44] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng*, 2012, p. 61.

[45] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proc. 9th Joint Meeting Foundations Softw. Eng.*, 2013, pp. 147–157.

[46] R. E. Schapire, Y. Freund, P. Barlett, and W. S. Lee, "Boosting the margin: A new explanation for the effectiveness of voting methods," in *Proc. 14th Int. Conf. Mach. Learn.*, 1997, pp. 322–330.
[47] S. Shivaji, J. E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve bug prediction," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2009, pp. 600–604.
[48] S. Sivanandam and S. Deepa, *Introduction to Genetic Algorithms*. New York, NY, USA: Springer, 2007.
[49] M. Tang, M. Kao, and M. Chen, "An empirical study on object-oriented metrics," in *Proc. 6th Int. Softw. Metrics Symp.*, 2009, pp. 242–249.
[50] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 386–396.
[51] B. Turhan, "On the dataset shift problem in software engineering prediction models," *Empirical Softw. Eng.*, vol. 17, no. 1/2, pp. 62–74, 2012.
[52] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.
[53] B. Turhan, A. T. Mısırlı, and A. Bener, "Empirical evaluation of the effects of mixed project data on learning defect predictors," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1101–1118, 2013.
[54] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics*, vol. 1, no. 6, pp. 80–83, 1945.
[55] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: Recovering links between bugs and changes," in *Proc. 19th ACM SIGSOFT Symp., 13th Eur. Conf. Foundations Softw. Eng.*, 2011, pp. 15–25.
[56] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*. Boca Raton, FL, USA: CRC Press, 2012.
[57] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 531–540.
[58] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 91–100.