Cross-Project Build Co-change Prediction

Xin Xia*, David Lo†, Shane McIntosh‡, Emad Shihab§, and Ahmed E. Hassan‡
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
‡School of Computing, Queen's University, Kingston, Canada
§Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
xxkidd@zju.edu.cn, davidlo@smu.edu.sg, mcintosh@cs.queensu.ca,
eshihab@cse.concordia.ca, and ahmed@cs.queensu.ca

***Abstract*—Build systems orchestrate how human-readable source code is translated into executable programs. In a software project, source code changes can induce changes in the build system (aka. build co-changes). It is difficult for developers to identify when build co-changes are necessary due to the complexity of build systems. Prediction of build co-changes works well if there is a sufficient amount of training data to build a model. However, in practice, for new projects, there exists a limited number of changes. Using training data from other projects to predict the build co-changes in a new project can help improve the performance of the build co-change prediction. We refer to this problem as *cross-project build co-change prediction*.**

In this paper, we propose *CroBuild*, a novel cross-project build co-change prediction approach that iteratively learns new classifiers. *CroBuild* constructs an ensemble of classifiers by iteratively building classifiers and assigning them weights according to its prediction error rate. Given that only a small proportion of code changes are build co-changing, we also propose an imbalance-aware approach that learns a threshold boundary between those code changes that are build co-changing and those that are not in order to construct classifiers in each iteration. To examine the benefits of *CroBuild*, we perform experiments on 4 large datasets including Mozilla, Eclipse-core, Lucene, and Jazz, comprising a total of 50,884 changes. On average, across the 4 datasets, *CroBuild* achieves a F1-score of up to 0.408. We also compare *CroBuild* with other approaches such as a basic model, AdaBoost proposed by Freund et al., and TrAdaBoost proposed by Dai et al.. On average, across the 4 datasets, the *CroBuild* approach yields an improvement in F1-scores of 41.54%, 36.63%, and 36.97% over the basic model, AdaBoost, and TrAdaBoost, respectively.

***Keywords*—Cross-project, Build Co-change Prediction, Transfer Learning, Imbalance Data**

I. Introduction

The build system is an indispensable component in a software system, which compiles source code, libraries and other data into executable programs by executing various compilers and other tools [1]. Build systems are complex, often describing how to assemble numerous software configuration [2] that are specified using hundreds of build files (e.g., *Makefile* or *ant* files) [3]. A prior study found that build maintenance increases software development costs by an additional 12%-36% [4].

When a software project evolves, its source code is often changed to add new features, refactor its structure, and fix bugs. Previous studies have shown that source code and build system tend to co-evolve [5], [6], i.e., source code changes[1] will induce changes in the build system (aka. build co-changes), and vice versa. Furthermore, developers have difficulty identifying the code changes that require build co-changes, which can cause build breakages that can slow the release process down [7]–[9]. For example, Seo et al. found that 29.7% and 37.4% of builds triggered by Java and C++ developers at Google on their local copies of the projects are broken, mainly due to neglected build maintenance [7].

To help developers identify build co-changes, our prior work trained classifiers that predict whether or not a source code change will be build co-changing [10]. However, to achieve high performance, these classifiers must be trained using a large amount of historical data that new projects may not have accrued yet. Fortunately, there are many long-lived software projects that have collected a plethora of historical data. Hence, we propose *cross-project build co-change prediction*, i.e., the use of training data from other projects (aka. source projects) to predict the build co-changes in a particular project of interest (aka. target project) to improve the performance of build co-change prediction in projects in the initial development phases.

Cross-project build co-change prediction is a challenging task for two main reasons. First, a classifier that is trained on source projects might not generalize to other target projects, due to the difference in the domains of the source and target projects. Indeed, one must build a classifier that captures generalizable properties of build co-changes, while discarding domain-specific properties that may not apply for a target project. Second, there is an imbalanced distribution of source code changes that are build co-changing and those that are not. For example, only 16.5%, 16.5%, 8.3%, and 10.2% of the source code changes in Eclipse-core, Jazz, Lucene, and Mozilla projects are build co-changes, respectively. We refer to this phenomenon as the class imbalance phenomenon [11].

In this paper, we propose *CroBuild*, a cross-project build co-change prediction approach that addresses the above challenges by iteratively building an ensemble of imbalanced classifiers. To address the challenge of generalizability, *CroBuild* is a transfer learning approach [12] that transfers the knowledge from source projects to the target projects. To that end, *CroBuild* operates in a setting where there are numerous historical code changes available in a source project, and limited historical code changes available in a target project.

[1]In this paper, we consider a source code change as a commit to a version control system.

We refer to this limited amount of historical data in the target project as target training data. To address the class imbalance challenge, in each iteration, a new classifier is trained and an effective decision boundary is learned. In a nutshell, *CroBuild* searches for an effective classifier threshold that maximizes F1-scores achieved in the training target data.

To evaluate *CroBuild*, we perform experiments on the large Mozilla, Eclipse-core, Lucene, and Jazz datasets containing a total of 50,884 changes. We measure the performance of the approaches in terms of F1-score and AUC values. On average, across the 4 datasets, *CroBuild* achieves a F1-score and AUC values to 0.408 and 0.738. We also compare *CroBuild* to other approaches such as a basic model[2], AdaBoost proposed by Freund et al. [13], and TrAdaBoost proposed by Dai et al. [14]. On average, across the 4 datasets, *CroBuild* improves F1-score and AUC values of basic model by 41.54% and 3.97%, of AdaBoost by 36.63% and 12.12%, of TrAdaBoost by 36.97% and 14.16%, respectively.

The main contributions of this paper are:

- We propose the problem of cross-project build co-change prediction. To the best of our knowledge, this is the first time the problem is proposed and studied. Also, considering the challenges in cross-project prediction, we propose a novel cross-project build co-change prediction approach named CroBuild, which iteratively learns new classifiers and good decision boundaries in each iterations.
- We evaluate our approach with other state-of-the-art approaches such as AdaBoost, and TrAdaBoost on 4 datasets containing a total of 50,884 instances. The experiment results show that our approach can achieve a substantial improvement over these baseline approaches.

Paper organization. The remainder of the paper is organized as follows. Section II provides the technical motivation for the problem of cross-project build co-change prediction. Section III describes the *CroBuild* architecture. Section IV details the *CroBuild* technique. Section V presents the results of our comparative evaluation of *CroBuild*. Section VI discusses the limitations of *CroBuild* and threats to the validity of our evaluation. Section VII surveys the related work. Finally, Section VIII concludes the paper.

II. Technical Motivation

New projects will have limited historical data. Yet plenty of data is available in other projects. We can leverage the limited data in a target project and also other data from other projects to help predict the build co-changes in the target project. Thus, the need of our CroBuild approach relies on findings of the following 2 investigations:

Investigation 1: *Can a prediction model built on a small number of changes (e.g., 5%) achieve similar performance as a prediction model built on a large number of changes (e.g., 90%)?*

[2]In the basic model, we build a random forest model by using data in the source project, and predict the changes in the target project.

TABLE I. The precision, recall, F1-score, and AUC values for the prediction model built on the 5%, 50%, and 90% number of changes in Mozilla.

Approach	Precision	Recall	F1-score	AUC
5%	0.657	0.478	0.554	0.793
50%	0.776	0.551	0.644	0.874
90%	0.788	0.564	0.658	0.868

TABLE II. The precision, recall, F1-score, and AUC values for the prediction model built on the Eclipse-core, Jazz, and Lucene.

Source=>Target	Precision	Recall	F1-score	AUC
Eclipse-core=>Mozilla	0.432	0.377	0.402	0.790
Jazz=>Mozilla	0.393	0.134	0.200	0.693
Lucene=>Mozilla	0.481	0.173	0.254	0.779

Approach. To perform Investigation 1, we first randomly select 5% of the changes from Mozilla, and build a classifier based on the selected changes. Then, we apply the classifier to predict build co-changes using the remaining 95% of the changes. We repeat the process 100 times, and record the average precision, recall, F1-score, and AUC values. We repeat the same process with 50% and 90% of the number of changes being used as training data. The reason we train with 5%, 50%, and 90% of the number of changes is to examine whether the performance of prediction models differs when learning from asmall (i.e., 5%), medium (i.e., 50%), or large (i.e., 90%) amount of changes. To align with the prior work by McIntosh et al. [10], we use the random forest algorithm [15] to build our classifiers.

Results. Table I presents the precision, recall, F1-score, and AUC values of prediction models built using 5%, 50%, and 90% of the number of changes in Mozilla. We observe that a prediction model built on a small number of training data achieves lower performance compared to a model built on a large training data. In Table I, the F1-score and AUC values for the classifier built on 5% of the number of changes are only 0.554 and 0.793 respectively, while those values for the classifier built on 90% of the number of changes are 0.658 and 0.868 respectively.

Investigation 2: *Can a prediction model built on change data from other projects be used to effectively predict build co-changes of a new project?*

Approach. To perform Investigation 2, we select the Mozilla project as the target project, and Eclipse-core, Jazz, and Lucene as the source projects. The domains of these 4 projects are different (i.e., they provide different functionalities). We first build 3 classifiers by using the change data in Eclipse-core, Lucene, and Jazz, respectively. Then, we apply the 3 classifiers to predict the build co-changes in Mozilla. Similar to Investigation 1, we also record the precision, recall, F1-score, and AUC values.

Results. Table II presents the precision, recall, F1-score, and AUC values for the prediction models built on Eclipse-core, Jazz, and Lucene, respectively. We observe that the prediction models do not work well to predict build co-changes in the target project. In Table II, the F1-scores for models built on Eclipse-core, Jazz, and Lucene are 0.377, 0.200, and 0.254, respectively, which is much lower than the prediction model built on Mozilla's own data.

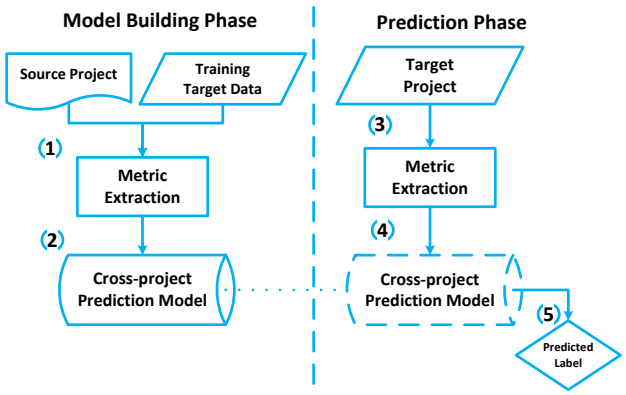


Fig. 1. Overall architecture of *CroBuild*.

Investigation Summary. The above preliminary experiments show that a build co-change classifier built using insufficient data (Investigation 1), or built using the data from another project (Investigation 2) do not perform well. In practice, new projects will not have accrued sufficient data to train a high performance build co-change classifier. To address these challenges, we need to maximize on the value of the limited data that may be available in the target project by combining it data available from other projects. However, the external data will need to be carefully processed in order to retain generalizable features of build co-changes while discarding the project-specific ones. Thus, in this paper, we propose *CroBuild*, which considers both the limited data in the target projects and data in the source projects. Furthermore, *CroBuild* bridges the domain difference between source projects and the target project by building an ensemble of classifiers.

III. CroBuild Architecture

Figure 1 presents the architecture of CroBuild, which contains two phases: a model building phase and a prediction phase. In the model building phase, our goal is to build a cross-project build co-change prediction model learned from instances in a source project and the training target data (i.e., 5% changes from the target project that are labeled as build or non-build). In the prediction phase, we apply this model to predict if a new change in the target project is a build co-change or not.

Our framework takes as inputs instances (i.e., changes) from a source project with known labels (i.e., *build* or *non-build*). Moreover, it also needs a small number of labeled instances from the target project (i.e., 5% of the instances). Note that the labels of these instances can be automatically decided by mining project historical data (e.g., checking whether changes contain modifications of build files). Next, it extracts various metrics from instances in the the source project and the training target data (Step 1). Table III shows the metrics we collected that are also used by McIntosh et al. [10]. Notice that we extract the same metrics from the source project and the target project. Then, our framework builds a cross-project build co-change prediction model based on the metrics from the source project and the training target data (Step 2). The model is a machine learning classifier which assigns labels (in our case: *build* or *non-build*) to an instance (in our case: a change) based on its metrics.

TABLE III. Extracted Metrics.

Metrics	Description
new_src	Number of source code files added in a change.
new_test	Number of test files added in a change.
del_src	Number of source code files deleted in a change.
del_test	Number of test files deleted in a change.
mod_src	Number of source code files modified in a change.
mod_test	Number of test files modified in a change.
ren_src	Number of source code files renamed in a change.
ren_test	Number of test files renamed in a change.
num_dependency	Number of dependencies added/removed on other code through #include preprocessor directives for C++ code or import statements in Java code.
num_ifdefs*	number of times that #if[n][def] preprocessor directives added or removed.
prior_build_changes*	For each source and test file involved in a change, the maximum proportions of prior changes that were build co-changes.
num_files*	The number of source and test files that are involved in a change.

*In Jazz, these 3 metrics are not collected due to privacy concerns.

After the model is constructed, in the prediction phase it is then used to predict whether an unlabeled instance (i.e., change) in the target project is a build co-change or not. For each 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). It outputs the prediction result, which is one of the following labels: *build* or *non-build* (Step 5).

IV. Our Proposed Approach

In this section, we present the details of CroBuild. CroBuild has two types of projects, source projects S and target projects T. The source project contains many instances, and an instance corresponds to a change. Each instance has two parts: a set of metrics x and a label y, which corresponds to whether a change is a build or non-build co-change ($y = 1$ represents *build*, $y = 0$ represents *non-build*). For unlabeled instances in the target project T, the goal of *CroBuild* is to predict the labels of the instances by using the model trained using instances in the source project S and a small number of instances from the target project whose labels are known (aka. training target data) T_t. In the following sections, we first propose our imbalanced classifier approach that is used as the underlying classifier of CroBuild in Section IV-A. Next, we propose CroBuild, which learns an ensemble of imbalanced classifiers to bridge the domain differences between the source projects and the target project in Section IV-B.

A. Imbalanced Classifier

In this work, we are interested in identifying build co-changes, and follow the prior work [10], [16], [17], we use random forest [15] to construct a classifier. In the model building phase, random forest constructs a number of decision trees by using instances in the training set. To predict the label for a new change, random forest inputs the instance into sets of decision trees, and predicts the label of the instance based on the majority voting of the outputs of the set of decision trees.

The output of random forest is a likelihood score between 0 and 1 that denotes the confidence that the instance is a build co-change. Then, the user determines a threshold that she/he will consider a change as being a build co-change. By default, a threshold of 0.5 is used. This means that if a change has a

likelihood score of 50% or higher, then it is considered a build co-change, otherwise it is not.

In our collected data, we notice that the distribution of build and non-build co-changes is skewed. Only 16.5%, 16.5%, 8.3%, and 10.2% of the source code changes in Eclipse-core, Jazz, Lucene, and Mozilla projects respectively are build co-changes. Due to the class imbalance phenomenon, a random forest classifier would be prone to predict a change as a non-build co-change if we set the threshold to 0.5. Hence, we need to automatically determine a good threshold value. Also, the threshold value can be different for different datasets and the value of the threshold affects the performance of the prediction models [18].

Algorithm 1 Pseudocode of Our Imbalanced Classifier.

```
1: EstimateThreshold(S, T_t)
2: Input:
3: S: Source Project
4: T_t: Training Target Data
5: Output: threshold
6: Method:
7: Built a random forest classifier Cla on S;
8: for all change c in T_t do
9:     Compute the likelihood score of c to be a build co-change by using Cla;
10: end for
11: for all threshold from 0 to 1, every time increase threshold by 0.01 do
12:     Predict the labels of changes in T_t according to Equation (1);
13:     Compute the F1-score on T_t;
14: end for
15: Return threshold which maximizes the F1-score for changes in T_t
```

Due to the class imbalance phenomenon and the importance of the threshold, we propose Algorithm 1 that automatically estimates a good threshold value in the model building phase. Our algorithm takes as input historical instances in the source project S and training target data T_t. It then builds a classifier on S (Line 7), and for each change c in T_t, it computes c's likelihood score to be a build co-change, i.e., $Score_{build}(c)$ (Lines 8 - 10). Finally, to tune a good threshold value, it gradually increases the threshold value from 0 to 1 (every time increasing the threshold by 0.01), and for each change c in T_t, we predict its label using the following equation:

$$Predict(c) = \begin{cases} \text{Build}, & \text{if Score}_{\text{build}}(c) \geq \text{threshold} \\ \text{Non-Build}, & \text{Otherwise} \end{cases} \quad (1)$$

Finally, we output a threshold which maximizes the F1-score for changes in T_t (Lines 11 - 15).

B. CroBuild: An Ensemble Learning Approach

Imbalance classifiers can help to deal with the class imbalance phenomenon. However, in cross-project build change prediction, we face another challenge: a prediction model that is trained on a source project might not generalize to another target project due to the differences in the domains of the source and the target project. To address this challenge, we adapt AdaBoost [13], which is one of the most popular and widely used ensemble learning algorithms in the machine learning literature.

AdaBoost iterates a number of times and builds a classifier for each iteration. In each iteration, it assigns a weight to the classifier according to its prediction error rate, and also assigns weights to the instances in the training set depending on whether the instances are correctly classified or not. CroBuild follows the principle of AdaBoost to generate the ensemble of classifiers. However, there are several differences between CroBuild and AdaBoost: (1) AdaBoost is designed for traditional supervised learning, while our approach is designed for transfer learning [12], where we transfer the knowledge from source project to the target project. (2) To adapt AdaBoost for transfer learning, we modify the way AdaBoost [13] assigns weights to the classifiers and the instances in each iteration. In AdaBoost, the instances are from the same project, however, in CroBuild we have instances from a source project and from training target data. CroBuild adjusts the weights of instances from the source project differently from those from the training target data. In general, CroBuild assigns higher weights for the wrongly classified instances in the training target data, compared to the weights assigned to the wrongly classified instances in the source projects. Also, during the iterations, CroBuild focuses on minimizing prediction errors on instances in the training target data, while AdaBoost tries to minimize prediction errors on all training instances.

The details of CroBuild are as follows. For each iteration k, we build an imbalance classifier Cla_k according to the data distributions in S and T_t. Next, we assign different weights to the data instances in S and T_t. For the instances that Cla_k predicts correctly, we assign low weights, and for the instances that Cla_k predicts wrongly, we assign high weights. We have different strategies in assigning weights to instances in training target data and those in the source project, since our goal is to minimize errors on instances in the training target data. In the next iteration $(k+1)$, since different data instances have different weights, Cla_k would prioritize data instances with higher weights. At the end of iteration k, we also assign a weight to Cla_k according to its prediction error rate ϵ_k on instances in the training target data T_t. ϵ_k is computed based on instances in T_t that are wrongly classified by Cla_k. We denote the i^{th} instance in T_t as $(x^i_{T_t}, y^i_{T_t})$, where $x^i_{T_t}$ denotes the set of metrics and $y^i_{T_t}$ denotes the label (i.e., build or non-build) of the i^{th} instance. Its weight is denoted as $w^i_{T_t}$. The prediction error rate is computed as follows:

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

In the above equation, $Cla_k(x)$ denotes the predicted label for an unlabeled instance, with a set of metrics x using the classifier Cla_k. For example, consider 3 instances with weights 0.2, 0.3, and 0.5, and labels 1, 0, and 1. After we run the imbalance classifier Cla_k, the predicted labels are 1, 1, and 0. Then, the error rate for Cla_k would be:

$$\epsilon(k) = \frac{0.2 * |1-1| + 0.3 * |0-1| + 0.5 * |1-0|}{0.2 + 0.3 + 0.5} = 0.8$$

At the end of the K iterations, we have a total of K imbalanced classifiers, and each imbalanced classifier has a weight. We refer to the combination of these 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 would output a predicted label.

Algorithm 2 presents the detailed steps of *CroBuild*. We denote the i^{th} instance in the source project as $\{x_S^i, y_S^i\}$ where x_S^i is the set of metrics of the i^{th} instance and y_S^i is its label. Moreover, we denote the weight of the i^{th} instance in S as w_S^i, and the weight of the i^{th} instance in the training target data T_t as $w_{T_t}^i$.

CroBuild first computes the total number of instances in S and T_t (n) (Line 8). Then, it initializes the *source project factor* (β_s) which would be used to reassign weights of instances in source projects (Line 9). We initialize the *source project factor* following the approach by Dai et al. [14]. Next, it initializes the weights of the instances in S and T_t (Line 10). After these initializations, we iterate K times and build K imbalanced classifiers to get the ensemble classifier. For each iteration k, we first normalize the weights of all instances following AdaBoost [13] (Line 12), and then input the instances in the source projects and training target data by using the imbalance classifier presented in Algorithm 1 to get the classifier Cla_k, and also record the threshold value $threshold_k$ (Line 13). For iteration k, we compute the error rate by running Cla_k on instances in T_t (Line 14). An error rate of more than 0.5 means that the performance of Cla_k is even lower than a random guess, so we terminate the process, and discard classifier Cla_k, and return all of the previous imbalance classifiers to form the ensemble classifier (Line 15). If the error rate is less than or equal to 0.5, *CroBuild* calculates weight β_k for Cla_k and also reassigns the weights of instances in the source project and training target data, respectively (Lines 16 and 17). The reassignments of weights of instances in the source project and training target data are done differently. For instances in the source project, if they are wrongly labeled, we still need to increase their weights, but lower than those in the target project. The weights of instances in the training target data are adjusted using β_k which is usually larger than β_s especially when the error rate is relatively high. At the end of the K iterations, we get the final ensemble classifier $\sum_{k=1}^{K} \beta_k Cla_k$, and the ensemble threshold $\sum_{k=1}^{K} \beta_k threshold_k$.

To predict the label of a new change c in the target project, we first compute the likelihood score for c to be a build co-change for each imbalanced classifier Cla_k (denoted as $Score_k(c)$). Then, we predict its label by using the following equation:

$$\text{Predict}(c) = \begin{cases} \text{Build}, & \text{if } \sum_k \beta_k \text{Score}_k(c) \geq \sum_k \beta_k \text{threshold}_k \\ \text{Non-Build}, & \text{Otherwise} \end{cases} \quad (3)$$

Algorithm 2 Pseudocode of *CroBuild*.

1: **CroBuild**(S, T_t, K)
2: **Input:**
3: S: Source project
4: T_t: Training target data
5: K: Maximum number of iterations
6: **Output:** Ensemble Classifier $\sum_{k=1}^{K} \beta_k \cdot Cla_k$.
7: **Method:**
8: Compute the number of instances in S and T_t: $n = |S| + |T_t|$;
9: Set $\beta_s = \frac{1}{2} \ln(1 + \sqrt{2 \ln \frac{ns}{K}})$;
10: Initialize the weights of instances in S, and T_t. We set the weights equally, i.e., $w_S^j = \frac{1}{n}$, and $w_{T_t}^j = \frac{1}{n}$;
11: **for all** Iteration k from 1 to K **do**
12: Normalize the weights in S, and T_t to 1;
13: Input S and T_t into the imbalance classifier (i.e., Algorithm 1) to get the classifier Cla_k, and also the $threshold_k$;
14: Compute the prediction the error rate ϵ_k of Cla_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, and T_t:

$$w_S^j = w_S^i \exp^{-\beta_s |Cla_k(x_S^i) - y_S^i|}, \ 1 \leq i \leq |S|$$

$$w_{T_t}^i = w_{T_t}^i \exp^{-\beta_k |Cla_k(x_{T_t}^i) - y_{T_t}^i|}, \ 1 \leq i \leq |T_t|$$

18: **end for**
19: **Output** Ensemble Classifier ($\sum_{k=1}^{K} \beta_k Cla_k$, $\sum_{k=1}^{K} \beta_k threshold_k$).

V. Experiments and Results

In this section, we evaluate the performance of CroBuild. The experimental environment is a Windows Server 2008, 64-bit, Intel Xeon 2.00GHz server with 80GB RAM.

A. Experiment Setup

We use the same datasets as McIntosh et al. [10], which contain source code changes from 4 open source software projects: Eclipse-core, Jazz, Lucene, and Mozilla. In total we analyze 50,884 source code changes, and among these changes, only 5,409 are build co-changes, which accounts for 10.6% of the total number of changes. Table IV presents the statistics of McIntosh et al.'s data. The columns correspond to the name of projects (**Project**), the time period (# Time), the number of source code changes (# **Change**), the number of build co-changes (# **Build**), and the percentage of build co-changes (% **Build**).

To evaluate CroBuild, we randomly select 5% of the instances in a target project to construct a training target data T_t. We set the number of iterations $K = 100$ to mitigate overfitting [13]. Since our approach involves randomness (i.e., we randomly select 5% of the target instances), similar to Arcuri and Briand [19], we run CroBuild 100 times and record the average performance across the multiple runs. To simulate the practical usage of our approach, when we consider a project as a target project, we choose the other projects as the source projects. For example, if we choose Eclipse-core as the target project, we use Jazz, Lucene, and Mozilla as the source projects, and build 3 prediction models based on the data from the 3 corresponding source projects. We denote them as Jazz⇒Eclipse-core, Lucene⇒Eclipse-core, and Mozilla⇒Eclipse-core, respectively.

We compare the performance of CroBuild to a basic model, AdaBoost [13], and TrAdaBoost [14]. In the basic model, we build a random forest model by using data in the source project, and predict the changes in the target project. In AdaBoost, we also iterate 100 times as we do for CroBuild, and in each iteration, we build a model using the data in the source project and minimize the prediction error rate of the changes in the source project. In TrAdaBoost, we also randomly select 5% of the instances in a target project to construct a sample of training target data T_t. We use the same setting for TrAdaBoost as we use for CroBuild: iterate 100 times and run TrAdaBoost

TABLE IV. STATISTICS OF THE COLLECTED DATA.

Project	Time	# Changes	# Build	% Build
Eclipse-core	2001 – 2010	2,309	382	16.5%
Jazz	2007 – 2008	2,309	382	16.5%
Lucene	2010 – 2013	2,817	234	8.3%
Mozilla	1998 – 2010	43,449	4,411	10.2%

10 times. For both AdaBoost and TrAdaBoost, we use random forest as the underlying classification algorithm.

B. Evaluation Metrics

In this paper, we use the F1-score and AUC values as the main evaluation metrics.

1) F1-score: There are four possible outcomes for a change in the test data: a change can be classified as a build co-change when it truly is a build co-change (true positive, TP); it can be classified as a build co-change when it is actually not a build co-change (false positive, FP); it can be classified as a non-build co-change when it is actually a build co-change (false negative, FN); or it can be classified as a non-build co-change and it truly is a non-build co-change (true negative, TN). Based on these possible outcomes, precision, recall and F1-score are defined as:

Precision: the proportion of changes that are correctly labeled as build co-changes among those labeled as build co-changes, i.e., $P = TP/(TP + FP)$

Recall: the proportion of build co-changes that are correctly labeled, i.e., $R = TP/(TP + FN)$.

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), i.e., $F = (2 \times P \times R)/(P + R)$.

There is a trade-off between precision and recall. One can increase precision by sacrificing recall (and vice versa). In CroBuild, we can sacrifice precision (recall) to increase recall (precision), by manually lowering (increasing) the value of the $threshold$ parameter in Equation (1). This trade-off causes difficulties to compare the performance of several prediction models by using only precision or recall alone [20]. For this reason, we compare the prediction results using the F1-score, which is the harmonic mean of precision and recall. This follows the setting used in the prior study [10] and many other software analytics studies [21]–[25]. In general, the higher the F1-score is, the better the performance of an approach is.

2) Area under the ROC curve (AUC): Due to the imbalance class phenomenon, area under the ROC curve (AUC) is one of the *de facto* performance measure that measures the likelihood that a build co-change is given a higher confidence score than a false positive (i.e., a non-build co-change). Values of AUC range between 0 (worst classifier performance) and 1 (best classifier performance). An AUC score of 0.7 or above is typically considered to be good [26]. In general, the higher the AUC value is, the better the performance of an approach is.

C. Research Questions

In order to evaluate CroBuild, we perform experiments that: (1) compare its performance to other state-of-the-art build co-change prediction techniques, (2) compare its performance to within-project build co-change prediction, and (3) evaluate the impact of the number of code changes used to train the CroBuild classifiers. We formalize our study with the following three research questions:

TABLE V. PRECISION FOR CROBUILD COMPARED WITH THE BASIC MODEL, ADABOOST, AND TRADABOOST.

Source⇒Target	CroBuild	Basic	AdaBoost	TrAdaBoost
Jazz⇒Eclipse	0.321	0.665	0.218	0.665
Lucene⇒Eclipse	0.271	0.517	0.333	0.517
Mozilla⇒Eclipse	0.307	0.370	0.306	0.370
Eclipse⇒Jazz	0.377	0.363	0.203	0.363
Lucene⇒Jazz	0.266	0.260	0.201	0.260
Mozilla⇒Jazz	0.229	0.295	0.167	0.295
Eclipse⇒Lucene	0.209	0.230	0.128	0.230
Jazz⇒Lucene	0.155	0.136	0.091	0.136
Mozilla⇒Lucene	0.194	0.205	0.150	0.205
Eclipse⇒Mozilla	0.445	0.433	0.197	0.433
Lucene⇒Mozilla	0.452	0.389	0.260	0.389
Jazz⇒Mozilla	0.626	0.485	0.109	0.485
Average.	0.321	0.362	0.197	0.362

TABLE VI. RECALL FOR CROBUILD COMPARED WITH THE BASIC MODEL, ADABOOST, AND TRADABOOST.

Source⇒Target	CroBuild	Basic	AdaBoost	TrAdaBoost
Jazz⇒Eclipse	0.672	0.311	0.924	0.311
Lucene⇒Eclipse	0.664	0.085	0.490	0.085
Mozilla⇒Eclipse	0.543	0.314	0.563	0.314
Eclipse⇒Jazz	0.704	0.580	0.990	0.580
Lucene⇒Jazz	0.541	0.318	0.746	0.318
Mozilla⇒Jazz	0.649	0.434	0.832	0.434
Eclipse⇒Lucene	0.525	0.242	0.829	0.242
Jazz⇒Lucene	0.417	0.094	0.803	0.094
Mozilla⇒Lucene	0.614	0.484	0.756	0.484
Eclipse⇒Mozilla	0.684	0.378	0.832	0.378
Lucene⇒Mozilla	0.594	0.132	0.677	0.132
Jazz⇒Mozilla	0.615	0.175	0.825	0.175
Average.	0.602	0.296	0.772	0.296

RQ1: How effective is CroBuild? How much improvement can it achieve over other state-of-the-art approaches?

Motivation. Developers can use CroBuild to identify code changes that require build co-changes. Developers using CroBuild expect it to be accurate. Hence, we first set out to evaluate the accuracy of CroBuild with respect to other state-of-the-art approaches.

Approach. To address RQ1, we compare CroBuild with the basic model, AdaBoost, and TrAdaBoost approaches. We

TABLE VII. F1-SCORES FOR CROBUILD COMPARED WITH THE BASIC MODEL, ADABOOST, AND TRADABOOST. THE STATISTICALLY SIGNIFICANT IMPROVEMENTS ARE MARKED IN BOLD.

Source⇒Target	CroBuild	Basic	AdaBoost	TrAdaBoost
Jazz ⇒ Eclipse	0.435	0.425	**0.353**	0.424
Lucene⇒ Eclipse	0.385	**0.144**	0.396	**0.147**
Mozilla ⇒ Eclipse	0.392	**0.341**	0.397	**0.340**
Eclipse ⇒ Jazz	0.491	**0.444**	**0.337**	**0.447**
Lucene ⇒ Jazz	0.356	**0.288**	**0.317**	**0.286**
Mozilla ⇒ Jazz	0.339	0.344	**0.278**	0.351
Eclipse ⇒ Lucene	0.299	**0.234**	**0.222**	0.340
Jazz ⇒ Lucene	0.226	**0.112**	**0.163**	**0.111**
Mozilla ⇒ Lucene	0.295	0.288	**0.250**	0.288
Eclipse ⇒ Mozilla	0.539	**0.402**	**0.319**	**0.403**
Jazz ⇒ Mozilla	0.621	**0.254**	**0.376**	**0.257**
Lucene ⇒ Mozilla	0.546	**0.200**	**0.193**	**0.198**
Average	0.401	0.290	0.300	0.299

TABLE VIII. AUC VALUES FOR CROBUILD COMPARED WITH THE BASIC MODEL, ADABOOST, AND TRADABOOST. THE STATISTICALLY SIGNIFICANT IMPROVEMENTS ARE MARKED IN BOLD.

Source⇒Target	CroBuild	Basic	AdaBoost	TrAdaBoost
Jazz ⇒ Eclipse	0.769	0.805	**0.663**	0.807
Lucene⇒ Eclipse	0.679	0.672	0.704	**0.553**
Mozilla ⇒ Eclipse	0.681	**0.673**	0.680	**0.620**
Eclipse ⇒ Jazz	0.794	**0.784**	**0.620**	**0.714**
Lucene ⇒ Jazz	0.645	**0.634**	**0.585**	**0.587**
Mozilla ⇒ Jazz	0.612	0.624	**0.503**	0.628
Eclipse ⇒ Lucene	0.754	**0.708**	**0.724**	**0.620**
Jazz ⇒ Lucene	0.674	**0.611**	**0.565**	**0.571**
Mozilla ⇒ Lucene	0.726	0.733	0.725	**0.695**
Eclipse ⇒ Mozilla	0.871	**0.790**	**0.811**	**0.717**
Jazz ⇒ Mozilla	0.882	**0.779**	**0.802**	**0.632**
Lucene ⇒ Mozilla	0.755	**0.693**	**0.506**	**0.603**
Average	0.737	0.709	0.657	0.646

compute the precision, recall, F1 and AUC scores to evaluate the performance of the 3 approaches on the 4 studied projects. Also, since we run these approaches 100 times, we apply the Wilcoxon signed-rank test [27] on the 100 rows of paired performance values to test whether the improvement of CroBuild over the 3 approaches is statistically significant ($\alpha = 0.05$).

Results. Tables V and VI present the precision and recall of the RQ1 experiment. On average across the 4 projects, CroBuild achieves precision and recall values of 0.321 and 0.602, respectively. Precision and recall are both important metrics for build co-change prediction since they measure quality in two aspects. Low precision means a high number of false labels. On the other hand, low recall means that most correct labels are not assigned to the changes. There is a trade off between precision and recall [20], we use the F1-score, which is the harmonic mean of precision and recall, to compare the performance of the different approaches.

Tables VII and VIII show the F1 and AUC scores of the RQ1 experiment. Statistically significant improvements are marked in bold. The F1 and AUC scores of CroBuild vary from 0.226 - 0.621 and 0.612 - 0.882, respectively. On average, across the 4 projects, CroBuild can achieve F1 and AUC scores of 0.401 and 0.731, respectively.

From Table VII, the improvement of CroBuild over the basic model, AdaBoost, and TrAdaBoost are substantial in terms of F1-score. On average, across the 4 projects, CroBuild improves over the basic model, AdaBoost, and TrAdaBoost by 41.54%, 36.63%, and 36.97%, respectively. Also, the Wilcoxon signed-rank tests show that CroBuild statistically significantly improves the basic model on 9 out of 12 source and target project pairs, AdaBoost on 10 out of 12 source and target project pairs, and TrAdaBoost on 8 out of 12 source and target project compositions.

From Table VIII, we notice on average across the 4 projects, CroBuild improves the AUC values of the basic model, AdaBoost, and TrAdaBoost by 3.97%, 12.12%, and 14.16%, respectively. Also, the Wilcoxon signed-rank tests show that CroBuild statistically significantly improves the basic model on 8 out of 12 source and target project pairs, AdaBoost on 9 out of 12 source and target project pairs, and TrAdaBoost on 10 out of 12 source and target project pairs.

TABLE IX. F1-SCORES FOR CROBUILD COMPARED WITH WITHIN-PROJECT PREDICTION (5%, 90%), AND THE BEST RESULTS REPORTED BY MCINTOSH ET AL. [10].

Source⇒Target	CroBuild	5%	90%	McIntosh et al.
Jazz ⇒ Eclipse	0.435	0.213	0.340	0.390
Lucene⇒ Eclipse	0.385			
Mozilla ⇒ Eclipse	0.392			
Eclipse ⇒ Jazz	0.491	0.146	0.231	0.310
Lucene ⇒ Jazz	0.356			
Mozilla ⇒ Jazz	0.339			
Eclipse ⇒ Lucene	0.299	0.150	0.138	0.360
Jazz ⇒ Lucene	0.226			
Mozilla ⇒ Lucene	0.295			
Eclipse ⇒ Mozilla	0.539	0.554	0.658	0.640
Jazz ⇒ Mozilla	0.621			
Lucene ⇒ Mozilla	0.546			
Average	0.401	0.266	0.342	0.425

TABLE X. AUC VALUES FOR CROBUILD COMPARED WITH WITHIN-PROJECT PREDICTION (5%, 90%), AND THE BEST RESULTS REPORTED BY MCINTOSH ET AL. [10].

Source⇒Target	CroBuild	5%	90%	McIntosh et al
Jazz ⇒ Eclipse	0.769	0.649	0.695	0.690
Lucene⇒ Eclipse	0.679			
Mozilla ⇒ Eclipse	0.681			
Eclipse ⇒ Jazz	0.794	0.580	0.720	0.610
Lucene ⇒ Jazz	0.645			
Mozilla ⇒ Jazz	0.612			
Eclipse ⇒ Lucene	0.754	0.723	0.761	0.790
Jazz ⇒ Lucene	0.674			
Mozilla ⇒ Lucene	0.726			
Eclipse ⇒ Mozilla	0.871	0.793	0.868	0.880
Jazz ⇒ Mozilla	0.882			
Lucene ⇒ Mozilla	0.755			
Average	0.737	0.686	0.761	0.743

In most cases, CroBuild achieves a statistically significant improvement compared to the basic model, AdaBoost, and TrAdaBoost in terms of F1 and AUC scores. On average, CroBuild improves F1-scores of the basic model, AdaBoost, and TrAdaBoost by 41.54%, 36.63%, and 36.97%, and AUC by 3.97%, 12.12%, and 14.16%, respectively.

RQ2 Can CroBuild outperform conventional within-project build co-change prediction?

Motivation. We refer to an approach that builds a model using only change data from a project and uses the model to predict other changes from the same project as an *within-project build co-change prediction* approach. Prior work by McIntosh et al. [10] falls under this category. Since we use some labeled training data from a target project (i.e., training target data), we also investigate whether *CroBuild* 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 classifier is used to label the other data from the target project. Thus, we are also interested to investigate whether our approach, which leverages change data from other projects can perform similar to within-project prediction, when a sufficient amount of within-project training data is available.

Approach. To address RQ2, we investigate two settings. First, since by default *CroBuild* requires 5% of the code changes from the target project to be labeled, we investigate the performance of conventional within-project prediction using

the same 5% of data. Second, we randomly select 90% of the instances from the target project, and build a classifier to predict the label of the remaining 10% of the instances. With 90% of the instances labeled, it is likely that conventional within-project predictions can train a high-performance model to predict the remaining 10% of the instances.

Results. Tables IX and X show the F1-scores and AUC values of *CroBuild* compared to those of within-project prediction with 5% and 90% labeled data, and also the best results reported by McIntosh et al. [10]. From Tables IX and X, the improvement of our approach over within-project prediction with 5% labeled data is substantial. On average across the 4 datasets, *CroBuild* outperforms the F1-score and AUC values of within-project prediction with 5% labeled data by 50.89% and 7.40% respectively. Moreover, *CroBuild* still improves the average F1-score of within-project prediction with 90% labeled data by 17.34%, and achieves similar AUC values as the within-project prediction with 90% labeled data. The average AUC of within-project prediction with 90% data is 0.761, while it is 0.737 for *CroBuild*. Note that *CroBuild* only requires 5% labeled data from the target project.

McIntosh et al. propose a re-sampling based approach to predict build co-changes, which removes samples from the non-build (majority) category and repeats samples in the build (minority) category. We also compare *CroBuild* with McIntosh et al.'s approach. In Tables IX and X, we record the best results reported by McIntosh et al. [10]. We notice McIntosh et al.'s approach achieve a slight better performance than *CroBuild*. The average F1-score and AUC values for McIntosh et al.'s approach are 0.425 and 0.743, while those values are 0.401 and 0.737 for *CroBuild*. Once again, it is important to note that McIntosh et al.'s approach requires 90% labeled data from the target project.

CroBuild achieves better performance compared to within-project prediction, when only 5% of the instances are used. Furthermore, CroBuild achieves similar performance as within-project prediction with 90% of instances, such as McIntosh et al.'s approach.

RQ3: Do different percentages of code changes selected from a target project affect the performance of CroBuild?

Motivation. *CroBuild* requires a small number of labeled data from the target project (i.e., training target data). By default, the number of code changes in the training target data is set to 5% of the total number of code changes in the target project. However, how the number of code changes affects the performance of CroBuild remains unknown. Knowing the impact of the number of code changes is important, since we would like to know the smallest amount of code changes that will produce the best results.

Approach. To address RQ3, we vary the number of code changes from 1% - 15% of the total number of code changes in the target project and measure the performance of CroBuild. Additionally, we investigate the performance of CroBuild when a limited budget is specified, i.e., an absolute number of code changes selected from a target project.

Results. Figure 2 presents the F1-scores across the 4 datasets when we vary the percentage of code changes (1% to 15%) from the target project. We notice that in general, when we increase the number of code changes in the training target data from 1% to 15% of the number of code changes in the target project, the F1-scores for CroBuild are slightly increased. For example, when we choose Eclipse as the source project and Mozilla as the target project, the F1-scores vary from 0.538 (4%) to 0.615 (15%). Figure 3 presents the AUC values in Mozilla with various percentages of code changes (1% to 15%) from the target project. Still, we find when we increase the number of instances in the training target data, the AUC values for CroBuild are slightly increased. For the other 3 datasets, we observe a similar result for AUC values. Due to the page limitation, we do show the AUC values in a table.

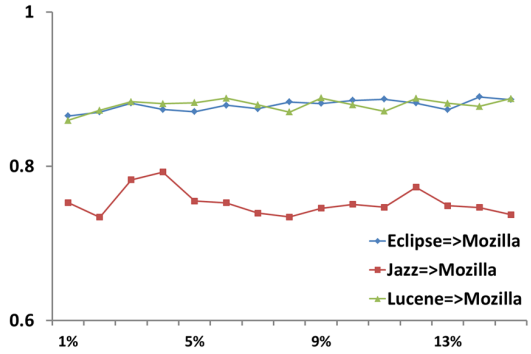


Fig. 3. AUC for different percentages of instances in the training target data (1% to 15%) in Mozilla.

TABLE XI. F1 AND AUC SCORES FOR CROBUILD WITH DIFFERENT NUMBER OF INSTANCES IN THE TRAINING TARGET DATA (100, 300, AND 500, RESPECTIVELY).

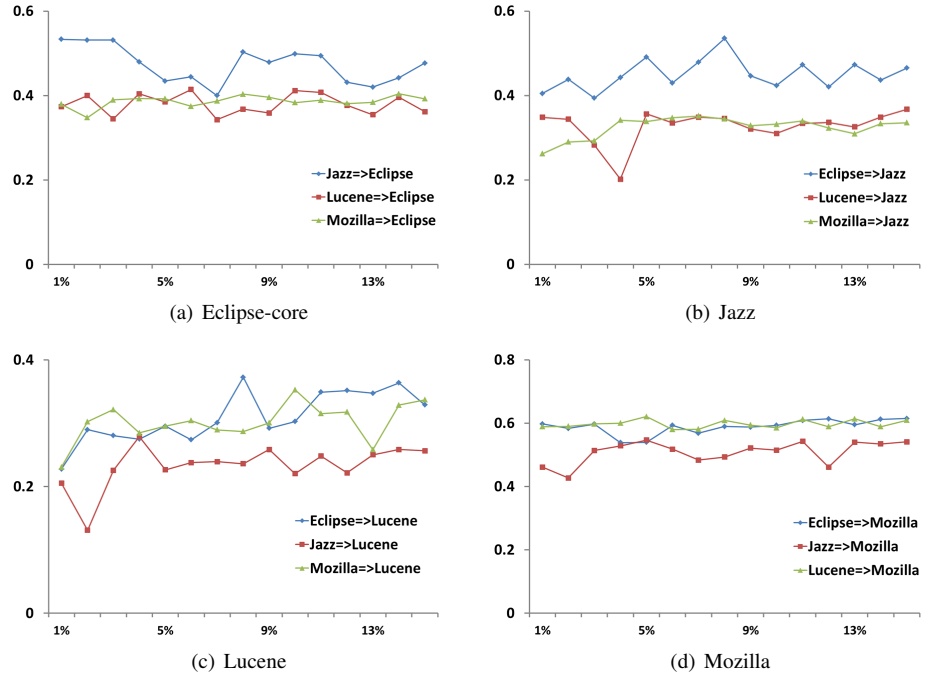
Source⇒Target	100		300		500	
	F1	AUC	F1	AUC	F1	AUC
Jazz ⇒ Eclipse	0.460	0.786	0.493	0.766	0.498	0.766
Lucene⇒ Eclipse	0.304	0.663	0.386	0.684	0.399	0.696
Mozilla ⇒ Eclipse	0.385	0.657	0.393	0.684	0.388	0.700
Eclipse ⇒ Jazz	0.500	0.788	0.432	0.754	0.484	0.789
Lucene ⇒ Jazz	0.340	0.654	0.326	0.632	0.362	0.622
Mozilla ⇒ Jazz	0.342	0.627	0.354	0.597	0.344	0.604
Eclipse ⇒ Lucene	0.253	0.719	0.206	0.756	0.374	0.768
Jazz ⇒ Lucene	0.223	0.634	0.240	0.676	0.301	0.727
Mozilla ⇒ Lucene	0.314	0.753	0.297	0.758	0.403	0.808
Eclipse ⇒ Mozilla	0.404	0.781	0.531	0.853	0.561	0.863
Jazz ⇒ Mozilla	0.391	0.749	0.400	0.755	0.433	0.746
Lucene ⇒ Mozilla	0.364	0.782	0.522	0.847	0.552	0.861
Average	0.357	0.716	0.382	0.730	0.425	0.746

Table XI presents the F1-scores and AUC of CroBuild when there are only 100, 300, and 500 code changes in the training target data. The average F1-scores and AUC for CroBuild vary from 0.357 to 0.425, and 0.716 to 0.746, respectively. With more instances in the training target data, the performance of CroBuild is better.

In general, an increase in the number of instances in the training target data increases the F1 and AUC scores.

VI. THREATS TO VALIDITY

Threats to internal validity relates to errors in our code and experiment bias. We have double-checked our code, still there could be errors that we did not notice. To reduce the training set selection bias, we run CroBuild 100 times, and record the average performance. Also, in our CroBuild, we randomly took 5% of the changes in the target project, which



(a) Eclipse-core (b) Jazz

(c) Lucene (d) Mozilla

Fig. 2. F1-scores for different percentages of instances in the training target data (1% to 15%).

is an approximation since in most cases, we will only have the first 5% of the data. Threats to external validity relate to the generalizability of our results. We have analyzed 50,884 changes from 4 different projects. In the future, we plan to reduce this threat further by analyzing even more change data from additional software projects. Threats to construct validity refer to the suitability of our evaluation measures. We use F1-score and AUC, which are also used by past studies to evaluate the effectiveness of build co-change prediction [10], and also various automated software engineering techniques [21]–[25]. Also, we consider all the false positives to have the same impact. In the future, we plan to reduce this threat by taking into consideration cost in our analysis.

VII. Related Work

Build system maintenance. A number of prior studies have shown that source code changes often require build co-changes. Adams et al. noticed that the source code and the `make`-based build system of the Linux kernel tend to co-evolve [5]. Similar results were also observed in a sample of Ant- and Maven-based build systems [6], [28]. Even for programs written in dynamic scripting languages like Ruby, the source code tends to co-evolve with its build system [29].

To support developers in maintaining build systems, researchers have proposed a variety of tools. For example, Adams et al. propose a reverse-engineering framework named MAKAO to analyze the dependencies in build systems [30]. Tamrawi et al. propose SYMAKE, which analyzes `make`-based build systems using symbolic execution [31]. Zhou et al. propose BuildPredictor, which predicts the missed dependencies in software build system by leveraging the links in the build graph and code graph [32]. Our work is orthogonal to the above studies – we propose CroBuild, which predicts build co-changes in a project by using other projects' data.

Source-build co-change. Prior studies have also shown that up to 27% of source code changes are accompanied by build co-changes [3]. These build co-changes are difficult for developers to identify. To support developers in identifying the code changes that require accompanying build changes, McIntosh et al. propose build co-change prediction [10]. They study the build co-changes in Eclipse-core, Jazz, Lucene, and Mozilla, extract a number of metrics such as the number of files added/removed/modified in a source code change, and propose a re-sampling approach to predict the build co-changes. Our work is different from their work in several ways: (1) the approach proposed in their work is a within-project prediction approach, while CroBuild is a cross-project prediction approach, (2) to address the class imbalance phenomenon, McIntosh et al. use a re-sampling strategy where the instances in the majority class (non-build) are under-sampled and the instances in the minority class (build) are over-sampled, while CroBuild addresses the class imbalance problem by learning an appropriate decision boundary to separate majority and minority classes.

Transfer learning. In the machine learning community, there have been a number of studies on transfer learning [12], [14], [33]. There are two categories for the techniques that focus on transfer learning: (1) supervised transfer learning, where a small amount of labeled data are available for the target task [14][3], and (2) unsupervised transfer learning, where only some unlabeled data are available for the target task [33]. TrAdaBoost is a supervised state-of-the-art transfer learning algorithm, which is most related to ours [14]. It also extends AdaBoost and transfers knowledge from a source project to a target project. Our proposed CroBuild is different from TrAdaBoost: TrAdaBoost does not consider the class imbalance phenomenon, while our CroBuild considers the class imbalance phenomenon in build co-change prediction, and learns a good threshold for the classifier built in each iteration. In the software engineering community, some studies

[3]In our setting, we refer to this small amount of labeled data as training target data

also leverage transfer learning approaches to solve the defect prediction problem [34]–[36]. In contrast, we focus on the problem of cross-project build co-change prediction.

VIII. Conclusion and Future Work

In this paper, we propose the problem of cross-project build co-change prediction - namely the prediction of a source code change that leads to a future change in the build code. We propose *CroBuild*, which iteratively learns new classifiers and a good decision boundary that collectively try to overcome the class imbalance phenomenon and project difference problem. *CroBuild* iterates a number of times, and in each iteration, it builds an imbalance classifier, and assigns a weight to the classifier according to its prediction error rate. In effect, it builds an ensemble of classifiers. To examine the benefits of *CroBuild*, we perform experiments on 4 large datasets including Mozilla, Eclipse-core, Lucene, and Jazz containing a total of 50,884 changes. On average across the 4 datasets, *CroBuild* achieves an F1-score of 0.408. We also compare *CroBuild* with other state-of-the-art approaches, i.e., basic model, AdaBoost proposed by Freund et al., TrAdaBoost proposed by Dai et al.. On average, across the 4 datasets, *CroBuild* results correspond to an improvement in the F1-scores of 41.54%, 36.63%, and 36.97% over the basic model, AdaBoost, and TrAdaBoost, respectively.

In the future, we plan to evaluate *CroBuild* with datasets from additional software projects, and develop a better technique that improves the prediction performance further, e.g., by combining some classical imbalance learning approaches such as SMOTE [11] and OSS [11] .

Acknowledgment. This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2014BAH24F02.

References

[1] P. Smith, *Software build systems: principles and experience*. Addison-Wesley Professional, 2011.

[2] C. AtLee, L. Blakk, J. ODuinn, and A. Gasparian, "Firefox release engineering," *The Architecture of Open Source Applications*, vol. 2.

[3] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *ICSE*, 2011, pp. 141–150.

[4] G. K. T. Epperly, "Software in the doe: The hidden overhead of the build," 2002.

[5] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," *Electronic Communications of the EASST*, vol. 8, 2008.

[6] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of java build systems," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 578–608, 2012.

[7] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: a case study (at google)." in *ICSE*, 2014, pp. 724–734.

[8] I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success? a study of coordination in a software project," *TSE*, pp. 307–324, 2011.

[9] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *ASE*, 2006, pp. 189–198.

[10] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *ICSME*, 2014, pp. 241–250.

[11] H. He and E. A. Garcia, "Learning from imbalanced data," *Knowledge and Data Engineering, IEEE Transactions on*, pp. 1263–1284, 2009.

[12] S. J. Pan and Q. Yang, "A survey on transfer learning," *Knowledge and Data Engineering, IEEE Transactions on*, pp. 1345–1359, 2010.

[13] Y. Freund, R. Schapire, and N. Abe, "A short introduction to boosting," *Journal-Japanese Society For Artificial Intelligence*, p. 1612, 1999.

[14] W. Dai, Q. Yang, G.-R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proceedings of the 24th international conference on Machine learning*, 2007, pp. 193–200.

[15] L. Breiman, "Random forests," *Machine learning*, pp. 5–32, 2001.

[16] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, 2015.

[17] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *MSR*, 2014, pp. 72–81.

[18] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *FSE*, 2012, p. 62.

[19] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE*, 2011, pp. 1–10.

[20] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan kaufmann, 2006.

[21] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, pp. 1005–1042, 2013.

[22] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, pp. 1–30, 2014.

[23] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008, pp. 346–355.

[24] H. V. Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *MSR*, 2014.

[25] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, pp. 1–35, 2014.

[26] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *ICSM*, 2011, pp. 303–312.

[27] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, pp. 80–83, 1945.

[28] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ant build systems," in *MSR*, 2010, pp. 42–51.

[29] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering*, pp. 1–47, 2014.

[30] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *ICSM*, 2007, pp. 114–123.

[31] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *ICSE*, 2012, pp. 650–660.

[32] B. Zhou, X. Xia, D. Lo, and X. Wang, "Build predictor: More accurate missed dependency prediction in build configuration files," in *COMPSAC*, 2014, pp. 53–58.

[33] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *Neural Networks, IEEE Transactions on*, pp. 199–210, 2011.

[34] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *ICSE*, 2013, pp. 382–391.

[35] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *MSR*, 2013, pp. 409–418.

[36] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, pp. 248–256, 2012.