Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

Cutting the Software Building Efforts in Continuous Integration by Semi-Supervised Online AUC Optimization*

Zheng Xie and **Ming Li**
National Key Laboratory for Novel Software Technology, Nanjing University
Collaborative Innovation Center for Novel Software Technology and Industrialization
Nanjing 210023, China
{xiez, lim}@lamda.nju.edu.cn

Abstract

Continuous Integration (CI) systems aim to provide quick feedback on the success of the code changes by keeping on building the entire systems upon code changes are committed. However, building the entire software system is usually resource and time consuming. Thus, build outcome prediction is usually employed to distinguish the successful builds from the failed ones to cut the building efforts on those successful builds that do not result in any immediate action of the developer. Nevertheless, build outcome prediction in CI is challenging since the learner should be able to learn from a stream of build events with and without the build outcome labels and provide immediate prediction on the next build event. Also, the distribution of the successful and the failed builds are often highly imbalanced. Unfortunately, the existing methods fail to address these challenges well. In this paper, we address these challenges by proposing a semi-supervised online AUC optimization method for CI build outcome prediction. Experiments indicate that our method is able to cut the software building efforts by effectively identify the successful builds, and it outperforms the existing methods that elaborate to address part of these challenges.

1 Introduction

Accurately identifying defective modules in software systems is crucial for software development. *Continuous Integration* (CI) [Booch, 1991; Duvall *et al.*, 2007] is an effective mechanism to get quick feedbacks on whether the committed code changes are problematic that result in a failure of system build. If the committed code fails the build or test, the developers can be warned immediately so that they can re-check or revise the code before plugging the defective code into the systems. However, in most cases, the successful builds make up the majority of the build events, and usually take several hours for large projects. Since the successful builds do not provide any guidance on immediate action of the developers, most of the build efforts are wasted.

In this circumstance, predicting whether a build will pass or fail can help to reduce the building efforts, since we can cut the building efforts on those successful builds and prioritize the resources for the failed builds. Such a task is known as continuous build outcome prediction [Hassan and Zhang, 2006] or continuous defect prediction [Madeyski and Kawalerowicz, 2017]. Efforts have been made on this task, e.g., Finlay *et al.* [2014] and Hassan and Zhang [2006] used tree-based methods with different group of features for prediction, Ni and Li [2017] used cascaded classifiers to improve the accuracy of finding failed builds.

However, although previous attempts have indicated that predicting software build outcome is possible and beneficial, few of them actually construct the learning model that perfectly fits the following characteristics of the CI build outcome prediction task:

- *Streaming data*. As the development of the software system goes on, every commit would trigger a CI event. The prediction model is supposed to learn from the series of CI events *from scratch* and at the same time predict the build outcome for each arrived CI event.
- *Few build outcome label*. Due to the heavy computational cost for CI builds and limited resources, the build will be actually conducted on a tiny little portion of the commits to get the build outcome labels. The learner has to exploit the unlabeled build records for model construction.
- *Imbalance in build outcomes*. In most cases, the developers will commit the code that are considered as correct for CI, which makes the builds more likely to success.
- *Suspiciousness of build event required*. To make the build outcome prediction flexible in practice, the suspiciousness of the outcome prediction for each build event is required, such that the workload of the CI server can be configured and adjust as required based on the provided suspiciousness.

These characteristics have place great challenges to the model construction for CI build outcome prediction. Unfortu-

*This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61422304).

nately, few existing studies, to our best knowledge, have ever address all these challenges simultaneously in model construction.

In this paper, we propose a novel semi-supervised online AUC optimization method, named SOLA, for continuous build outcome prediction, which is capable of handling all the challenges mentioned above. Optimizing AUC makes the classifier intrinsically insensitive to the imbalanced data, and have the ability of ranking the build events according to the suspiciousness. To enable AUC optimization to exploit both labeled and unlabeled data in an online situation, we leverage the nature of semi-supervised AUC optimization that the unlabeled data can be used without estimating their possible labels, and formulate the problem into a saddle point problem to solve it online. Experiments show that our method can cut the building efforts in CI by accurately identifying the failed builds, and outperforms the existing methods that only partially address the challenges.

2 Background and Related Work

2.1 Continuous Integration

Continuous Integration was first proposed by Booch to prevent integration problems in extreme programming [Booch, 1991]. Together with automated system build and test, it can avoid the risk of one developer's defective work affecting another developer's copy. Nowadays, CI systems provide multiple services including testing, deploying, feedback, etc. Besides CI systems like Jenkins and Microsoft Team Foundation Server that can be deployed locally, CI cloud platforms like Travis CI have drawn much attention, especially among open-source projects.

A typical continuous integration pipeline is shown in Figure 1. Once a developer commits his change to the version control system, the CI server will pull the latest code and execute the build script, which usually consist of compilation, testing, inspection, and deployment. The build results are returned to all the related developers, to help them localize the software defects. CI saves a lot of human efforts, as well as reduces risk, by automatically executing the repetitive testing and deployment process. According to [Hilton *et al.*, 2016], CI increases the software release frequency by more than 200%.

However, a major drawback of using CI is *unproductive time* [Hilton *et al.*, 2016], that is, the developers have to wait the CI system's feedback to determine whether to re-check the code, which may takes hours to days. In this circumstance, build outcome prediction is employed to distinguish the successful builds from the failed ones to cut the building efforts on those successful builds that do not result in any immediate action of the developer.

Researchers have proposed different approaches for build outcome prediction. Finlay *et al.* [2014] used a tree-based method with a group of features for build outcome prediction, and Hassan and Zhang [2006] used Hoeffding Tree to enable the build outcome prediction in an online circumstance. Ni and Li [2017] used cost-sensitive cascaded classifiers to improve the accuracy of finding failed builds and reduce the CI cost. There are also researches attempting to reduce the time

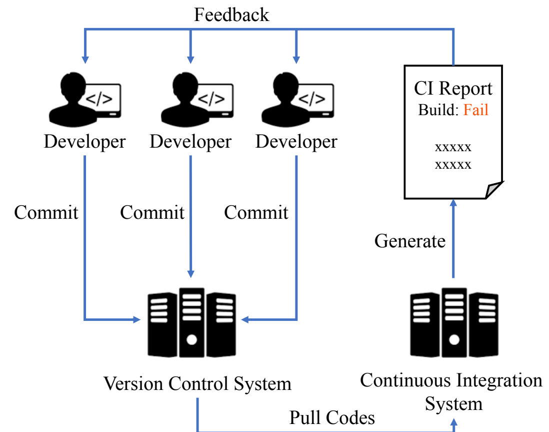

Figure 1: Continuous Integration Pipeline.

cost by improving the unit test in CI. For example, Campos *et al.* [2014] leveraged the historical data to help the budget allocation among code units and the seeding process for the test case generation. All existing researches that aim to improve the prediction accuracy tried to utilize different machine learning techniques to address different challenges. However, none of them pointed all of the four challenges in CI build outcome prediction, and they can only partially address the challenges.

A related task of continuous build outcome prediction is called just-in-time defect prediction, which aims to conduct accurate defect prediction on change level. Just-in-time defect prediction can also be deployed on CI systems, making predictions after the changes are made with commit information. Existing researches on just-in-time defect prediction mainly focus on building learning pipelines to improve the prediction performance, but do not consider about exploiting the unlabeled data or updating model efficiently and incrementally [Kamei *et al.*, 2016; Yang *et al.*, 2017].

2.2 AUC Optimization

AUC is a widely-used performance measure for classifiers, especially for problems that are highly imbalanced. Optimizing AUC is a common method to learn classifiers that rank the positive data before the negative data. Due to the nature of AUC optimization, it is a suitable way to handle the last two challenges, i.e., handling the imbalance and providing the suspiciousness. However, none of the methods of AUC optimization proposed so far can exploit the unlabeled data in an online situation.

Owing to the non-convexity and discontinuousness of AUC risk, many surrogate losses are proposed to simplify the optimization. Gao and Zhou [2015] studied the consistency of the surrogate losses theoretically. For online AUC optimization, Zhao *et al.* [2011] first attempted to extend AUC optimization into online circumstance by maintaining a reservoir. Later, Gao *et al.* [2013] proposed a method that maintains a covariance matrix to optimize AUC online, and Ying *et al.* [2016] formulated the online AUC optimization problem as a stochastic saddle point problem to solve it with stochastic gradient based algorithm.

For AUC optimization in semi-supervised circumstance, Amini *et al.* [2008] extended the RankBoost algorithm to semi-supervised case to learn a ranking function. Fujino and Ueda [2016] used a generative model based algorithm to exploit unlabeled data to optimize AUC. Sakai *et al.* [2018] proposed a semi-supervised AUC optimization method by reweighting the unlabeled data, from a positive-unlabeled learning perspective. Xie and Li [2018] proved that in semi-supervised AUC optimization problem, directly using unlabeled data as both positive and negative data can lead to unbiased AUC optimization.

Although there are researches that extend the AUC optimization into online or semi-supervised learning, none of the researches combines them up since it is difficult to estimate the distribution of the data when the instances come sequentially. We overcome this obstacle by leveraging the nature of semi-supervised AUC optimization that the unlabeled data can be used without estimating their possible label [Xie and Li, 2018], and propose the first semi-supervised online AUC optimization method.

3 Semi-Supervised Online AUC Optimization

To address the continuous build outcome prediction problem, in this section, we describe our SOLA (Semi-Supervised Online AUC Optimization) method.

Let $\mathcal{X}_\text{P}$ and $\mathcal{X}_\text{N}$ denote the set of failed and successful build events, respectively. Since not all of the build events are executed due to the limited computational resources, we also have a set of unlabeled instances, $\mathcal{X}_\text{U}$. AUC over labeled instances can be formalized as:

$$\text{AUC} = 1 - \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_\text{P}}\big[\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_\text{N}}[\ell_{01}(\boldsymbol{w}^\top(\boldsymbol{x}-\boldsymbol{x}'))]\big] . \tag{1}$$

And thus the AUC optimization problem can be formulated as an empirical risk minimization problem. With an ℓ_2 regularizer, the minimization of AUC risk can be formulated as:

$$\min_{||\boldsymbol{w}||<R} \quad R_\text{PN}(\boldsymbol{w}) , \tag{2}$$

where

$$R_\text{PN} = \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_\text{P}} \mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_\text{N}} \ell(\boldsymbol{w}^\top(\boldsymbol{x}-\boldsymbol{x}')) \tag{3}$$

is the supervised (positive vs. negative) AUC risk. Here n_P and n_N are the number of failed and successful build events, respectively. In practice, we use the square loss $\ell(z) = (1-z)^2$ instead of the 0-1 loss, because the 0-1 loss is discrete and difficult to optimize. It has been proved that the square loss is consistent with AUC risk [Gao and Zhou, 2015]. The AUC can be regarded as a pairwise ranking loss, which makes it naturally suitable for *handling imbalanced data* and *ranking the build events by suspiciousness*.

Since the estimation of the data distribution can be hard when data comes sequentially, it is difficult to exploit the unlabeled data for online AUC optimization. We try to overcome this obstacle by leveraging the nature of semi-supervised AUC optimization that optimizing the probability of a randomly drawn positive instance being ranked before a randomly drawn unlabeled instance (or an unlabeled instance being ranked before a negative instance) is equivalent to an unbiased AUC optimization [Xie and Li, 2018]. With this nature, to *exploit the unlabeled data*, the semi-supervised AUC optimization can be formulated as a minimization problem of the following loss function consisting of three risk estimators:

$$\min_{||\boldsymbol{w}||<R} \quad \gamma R_\text{PN} + (1-\gamma)(R_\text{PU} + R_\text{NU}) , \tag{4}$$

where $\gamma \in [0,1]$ is the trade-off parameter of supervised risk and semi-supervised risk, and

$$R_\text{PU} = \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_\text{P}} \mathop{\mathbb{E}}_{\boldsymbol{x}''\in\mathcal{X}_\text{U}} \ell(\boldsymbol{w}^\top(\boldsymbol{x}-\boldsymbol{x}'')) \text{, and} \tag{5}$$

$$R_\text{NU} = \mathop{\mathbb{E}}_{\boldsymbol{x}''\in\mathcal{X}_\text{U}} \mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_\text{N}} \ell(\boldsymbol{w}^\top(\boldsymbol{x}''-\boldsymbol{x}')) \tag{6}$$

are semi-supervised risk estimators.

Directly applying Eq. (4) is still not scalable for streaming data, but it provides us a way to exploit the unlabeled data without estimating the data distribution. Another obstacle to extending Eq. (4) into online circumstance is that the risk of AUC is defined over instance pairs, so it is infeasible to solve the problem by directly using stochastic gradient descent. To overcome this problem, we rewrite Eq. (4) into a saddle point problem, and propose an algorithm that can *update the model online* by a mini-max procedure.

We construct the following saddle point problem, which is equivalent to Eq. (4):

$$\begin{aligned} \min_{\substack{||\boldsymbol{w}||<R,\\ a_1,b_1,\\ a_2,b_2,\\ a_3,b_3}} \max_{\alpha_1,\alpha_2,\alpha_3} \quad & \gamma f_\text{PN}(\boldsymbol{w},a_1,b_1,\alpha_1) \\ & + (1-\gamma)\big(f_\text{PU}(\boldsymbol{w},a_2,b_2,\alpha_2) \\ & + f_\text{NU}(\boldsymbol{w},a_3,b_3,\alpha_3)\big) , \end{aligned} \tag{7}$$

where

$$\begin{aligned} f_\text{PN}(\boldsymbol{w},a,b,\alpha) &= \mathop{\mathbb{E}}_{\boldsymbol{x}}[F_\text{PN}(\boldsymbol{w},a,b,\alpha;\boldsymbol{x})] , \\ f_\text{PU}(\boldsymbol{w},a,b,\alpha) &= \mathop{\mathbb{E}}_{\boldsymbol{x}}[F_\text{PU}(\boldsymbol{w},a,b,\alpha;\boldsymbol{x})] , \\ f_\text{NU}(\boldsymbol{w},a,b,\alpha) &= \mathop{\mathbb{E}}_{\boldsymbol{x}}[F_\text{NU}(\boldsymbol{w},a,b,\alpha;\boldsymbol{x})] , \end{aligned}$$

and

$$\begin{aligned} F_\text{PN}(\boldsymbol{w},a,b,\alpha;\boldsymbol{x}) &= -\frac{n_\text{P} n_\text{N}}{n_\text{P}+n_\text{N}}\alpha^2 \\ &+ n_\text{N}\mathbb{I}_{[\boldsymbol{x}\in\mathcal{X}_\text{P}]}\left((\boldsymbol{w}^\top\boldsymbol{x}-a)^2 - 2(1+\alpha)\boldsymbol{w}^\top\boldsymbol{x}\right) \\ &+ n_\text{P}\mathbb{I}_{[\boldsymbol{x}\in\mathcal{X}_\text{N}]}\left((\boldsymbol{w}^\top\boldsymbol{x}-b)^2 + 2(1+\alpha)\boldsymbol{w}^\top\boldsymbol{x}\right) , \\ F_\text{PU}(\boldsymbol{w},a,b,\alpha;\boldsymbol{x}) &= -\frac{n_\text{P} n_\text{U}}{n_\text{P}+n_\text{U}}\alpha^2 \\ &+ n_\text{U}\mathbb{I}_{[\boldsymbol{x}\in\mathcal{X}_\text{P}]}\left((\boldsymbol{w}^\top\boldsymbol{x}-a)^2 - 2(1+\alpha)\boldsymbol{w}^\top\boldsymbol{x}\right) \\ &+ n_\text{P}\mathbb{I}_{[\boldsymbol{x}\in\mathcal{X}_\text{U}]}\left((\boldsymbol{w}^\top\boldsymbol{x}-b)^2 + 2(1+\alpha)\boldsymbol{w}^\top\boldsymbol{x}\right) , \\ F_\text{NU}(\boldsymbol{w},a,b,\alpha;\boldsymbol{x}) &= -\frac{n_\text{U} n_\text{N}}{n_\text{U}+n_\text{N}}\alpha^2 \\ &+ n_\text{N}\mathbb{I}_{[\boldsymbol{x}\in\mathcal{X}_\text{U}]}\left((\boldsymbol{w}^\top\boldsymbol{x}-a)^2 - 2(1+\alpha)\boldsymbol{w}^\top\boldsymbol{x}\right) \\ &+ n_\text{U}\mathbb{I}_{[\boldsymbol{x}\in\mathcal{X}_\text{N}]}\left((\boldsymbol{w}^\top\boldsymbol{x}-b)^2 + 2(1+\alpha)\boldsymbol{w}^\top\boldsymbol{x}\right) . \end{aligned}$$

Next, we prove the equivalency of Eq. (4) and Eq. (7) and then give the algorithm to solve the Eq. (7).

The proof of the equivalency of Eq. (4) and Eq. (7) is based on [Ying *et al.*, 2016]. We first prove that $R_{\mathrm{PN}} = \min_{||\boldsymbol{w}||<R,a,b} \max_{\alpha} \mathbb{E}_{\boldsymbol{x}}[F_{\mathrm{PN}}(\boldsymbol{w}, a, b, \alpha; \boldsymbol{x})]$. The AUC risk R_{PN} can be rewritten as:

$$\begin{aligned}
R_{\mathrm{PN}} &= \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}} \mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}} \ell(\boldsymbol{w}^\top(\boldsymbol{x}-\boldsymbol{x}')) \\
&= \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[(\boldsymbol{w}^\top\boldsymbol{x})^2] + \mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[(\boldsymbol{w}^\top\boldsymbol{x}')^2] - \\
&\quad 2\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}] + 2\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'] - \\
&\quad 2\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}]\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'] + 1 \\
&= 1 + (\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[(\boldsymbol{w}^\top\boldsymbol{x})^2] - (\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}])^2) + \\
&\quad (\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[(\boldsymbol{w}^\top\boldsymbol{x}')^2] - (\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'])^2) - \\
&\quad 2\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}] + 2\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'] + \\
&\quad (\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}] - \mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'])^2 .
\end{aligned}$$

Notice that

$$\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[(\boldsymbol{w}^\top\boldsymbol{x})^2] - (\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}])^2 = \min_a \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[(\boldsymbol{w}^\top\boldsymbol{x}-a)^2] ,$$

$$\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[(\boldsymbol{w}^\top\boldsymbol{x}')^2] - (\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'])^2 = \min_b \mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[(\boldsymbol{w}^\top\boldsymbol{x}'-b)^2] ,$$

where the minimization is achieved by $a = \mathbb{E}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}]$, and $b = \mathbb{E}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}']$. Also we have

$$\begin{aligned}
&(\mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}] - \mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'])^2 = \\
&\qquad \max_\alpha 2\alpha(\mathop{\mathbb{E}}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'] - \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}]) - \alpha^2 ,
\end{aligned}$$

the maximization is achieved by $\alpha = \mathbb{E}_{\boldsymbol{x}'\in\mathcal{X}_{\mathrm{N}}}[\boldsymbol{w}^\top\boldsymbol{x}'] - \mathbb{E}_{\boldsymbol{x}\in\mathcal{X}_{\mathrm{P}}}[\boldsymbol{w}^\top\boldsymbol{x}]$. Above all, we have

$$R_{\mathrm{PN}} = 1 + \min_{\substack{||\boldsymbol{w}||<R,\\ a,b}} \max_\alpha \mathop{\mathbb{E}}_{\boldsymbol{x}}[F_{\mathrm{PN}}(\boldsymbol{w}, a, b, \alpha; \boldsymbol{x})] . \tag{8}$$

With a similar proof, we have

$$R_{\mathrm{PU}} = 1 + \min_{\substack{||\boldsymbol{w}||<R,\\ a,b}} \max_\alpha \mathop{\mathbb{E}}_{\boldsymbol{x}}[F_{\mathrm{PU}}(\boldsymbol{w}, a, b, \alpha; \boldsymbol{x})] , \tag{9}$$

$$R_{\mathrm{NU}} = 1 + \min_{\substack{||\boldsymbol{w}||<R,\\ a,b}} \max_\alpha \mathop{\mathbb{E}}_{\boldsymbol{x}}[F_{\mathrm{NU}}(\boldsymbol{w}, a, b, \alpha; \boldsymbol{x})] , \tag{10}$$

and thus Eq. (4) equivalent to Eq. (7).

Since function f_{PN}, f_{PU}, and f_{NU} is convex in primal variables $(\boldsymbol{w}, a, b)$ and concave in dual variable α, Eq. (7) can be solved by gradient descent in $(\boldsymbol{w}, a_1, b_1, a_2, b_2, a_3, b_3)$ and gradient ascent in $(\alpha_1, \alpha_2, \alpha_3)$. Instead of using a full gradient of f_{PN}, f_{PU}, and f_{NU}, we use the gradient of F_{PN}, F_{PU}, and F_{NU} as an unbiased gradient estimator to update the model with each instance, which can be calculated on every single instance.

Each labeled or unlabeled instance effects two of F_{PN}, F_{PU}, and F_{NU}. Thus, at each iteration, the gradient of two functions should be computed, gradient descent in the primal variables and gradient ascent in the dual variables. The update rule for the model weight $\boldsymbol{w}$ is:

$$\begin{aligned}
\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} &- \eta^{(t)}\gamma \frac{\partial F_{\mathrm{PN}}(x^{(t)})}{\partial \boldsymbol{w}^{(t)}} \\
&- \eta^{(t)}(1-\gamma)(\frac{\partial F_{\mathrm{PU}}(x^{(t)})}{\partial \boldsymbol{w}^{(t)}} + \frac{\partial F_{\mathrm{NU}}(x^{(t)})}{\partial \boldsymbol{w}^{(t)}}) ,
\end{aligned} \tag{11}$$

For other optimization variables, the update rule for F_{PN} is:

$$(a_1^{(t+1)}, b_1^{(t+1)}) \leftarrow (a_1^{(t)}, b_1^{(t)}) - \eta^{(t)}\gamma \frac{\partial F_{\mathrm{PN}}(x^{(t)})}{\partial (a_1^{(t)}, b_1^{(t)})} , \tag{12}$$

$$\alpha_1^{(t+1)} \leftarrow \alpha_1^{(t)} + \eta^{(t)}\gamma \frac{\partial F_{\mathrm{PN}}}{\partial \alpha_1^{(t)}} . \tag{13}$$

The update rule for F_{PU} is:

$$(a_2^{(t+1)}, b_2^{(t+1)}) \leftarrow (a_2^{(t)}, b_2^{(t)}) - \eta^{(t)}(1-\gamma) \frac{\partial F_{\mathrm{PU}}(x^{(t)})}{\partial (a_2^{(t)}, b_2^{(t)})} , \tag{14}$$

$$\alpha_2^{(t+1)} \leftarrow \alpha_2^{(t)} + \eta^{(t)}(1-\gamma) \frac{\partial F_{\mathrm{PU}}}{\partial \alpha_2^{(t)}} . \tag{15}$$

And the update rule for F_{NU} is:

$$(a_3^{(t+1)}, b_3^{(t+1)}) \leftarrow (a_3^{(t)}, b_3^{(t)}) - \eta^{(t)}(1-\gamma) \frac{\partial F_{\mathrm{NU}}(x^{(t)})}{\partial (a_3^{(t)}, b_3^{(t)})} , \tag{16}$$

$$\alpha_3^{(t+1)} \leftarrow \alpha_3^{(t)} + \eta^{(t)}(1-\gamma) \frac{\partial F_{\mathrm{NU}}}{\partial \alpha_3^{(t)}} . \tag{17}$$

The procedure of the algorithm is shown in Algorithm 1. It is noteworthy that even if one of the three types (i.e., positive, negative, and unlabeled) of data is missing, SOLA can still work and learn meaningful models, since it is updated over single instances.

Algorithm 1 SOLA

1: Initialize $t \leftarrow 0, n_{\mathrm{P}}^{(0)} \leftarrow 0, n_{\mathrm{N}}^{(0)} \leftarrow 0, n_{\mathrm{U}}^{(0)} \leftarrow 0$
2: Initialize optimization variables at random
3: **for** each $\boldsymbol{x}^{(t+1)}$ available **do**
4:     **if** $\boldsymbol{x}^{(t+1)}$ is a positive labeled instance **then**
5:         $n_{\mathrm{P}}^{(t+1)} \leftarrow n_{\mathrm{P}}^{(t)} + 1$
6:         Update (a_1, b_1, α_1) by Eq. (12) and Eq. (13)
7:         Update (a_2, b_2, α_2) by Eq. (14) and Eq. (15)
8:     **if** $\boldsymbol{x}^{(t+1)}$ is a negative labeled instance **then**
9:         $n_{\mathrm{N}}^{(t+1)} \leftarrow n_{\mathrm{N}}^{(t)} + 1$
10:        Update (a_1, b_1, α_1) by Eq. (12) and Eq. (13)
11:        Update (a_3, b_3, α_3) by Eq. (16) and Eq. (17)
12:    **if** $\boldsymbol{x}^{(t+1)}$ is an unlabeled instance **then**
13:        $n_{\mathrm{U}}^{(t+1)} \leftarrow n_{\mathrm{U}}^{(t)} + 1$
14:        Update (a_2, b_2, α_2) by Eq. (14) and Eq. (15)
15:        Update (a_3, b_3, α_3) by Eq. (16) and Eq. (17)
16:    Calculate $\boldsymbol{w}^{(t+1)}$ by Eq. (11)
17:    **if** $||\boldsymbol{w}^{(t+1)}|| > R$ **then**
18:        $\boldsymbol{w}^{(t+1)} \leftarrow R\boldsymbol{w}^{(t+1)}/||\boldsymbol{w}^{(t+1)}||$
19:    $t \leftarrow t + 1$

4 Experiments

4.1 Datasets

To conduct continuous build outcome prediction, two source of data should be used: continuous integration systems and version control systems. The former provides build results and the other build information with corresponding commit ID, while we can obtain the information about the project, code changes and developers from the latter. Madeyski and Kawalerowicz [2017] proposed a dataset for continuous build outcome prediction, which consists of 1265 open source projects using CI and being hosted on GitHub. Those projects use CI services provided by three mainstream CI platforms: Jenkins, Travis CI, and TeamCity. Besides the build records with clear outcome label, i.e., success and failure, all of the three platforms provide unclear build outcome labels including "not_build", "aborted", "unstable" on Jenkins, "errored" on Travis CI, and "error", "warning", "unknown" on TeamCity. Build records with unclear labels are marked as "unknown" in the dataset, which can be regarded as unlabeled instances during learning process.

The data of different project varies significantly in terms of the total number of build records, the percentage of labeled records, and the ratio of successful build to failed build. To study the performance of the methods in different cases, we choose 8 projects with different imbalanced ratio and size to evaluate our method and compared methods. Projects with less than 200 labeled records are not chosen since we have to hold some labeled records out for evaluation. The statistics of the projects are shown in Table 1. We use the features provided in [Madeyski and Kawalerowicz, 2017], as well as some other features computed from the commit log. Features that irrelevant to the learning like commit ID and build time are removed. Table 2 shows the features used to build the model.

Project Name	#Success	#Failure	#Unknown	S:F
deeplearning4j	5	426	422	0.01
capybara	173	185	106	0.94
killbill	841	332	790	2.53
rails	8,048	2,792	805	2.88
codetriage	242	32	15	7.56
oryx	346	40	31	8.65
phony	332	29	24	11.45
stringer	293	9	9	32.56

Table 1: Statistics of the projects. For each project, the numbers of successful, failed, and unknown build records are shown. S:F refers to the ratio of successful builds to failed builds.

Abbr.	Feature
LBO	Last Build Outcome
NR	Number of Revisions
NRC	Number of Revised Code Files
NML	Number of Modified Lines
NMLC	Number of Modified Lines in Code Files
NDC	Number of Distinct Committers
NC	Number of Commits

Table 2: Used features of each build record.

4.2 Compared Methods

We compare our method with a number of competing baseline methods to demonstrate that it is necessary to handle the four challenges simultaneously. The compared methods are:

- Hoeffding Tree [Finlay *et al.*, 2014]: a Hoeffding Tree based method for handling streaming data of software build outcome. This method is capable for handling streaming data, but *fails to handle the other three challenges*.
- OMR [Goldberg *et al.*, 2008]: a semi-supervised method that learns from sequential labeled and unlabeled data, by conducting online manifold regularization. It *does not take the imbalance of data into consideration*, which may harm the performance.
- SOLAM [Ying *et al.*, 2016]: an online AUC optimization method. By leveraging AUC optimization, SOLAM can also handle the imbalanced data, as well as providing the suspiciousness of the build events. However, it *cannot exploit the unlabeled data*.
- SAMULT [Xie and Li, 2018]: a batch semi-supervised AUC optimization method. Although it *cannot handle the streaming data* and can be ineffective in practice, we use it as an upper bound reference of our method in the experiments.

4.3 Performance

We evaluate the performance of compared methods. For each project, the last 100 labeled instances are used as test set, and the instances before the 100th last labeled instance are used as training data. The unlabeled instances after the 100th last labeled instance are ignored. We use the AUC as performance measure, which reflects the ranking ability of the methods. The performance of compared methods are shown in Table 3.

Compared with the method designed for build outcome prediction. SOLA beats Hoeffding Tree on most of the projects. It is noteworthy that when the data is highly imbalanced, e.g., *deeplearning4j*, *phony*, *stringer*, and so on, Hoeffding Tree fails to learn meaningful models, which leads to a 0.5 AUC score or lower. When the dataset is relatively balanced, Hoeffding Tree can learn a reasonable model from relative sufficient positive and negative data. However, by optimizing AUC, SOLA can learn meaningful models whether the data is balanced or not. SOLA achieves 42.1% higher AUC than Hoeffding Tree on average.

Compared with the method without imbalance handling. OMR is not designed for imbalance case either, so it fails to generate meaningful output when the data is highly imbalanced, just like Hoeffding tree. Only on the relative balanced projects like *capybara* and *killbill*, OMR can learn a reasonable model, but the performance is still poor. SOLA achieves

Methods	Online Methods				Batch Method
	Hoeffding Tree	OMR	SOLAM	SOLA	SAMULT
	—	Semi.	AUC-Opt.	AUC-Opt./Semi.	AUC-Opt./Semi.
deeplearning4j	0.429	0.500†	**0.832**	**0.832**	**0.848**
capybara	0.612	0.604	**0.718**	**0.727**	0.658
killbill	**0.692**	0.686	0.688	0.688	**0.710**
rails	0.531	0.529	**0.611**	**0.616**	**0.627**
codetriage	0.640	0.495	0.648	**0.707**	**0.704**
oryx	0.500†	0.500†	0.665	**0.737**	**0.757**
phony	0.500†	0.500†	0.808	**0.980**	**0.979**
stringer	0.500†	0.500†	0.954	**0.975**	**0.975**
Average	0.551	0.539	0.741	0.783	0.782

Table 3: Average AUC of compared methods on different projects. The boldfaces denote the best or comparable methods on each dataset, according to pairwise t-test at the significance level 5%. The daggers (†) denote the methods that fail to output any meaningful predictions by classifying all instances into one class or assigning them with a same score.

45.3% higher AUC than Hoeffding Tree on average. Comparing OMR and Hoeffding Tree to SOLA, we can conclude that AUC optimization is crucial for learning models from imbalanced data.

Compared with the method without exploiting unlabeled data. Since SOLAM handles imbalance by optimizing AUC, it can learn meaningful models in all the projects. SOLA outperforms SOLAM in most of the case due to the unlabeled data. Note that SOLAM achieves comparable performance to SOLA on *deeplearning4j*, *killbill*, and *rails*, the reason might be there are enough labeled data for learning the model. However, by exploiting the unlabeled data, SOLA achieves 5.7% improvement than SOLAM on average.

Compared with the batch method. SAMULT is a batch method, which can be regarded as a upper bound of proposed online method. As shown in Table 3, SOLA achieves similar performance to SAMULT, which means that SOLA learns well from sequential data by just processing each instance once. However, by leveraging an online updating algorithm, the running time of SOLA is significantly reduced, which is discussed in the next subsection.

In summary, the experimental results show that SOLA outperforms the existing online method by *handling imbalanced data*, *learning suspiciousness of the events*, and *exploit the unlabeled data*. SOLA also saves significant amount of time by *using an online algorithm*. Such results suggest that addressing four challenges simultaneously in CI build outcome prediction is beneficial.

4.4 Running Time

To study the running time reduced by adapting the semi-supervised AUC optimization method into an online situation, we evaluate the running time of SOLA and other competitive methods. We update the model every time a new instance arrives, and measure the time of model updating. The

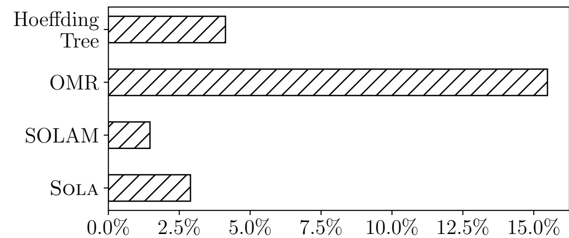

Figure 2: Average relative model updating time of the online models w.r.t. the batch method, SAMULT, on *rails* dataset.

updating process is repeated 100 times to eliminates the impact of the randomness. We record the average relative model updating time of each method w.r.t. the batch method, SAMULT, to show how much time is saved by formulating the continuous build outcome prediction as an online problem. Due to the space limitation, we only report the results on the largest dataset, *rails*, in Figure 2. Similar trends can be find on the other datasets.

It can be observed in Figure 2 that SOLA saves more than 97% of the time during model updating, compared with the batch method SAMULT. Only SOLAM achieves a slightly better time consumption than SOLA, since SOLAM simply dismisses the unlabeled data while SOLA uses them to update the model. By adopting an online algorithm, much time for model updating is saved.

4.5 Threats to Validity

We have only evaluated our method on open-source projects, and only 8 projects are used. Evaluation on more projects, including closed-source projects may lead to more reliable results. Also, the projects are all Java projects. In the future, we plan to reduce these threats by experimenting on more software projects.

5 Conclusion

Continuous build outcome prediction can be applied to cut the effort in Continuous Integration. In this paper, we ar-

gue that four challenges, including streaming data, few build outcome label, imbalance in build outcomes, and suspiciousness of build event required, should be properly handled. We propose SOLA, a semi-supervised online AUC optimization method, to address the continuous build outcome prediction problem by handling the four challenges simultaneously. Experiments show that SOLA can reduce the building efforts by accurately identifying the successful and failed builds, and outperforms the existing methods in terms of performance and running time.

It is worth mentioning that SOLA is not closely coupled with the features used in this paper. By using a better group of features or features learned by deep models, the performance may be further improved, which may be further explored in the future. Besides, SOLA may be adapted to other tasks with similar settings, such as just-in-time defect prediction.

References

[Amini *et al.*, 2008] Massih Reza Amini, Tuong Vinh Truong, and Cyril Goutte. A boosting algorithm for learning bipartite ranking functions with partially labeled data. In *Proceedings of the Thirty-First Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 99–106, 2008.

[Booch, 1991] Grady Booch. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, California, 1991.

[Campos *et al.*, 2014] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the Twenty-Ninth ACM/IEEE International Conference on Automated Software Engineering*, pages 55–66, New York, New York, 2014.

[Duvall *et al.*, 2007] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[Finlay *et al.*, 2014] Jacqui Finlay, Russel Pears, and Andy M. Connor. Data stream mining for predicting software build outcomes using source code metrics. *Information and Software Technology*, 56(2):183 – 198, 2014.

[Fujino and Ueda, 2016] Akinori Fujino and Naonori Ueda. A semi-supervised AUC optimization method with generative models. In *IEEE Sixteenth International Conference on Data Mining*, pages 883–888, 2016.

[Gao and Zhou, 2015] Wei Gao and Zhi-Hua Zhou. On the consistency of AUC pairwise optimization. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 939–945, 2015.

[Gao *et al.*, 2013] Wei Gao, Rong Jin, Shenghuo Zhu, and Zhi-Hua Zhou. One-pass AUC optimization. In *Proceedings of the Thirtieth International Conference on Machine Learning*, volume 28, pages 906–914, 2013.

[Goldberg *et al.*, 2008] Andrew B. Goldberg, Ming Li, and Xiaojin Zhu. Online manifold regularization: A new learning setting and empirical study. In *Machine Learning and Knowledge Discovery in Databases*, pages 393–407, Berlin, Heidelberg, 2008.

[Hassan and Zhang, 2006] Ahmed E. Hassan and Ken Zhang. Using decision trees to predict the certification result of a build. In *Proceedings of the Twenty-First IEEE/ACM International Conference on Automated Software Engineering*, pages 189–198, Washington, DC, 2006.

[Hilton *et al.*, 2016] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the Thiry-First IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437, New York, New York, 2016.

[Kamei *et al.*, 2016] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[Madeyski and Kawalerowicz, 2017] Lech Madeyski and Marcin Kawalerowicz. Continuous defect prediction: The idea and a related dataset. In *Proceedings of the Fourteenth International Conference on Mining Software Repositories*, pages 515–518, Piscataway, New Jersey, 2017.

[Ni and Li, 2017] Ansong Ni and Ming Li. Cost-effective build outcome prediction using cascaded classifiers. In *Proceedings of the Fourteenth International Conference on Mining Software Repositories*, pages 455–458, Piscataway, New Jersey, 2017.

[Sakai *et al.*, 2018] Tomoya Sakai, Gang Niu, and Masashi Sugiyama. Semi-supervised AUC optimization based on positive-unlabeled learning. *Machine Learning*, 107(4):767–794, 2018.

[Xie and Li, 2018] Zheng Xie and Ming Li. Semi-supervised AUC optimization without guessing labels of unlabeled data. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[Yang *et al.*, 2017] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206 – 220, 2017.

[Ying *et al.*, 2016] Yiming Ying, Longyin Wen, and Siwei Lyu. Stochastic online AUC maximization. In *Advances in Neural Information Processing Systems 29*, pages 451–459. Curran Associates, Inc., 2016.

[Zhao *et al.*, 2011] Peilin Zhao, Steven C. H. Hoi, Rong Jin, and Tianbao Yang. Online AUC maximization. In *Proceedings of the Twenty-Eighth International Conference on International Conference on Machine Learning*, pages 233–240, 2011.