The Journal of Systems and Software 214 (2024) 112077

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Cross-project concurrency bug prediction using domain-adversarial neural network[☆]

Fangyun Qin[a,b], Zheng Zheng[c,*], Yulei Sui[d], Siqian Gong[e,b], Zhiping Shi[a], Kishor S. Trivedi[f]

[a] College of Information Engineering, Capital Normal University, Beijing, 100048, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, PR China
[c] School of Automation Science and Electrical Engineering, Beihang University, Beijing, 100191, China
[d] School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
[e] School of Mechanical, Electronic and Control Engineering, Beijing Jiaotong University, 100044, China
[f] Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA

ARTICLE INFO

Keywords:
Software bug prediction
Cross-project bug prediction
Concurrency bug
Domain adaptation

ABSTRACT

In recent years, software bug prediction has shown to be effective in narrowing down the potential bug modules and boosting the efficiency and precision of existing testing and analysis tools. However, due to its non-deterministic nature and low presence, concurrency bug labeling is a challenging task, which limits the implementation of within-project concurrency bug prediction. This paper proposes DACon, a Domain-Adversarial neural network-based cross-project Concurrency bug prediction approach to tackle this problem by leveraging information from another related project. By combining a set of designed concurrency code metrics with widely used sequential code metrics, DACon uses SMOTE (Synthetic Minority Over-sampling TEchnique) and domain-adversarial neural network to mitigate two challenges including the severe class imbalance between concurrency bug-prone samples and concurrency bug-free samples, and shift between source and target distribution during bug prediction implementation. Our evaluation on 20 pair-wise groups of experiments constructed from 5 real-world projects indicates that cross-project concurrency bug prediction is feasible, and DACon can effectively predict concurrency bugs across different projects.

1. Introduction

In the multicore era, concurrent programming, an effective way of utilizing computation resources, is more important than ever. However, concurrency bugs, which are hard to diagnose and fix due to their non-deterministic nature (Yu et al., 2019; Chandra and Chen, 2000), pose a big threat to concurrent programming. They can only be triggered under specific thread interleavings. Compared with sequential bugs, concurrency bugs take twice as long to be fixed, 4 times more patches, 17% more developers involved, and 17% higher severity (Zhou et al., 2015). One third of concurrency bugs in production software can cause serious failure effects (e.g., crash or hang) (Fonseca et al., 2010). Therefore, accurately predicting and detecting concurrency bugs in the early stage of software development cycle is critically important.

Background. Software bug prediction is the process of developing models to predict source files or code regions that may contain bugs to improve the overall software quality before a software product is deployed (Qu et al., 2021; Herbold, 2019; Qu and Yin, 2021; Chakraborty and Chakraborty, 2020). Bug prediction can also guide later testing and verification by boosting their performance and precision given limited availability of testing resources and budgets. A wide variety of machine learning techniques have been used for predicting bugs in sequential programs (Di Nucci et al., 2018; Wu et al., 2018; Malhotra, 2015). However, only limited efforts have been made to predict concurrency bugs due to the complicated features of concurrent programs and scarcity of ground truth samples. In 2013, Carrozza et al. (2013) investigated Mandelbug (Qin et al., 2017; Cotroneo et al., 2013a) (a larger category that includes concurrency bug) prediction in an industrial software system. Then Yu et al. (2019) proposed an approach named Conpredictor to perform concurrency bug prediction in the source code level. These two studies are mainly related to or focused on within-project concurrency bug prediction. Like all the existing within-project bug prediction models, within-project concurrency bug prediction works well if a large number of training data are available. However, in practice, training data for concurrency bug prediction are

☆ Editor: Dr Gabriele Bavota.
* Corresponding author.

E-mail addresses: fyqin@cnu.edu.cn (F. Qin), zhengz@buaa.edu.cn (Z. Zheng), y.sui@unsw.edu.au (Y. Sui), gongsq@bjtu.edu.cn (S. Gong), shizp@cnu.edu.cn (Z. Shi), ktrivedi@duke.edu (K.S. Trivedi).

https://doi.org/10.1016/j.jss.2024.112077
Received 3 April 2023; Received in revised form 7 April 2024; Accepted 22 April 2024
Available online 25 April 2024
0164-1212/© 2024 Elsevier Inc. All rights reserved.

difficult to collect. First, due to its non-deterministic nature (Musuvathi and Qadeer, 2007; Asadollah et al., 2017; Liu et al., 2014; Taylor et al., 1992), concurrency bug labeling is a very challenging task that requires intensive knowledge. Second, only a few concurrency bugs are scattered across a large codebase (Asadollah et al., 2017; Cotroneo et al., 2016). Therefore, a large amount of time and effort is wasted in analyzing the non-concurrency code. Third, training data may not be available for the projects in the initial development phases or without archived historical data. This paper focuses on cross-project concurrency bug prediction in real-world software projects by utilizing labeled data from other project (i.e., source project) to predict concurrency bugs in the particular project (i.e., target project). Our aim is to predict the existence of concurrency bugs by identifying the source code files that likely contain bugs. The results can be utilized for programmers to narrow down the failures and choose the most suitable testing tools, or used as a complementary tool to boost the performance and precision of existing concurrency bug detectors.

Challenges and Insights. Accurate cross-project concurrency bug prediction is challenging. First, a prediction classifier learned from one project might not be general enough for another project. The main reason comes from the shift between source and target distributions (Briand et al., 2002; Li et al., 2012; Zhang et al., 2013). Second, the number of concurrency bugs only consists of a small fraction of the total bugs reported in real-world projects (e.g., 7.56% in MySQL and 5.26% in Apache (Cotroneo et al., 2016)), resulting in a severe class imbalance which has a negative impact on the performance of prediction classifier (Song et al., 2019; Tantithamthavorn et al., 2020; Gong et al., 2020). Yu et al. (2019) also validated the prediction performance of ConPredictor in the cross-project setting, but the prediction performance was moderately effective since they did not consider the distribution shift between source and target projects.

Our key insight for cross-project concurrency bug prediction is that although different projects have their own properties (e.g., varying developer experiences and user requirements), concurrency bugs across different projects are alike and predictable. They share some common features including both thread-aware features (e.g., code patterns/metrics by considering threads) and thread-unaware ones (e.g., code patterns/metrics without considering threads). Moreover, programmers tend to make similar mistakes when writing concurrent programs across different projects. If the shift between source and target distributions is mitigated, (i.e., a good representation is found so that the source and target projects have the same or similar distributions), the prediction classifier learned from the representation of the source project can be applicable to the representation of the target project. By exploiting this good representation, we can capture the association between program features and concurrency bugs across different projects for accurate cross-project concurrency bug prediction.

Our Solution. To address the above challenges, we propose DACon, a Domain-Adversarial neural network (Ganin et al., 2016) based cross-project Concurrency bug prediction approach. Specifically, we first propose a set of concurrency code metrics specific to concurrent programs by taking basic concurrent programming into account. Based on these metrics, DACon uses SMOTE (Synthetic Minority Over-sampling TEchnique) (Chawla et al., 2002) to create more concurrency bug-related samples to alleviate the class imbalance problem in the source project. Then we train a domain-adversarial neural network which combines distribution shift mitigation and concurrency bug discrimination. Specifically, the domain-adversarial neural network strives to find a good representation for samples in source and target project which cannot discriminate the origin of samples (source or target project) while preserving a good concurrency bug prediction performance on samples in the source project.

To evaluate the feasibility and effectiveness of DACon, we perform experiments with five real-world projects that include 20 pair-wise group cases. The results show that DACon can effectively predict concurrency bugs across different projects. In addition, it significantly improves the prediction performance when compared with (1) baseline models, (2) models using class imbalance or distribution shift mitigation separately, and (3) models with sequential code metrics. Furthermore, the comparison between DACon and the within-project setting demonstrates that DACon achieves comparable performance as the within-project setting.

The contributions of this work are the following:

- We present DACon, a Domain-Adversarial neural network-based cross-project Concurrency bug prediction approach to address the cross-project concurrency bug prediction problem. To the best of our knowledge, this is the first approach aiming at cross-project concurrency bug prediction.
- A set of concurrency code metrics is proposed by considering the comprehensive features of concurrent programs. The proposed fine-grained set of metrics capture the commonly used concurrent programming utilities and patterns.
- We have implemented our approach and conducted 20 pair-wise groups of experiments on five real-world projects to demonstrate its effectiveness. The experimental results show that DACon has the potential to get a good prediction performance in cross-project bug prediction setting.

The rest of the paper is organized as follows. We begin with the approach in Section 2. Section 3 presents the experimental setup and Section 4 demonstrates the experiment implementation and results. Section 5 discusses the metrics and results compared with previous studies. Section 6 identifies threats to validity. After discussing related work in Section 7, we wrap up with conclusion and future work in Section 8.

2. Approach

In this section, we first present the overall framework of Domain-Adversarial neural network-based cross-project Concurrency bug prediction approach (DACon) and then elaborate on each part of the proposed framework.

2.1. Overall framework

Fig. 1 illustrates the overview framework of the DACon. This approach mainly consists of four parts: metric extraction, data preprocessing, domain-adversarial neural network training and target label prediction.

In the first step, we extract both sequential and concurrency code metrics to represent each source code file with numerical values. Here, we mainly introduce the proposed concurrency code metrics which are able to show the characteristics of concurrent programming better. The sequential code metrics will be introduced in Section 3.2 in detail. In the second step, we perform data preprocessing which includes z-score normalization and class imbalance mitigation method SMOTE (Chawla et al., 2002). In detail, we first conduct z-score normalization to give all metrics in each dataset an equal weight, in order to reduce the effect of heterogeneity during model training. Then we split the normalized source project dataset into 3 folds where 2 folds are used as training set and 1 fold is used as validation set. Since the severe class imbalance between concurrency bug-prone samples and concurrency bug-free samples has a negative effect on concurrency bug prediction, we adopt widely used SMOTE (Chawla et al., 2002) to manually generate concurrency bug-prone samples to alleviate the influence of class imbalance. In the third step, we use all the normalized labeled training data in the source project and normalized unlabeled data in the target project to train the Domain-Adversarial Neural Network (DANN) (Ganin et al., 2016) whose internal representation is discriminative for the concurrency bug classification task on the source project and indiscriminative with respect to the shift between source project and target project distributions. Then we choose the optimal hyperparameter based on the prediction performance in the validation set. In the final step, we feed the normalized target project data into the learned DANN network and get their predicted label.

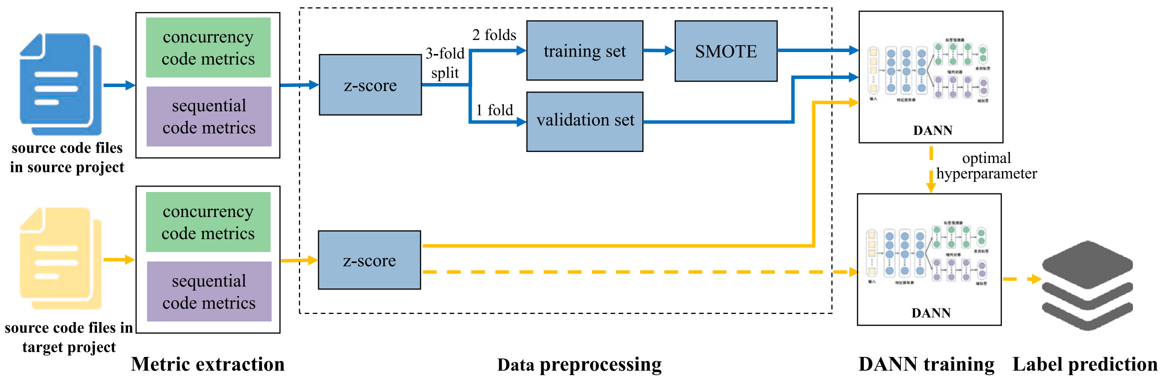

Fig. 1. The overview framework of DACon.

2.2. *Concurrency code metrics*

Concurrency bugs are hard to diagnose and fix due to their non-deterministic nature (Yu et al., 2019; Chandra and Chen, 2000). They can only be triggered under specific thread interleavings. Undoubtedly, the development of concurrent programs relies on the concurrent programming utilities provided by the programming language. Based on the investigation of the characteristics of concurrent programming in Java (Java, 0000; Bradbury et al., 2006; Lea, 2000; Goetz et al., 2006), we propose eight suites of concurrency code metrics to have a high potential of capturing concurrent programming, as summarized in Table 1. Note that these metrics are tailored to the Java language, which is adopted in the software products studied in this paper, although they can potentially be extended to other programming languages such as C++.

Package level metrics (M1–M4) metric suite reflects the complexity of concurrent programming from the number of instances or methods in the four concurrent packages provided in the Java platform. The Java platform provides a powerful, extensible framework for high-performance threading by utilizing packages of concurrent programming with both low-level primitives and advanced programming patterns. The four packages are listed and introduced as follows. (1) `java.util.concurrent` package provides utility classes commonly used in concurrent programming. It offers a few small standardized extensible frameworks, as well as some classes that provide useful functionality. (2) `java.util.concurrent.locks` package provides a framework for locking and waiting for conditions that differ from built-in synchronization and monitors. It gives much greater flexibility during the use of locks and conditions, at the expense of more awkward syntax. (3) `java.util.concurrent.atomic` package includes a number of atomic classes that support lock-free thread-safe programming on single variables. (4) `Collections.SynchronizedXxx` package offers synchronization wrappers that are used to add synchronization to some unsynchronized collections. Basically, the more instances or methods in these packages involved in concurrent programming, the higher the chance to have a complicated concurrent programming structure.

Thread (M5–M12) metric suite consists of a list of metrics that are related to thread and can affect the status of a thread. Metrics include the number of class that inherits `Thread` class or implements the `Runnable` or `Callable` interface, and the number of times `start`, `sleep`, `yield`, `join`, `wait`, `notify` and `notifyAll` method called in each file respectively because these classes are related with the content of threads, and these methods have influence on the status of a thread. Java concurrency is built based on the notion of multi-threaded programs, and the state of the thread can affect interleavings and further has a close relationship with concurrency bugs.

Synchronization (M13) metric suite captures the number of times the `synchronized` keyword are used, including not only the apparently `synchronized` keyword used times, but also the number of called methods recursively containing the `synchronized` keyword. The `synchronized` keyword is a widely used built-in keyword for concurrency and can be used to synchronize both methods and statements. With the more frequent usage of the keyword `synchronized`, the program is more likely to contain concurrency bugs. Based on our practical experience, the object that is synchronized using the `synchronized` keyword is not always obviously observed from the code, and it is often hidden in the callee methods.

Explicit lock (M14–M18) metric suite takes into account the concurrent programming complexity resulting from the explicit lock, i.e., the number of times some important methods like `lock`, `unlock`, `await`, `signal` and `signalAll` are used in a file. The explicit lock provides the same semantics as the implicit monitor locks used by synchronized code. As with implicit monitor locks, only one thread can own a Lock object at a time. Nevertheless, the explicit lock provides additional functionality such as supporting a `wait/notify` mechanism through their associated Condition objects. Basically, the frequent use of these methods makes it harder to track the thread interleaving and guarantee the correctness of the concurrent program. It is not easy to decide when the thread should acquire the lock and properly release the lock.

Synchronizers (M19–M23) metric suite characterizes the usage of synchronizers in concurrent programming, i.e., the number of methods that have a close relationship with resource acquire and release in `Semaphore`, and those control the state of thread in `CountDownLatch` and `Barrier`. Unlike `synchronized` keyword or explicit lock that only enable one thread to visit some resource, these synchronizer utilities allow multiple threads to visit some resource or make a set of threads wait until restrictions are satisfied in other threads or these threads. This gives more flexibility accompanied in concurrent programming, as well as the difficulty in promising the right interleaving of concurrent programming.

Task scheduling framework (M24–M35) metric suite captures the concurrency complexity due to Executors which separate thread management and creation from the rest of the application, i.e., the number of methods that can influence the thread status, including creating a new thread or thread pool, executing or submitting the given task, blocking task, shutting down the task, etc.. In the task scheduling framework, Executor Interface defines three executor object types, including `Executor`, `ExecutorService` and `ScheduledExecutorService`. Thread Pools are the most common kind of executor implementation. According to previous analyses, the methods in these classes or interfaces make concurrent programming more complicated and can increase the risk of concurrency bugs.

Legacy implementations (M36–M37) metric suite includes the number of instances of `Vector` or `Hashtable` class. These two classes are legacy implementations of the Java Collections Framework

Table 1
Proposed concurrency code metrics.

Metric suite	ID	Metric name	Metric description
Package level metrics	M1	Con	# of instances that inherit or implement the class or interface in the `java.util.concurrent` package.
	M2	ConLock	# of instances that inherit or implement the class or interface in the `java.util.concurrent.locks` package.
	M3	Atomic	# of variables that belong to the class in the `java.util.concurrent.atomic` package.
	M4	CollSyn	# of methods in `Collections.Synchronizedxxx` called in the file.
Thread	M5	DesThread	# of class that inherits `Thread` class or implement the `Runnable` or `Callable` interface.
	M6	Start	# of `start` method of the `Thread` class called in the file.
	M7	Sleep	# of `sleep` method of the `Thread` class called in the file.
	M8	Yield	# of `yield` method of the `Thread` class called in the file.
	M9	Join	# of `join` method of the `Thread` class called in the file.
	M10	Wait	# of `wait` method of the `Thread` class called in the file.
	M11	Notify	# of `notify` method of the `Thread` class called in the file.
	M12	NotifyAll	# of `notifyAll` method of the `Thread` class called in the file.
Synchronization	M13	Syn	# of `synchronized` keyword and methods recursively containing this keyword.
Explicit lock	M14	LockLock	The sum of # of `lock`, `tryLock`, `lockInterruptibly` method of the `Lock` interface and `ReentrantLock` class called in the file.
	M15	LockUnlock	# of `unlock` method of the `Lock` interface and `ReentrantLock` class called in the file.
	M16	ConAwait	The sum of # of `await`, `awiatNanos`, `awaitUninterruptibly`, `awaitUntil` method of the `Condition` class called in the file.
	M17	ConSignal	# of `signal` method of the `Condition` class called in the file.
	M18	ConSignalAll	# of `signalAll` method of the `Condition` class called in the file.
Synchronizers	M19	SemAcquire	Sum of # of `acquire`, `acquireUninterruptibly`, `tryAcquire` method of the `Semaphore` class called in the file.
	M20	SemRelease	# of `release` method of the `Semaphore` class called in the file.
	M21	LatchAwait	# of `await` method of the `CountdownLatch` class called in the file.
	M22	LatchCDown	# of `countDown` method of the `CountdownLatch` class called in the file.
	M23	BarrierAwait	# of `await` method of the `Barrier` class called in the file.
Task scheduling framework	M24	ThrFctNewThr	# of `newThread` method of the `ThreadFactory` interface called in the file.
	M25	Execute	# of method that executes tasks in the taskscheduling framework in the file.
	M26	AwaitTermi	# of `awaitTermination` method in the taskscheduling framework called in the file.
	M27	Shutdown	# of `shutdown` method in the taskscheduling framework called in the file.
	M28	ShutdownNow	# of `shutdownNow` method in the taskscheduling framework called in the file.
	M29	ThrPExecFinal	# of `finalize` method of the `ThreadPoolExecutor` class called in the file.
	M30	StartThread	# of `prestartAllCoreThreads` and `prestartCoreThreads` method of `ThreadPExecutor` class called in the file.
	M31	ThrPExecRej	# of handler for unexecutable of the `ThreadPoolExecutor` class called in the file.
	M32	NewThrPool	# of methods used for creating thread pool in the task scheduling framework called in the file.
	M33	FutureCancel	# of `cancel` method called of the `Future` and `FutureTask` interface.
	M34	FutureGet	# of `get` method called of the `Future` and `FutureTask` interface.
	M35	FutureRun	# of `run` method called of the `RunnableFuture` and `FutureTask` interface.
Legacy implementations	M36	VectorMetric	The sum of # of instances of `Vector` class and class/interface that extends `Vector` class.
	M37	HashTabMetric	The sum of # of instances of `Hashtable` class and class/interface that extends `HashTable` class.
Others	M38	ThreadLocal	# of thread-local variables provided with `ThreadLocal` class in the file.
	M39	Volatile	# of `volatile` keyword used in the file.
	M40	Final	# of `final` keywork used in the file.
	M41	Finally	# of `finally` keyword used in the file.

in Java2. However, when they are used in the parallel accesses, the concurrency problem sometimes occurs. Although there are new built-in concurrent data structures provided in the new java version such as `ConcurrentHashMap`, some developers still use these two old classes without careful consideration and result in concurrency bugs.

Other classes and keywords for concurrent programming (M38–M41) metric suite collects other classes and keywords used in concurrent programming, mainly focusing on `ThreadLocal` class, `volatile`, `final` and `finally` keywords. `ThreadLocal` can give the variable in each thread an initialized copy of the variable's value and ensure the thread safety with more resources. The `volatile` keyword is used to ensure visibility from other tasks/threads after it is written. If a field is declared as `volatile`, any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation. Reader threads must reload the values of volatile fields upon each access. The `final` keyword supports the construction of immutable objects which can sometimes provide a weak form of atomicity. The `finally` keyword is very important in releasing explicit locks. For example, if the `unlock()` method is called in the statements wrapped with `finally`, it can ensure that the `unlock` method is called whether or not an exception is thrown. But if the `unlock()` method is not wrapped by the `finally`, this may cause a suspension concurrency bug.

2.3. *Data preprocessing*

Data preprocessing here mainly refers to z-score normalization and SMOTE class imbalance mitigation method. We will introduce these two techniques in detail in this subsection.

2.3.1. *Z-score normalization*

Since different metrics have different units and thus their range values may vary greatly. We apply z-score normalization, a widely used normalization method in machine learning and deep learning (Rajapaksha et al., 2021; Chen et al., 2020; Hosseini et al., 2019), to rescale the metric values into a specific interval before feeding them to the following models. Let $x_j = \{x_{1j}, x_{2j}, \dots, x_{pj}\}$ be the jth metric values for all the samples in a project where p refers to the number of samples. The corresponding z-score normalized metric value for the ith sample is represented as

$$x'_{ij} = \frac{x_{ij} - mean(x_j)}{std(x_j)}, \tag{1}$$

where $mean(x_j)$ and $std(x_j)$ represent the mean value and standard deviation of the metric vector x_j.

2.3.2. SMOTE

In this subsection, we focus on mitigating the severe class imbalance existed in cross-project concurrency bug prediction. Previous empirical studies showed that the concurrency bug-prone class contains significantly fewer bugs than the concurrency bug-free class in the analyzed projects, such as 7.56% in MySQL and 5.26% in Apache (Cotroneo et al., 2016). In such a scenario, learners will typically pay more attention to the concurrency bug-free class, making it difficult to accurately predict the concurrency bug-prone class which is the class of interest (Song et al., 2019; Tantithamthavorn et al., 2020; Gong et al., 2020; Buda et al., 2018).

Algorithm 1: SMOTE (*ftr*, *label*)

```
Input: Features ftr; Labels label
Output: New features ftr_new; New labels label_new
1  ftr_syn[ ]=synthetic feature array;
2  label_syn[ ]=synthetic label array;
3  Get concurrency bug-prone samples (ftr_con, label_con);
4  Get concurrency bug-free samples (ftr_non_con, label_non_con);
5  Compute sample number difference diff between two classes;
6  for $i \leftarrow 1$ to (int)(diff/2) do
7      Randomly pick a sample from concurrency bug-prone class:
         (ori_ftr, ori_label);
8      for $j \leftarrow 1$ to 2 do
9          Randomly pick a sample from (ori_ftr, ori_label)'s 5
             concurrency bug-prone nearest neighbors, call it (nei_ftr,
             nei_label);
10         Compute: dis=nei_ftr-ori_ftr;
11         Compute: gap=random number in (0,1);
12         Compute: $idx=2*(i-1)+j$;
13         Compute: $ftr\_syn[idx]=ori\_ftr+gap*dis$;
14         Compute: $label\_syn[idx]=ori\_label$;
15     end
16 end
17 Combine ftr_con and ftr_syn as ftr_new;
18 Combine label_con and label_syn as label_new;
19 return ftr_new, label_new;
```

To solve the class imbalance problem, we use SMOTE (Synthetic Minority Over-sampling TEchnique) (Chawla et al., 2002) to manually generate concurrency bug-prone samples along the line segments connecting randomly selected samples in concurrency bug-prone class and any/all of their k concurrency bug-prone class nearest neighbors. Depending on the number of synthetic samples required, neighbors from the k nearest neighbors are randomly chosen. In our paper, we aim to make two classes have the same number of samples, so the number of synthetic concurrency bug-prone samples equals the difference (*diff*) between two class sample numbers. For each iteration, we randomly pick a sample with replacement from the concurrency bug-prone class and two neighbors from its five nearest neighbors, and one synthetic sample is generated in the direction of each. Specifically, for the picked concurrency bug-prone sample and nearest neighbor sample, we calculate their difference and multiply the difference with a random number between 0 and 1, and add it to the concurrency bug-prone sample. Algorithm SMOTE lists the pseudo-code for the implementation in detail.

2.4. Domain-adversarial neural network

As stated in previous sections, one of the main challenges in cross-project concurrency bug prediction is the shift between source and target distributions. In such a situation, traditional classifiers trained from source project cannot be used for the concurrency bug prediction in the target project. Domain-Adversarial Neural Network (DANN) was proposed to tackle the machine learning problem where data at training and testing time come from different distributions (Ganin et al., 2016).

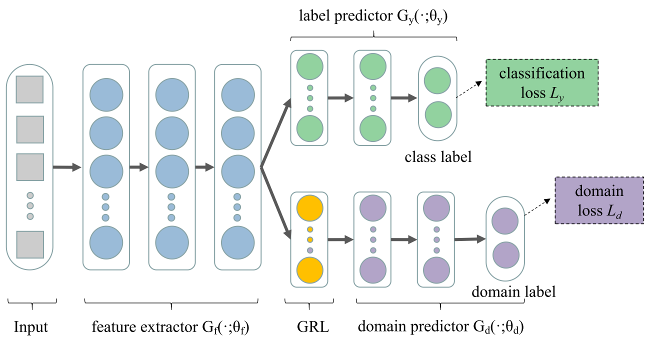
label predictor $G_y(\cdot;\theta_y)$
classification loss L_y
class label
domain loss L_d
domain label
Input
feature extractor $G_f(\cdot;\theta_f)$
GRL
domain predictor $G_d(\cdot;\theta_d)$

Fig. 2. Domain-adversarial neural network architecture.

It strives to learn a model that can generalize well from one project to another, i.e., ensuring the learned representation can be used to predict the class label of source project samples while containing no discriminative information about the origin of the input (source or target). Under such representation, the class label of target project samples can be predicted with the trained network without being hindered by the shift between source and target projects. The structure and details of the DANN network we use in this study are depicted in Fig. 2.

As shown in Fig. 2, DANN achieves the goal of both performing prediction and mitigating the distribution shift between source and target project with four parts: feature extractor, label predictor, domain predictor and Gradient Reversal Layer (GRL).

Feature extractor: Feature extractor maps the input feature into a new dimensional representation. In this new representation, the following label predictor can accurately classify whether the source project sample belongs to the concurrency bug-prone or concurrency bug-free class. However, the following domain predictor fails to detect whether a sample belongs to the source or target project. Suppose each input x_i is an m-dimensional real vector and the output dimension of feature extractor is D. Let $G_f(\cdot;\theta_f)$ be the D-dimensional neural network feature extractor with parameters θ_f. Then the output of feature extractor is $z_i = G_f(x_i;\theta_f)$. In this work, we use three fully connected layers followed by ReLU non-linear activation function to work as the feature extractor. The size of three layers from left to right are (m, 64), (64, 32), and (32, 16), respectively.

Label predictor: Label predictor predicts the class labels of samples, i.e., whether the sample belongs to concurrency bug-prone or concurrency bug-free class. It is used to evaluate label prediction performance of source project samples in the training phase and predict the class label of target project samples in the testing phase. Let $G_y(\cdot;\theta_y)$ be the *2*-dimensional neural network label predictor with parameters θ_y. Then the output of label predictor is $G_y(z_i;\theta_y)$. In this paper, we use two fully connected layers followed by ReLU non-linear activation function for the first layer and softmax function for the second layer to compute the class label output. The size of two layers is (16, 8) and (8, 2) respectively.

Domain predictor: Domain predictor discriminates the origin of samples during the training phase, i.e., whether the sample comes from source or target project. Let $G_d(\cdot;\theta_d)$ be the *2*-dimensional neural network domain predictor with parameters θ_d. Then the output of domain predictor is $G_d(z_i;\theta_d)$. In this paper, we use two fully connected layers followed by ReLU non-linear activation function for the first layer and softmax function for the second layer to compute the domain output. The size of two layers is (16, 8) and (8, 2) respectively.

Gradient Reversal Layer (GRL): Since the parameters of label predictor and domain predictor are optimized in the direction of minimizing their prediction error on the training set, and the parameters of feature extractor are optimized in the direction of minimizing the prediction error of label predictor and maximizing the prediction error

Table 2
Concurrency bug information in each project.

Project	#Con_Bugs	#Files	#Con_Prone_Files	#Con_Free_Files	Con_Prone_Files%
Accumulo	21	3007	51	2956	1.70%
Oozie	18	1325	99	1226	7.47%
Zookeeper	83	967	130	837	13.44%
Hadoop1	154	4014	224	3790	5.58%
Hadoop2	66	8547	104	8443	1.21%

of domain predictor, Gradient Reversal Layer (GRL) is inserted between feature extractor and domain predictor to solve the domain adversarial requirement for domain predictor. By acting as an identity transformation during forward propagation and multiplying the gradient by a certain negative constant during backward propagation, GRL ensures the feature extractor learn domain-invariant features.

We use cross-entropy loss to compute the loss between two predictors' prediction result and the ground truth label. Then the label predictor loss $\mathcal{L}_y$ and domain predictor loss $\mathcal{L}_d$ can be written as:

$$\begin{aligned}\mathcal{L}_y\left(\theta_f,\theta_y\right) &= \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_y^i\left(G_y(z_i;\theta_y),y_i\right)\\ &= \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_y^i\left(G_y(G_f(x_i;\theta_f);\theta_y),y_i\right)\end{aligned} \tag{2}$$

$$\begin{aligned}\mathcal{L}_d\left(\theta_f,\theta_d\right) &= \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_d^i\left(G_d(z_i;\theta_d),d_i\right)+\frac{1}{n'}\sum_{j=1}^{n'}\mathcal{L}_d^j\left(G_d(z_j;\theta_d),d_j\right)\\ &= \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_d^i\left(G_d(G_f(x_i;\theta_f);\theta_d),d_i\right)\\ &\quad+\frac{1}{n'}\sum_{j=1}^{n'}\mathcal{L}_d^j\left(G_d(G_f(x_j;\theta_f);\theta_d),d_j\right)\end{aligned} \tag{3}$$

where n and n' refer to the number of training samples in source and target project, respectively. d_k indicates the original of the sample. We set $d_k = 0$ if the sample comes from source project and $d_k = 1$ if the sample comes from target project.

Combining label predictor loss and domain predictor loss, the overall loss of DANN is as follows:

$$\mathcal{L} = \mathcal{L}_y(\theta_f,\theta_y) + \lambda\mathcal{L}_d(\theta_f,\theta_d) \tag{4}$$

where λ is domain regularization parameter.

Based on the learned DANN parameters, the trained DANN model can be used to predict the class label for samples in the target project. Given a sample in the target project, DACon feeds the z-scored features into the DANN model. The output layer of the label predictor finally gives the concurrency bug-prone and concurrency bug-free probability for the sample. The class with a higher probability is assigned as the label of the input sample.

3. Experimental setup

This section presents the details of our experimental setup, including research questions, datasets, model evaluation criteria and statistical testing method.

3.1. Research questions

In this paper, we design experiments to answer the following four research questions:

• **RQ1:** How does DACon perform in cross-project concurrency bug prediction?

• **RQ2:** Does the combination of class imbalance mitigation and distribution shift mitigation have an improvement over using any one of them individually in cross-project concurrency bug prediction?

• **RQ3:** How effective are the concurrency code metrics to the performance of DACon?

• **RQ4:** How does DACon perform in comparison to within-project concurrency bug prediction?

3.2. Datasets

We collect the datasets from four open-source software products: Accumulo,[1] Oozie,[2] Zookeeper[3] and Hadoop.[4] Accumulo is a sorted, distributed key–value store that provides robust, scalable data storage and retrieval. Oozie is a server-based workflow scheduler system to manage Apache Hadoop jobs. Zookeeper is a software project providing centralized service for distributed systems with configuration information, naming maintenance, distributed synchronization and group services. Hadoop is a software framework for distributed storage and processing of big data using the MapReduce programming model. We use Hadoop1 and Hadoop2 to denote two different projects in Hadoop, where Hadoop1 contains versions before 0.20.205 and after 1.0.0, Hadoop2 includes versions 0.23.x and after 2.0.0 since version 2.0.0 is developed based on 0.23.x and 0.20.205 is actually renamed to 1.0.0. Note that although Hadoop1 and Hadoop2 belong to the Hadoop product and share certain attributes, they show a large difference in the framework. Hadoop2 adopted a resource management architecture named YARN which decouples the resource management infrastructure from the processing components, and it increases the flexibility in big data clusters (Polato et al., 2015; Mohamed et al., 2019). Therefore, we view Hadoop1 and Hadoop2 as different projects.

The collection procedure of the concurrency bug-prone files in each project is shown in Fig. 3. We first extract concurrency bug reports from the bug issue tracking system, a collection of reported issues by users and reporters. In this paper, these concurrency bug reports are provided by research (Asadollah et al., 2017). Given filtered fixed and closed "Bug" type issue tracking reports, they were further filtered based on the concurrency-related keywords (e.g., thread, race, deadlock, concurrency, synchronization, order violation, multi-thread) which had been utilized in similar previous studies (Lu et al., 2008; Asadollah et al., 2015). The reports containing at least one concurrency keyword in their description were kept. Then, the obtained reports were manually analyzed to exclude those that did not describe concurrency bugs or had very little information to confirm. We provide a summary of concurrency bug reports in Table 2.

In the next step, we map concurrency bug reports to the relevant source code files. From a concurrency bug report, we can usually get the patches to remove the concurrency bugs (i.e., the files that contain concurrency bugs). A file is marked as concurrency bug-prone if it contains at least one concurrency bug that was reported in the concurrency bug reports, and a file without containing a concurrency bug is labeled concurrency bug-free. Note that we neglect the concurrency bug reports where no patches can be located. In addition, a bug may affect multiple source files, so the number of concurrency bug-prone files could be larger than the number of concurrency bug reports. Meanwhile, as our main objective is to perform cross-project concurrency bug prediction, the same file affected by multiple bugs is counted only once. Table 2 also presents the information we collect after source file mapping. From the table, we can see that compared with concurrency bug-free files, concurrency bug-prone files constitute only a tiny part of the whole files in the project, ranging from 1.21% to 13.44%. This indicates the severe

[1] https://issues.apache.org/jira/browse/ACCUMULO.
[2] https://issues.apache.org/jira/browse/OOZIE.
[3] https://issues.Apache.org/jira/browse/ZOOKEEPER.
[4] https://issues.apache.org/jira/browse/HADOOP.

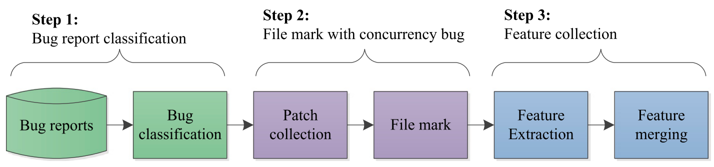

Fig. 3. The concurrency bug-prone file collection procedure.

Table 3
Summary of metrics.

Category	Metric suite	Metrics
Seq.	Program size	AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, CountLine, CountLineBlank, CountLineCode, CountStmtExe, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountSemicolon, CountStmt, CountStmtDecl, RatioCommentToCode
	McCabe complexity	AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxEssential, MaxNesting, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential
	Halstead	Program Volume, Program Length, Program Vocabulary, Program Difficulty, Effort, N1, N2, n1, n2
	Object-oriented metrics	CountDeclClass, CountDeclClassMethod, CountDeclClassVariable, CountDeclExecutableUnit, CountDeclFunction, CountDeclInstanceMethod,CountDeclInstanceVariable, CountDeclMethod, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethodProtected, CountDeclMethodPublic
Con.	Package level metrics	ConMetric, ConLockMetric, AtomicMetric, CollSynMetric
	Thread	DesThread, Start, Sleep, Yield, Join, Wait, Notify, NotifyAll
	Synchronization	Syn
	Explicit lock	LockLock, LockUnlock, ConAwait, ConSignal, ConSignalAll
	Synchronizers	SemAcquire, SemRelease, LatchAwait, LatchCountDown, BarrierAwait
	Task scheduling framework	ThrFctNewThr, Execute, AwaitTermi, Shutdown, ShutdownNow, ThrPoolExecFinal, StartThread, ThrPoolExecRej, NewThrPool, FutureCancel, FutureGet, FutureRun
	Legacy implementation	VectorMetric, HashTableMetric
	Others	ThreadLocal, Volatile, Final, Finally

class imbalance between concurrency bug-prone files and concurrency bug-free files, which poses a challenge for the predictive model.

In addition to concurrency code metrics proposed in this paper, we also utilize sequential source code complexity metrics which were widely used in previous studies and shown to be effective in reflecting the quality of code (Menzies et al., 2007b; D'Ambros et al., 2012; Song et al., 2011). Sequential code metrics measure how complex source code is without taking the unique properties of concurrent programming into consideration, but they show the code complexity in a general case. So they can also reflect the difficulty in concurrent programming and further are related to concurrency bugs, especially when combined with concurrency code metrics. For example, suppose two files have the same concurrency code metric values. The file with more lines, complex structure, operators and operands, etc., makes the program harder to understand, decreasing the probability of writing correct concurrent programs. Table 3 summarizes the sequential (Seq.) and concurrency (Con.) metrics (features) we use in this paper. The concurrency code metrics have already been elaborately introduced in Section 2.2. In sequential code metrics, program size metrics refer to information about the number of lines of code, statements, and declarations in files, McCabe complexity metrics count the number of independent paths through model's flow graph, Halstead metrics estimate reading complexity by counting operators and operands in a module and object-oriented metrics reflect the information about variable, method, class in object-oriented programs. Readers can refer to Understand[5] for a detailed description of code attributes. In our study, all metrics are automatically extracted with the *Understand* tool. Since the projects we study contain several versions, we extract all the metrics listed in Table 3 in all related versions and compute the average value across these versions as the final metric used in our prediction, similarly as the way in Mockus et al. (2009).

[5] https://scitools.com/

Table 4
Confusion matrix.

		Predicted class	
		Con_prone	Con_free
Actual class	Con_prone	TP	FN
	Con_free	FP	TN

3.3. Model evaluation criteria

To evaluate the prediction performance of our approach, we employ five widely used model evaluation measures (Yang et al., 2022). We first show a confusion matrix in Table 4, based on which evaluation measures are defined as below.

Pre (Precision): It measures the proportion of predicted concurrency bug-prone files that are actually concurrency bug-prone. A high value indicates that actually concurrency bug-prone files constitute a high percentage of the predicted concurrency bug-prone files. It is defined as $Pre=\frac{TP}{TP+FP}$.

Re (Recall): It represents the percentage that concurrency bug-prone files are predicted as concurrency bug-prone. A high value indicates that many concurrency bug-prone files are predicted correctly. It is defined as $Re=\frac{TP}{TP+FN}$.

F1 (F1-measure): It is the harmonic mean of Pre and Re and is defined as $F1=\frac{2*Pre*Re}{Pre+Re}$. It symmetrically represents both precision and recall in one metric.

Bal (Balance): It is a trade-off of Re and PF (the probability that concurrency bug-free files are predicted as concurrency bug-prone $PF = \frac{FP}{FP+TN}$), and is defined as $Bal = 1-\frac{\sqrt{(0-PF)^2+(1-Re)^2}}{\sqrt{2}}$. It is a commonly used evaluation measure to evaluate bug prediction results by classification models in class imbalance tasks (Menzies et al., 2007b; Wang and Yao, 2013). The higher the value, the better the prediction result.

AUC (Area Under the Curve): It is the area under the receiver operating characteristics curve. It is independent of the cutoff value and ranges between 0 and 1. A larger AUC value represents a better prediction performance. The prediction model with an AUC value above 0.5 is regarded more effective than the random guessing and 0.7 is reasonably good (Yu et al., 2019; Hosmer et al., 2013).

3.4. Statistical testing

To compare the performance of different prediction models across 20 pair-wise groups of experiments, we apply two statistical testing methods, including one-sided Wilcoxon signed-rank test (Wilcoxon, 1945) and double Scott-Knott Effective Size Difference (ESD) test (Tantithamthavorn et al., 2017, 2019). Wilcoxon signed-rank test (Wilcoxon, 1945) is a widely used non-parametric test to check whether the difference in two data groups is statistically significant or not. In this study, we use one-sided Wilcoxon signed-rank test to detect whether DACon statistically outperforms its comparison method on each pair-wise experimental group at the confidence level of 95%. Scott-Knott ESD test (Tantithamthavorn et al., 2017, 2019) leverages a hierarchical clustering to divide a set of values into statistically distinct groups (ranks) with a non-negligible difference. Different from the standard Scott-Knott test (Scott and Knott, 1974), it makes no assumption about the underlying distribution and merges any two statistically different groups having a negligible effect size into one group. To test the predictive power of different prediction models among all the pair-wise experimental groups, we perform a double Scott-Knott ESD test. Specifically, we first generate Scott-Knott ESD ranks for each experimental group. Based on the prediction results of 100 iterations of each prediction model, a list of ranks for each experimental group is derived (i.e., each prediction model has a rank value). A smaller rank value indicates that the prediction model has a better prediction performance. Then, we summarize the ranks of prediction models for all the experimental groups. A second run of the Scott-Knott ESD test on these list of ranks across all the studied experimental groups is conducted. So a statistically distinct rank of prediction models across all the experimental groups is obtained. The implementation of the Scott-Knott ESD test can be obtained in CRAN[6] and GitHub.[7]

4. Experiment implementation and results

In this section, we present our motivation, implementation and findings for each research question introduced in Section 3.1.

4.1. RQ1: How does DACon perform in cross-project concurrency bug prediction?

Motivation. Unlike traditional bug prediction, cross-project concurrency bug prediction has to deal with the shift between source and target distributions and the class imbalance between concurrency bug-prone files and concurrency bug-free files. Hence, it is of significant interest to investigate if cross-project concurrency bug prediction is feasible, that is, the prediction performance of DACon in our case.

Implementation. To evaluate the performance of DACon, we should provide ways to set hyper-parameters (such as the domain regularization parameter λ, the learning rate l). Note that the hyper-parameter tuning is performed individually for each model evaluation measure. In our experiment, we fix λ at 10, and set l as follows. Given the z-scored labeled source samples S and unlabeled target samples T, we split samples in the source project S with stratified random sampling into training set S_{train} (containing 2/3 samples) and validation set S_{valid} (containing remaining 1/3 samples). We use S_{train} and T to learn the domain predictor and S_{train} to learn the label predictor. Meanwhile, we use S_{valid} to evaluate the performance of DACon. The

Table 5
The prediction performance of DACon.

Group	Pre	Re	F1	Bal	AUC
A⇒H1	0.312	0.994	0.207	0.687	0.707
A⇒H2	0.048	0.994	0.082	0.705	0.734
A⇒O	0.210	0.996	0.139	0.626	0.664
A⇒Z	0.237	0.992	0.245	0.605	0.644
H1⇒A	0.086	1.000	0.091	0.613	0.629
H1⇒H2	0.033	1.000	0.090	0.761	0.774
H1⇒O	0.058	1.000	0.243	0.698	0.698
H1⇒Z	0.093	1.000	0.307	0.622	0.635
H2⇒A	0.081	0.997	0.075	0.607	0.632
H2⇒H1	0.361	0.999	0.239	0.724	0.730
H2⇒O	0.172	0.999	0.182	0.685	0.695
H2⇒Z	0.238	0.995	0.230	0.617	0.631
O⇒A	0.085	0.999	0.046	0.555	0.607
O⇒H1	0.313	0.995	0.201	0.670	0.698
O⇒H2	0.058	0.997	0.062	0.687	0.713
O⇒Z	0.285	0.999	0.257	0.592	0.626
Z⇒A	0.081	0.998	0.063	0.552	0.586
Z⇒H1	0.467	0.995	0.225	0.692	0.690
Z⇒H2	0.065	0.998	0.063	0.702	0.723
Z⇒O	0.183	1.000	0.231	0.662	0.677
Avg.	0.173	0.997	0.164	0.653	0.675

process is repeated with 100 random values sampled using the Tree-structured Parzen Estimator algorithm (Bergstra et al., 2011) in the set [1e−5, 10] in the log domain with Optuna (Akiba et al., 2019), and the selected l value is the one with maximum model evaluation measure value in S_{valid}. We train DACon using SGD optimizer for 500 epochs with PyTorch. Note that the performance on the validation set S_{valid} is also used as an early stopping criterion to avoid overfitting problem during the training process with patience value set as 5. We run the above procedure 100 repetitions and compute the average evaluation measure value in the target project T.

In addition, we compare our approach against the plain model where both SMOTE and DANN are not included. In detail, compared with the structure of DACon, the plain model only contains metric extraction, z-score normalization, and a standard neural network that comprises feature extractor and label predictor parts in DANN (i.e., neither GRL nor domain predictor is included). For a fair comparison, the standard neural network has the same layer number, layer size and activation function as the relative DANN part. Since no domain predictor is included in the neural network, the unlabeled target data are not used for training. That is, only S_{train} is used to learn the label predictor and S_{valid} to evaluate the performance of the plain model to get the best hyper-parameter. The whole procedure is also repeated with 100 runs with the same implementation details and records the average Bal value in target project T.

Besides, we also reproduce the prediction models in ConPredictor (Yu et al., 2019) with Weka (Hall et al., 2009) and show the prediction performance. The feature selection is first performed on the source project to select effective metrics. Since the feature selection method WrapperSubsetEval used in Yu et al. (2019) deletes all metrics after its processing in our dataset, we use CfsSubsetEval instead. Similar to WrapperSubsetEval, CfsSubsetEval also evaluates the worth of a subset of metrics by considering the individual predictive power of each metric along with the degree of redundancy between them. Then, Weka's SMOTE filter is performed on the source samples S to increase the samples of the concurrency bug-prone class to the number of concurrency bug-free class. We use Bayesian Network, J48 Decision Tree, Logistic Regression, and Random Forest as classification algorithms with default hyper-parameter values. Since the source and target set are fixed in the prediction task without random sampling and no hyper-parameters to be set, the prediction process with each prediction model is only performed once.

Findings. Our DACon achieves good results for cross-project concurrency bug prediction. Table 5 shows the performance of DACon in 20 pair-wise groups in terms of five evaluation measures. For

[6] https://cran.r-project.org/web/packages/ScottKnottESD.
[7] http://github.com/klainfo/ScottKnottESD/.

Table 6
The comparison between DACon and baseline models in terms of Bal.

Group	DACon	Plain	Impv.	BN	Impv.	DT	Impv.	LR	Impv.	RF	Impv.
A⇒H1	0.687	0.416	**65.12%**	0.642	**7.01%**	0.485	**41.66%**	0.679	**1.28%**	0.409	**68.17%**
A⇒H2	0.705	0.426	**65.53%**	0.560	**25.75%**	0.474	**48.73%**	0.668	**5.52%**	0.380	**85.32%**
A⇒O	0.626	0.381	**64.57%**	0.442	**41.64%**	0.342	**83.24%**	0.385	**62.84%**	0.378	**65.49%**
A⇒Z	0.605	0.389	**55.47%**	0.595	**1.71%**	0.477	**26.89%**	0.632	−4.24%	0.445	**36.03%**
H1⇒A	0.613	0.418	**46.76%**	0.554	**10.76%**	0.472	**30.03%**	0.482	**27.31%**	0.431	**42.25%**
H1⇒H2	0.761	0.486	**56.61%**	0.588	**29.29%**	0.535	**42.33%**	0.641	**18.70%**	0.503	**51.32%**
H1⇒O	0.698	0.426	**63.87%**	0.428	**63.11%**	0.384	**81.50%**	0.391	**78.37%**	0.321	**117.17%**
H1⇒Z	0.622	0.483	**28.57%**	0.600	**3.56%**	0.529	**17.51%**	0.546	**13.79%**	0.466	**33.38%**
H2⇒A	0.607	0.381	**59.39%**	0.541	**12.15%**	0.417	**45.59%**	0.559	**8.70%**	0.362	**67.64%**
H2⇒H1	0.724	0.432	**67.60%**	0.604	**19.93%**	0.431	**67.95%**	0.634	**14.18%**	0.397	**82.45%**
H2⇒O	0.685	0.409	**67.27%**	0.428	**59.99%**	0.328	**108.52%**	0.630	**8.73%**	0.293	**133.97%**
H2⇒Z	0.617	0.425	**45.18%**	0.537	**14.81%**	0.384	**60.58%**	0.598	**3.07%**	0.353	**74.85%**
O⇒A	0.555	0.387	**43.42%**	0.527	**5.37%**	0.466	**19.04%**	0.535	**3.80%**	0.416	**33.30%**
O⇒H1	0.670	0.446	**50.25%**	0.557	**20.40%**	0.529	**26.69%**	0.597	**12.37%**	0.497	**34.89%**
O⇒H2	0.687	0.446	**54.29%**	0.549	**25.22%**	0.479	**43.47%**	0.627	**9.59%**	0.455	**51.14%**
O⇒Z	0.592	0.404	**46.50%**	0.586	**0.93%**	0.448	**32.13%**	0.618	−4.31%	0.427	**38.45%**
Z⇒A	0.552	0.508	**8.66%**	0.552	−0.09%	0.551	**0.08%**	0.624	−11.54%	0.510	**8.22%**
Z⇒H1	0.692	0.624	**10.97%**	0.659	**5.10%**	0.523	**32.50%**	0.689	**0.49%**	0.562	**23.13%**
Z⇒H2	0.702	0.624	**12.53%**	0.580	**21.09%**	0.471	**48.88%**	0.699	**0.35%**	0.491	**43.05%**
Z⇒O	0.662	0.558	**18.72%**	0.550	**20.33%**	0.509	**30.19%**	0.565	**17.17%**	0.468	**41.56%**
Avg.	0.653	0.453	**46.56%**	0.554	**19.40%**	0.462	**44.38%**	0.590	**13.31%**	0.428	**56.59%**
Wilcoxon		4.26E−07		6.65E−06		3.15E−08		1.63E−02		4.26E−08	

simplicity, A, H1, H2, O, Z represent Accumulo, Hadoop1, Hadoop2, Oozie and Zookeeper, respectively. The column “Group” represents each cross-project concurrency bug prediction group. For example, A⇒H1 represents using data from Accumulo to predict the concurrency bug proneness in Hadoop1. “Avg.” refers to the average prediction value among all groups. From the table, we can see that when using DACon as the prediction classifier, most Bal values are larger than 0.6, and the average value is 0.653. In detail, among all the 20 groups, there are only 3 groups that show the Bal value less than 0.6. In some groups, the Bal is larger than 0.7, such as A⇒H2, H1⇒H2, H2⇒H1, and Z⇒H2, and the largest value can reach 0.761. In terms of Re, the prediction values are larger than 0.99 in all the groups and even reach 1 in some groups. This demonstrates that DACon can almost correctly predict all the concurrency bug-prone files when using Re as the evaluation measure. In terms of AUC, we can observe that the values are larger than 0.5 in all the groups, and 6 values are above 0.7. This indicates that DACon performs better than random guessing and reasonably good in 30% groups when using AUC as the evaluation measure. However, we can also observe that the prediction performance is undesirable when using Pre or F1 as the evaluation measure. This phenomenon is consistent with other studies on software bug prediction (Menzies et al., 2007b; Cotroneo et al., 2013b; Gesi et al., 2021; Huang et al., 2019; Keshavarz and Rodríguez-Pérez, 2024): for the datasets with severe class imbalance, it is a very challenging problem for a prediction classifier to be precise enough to identify few concurrency bug-prone files without incurring concurrency bug-free files. As a consequence, the Pre, and further the F1 measure, of classifiers tend to be low. In addition, in the software bug prediction task, researchers and practitioners seek to identify as many bugs as possible because the cost of missing the bug-prone files is much more expensive than checking the false alarms, Re is regarded to be more important than Pre and low precision classifiers are useful (Cotroneo et al., 2013b; Menzies et al., 2007a; Zhao et al., 2021).

Table 6 displays the comparison between DACon and baseline models in terms of Bal. In the table, Plain, BN, DT, LR, and RF denote plain model, Bayesian Network, J48 Decision Tree, Logistic Regression and Random Forest baseline model respectively. The “Impv.” column shows the improvement of DACon over the baseline models. Here we use $(b-a)/a\%$ as the improvement where a and b refer to the average Bal of DACon and baseline model, respectively. Impv. values showing that DACon improves the prediction performance over the baseline model are highlighted in boldface. It is clear in Table 6 that our approach improves Bal over the five baseline models in most groups. In detail, the improvement is observed in 20, 19, 20, 17, 20 groups when comparing DACon with plain model, Bayesian Network, Decision Tree, Logistic Regression and Random Forest respectively, ranging from 8.66% to 67.60%, from 0.93% to 63.11%, from 0.08% to 108.52%, from 0.35% to 78.37%, from 8.22% to 133.97% with an average improvement of 46.56%, 19.40%, 44.38%, 13.31%, and 56.59%. To check if the performance of DACon is statistically better than the baseline models, we run one-sided Wilcoxon signed-rank test on the prediction results and list the results in Table 6. From the table, we can see that DACon statistically significantly improves the prediction performance at the confidence level of 95% when compared with the five baseline models.

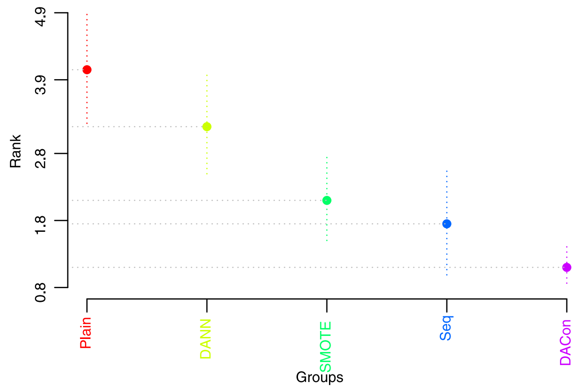

Fig. 4. The results of double Scott-Knott ESD test for DACon and other methods.

Fig. 4 presents the results of the double Scott-Knott ESD test in Bal values for DACon and other models. Since there is no experimental repetition with Bayesian Network, Decision Tree, Logistic Regression and Random Forest in each group, we do not perform double Scott-Knott ESD test with these models. Each name on the horizontal axis represents a model, while the vertical axis represents the rank value. The vertical dotted line refers to the rank range between *mean-std.* and *mean+std.* for a specific model, and the big dot on the line refers to the mean ranking of model. Different colors denote different groups where models in the same group have no significant difference while those in distinct groups have significant differences at a significance level of 0.05. From the figure, we can see that DACon and plain model belong to different ranking groups and DACon has a smaller ranking.

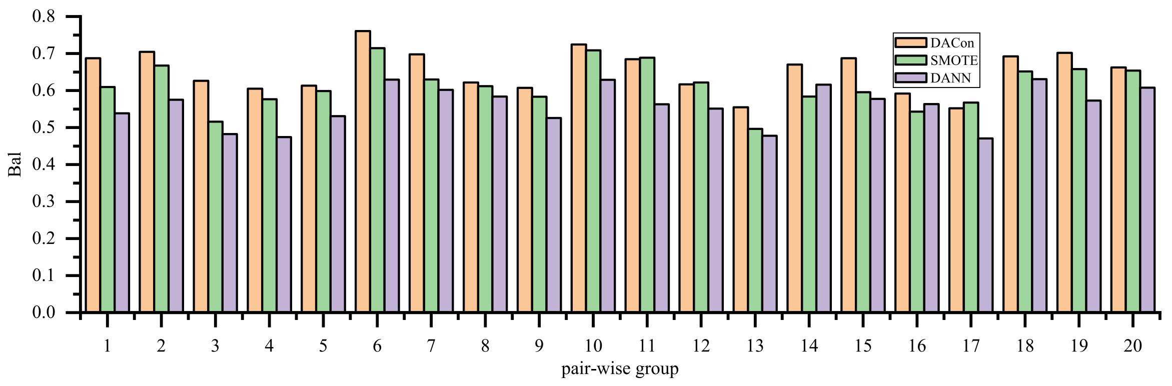

Fig. 5. The comparison between DACon, SMOTE alone and DANN alone.

This indicates that DACon is statistically significantly superior to the plain model.

From these results, we can conclude that DACon has the potential to get a good prediction performance in cross-project concurrency bug prediction.

4.2. RQ2: Does the combination of class imbalance mitigation and distribution shift mitigation have an improvement over using any one of them individually in cross-project concurrency bug prediction?

Motivation. As two major challenges in cross-project concurrency bug prediction are severe class imbalance between concurrency bug-prone and concurrency bug-free files, and shift between source and target distributions, we are interested to find if SMOTE and DANN help in improving the prediction performance comparing with using them individually.

Implementation. To address this research question, we need to get the performance when SMOTE and DANN are used individually. In terms of the SMOTE only situation (SMOTE), the prediction model is built without GRL and domain predictor branch included. That is, the DANN part is replaced with the standard neural network sharing the same architecture in feature extractor and label predictor. In such a situation, the prediction model is trained without consideration for target domain data, similar to the plain model in Section 4.1. In terms of the DANN only situation (DANN), the SMOTE part is not included in contrast to DACon. In such a situation, the prediction model is trained using the same procedure as DACon. Both SMOTE and DANN are repeated 100 times, and their average Bal values in the target project are recorded.

Findings. Both SMOTE and DANN parts are valuable to DACon. The prediction performance will decrease if one of them is not used. Fig. 5 gives the comparison of DACon with the approach using individual SMOTE and DANN. In the figure, x-axis denotes different pair-wise group cases and y-axis denotes the average Bal measures with 100 repetitions. As shown in the figure, most groups using DACon show a better prediction performance over the approach using SMOTE or DANN alone. As for the approach using DANN alone, DACon shows its superiority over all the pair-wise group cases. As for the approach using SMOTE alone, DACon outperforms in 17 out of 20 groups. In the rest 3 groups, the average prediction values of DACon is only 0.004, 0.005 and 0.016 less than those of SMOTE, with 0.016 as the upper limit. To better demonstrate the superiority of DACon over DANN and SMOTE, we also apply Wilcoxon one-tailed signed-rank test in each pair-wise group. The p-values are less than 0.05 in all the 20 groups compared with DANN, and 17 out of 20 groups compared with SMOTE. Furthermore, the double Scott-Knott ESD test result in Fig. 4 shows that DANN and SMOTE are not in the same ranking group as DACon. This demonstrates that DACon is statistically significantly better than DANN and SMOTE and further shows that both class imbalance mitigation and distribution shift mitigation are valuable to our approach.

Table 7
The comparison between DACon and model using sequential code metrics alone.

Group	DACon	Seq	Diff	Group	DACon	Seq	Diff
A⇒H1	0.687	0.679	**−0.008**	H2⇒O	0.685	0.685	0.000
A⇒H2	0.705	0.689	**−0.015**	H2⇒Z	0.617	0.597	**−0.020**
A⇒O	0.626	0.633	0.006	O⇒A	0.555	0.549	**−0.006**
A⇒Z	0.605	0.572	**−0.033**	O⇒H1	0.670	0.667	**−0.003**
H1⇒A	0.613	0.559	**−0.054**	O⇒H2	0.687	0.670	**−0.017**
H1⇒H2	0.761	0.726	**−0.034**	O⇒Z	0.592	0.590	**−0.002**
H1⇒O	0.698	0.691	**−0.007**	Z⇒A	0.552	0.566	0.014
H1⇒Z	0.622	0.605	**−0.016**	Z⇒H1	0.692	0.681	**−0.012**
H2⇒A	0.607	0.566	**−0.041**	Z⇒H2	0.702	0.644	**−0.058**
H2⇒H1	0.724	0.705	**−0.019**	Z⇒O	0.662	0.680	0.017
Avg.	0.653	0.638	−0.015	W/T/L	–	15/5/0	–

4.3. RQ3: How effective are the concurrency code metrics to the performance of DACon?

Motivation. In this paper, we propose eight suits of concurrency code metrics to capture the characteristics of concurrent programming. Thus, we wonder whether the consideration of concurrency code metrics is helpful for cross-project concurrency bug prediction.

Implementation. To answer this research question, we should present the prediction results where only sequential code metrics are used. For a fair comparison, we use the same framework as DACon and substitute its metrics with sequential code metrics alone. Similarly, the experiments on all the pair-wise groups are repeated 100 times.

Findings. The combination of concurrency and sequential code metrics is more effective than using sequential code metrics alone. As shown in Table 7, the "Seq" column refers to the approach using sequential code metrics alone. "Diff" represents the difference between Seq and DACon in each group. The numbers marked in boldface indicate that the performance of Seq is inferior to DACon. "Avg." denotes the average results over all the groups. The blue or grey cells imply significantly better performance or equality in one-sided Wilcoxon signed-rank test when comparing DACon with Seq. "W/T/L" shows the number of times DACon wins, ties, and losses in the one-sided Wilcoxon signed-rank test in the 20 pair-wise groups of experiments.

As data shown in Table 7, there is much more degradation than improvement when removing concurrency code metrics. Specifically, the decrease ranges from 0.002 to 0.058, accounting for 80.0% groups. This phenomenon indicates that concurrency code metrics make a substantial contribution to our approach. In addition, the one-sided Wilcoxon signed-rank test shows that DACon wins, ties and loses on 15, 5 and on 0 groups. Furthermore, the double Scott-Knott ESD test result shown in Fig. 4 indicates that Seq ranks in the penultimate group, which further demonstrates that Seq is statistically significantly inferior to DACon.

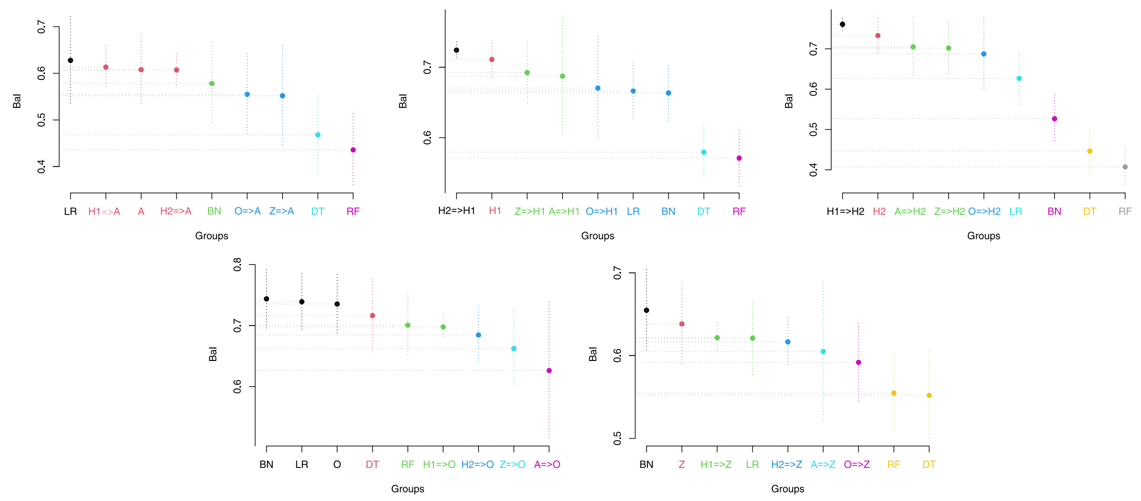

Fig. 6. The Scott-Knott ESD test for cross-project concurrency bug prediction and within-project setting. (a) Accumulo (b) Hadoop1 (c) Hadoop2 (d) Oozie (e) Zookeeper.

4.4. RQ4: How does DACon perform in comparison to within-project concurrency bug prediction?

Motivation. In comparison to cross-project concurrency bug prediction, the within-project setting does not suffer from data distribution shift between the training and testing phase. As cross-project concurrency bug prediction can solve the lack of labeled data in the target project, we are interested in finding if our DACon approach can compete with the within-project setting.

Implementation. In this research question, we use two kinds of models to get the within-project prediction results. The first kind of model has the same model architecture as SMOTE only situation in Section 4.2, i.e., only GRL and domain predictor branch is not included compared with DACon. All the samples P of a project are separated into training set P_{train_1}, validation set P_{valid_1} and testing set P_{test_1}. Here, we apply stratified random sampling and split the samples into three folds where each fold consists of 1/3 samples. We use P_{train_1} to train the model and P_{valid_1} to get the best hyper-parameter. To deal with the randomness of sampling, we repeat the random splits 100 times and compute the average Bal values in P_{test_1}. The second kind of model is the prediction models in ConPredictor (Yu et al., 2019). It also includes the feature selection, SMOTE, and classification algorithms (Bayesian Network, J48 Decision Tree, Logistic Regression, and Random Forest) as in Section 4.1. During the implementation, all the samples P of a project are separated into training set P_{train_2} (2/3) and testing set P_{test_2} (1/3) with stratified random sampling. The process is also repeated 100 times for each classification algorithm to avoid sampling bias.

Findings. The performance of DACon is comparable with the within-project setting, and even outperforms in some cases. Fig. 6 presents the Scott-Knott ESD test result in five studied projects. Since four cross-project alternatives are available for each target project, we only perform the Scott-Knott ESD test once. Thus the y-axis denotes the Bal performance measure and the vertical dotted line denotes the Bal range between *mean-std.* and *mean+std.* for the specific model. We use the abbreviations of the target project to stand for the first kind of model in the within-project case. For example, Fig. 6(a) illustrates the Scott-Knott ESD test result with Accumulo as the target project. In the subfigure, “A” refers to the first kind of within-project result, BN, DT, LR, and RF represent the second kind of within-project result, and H1⇒A, H2⇒A, O⇒A, and Z⇒A denote the four cross-project results.

It can be observed from Fig. 6 that the within-project methods are not always divided into the first several groups in which models achieve the best performance without including any cross-project alternative. Specifically, when Oozie serves as the target project, all the cross-project cases show a worse prediction ability than the within-project cases with the smallest decrease of 0.003. When Accumulo and Zookeeper serve as the target project, the cross-project cases scatter among the middle groups, indicating neither the best nor worst prediction performance compared to within-project cases. When Hadoop1 and Hadoop2 serve as the target project, cross-project cases generally have a better performance than within-project cases. H2⇒H1 and H1⇒H2 belong to the first group in two target projects, respectively and the best within-project case belongs to the second group. This demonstrates that at least one cross-project case shows statistically significantly better prediction performance than within-project setting. In addition, we calculate the average prediction performance of 25 within-project results and 20 cross-project results among five target projects and get the values of 0.612 and 0.653, respectively. This indicates that the average cross-project prediction result is even better than the within-project result. Thus, the experimental results imply that we can leverage data from another project to conduct concurrency bug prediction with DACon, which may get prediction performance comparable or even superior to the within-project setting.

5. Discussion

This paper presents DACon, a cross-project concurrency bug prediction approach to predict files likely to contain concurrency bugs using training data from a different project. To the best of our knowledge, ConPredictor (Yu et al., 2019) represents state-of-the-art concurrency bug prediction research and mainly targets within-project bug prediction. Both studies propose a set of metrics specific to concurrent programs. However, there is a large difference between them. The metrics in ConPredictor are heavyweight. Firstly, the dynamic metric collection requires the program to be runnable, and have test cases and test oracles. Secondly, the static code metric collection needs intensive program analysis knowledge to construct and analyze the CCFG (concurrent control flow graph). Thirdly, the whole procedure depends on more tools (Clang/LLVM platform and CCMutator). Fourthly, it is time-consuming to get both the dynamic metrics by running test cases on all the mutants of the object program and static code metrics by constructing and transforming CCFGs. Comparatively, the metrics proposed in this paper are lightweight. They are all static, having no restrictions on the program’s runnability and testability, relying on fewer tools and prior knowledge, and are less time-consuming.

In this paper, we make an effort to perform a comparison between DACon and ConPredictor (Yu et al., 2019). As our aim, which is different from that in Yu et al. (2019), is to verify the feasibility and performance of cross-project concurrency bug prediction, we do not reproduce the heavy metrics in Yu et al. (2019) in baselines and only reimplement the four prediction models. Thus, we cannot state whether DACon advances or underperforms ConPredictor. We assess the performance of DACon mainly through Bal which is widely used in class imbalance scenarios (Menzies et al., 2007b; Wang and Yao, 2013). A high Bal indicates a classifier can identify a high percentage of concurrency bug-prone files while inducing a low probability of false alarms. Our results indicate DACon has the potential to get a good prediction performance in cross-project concurrency bug prediction scenarios and its performance is comparable or even superior to the within-project setting. A point to note is that Yu et al. (2019) also performed a tentative cross-project concurrency bug prediction and showed moderate effectiveness in terms of F1-measure. Our results contradict this conclusion in terms of F1: we obtain an average F1 of 0.164 across 20 pairwise cases. Besides metrics, the studied projects and prediction granularity are all potential reasons affecting our results. This is only a speculation and remains to be verified in the future work.

6. Threats to validity

Threats to construct validity mainly refer to concurrency bug classification, extracted metrics, and the suitability of our evaluation measures in this study. First, the concurrency bug report labeling was gathered from the previous study (Asadollah et al., 2017). As the authors stated in their study, there might be concurrency bugs that were not reported in the issue trackers, not included in the studied reports, or labeled incorrectly. These factors may influence the concurrency bug-prone file identification and further the prediction performance of the proposed method. Second, the accuracy of extracted metrics, including concurrency code metrics and sequential code metrics, depends on the tool used. We adopt Understand, a mature commercial tool widely used in research (Cotroneo et al., 2013b; Qin et al., 2019), to extract the metrics. Third, we employ five measures to evaluate the prediction performance of the model. Nonetheless, other evaluation measures can also be considered. We describe our method in detail. Readers can easily replicate our experiments and choose the evaluation measure they concern.

Threats to internal validity primarily focus on factors that could affect our experimental results. We reduce the influence of randomness by repeating our experiments 100 times with different choices of training and validation set in the source project when building the prediction modules. In addition, two statistical testing methods, including one-sided Wilcoxon signed-rank test and double Scott-Knott ESD test, are used to test the prediction difference among different models.

Threats to external validity focus on the generalization of the obtained results. We perform experiments on 20 pair-wise groups in five projects to show the feasibility of cross-project concurrency bug prediction and the superiority of our proposed approach DACon. Although we exhaustively explore the experiments in five open source projects, we cannot make sure how well it will generalize to other cross-project settings. In the near future, we plan to reduce this threat further by performing experiments on more projects.

7. Related work

7.1. Software bug prediction

Software bug prediction techniques utilize software information such as code complexity, software development history and code authors to build a prediction model to determine whether new software modules are bug-prone or not (Qu et al., 2021; Herbold, 2019; Qu and Yin, 2021; Chakraborty and Chakraborty, 2020; Di Nucci et al., 2018; Wu et al., 2018; Malhotra, 2015). This is helpful for software practitioners to target the limited testing resources on bug-prone modules in the early stage of software development.

Many studies focus on within-project bug prediction in which training and testing datasets all belong to the same project (Malhotra, 2015; Nam, 2014; Panichella et al., 2014). Within-project bug prediction can get a good prediction result when enough training data are available. Plenty of models have been proposed and used to perform the bug prediction. They can be categorized as decision trees, Bayesian method, regression function, kernel-based algorithm, instance-based algorithm, clustering algorithm, association rule learning algorithms, artificial neural network, and ensemble algorithm (Malhotra, 2015; Nam, 2014).

In recent years, cross-project bug prediction has been attracting more and more attention to address the difficulty in training data collection in within-project bug prediction (Zimmermann et al., 2009; Qin et al., 2015; Pan and Yang, 2010). It applies the model built on one project and conducts the bug prediction on another project. However, a great challenge for cross-project bug prediction is the distribution difference between source and target projects, which will influence the prediction performance of classifiers. Different methods have been proposed to solve this challenge. Turhan et al. (2009) used the nearest neighbor filtering to conduct source instances selection according to their similarity to instances in the target project. Ma et al. (2012) proposed Transfer Naive Bayes (TNB) which built a weighted Naive Bayes classifier by allocating different weights to instances in the source project according to their similarity to instances in the target project. Nam et al. (2013) proposed a TCA+ approach by combining normalization selection and Transfer Component Analysis (TCA) (Pan et al., 2011) which reduces the distribution difference by transferring source and target dataset into a latent space with the maximum mean discrepancy. Ryu et al. (2017) proposed a transfer cost-sensitive boosting method that takes both knowledge transfer and class imbalance into consideration with a small amount of available labeled target dataset. Qiu et al. (2019) put forward a joint distribution matching approach named JDM to minimize the marginal distribution discrepancy and conditional distribution discrepancy simultaneously between source and target project. Different from Qiu et al. (2019), Xu et al. (2019) also considered two kinds of distribution differences but adaptively assigned different weights to them.

7.2. Concurrency bugs

Since the pervasiveness of multi-core hardware, concurrent programs have become more and more popular. Unfortunately, writing correct concurrent programs is not easy. Previous studies showed that compared with non-concurrency bugs, it is more difficult to fix concurrency bugs (Zhou et al., 2015). To understand the characteristic of concurrency bugs, several empirical studies have been conducted. Although the number is not large, they provide precious information to help improve the understanding of concurrency bugs and software reliability caused by concurrency bugs from many aspects, such as concurrency bug detection, fixing, and testing. Lu et al. (2008) analyzed 105 real-world concurrency bugs from 4 large open-source applications and studied their bug pattern, manifestation and fix. The findings and implications gave practical and valuable suggestions for concurrency bug detection, testing, fixing and language design. Fonseca et al. (2010) explored 80 concurrency bugs in MySQL and centred their study on the internal and external effects. The findings provide insights into the detection or tolerance of concurrency bugs. Asadollah et al. (2017) performed a comprehensive study on concurrency bugs in five open-source software systems, aiming at understanding the differences between concurrency bugs and non-concurrency bugs in terms of proportion, reproducibility, fix time and severity. The findings in these empirical studies are helpful for developers and designers to understand concurrency bugs and give valuable suggestions for concurrency bug detection, testing, fixing and tolerance.

Recently, a few studies have started to perform concurrency bug prediction. Zhou et al. (2015) utilized the textural description in the bug report to predict whether a new bug report is related to concurrency bugs or not. Ciancarini et al. (2017) leveraged information from the bug tracking system and code versioning system to triage concurrency bugs. The above studies mainly focus on bug report triaging. In 2016, Ciancarini et al. (2016) realized the importance of concurrency bug prediction in the source code and proposed some concurrency-related source code metrics. Then Yu et al. (2019) first proposed an approach named ConPredictor to mainly perform within-project concurrency bug prediction in the source code level. They proposed 6 code metrics and 18 mutation metrics and built the classifier based on these 24 metrics. TehraniJamsaz et al. (2021) proposed DeepRace to predict three patterns of data race bugs in the source codes via convolutional neural network. Habib and Pradel (2018) proposed TSFinder to automatically classify a given class as thread-safe or thread-unsafe to address the problem of missing thread safety documentation. In contrast, the main purpose of our paper is cross-project concurrency bug prediction where no historical labeled concurrency bug data are available in the target project. Moreover, our proposed concurrency code metrics are all static metrics that are easier to collect than those in Yu et al. (2019). To the best of our knowledge, this is the first paper targeting the cross-project concurrency bug prediction.

8. Conclusion and future work

This paper proposes a Domain-Adversarial neural network-based cross-project Concurrency bug prediction approach (DACon) to tackle the concurrency bug labeling problem in within-project concurrency bug prediction. The effectiveness of the proposed approach is validated with 20 pair-wise groups of experiments in five open-source projects. The experimental results show that DACon is effective in performing cross-project concurrency bug prediction, with prediction performance comparable with within-project setting and even outperforms in some cases. We are optimistic that this approach can give a new solution and choice in concurrency bug prediction when training data in the target project are difficult to collect, and it will guide optimal resource allocation strategies during software testing based on which practitioners can make a more comprehensive decision, which may reduce the cost of software testing and increase software testing process effectiveness.

In the near future, we first plan to validate the effectiveness of DACon with more projects. Second, we will further explore the cross-project concurrency bug prediction by leveraging data from multiple projects. Third, we plan to develop new concurrency code metrics from the perspective of complex network based on the concurrency-related interactions among files.

CRediT authorship contribution statement

Fangyun Qin: Formal analysis, Methodology, Software, Writing – original draft, Writing – review & editing. **Zheng Zheng:** Conceptualization, Funding acquisition, Supervision, Writing – original draft, Writing – review & editing. **Yulei Sui:** Conceptualization, Data curation, Investigation, Writing – original draft, Writing – review & editing. **Siqian Gong:** Formal analysis, Investigation, Validation. **Zhiping Shi:** Project administration, Resources, Writing – review & editing. **Kishor S. Trivedi:** Conceptualization, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (grant numbers 62372021, 61772055, 61876111, 61872169 and 62103034), and R&D Program of Beijing Municipal Education Commission (grant number KZ20231002822).

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M., 2019. Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 2623–2631.
Asadollah, S.A., Hansson, H., Sundmark, D., Eldh, S., 2015. Towards classification of concurrency bugs based on observable properties. In: 2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems. COUFLESS, IEEE, pp. 41–47.
Asadollah, S.A., Sundmark, D., Eldh, S., Hansson, H., 2017. Concurrency bugs in open source software: a case study. J. Internet Serv. Appl. 8 (1), 1–15.
Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., 2011. Algorithms for hyper-parameter optimization. Adv. Neural Inf. Process. Syst. 24.
Bradbury, J.S., Cordy, J.R., Dingel, J., 2006. Mutation Operators for Concurrent Java (J2SE 5.0). Tech. rep., Queen's University.
Briand, L.C., Melo, W.L., Wust, J., 2002. Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans. Softw. Eng. 28 (7), 706–720.
Buda, M., Maki, A., Mazurowski, M.A., 2018. A systematic study of the class imbalance problem in convolutional neural networks. Neural Netw. 106, 249–259.
Carrozza, G., Cotroneo, D., Natella, R., Pietrantuono, R., Russo, S., 2013. Analysis and prediction of mandelbugs in an industrial software system. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. IEEE, pp. 262–271.
Chakraborty, T., Chakraborty, A.K., 2020. Hellinger net: A hybrid imbalance learning model to improve software defect prediction. IEEE Trans. Reliab. 70 (2), 481–494.
Chandra, S., Chen, P.M., 2000. Whither generic recovery from application faults? A fault study using open-source software. In: Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on. IEEE, pp. 97–106.
Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357.
Chen, H., Jing, X.-Y., Li, Z., Wu, D., Peng, Y., Huang, Z., 2020. An empirical study on heterogeneous defect prediction approaches. IEEE Trans. Softw. Eng..
Ciancarini, P., Poggi, F., Rossi, D., Sillitti, A., 2016. Improving bug predictions in multicore cyber-physical systems. In: Proceedings of 4th International Conference in Software Engineering for Defence Applications. Springer, pp. 287–295.
Ciancarini, P., Poggi, F., Rossi, D., Sillitti, A., 2017. Analyzing and predicting concurrency bugs in open source systems. In: Neural Networks (IJCNN), 2017 International Joint Conference on. IEEE, pp. 721–728.
Cotroneo, D., Grottke, M., Natella, R., Pietrantuono, R., Trivedi, K.S., 2013a. Fault triggers in open-source software: An experience report. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 178–187.
Cotroneo, D., Natella, R., Pietrantuono, R., 2013b. Predicting aging-related bugs using software complexity metrics. Perform. Eval. 70 (3), 163–178.
Cotroneo, D., Pietrantuono, R., Russo, S., Trivedi, K., 2016. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. J. Syst. Softw. 113, 27–43.
D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. 17 (4–5), 531–577.
Di Nucci, D., Palomba, F., De Rosa, G., Bavota, G., Oliveto, R., De Lucia, A., 2018. A developer centered bug prediction model. IEEE Trans. Softw. Eng. 44 (1), 5–24.
Fonseca, P., Li, C., Singhal, V., Rodrigues, R., 2010. A study of the internal and external effects of concurrency bugs. In: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on. IEEE, pp. 221–230.
Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., Marchand, M., Lempitsky, V., 2016. Domain-adversarial training of neural networks. J. Mach. Learn. Res. 17 (59), 1–35.
Gesi, J., Li, J., Ahmed, I., 2021. An empirical examination of the impact of bias on just-in-time defect prediction. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–12.
Goetz, B., Peierls, T., Lea, D., Bloch, J., Bowbeer, J., Holmes, D., 2006. Java Concurrency in Practice. Pearson Education.
Gong, L., Jiang, S., Bo, L., Jiang, L., Qian, J., 2020. A novel class-imbalance learning approach for both within-project and cross-project defect prediction. IEEE Trans. Reliab. 69 (1), 40–54.
Habib, A., Pradel, M., 2018. Is this class thread-safe? inferring documentation using graph-based learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 41–52.
Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The WEKA data mining software: an update. ACM SIGKDD Explor. Newsl. 11 (1), 10–18.

Herbold, S., 2019. On the costs and profit of software defect prediction. IEEE Trans. Softw. Eng. 47 (11), 2617–2631.

Hosmer, Jr., D.W., Lemeshow, S., Sturdivant, R.X., 2013. Applied Logistic Regression, vol. 398, John Wiley & Sons.

Hosseini, S., Turhan, B., Gunarathna, D., 2019. A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans. Softw. Eng. 45 (2), 111–147.

Huang, Q., Xia, X., Lo, D., 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empir. Softw. Eng. 24, 2823–2862.

Java platform standard edition 7 documentation. URL https://docs.oracle.com/javase/7/docs/.

Keshavarz, H., Rodríguez-Pérez, G., 2024. JITGNN: A deep graph neural network framework for Just-In-Time bug prediction. J. Syst. Softw. 210, 111984.

Lea, D., 2000. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley Professional.

Li, M., Zhang, H., Wu, R., Zhou, Z.-H., 2012. Sample-based software defect prediction with active and semi-supervised learning. Autom. Softw. Eng. 19 (2), 201–230.

Liu, P., Tripp, O., Zhang, C., 2014. Grail: context-aware fixing of concurrency bugs. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 318–329.

Lu, S., Park, S., Seo, E., Zhou, Y., 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. Oper. Syst. Rev. 42 (2), 329–339.

Ma, Y., Luo, G., Zeng, X., Chen, A., 2012. Transfer learning for cross-company software defect prediction. Inf. Softw. Technol. 54 (3), 248–256.

Malhotra, R., 2015. A systematic review of machine learning techniques for software fault prediction. Appl. Soft Comput. 27, 504–518.

Menzies, T., Dekhtyar, A., Distefano, J., Greenwald, J., 2007a. Problems with Precision: A Response to" comments on'data mining static code attributes to learn defect predictors"'. IEEE Trans. Softw. Eng. 33 (9), 637–640.

Menzies, T., Greenwald, J., Frank, A., 2007b. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33 (1), 2–13.

Mockus, A., Nagappan, N., Dinh-Trong, T.T., 2009. Test coverage and post-verification defects: A multiple case study. In: Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on. IEEE, pp. 291–301.

Mohamed, S.H., El-Gorashi, T.E., Elmirghani, J.M., 2019. A survey of big data machine learning applications optimization in cloud data centers and networks. arXiv preprint arXiv:1910.00731.

Musuvathi, M., Qadeer, S., 2007. Iterative context bounding for systematic testing of multithreaded programs. ACM Sigplan Not. 42 (6), 446–455.

Nam, J., 2014. Survey on Software Defect Prediction. Tech. Rep., Department of Compter Science and Engineerning, the Hong Kong University of Science and Technology.

Nam, J., Pan, S.J., Kim, S., 2013. Transfer defect learning. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 382–391.

Pan, S.J., Tsang, I.W., Kwok, J.T., Yang, Q., 2011. Domain adaptation via transfer component analysis. IEEE Trans. Neural Netw. 22 (2), 199–210.

Pan, S.J., Yang, Q., 2010. A survey on transfer learning. IEEE Trans. Knowl. Data Eng. 22 (10), 1345–1359.

Panichella, A., Oliveto, R., De Lucia, A., 2014. Cross-project defect prediction models: L'union fait la force. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on. IEEE, pp. 164–173.

Polato, I., Barbosa, D., Hindle, A., Kon, F., 2015. Hybrid HDFS: Decreasing energy consumption and speeding up hadoop using SSDs. PeerJ PrePr. 3, e1320v1.

Qin, F., Zheng, Z., Bai, C., Qiao, Y., Zhang, Z., Chen, C., 2015. Cross-project aging related bug prediction. In: Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on. IEEE, pp. 43–48.

Qin, F., Zheng, Z., Li, X., Qiao, Y., Trivedi, K.S., 2017. An empirical investigation of fault triggers in android operating system. In: 2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing. PRDC, IEEE, pp. 135–144.

Qin, F., Zheng, Z., Qiao, Y., Trivedi, K.S., 2019. Studying aging-related bug prediction using cross-project models. IEEE Trans. Reliab. 68 (3), 1134–1153.

Qiu, S., Lu, L., Jiang, S., 2019. Joint distribution matching model for distribution–adaptation-based cross-project defect prediction. IET Softw. 13 (5), 393–402.

Qu, Y., Yin, H., 2021. Evaluating network embedding techniques' performances in software bug prediction. Empir. Softw. Eng. 26 (4), 1–44.

Qu, Y., Zheng, Q., Chi, J., Jin, Y., He, A., Cui, D., Zhang, H., Liu, T., 2021. Using k-core decomposition on class dependency networks to improve bug prediction model's practical performance. IEEE Trans. Softw. Eng. 47 (2), 348–366.

Rajapaksha, D., Tantithamthavorn, C., Bergmeir, C., Buntine, W., Jiarpakdee, J., Grundy, J., 2021. SQAPlanner: Generating data-informed software quality improvement plans. IEEE Trans. Softw. Eng..

Ryu, D., Jang, J.-I., Baik, J., 2017. A transfer cost-sensitive boosting approach for cross-project defect prediction. Softw. Qual. J. 25 (1), 235–272.

Scott, A.J., Knott, M., 1974. A cluster analysis method for grouping means in the analysis of variance. Biometrics 507–512.

Song, Q., Guo, Y., Shepperd, M., 2019. A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans. Softw. Eng. 45 (12), 1253–1269.

Song, Q., Jia, Z., Shepperd, M., Ying, S., Liu, J., 2011. A general software defect-proneness prediction framework. IEEE Trans. Softw. Eng. 37 (3), 356–370.

Tantithamthavorn, C., Hassan, A.E., Matsumoto, K., 2020. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. Softw. Eng. 46 (11), 1200–1219.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. 43 (1), 1–18.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2019. The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng. 45 (7), 683–711.

Taylor, R.N., Levine, D.L., Kelly, C.D., 1992. Structural testing of concurrent programs. IEEE Trans. Softw. Eng. 18 (3), 206–215.

TehraniJamsaz, A., Khaleel, M., Akbari, R., Jannesari, A., 2021. Deeprace: A learning-based data race detector. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, pp. 226–233.

Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J., 2009. On the relative value of cross-company and within-company data for defect prediction. Empir. Softw. Eng. 14 (5), 540–578.

Wang, S., Yao, X., 2013. Using class imbalance learning for software defect prediction. IEEE Trans. Reliab. 62 (2), 434–443.

Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biom. Bull. 1 (6), 80–83.

Wu, F., Jing, X.-Y., Sun, Y., Sun, J., Huang, L., Cui, F., Sun, Y., 2018. Cross-project and within-project semisupervised software defect prediction: A unified approach. IEEE Trans. Reliab. 67 (2), 581–597.

Xu, Z., Pang, S., Zhang, T., Luo, X.-P., Liu, J., Tang, Y.-T., Yu, X., Xue, L., 2019. Cross project defect prediction via balanced distribution adaptation based transfer learning. J. Comput. Sci. Tech. 34 (5), 1039–1062.

Yang, Y., Xia, X., Lo, D., Bi, T., Grundy, J., Yang, X., 2022. Predictive models in software engineering: Challenges and opportunities. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (3), 1–72.

Yu, T., Wen, W., Han, X., Hayes, J.H., 2019. ConPredictor: Concurrency defect prediction in real-world applications. IEEE Trans. Softw. Eng. 45 (6), 558–575.

Zhang, F., Mockus, A., Zou, Y., Khomh, F., Hassan, A.E., 2013. How does context affect the distribution of software maintainability metrics? In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 350–359.

Zhao, K., Xu, Z., Zhang, T., Tang, Y., Yan, M., 2021. Simplified deep forest model based just-in-time defect prediction for android mobile apps. IEEE Trans. Reliab. 70 (2), 848–859.

Zhou, B., Neamtiu, I., Gupta, R., 2015. Predicting concurrency bugs: how many, what kind and where are they? In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering. ACM, pp. 1–10.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B., 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM, pp. 91–100.

Fangyun Qin received the Ph.D degree in 2020 from Beihang University, Beijing, China. In 2017, she was with the Department of Electrical and Computer Engineering, Duke University as a visiting scholar. She is currently a lecturer in Capital Normal University, Beijing, China. Her research interests include software bug prediction, software engineering, software reliability and machine learning.

Zheng Zheng received the Ph.D. degree in 2006 from Chinese Academy of Science, Beijing, China. In 2014, he was with the Department of Electrical and Computer Engineering, Duke University as a research scholar. He is currently a professor at Beihang University, Beijing, China. His research interests include software engineering, software dependability modeling and software fault localization.

Yulei Sui received the Ph.D degree in 2014 from the University of New South Wales, Sydney, Australia. From 2014 to 2023, he was on the Faculty at the University of Technology Sydney, Sydney, Australia. He is currently an associate professor at University of New South Wales, Sydney, Australia. His research interests include program analysis, secure software engineering and machine learning.

Siqian Gong received the Ph.D. degree in 2019 from Beihang University, Beijing, China. In 2017, she was with Arizona State University as a research scholar. She is currently an associate professor in Beijing Jiaotong University, Beijing, China. Her research interests include machine learning, allocation in cloud computing and control theory and technology of intelligent manufacturing.

Zhiping Shi received the Ph.D. degree in 2005 from Chinese Academy of Science, Beijing, China. From 2005 to 2010, he was on the Faculty at the Institute of Computing Technology, Chinese Academy of Science. He is currently a professor at Capital Normal University, Beijing, China. His research interests include machine learning, formal verification and visual information analysis.

Kishor S. Trivedi received the Ph.D. degree in 1974 from the University of Illinois at Urbana-Champaign, Champaign, IL, USA. He is the Fitzgerald Hudson Chair with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA. Since 1975, he has been on the Duke faculty. He is the author of a well-known text entitled Probability and Statistics with Reliability, Queuing and Computer Science Applications (Prentice-Hall, 1982); He has authored or coauthored three other books entitled Performance and Reliability Analysis of Computer Systems (Kluwer, 1996), Queueing Networks and Markov Chains (Wiley, 1998), and Reliability and Availability Engineering (Cambridge Univ. Press, 2017). His research interests include reliability, availability, performance, and survivability of computer and communication systems, and software dependability.