2023 IEEE/ACM International Conference on Automation of Software Test (AST) | 979-8-3503-2402-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/AST58925.2023.00023

Test Case Prioritization using Transfer Learning in Continuous Integration Environments

Rezwana Mamata*, Akramul Azim*, Ramiro Liscano*, Kevin Smith†, Yee-Kang Chang†,
Gkerta Seferi†, and Qasim Tauseef†
**Ontario Tech University*, Oshawa, Ontario, Canada
{rezwana.mamata, akramul.azim, ramiro.liscano}@ontariotechu.ca
†*International Business Machines Corporation (IBM)*, United Kingdom
smithk6@uk.ibm.com, YeeKang.Chang@ibm.com, Gkerta.Seferi@ibm.com, Qasim.Tauseef@ibm.com

***Abstract*—The continuous Integration (CI) process runs a large set of automated test cases to verify software builds. The testing phase in the CI systems has timing constraints to ensure software quality without significantly delaying the CI builds. Therefore, CI requires efficient testing techniques such as Test Case Prioritization (TCP) to run faulty test cases with priority. Recent research studies on TCP utilize different Machine Learning (ML) methods to adopt the dynamic and complex nature of CI. However, the performance of ML for TCP may decrease for a low volume of data and less failure rate, whereas using existing data with similar patterns from other domains can be valuable. We formulate this as a transfer learning (TL) problem. TL has proven to be beneficial for many real-world applications where source domains have plenty of data, but the target domains have a scarcity of it. Therefore, this research investigates leveraging the benefit of transfer learning for test case prioritization (TCP). However, only some industrial CI datasets are publicly available due to data privacy protection regulations. In such cases, model-based transfer learning is a potential solution to share knowledge among different projects without revealing data to other stakeholders. This paper applies *TransBoost*, a tree-kernel-based TL algorithm, to evaluate the TL approach for 24 study subjects and identify potential source datasets.**

***Index Terms*—Transfer Learning (TL), Test Case Prioritization (TCP), Continuous Integration (CI).**

I. Introduction

Testing is an essential part of software development and highly impacts the overall development costs of a project. Continuous Integration (CI) automates compiling, building, and testing phase (CI cycle) of a software project to minimize integration problems and development time. In CI, developers frequently integrate their code changes with the mainline codebase, which activates a CI cycle, and after the compilation of each cycle, developers receive feedback. The compilation time of each cycle depends on the volume of tests, which may vary based on the codebase size. A large volume of tests requires significant computational resources and time to execute, which will delay the CI builds and prolong developers' feedback time.

Test Case Prioritization (TCP) is a widely adopted technique to improve the overall performance (e.g., handling the long execution time) of software testing in CI. In TCP, the test cases are reordered based on priorities instead of removing any test cases during the test case execution. In the literature, two types of TCP are available (e.g., heuristic-based and ML-based techniques). Heuristic-based TCP often works with a single feature (either analyze code coverage or test execution history). Unlike heuristic-based TCP, ML-based TCP operates on a set of features collected from various resources and can adapt to new changes (e.g., retraining or incremental learning). Therefore, recent studies have employed different ML techniques for TCP to handle CI's dynamic and complex nature.

The ML models utilize different data sources, such as CI test execution history [1], code coverage information [2], test dependency information [3], version control system (VCS) metadata [4]. Collecting the coverage and test dependency information requires white-box program analysis and is often too costly (e.g., feature extraction time) in large-scale codebases with rapid CI testing [1]. During the software development process, CI test execution history and VCS metadata generate automatically. These pieces of information are easily accessible and inexpensive (require little time for feature extraction) [4]. In addition, the CI and VCS metadata are not dependent on software types (e.g., mobile or web applications) or language (e.g., java, c, or PHP). Therefore, the knowledge of ML models trained with VCS and CI metadata can be transferred to other models, which may be beneficial for better test case prioritization.

Only a few existing datasets [4], [5] include some unique features extracted from both the VCS and CI metadata for TCP. However, most projects have a small volume of test execution history, which may be inadequate for effective TCP. For instance, Daniel et al. [4] publish a dataset that contains both VCS and CI data from 20 open-source projects and three industrial projects. The open-source projects have less than 1 million test execution histories except one, and among three industrial projects, two are large-scale with around 3.5M test execution histories. Some other large-scale datasets are available, but they avoid the inclusion of Github data (e.g., Google Shared Dataset of Test suite Results (GSDTSR) [1]).

The test case prioritization task becomes challenging for traditional ML algorithms because of fewer test suite failures (e.g., industrial projects) or low volumes of data (e.g., new projects). Transfer learning (TL) algorithms can be a possible solution to solve the problems of training data scarcity in the target domain [6]. TL enables tasks to adapt quickly to unknown situations, environments, and tasks by attaining

979-8-3503-2402-0/23/$31.00 ©2023 IEEE
DOI 10.1109/AST58925.2023.00023

knowledge from a related task [7]. Hence, it necessitates the improvement of TCP for unbalanced low-volume software projects by leveraging transfer learning.

Based on domain similarity, transfer learning algorithms can be categorized into two main classes, e.g., homogeneous transfer learning and heterogeneous transfer learning. The homogeneous transfer learning algorithms consider that the source and target domain have the same feature space, but the data distribution between these two domains is different. Homogeneous transfer learning algorithms try to minimize that difference. On the other hand, heterogeneous transfer learning algorithms assume that the source and target domain have different feature spaces. In our case, the source and target domain have the same feature space; thus, we consider the homogeneous transfer learning algorithms for test suite failure prediction. The homogeneous transfer learning algorithms can be divided into five subcategories (i.e., instance-based, feature-based, relational-knowledge-based, and hybrid). Software companies are unwilling to leak any testing information to their competitors. Therefore, model-based transfer learning is the only solution, instead of other approaches. Moreover, the model-based methods also work well when the storage capacity and transmission throughout the target domain are limited.

Therefore, this paper presents a model-based transfer learning approach for TCP, considering both VCS and CI metadata. The proposed approach applies *TransBoost* [8], a kernel-based transfer learning algorithm, and we refer to this technique as *TCP_TB*. This approach is evaluated with 24 study subjects and compared its performance with ten commonly used supervised ML algorithms. The experimental results demonstrate the effectiveness of using *TCP_TB* by improving the mean APFD value on average by 2.82% and 5.04% for large-scale and small-scale projects, respectively. In addition, we define the general criteria for selecting the source dataset for smaller projects and propose internal domain knowledge transfer for large-scale datasets as they have sufficient historical data.

The novelty of this work lies in using the TL method for test case prioritization employing both GitHub and CI data. In addition, our study comprehensively compares different CI and GitHub datasets. It helps to identify which source datasets have the potential for transferring knowledge to improve TCP performance. Moreover, it explores the applicability of internal domain knowledge to prioritize test cases for large-scale projects.

The remainder of this paper is organized as follows. We discussed the related work in Section II. Section III illustrates the background of this work. The detail of our proposed approach is explained in Section IV. The results and analysis of various experiments are demonstrated in Section V. Finally, Section VII concludes the paper.

II. Related Work

Prior studies describe test case prioritization (TCP) as a ranking problem. To solve this problem, three different ranking models [9] (i.e., pointwise, pairwise, and listwise ranking) are commonly available. The pointwise ranking model considers a single test case and applies a prediction model to determine the priority score for this test case to fail. Then, it sorts the test cases based on priority value to get the final ranking. The pairwise ranking model takes a pair of test cases simultaneously and assigns a priority score. After that, it takes all the pairs, compares them with the ground truth, and determines an optimal ordering for all test cases. Unlike pointwise and pairwise, the listwise model evaluates a list of test cases simultaneously and ranks each test case compared with others.

Bertolino et al. [10], Lachmann et al. [11], and Tonella et al. [12] use pairwise ranking models for TCP. Bertolino et al. [10] evaluate the performance of Random Forest (RF), Multiple Additive Regression Tree (MART), L-MART, RankBoost, RankNet, and Coordinate ASCENT (CA) for TCP, and MART achieves better performance among these models. Lachmann et al. [11] consider SVM Rank [13], capable of handling large input vectors and, based on training inputs, returns a ranked classification function. Tonella et al. [12] apply the Rankboost algorithm for TCP, utilizing statement coverage, cyclomatic complexity, and developers' ranking scores for model training.

On the othe hand, Busjaeger and Xie [14] use the listwise ranking model SVM MAP for TCP. Many papers consider a pointwise ranking algorithm mainly XGBoost [15], Recurrent Neural Network (RNN) [16], Neural Network (NN) [17], K Nearest Neighbour (KNN) [18], Bayesian Network [19], Logistic Regression [20], [21], and SVM [22] for TCP. In another work [23], the authors employ natural language processing (NLP) for software requirement analysis concentrating on test case prioritization.

Some recent studies [24], [25] investigate reinforcement learning (RL) techniques for TCP, proving that RL models performed better than baseline models. To the best of our knowledge, only one study by Rosenbauer [6] introduces transfer learning for test case Prioritization. The authors of this work use the XCSF algorithm, a reinforcement learning-based classifier system for transferring knowledge. They design a simplistic population transformation and evaluate their approach with four study subjects considering CI metadata. Table I lists existing techniques used for test case prioritization.

Both RL and Supervised Learning (SL) deliver good performance for TCP. However, RL may not be the most suitable approach for TCP as it has well-defined outcomes and goals. Instead, SL methods such as gradient-boosting trees, random forests, or neural networks may be more appropriate for the TCP problem. Moreover, the time complexity of SL algorithms is generally lower than RL algorithms. It is important to note that the time complexity highly depends on the choice of algorithms and data volume.

Pan et al. [26] deliver a systematic literature review of ML-based TCP and test case selection (TCS). The paper reviews 29 research papers published from 2006 to 2020 and addresses the variation in the machine learning models, feature sets, evaluation metrics, and the earlier work's reproducibility. It is shown that most of the prior studies have used SL algorithms. Moreover, Researchers explore various information resources

	#Projects	Approach
Elbaum et al. [1]	1	Heuristic
Zhu et al. [27]	2	Heuristic
Liang et al. [28]	2	Heuristic
Mattis et al. [5]	20	Heuristic
Busjaeger et al. [14]	1	SL & NLP
Lachmann et al. [11]	2	SL(e.g.,SVM rank) & NLP
Bertolino et al. [10]	6	RL & SL
Noor et al. [20]	5	SL (e.g.,LR)
Chen et al. [15]	50	SL(e.g.,XGBoost)
Mahdieh et al. [17]	5	SL(e.g.,Neural Network)
Hasnain et al. [16]	5	SL(e.g., RNN)
Palma et al. [21]	5	SL(e.g., LR)
Elsner et al. [4]	23	Heuristic and SL
Pan et al. [29]	242	SL(e.g., Light GBM)
Spieker et al. [24]	3	RL
Bagherzadeh et al. [25]	8	RL
Shi et al. [30]	3	RL
Carlson et al. [31]	1	UL
Rosenbauer et al. [32]	3	RL
Rosenbauer et al. [6]	4	TL

TABLE I
COMPARISON OF EXISTING WORKS ON TCP

for both Heuristic and ML-based TCP. For example, test traces, build dependencies, test coverage, version control systems metadata, and test histories are different sources for collecting data. A study by Elsner et al. [4] reports that CI and GitHub data are easily accessible and inexpensive. Therefore, we select *TransBoost* [8], a transfer learning algorithm that integrates the XGBoost classifier (SP algorithm) for TCP, considering VCS and CI metadata.

III. BACKGROUND

This section explains how TL-based test case prioritization (TCP) suits the continuous integration (CI) environment. Then we discuss the significance of model-based transfer learning algorithms in TCP. Finally, a brief introduction to the *TransBoost*, a boosting tree kernel-based transfer learning algorithm [8]

A. TL-based Test Case Prioritization in CI

Most software companies operate in continuous integration (CI) environment, which runs automated regression test cases to ensure software quality. Test case prioritization (TCP) reorders tests to locate faults as early as possible. Machine learning, ML-based approaches are showing promising outcomes in solving test case prioritization problems in CI because of their dynamic adaptability. However, in some instances, they face challenges in creating usable models because of the limited available data of newer projects. Transfer learning (TL), a type of ML algorithm, solves these problems (e.g., low volume of data, class imbalance). It leverages knowledge from a source domain with much data to create a usable model for the target domain.

Every software runs a different set of tests to examine code modifications. Even though these tests are different, they have some common failure characteristics because of the same testing environment. For instance, if a test is failing more in the recent CI cycles, it has a high probability of failure in the next CI cycle. These statistical features of test failures are independent of software types (e.g., web application or car application) and languages(e.g., java or C++). Therefore, these features can be shared for TCP across different software projects to a certain degree.

The giant software companies want to keep the failure information of their project private. Failure information from these projects can help to build efficient testing models for newer projects. Thus, only the model-based TLs are applicable to the above scenario, where the source domain is referential, but the data points are not available to maintain privacy.

B. TransBoost

Homogenous transfer learning algorithms consider the source and the target domain to have the same feature space, X_s=X_t but different data distributions (e.g., marginal or conditional). For instance, the number of test runs in one project for each CI cycle may vary from others. Thus, they will have different marginal distributions. Minimizing these distribution discrepancies across domains is a crucial task for transfer learning.

This paper selects TransBoost [8], a tree-kernel-based transfer learning algorithm. This algorithm is specially designed considering commercial tabular datasets and has proven efficient, robust, and interpretable for real-world transfer learning applications. TransBoos algorithm leverages the instances of developed products to enhance the performance of the newer target domain. facefaThe working process of the TransBoost algorithm is illustrated in Figure 1. TransBoost generates two parallel boosting-tree models with identical tree structures but different node weights. This specific design keeps the merit of tree-based models robust and interpretable. This algorithm considers gradient-boosting trees as a base learner, which proved to be a very effective ML algorithm. It trains models and reduces the discrepancy between domains simultaneously within $O(n)$ complexity compared to the traditional kernel method.

IV. PROPOSED APPROACH

We propose *TCP_TB*, which aims to achieve better test prioritization in the CI environment using transfer learning (TL). This approach considers both VCS (version control system) and CI metadata. First, let us define the problem and notations; before discussing details about the proposed approach. Let S be a system under test, and a code change, C is pushed to the codebase to construct S', and T_s be a set of test suites. For each test suite, $T \in T_s$, the transfer learning model, M, calculates the failure probability value

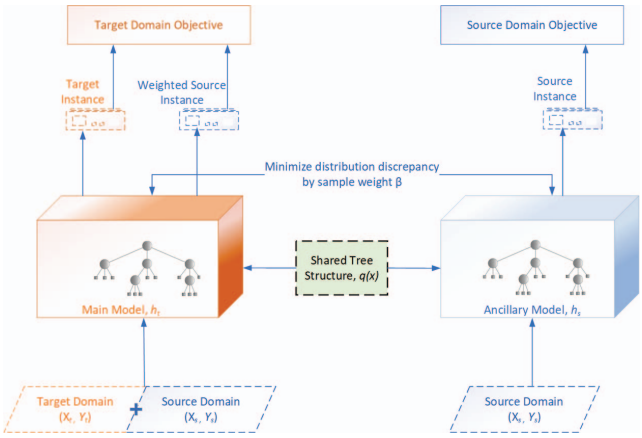

Fig. 1. TranBoost

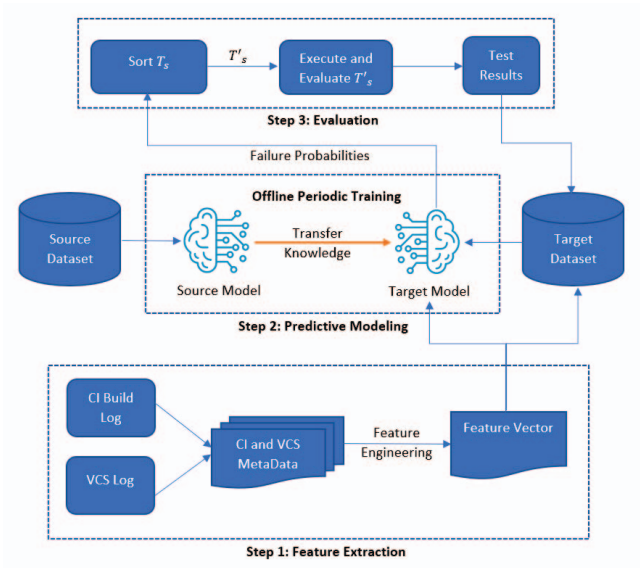

Fig. 2. System model of our proposed approach

based on a feature vector, F. Then, the probability values are sorted in descending order to rank test suites in T_s. Depending on the test budget, the complete prioritized list (TCP), T'_s or a subset of it (TCS) will be executed to evaluate S' [4], [14], [33]. After evaluating S', the feature vector and test results are added to the dataset as historical data. The system model of our proposed approach is shown in Figure 2, and it consists of three successive steps, i.e., feature extraction, predictive modeling, and evaluation. The details of these steps are discussed in the following subsections.

A. Feature Extraction

Software projects that follow CI practice use VCS to keep track of every modification to the codebase and store them in a database. Different types of VCS are available, such as distributed (e.g., Git, Bazaar) or centralized (e.g., Microsoft Team Foundation Server, Apache Subversion). However, they share the same concept of a commit (i.e., a code modification pushed by a single author). Each commit will have the following information: Commit identifier, author, commit timestamp, message, and a changeset(e.g., number of files added, modified, or deleted) stored in a VCS log.

Simultaneously, CI systems generate a log after the compilation of each CI cycle (building, testing, and deploying). The CI log contains the following information, e.g., build identifier, build result, build duration, test identifier, number of assertions in a test, test result, and test duration. The testing frameworks in CI systems (or plugins) often generate structured test logs in various output formats. However, through regular expressions, the testing and building information can be extracted from raw textual CI logs [4], [5]. Based on the configuration, a CI cycle can be triggered either after a code change push or after the previous cycle. The metadata of CI and VCS changeset is not software language (e.g., Java or python) or type (e.g., web application or car software) specific. Therefore, the knowledge gained by an ML model using features from CI and VCS metadata can be transferred to other ML models. We extract nine CI and three VCS features for *TCP_TB* from the collected metadata. Similar to prior studies [4], we provide the definition of each feature is given below:

CI Features:

- Test_Identifier: Unique integer value to represent each test suite.
- Execution_Count: Integer represents the test suite execution number.
- Test_Total: Integer Represents the number of test cases in the test suite.
- Failure_Count: Total number of times a test suite failed.
- Last_Failure_Age: The amount of CI builds since the last failure.
- Last_Failure: The boolean value indicates whether the test suite has failed in the last build or not.
- Failure_Rate: The percentage of test suite failure in the previous run.
- Transition_Count: The total number of times a test suite changed its state (e.g., pass to fail or fail to pass).
- Last_Transition: The amount of CI builds since the last transition.
- Failed: The boolean value indicates whether the test suite has failed or passed.

VCS Features:

- #Files_Changed: The number of files changed because of a specific change.
- #Lines_Inserted: The lines of code added in all modified files because of a specific change.
- #Lines_Deleted: The lines of code deleted in all modified files because of a specific change.

B. Predictive Modeling

The most straightforward way to create a predictive transfer model is to train a model on the source dataset and then transfer it to the target domain for further training according to the target dataset. The ultimate model inherits the source

domain informative patterns and fine-tunes them with the target data points. If the data distribution in the source and target domain differs, then directly sharing the model can be ineffective due to the data drifting in the previous analysis. Therefore this paper selects *TransBoost*, which uses XGBoost to generate two parallel boosting-tree models (e.g., source and target) with identical tree structures but different node weights. The details of *TransBoost* are discussed in section III-B.

The source model trains with a source dataset with a comparatively large data volume, a high test failure rate, or both and then shares its knowledge with the target model for which we want to improve the TCP. The performance of the predictive TL model decreases with time; however, retraining them in each cycle can cause computational overhead to the CI build. Therefore, we periodically retrain the TL model in an offline environment to maintain the TL performance and avoid computational overhead. A feature vector generated from historical CI test data and VCS changeset passes to the pre-trained TL model as an input matrix to predict the failure probability score of the current test set. The failure probabilities range from 0 to 1, and a probability value close to 1 indicates a higher probability of failure. Then the list of probability scores is sorted in descending order.

When training a predictive transfer learning (TL) model, the first challenge is the selection of the source dataset. In natural language processing and computer vision, the standard practice is to select a pre-trained model that performs sensibly well on a large dataset [34], [35]. Hence, this paper selects projects with at least 100k test execution from Table II as a possible source. To identify the best source dataset, we train each selected dataset as a source of the *TCP_TB* and apply it to the remaining datasets. This paper also attempts internal domain knowledge transfer by dividing the training dataset into source and target. *TCP_TB* uses the TranBoost algorithm, which is built upon the source code of the XGBoost. Thus, the TransBoost and XGBoost have the same hyperparameters; however, TransBoost has six additional parameters (i.e., transfer_decay_ratio, transfer_rebalance, transfer_margin_estimation, transfer_min_leaf_size, transfer_prior_margin, transfer_velocity), particularly for transfer learning. This paper uses the default hyperparameter of TransBoost for *TCP_TB*. The algorithm 1 represents the working algorithm of *TCP_TB*.

Algorithm 1: Test prioritization model, TCP_TB

Input: T_s, E_t , C, $model$, $\{D_S, D_T\}$.
Result: Prioritized test execution order, T'_s.
T_s = List of test suites;
E_t = Test execution history;
C = code change informations;
model = $TransBoost()$;
source and target dataset = $\{D_S, D_T\}$;
Function `feature_extraction`(T_s, E_t, C)**:**
  for $i = 0$; $i < length(T_s)$; i++ **do**
    $f_s[i] \leftarrow E_i$;
  end
  $f_s \leftarrow C$;
  return f_s;
feature_vector, f_s = feature_extraction(T_s, E_t , C) ;
Function `predictive_model`$(T_s, TransBoost, f_s)$**:**
  while $(retraining == True)$ **do**
    clf = $TransBoost()$;
    $model$ = $clf.fit(D_S, D_T)$;
  end
  $t\_prediction$=$model.predict(f_s)$;
  $T'_s = sort(t\_prediction, ascending = false)$;
  return T'_s;

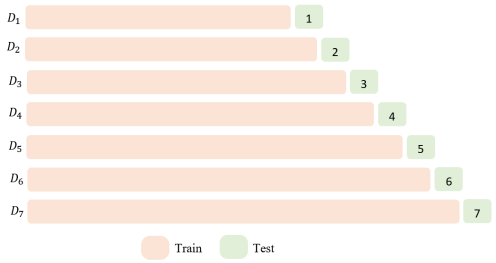

Fig. 3. Split dataset, D using *forward-chaining*

C. Evaluation

1) Training and Test Splits: There is no generic way to determine a dataset's training-testing split ratio or the volume of training data that would be sufficient to prepare a suitable model. For example, one study [36] utilized one-year data for training the model and tested it with the last two months of data. The performance of the predictive model decays with time; thus, it requires a periodic update. This paper does periodic offline training and, thus, uses all the available historical data for training. Because of the temporal dependency between CI cycles, it is not reasonable to perform k-fold cross-validation. Therefore, it is more realistic to train models on past data and test on more recent data [36]. Figure 3 displays the seven training testing splits this paper chooses to divide a time-series dataset, D. It follows the *forward chaining* method for splitting, which does the cross-validation on a rolling basis. First, it takes a small subset of data for training and predicts the probability of failure for the following data points. Then the same predicted data points are incorporated into the following training dataset, and subsequent data points are tested. Except for the *jetty.project*, we apply month *forward chaining* for the last seven months of each project for cross-validation. As the *jetty.project* has only 63 days of information available, we apply day *forward chaining* for the last seven days of it for cross-validation.

2) Evaluation Metric: The datasets are imbalanced; therefore, accuracy is not an efficient way to measure the prediction results. Also, Recall, F-measure, AUC, and MCC metrics are well for prediction model evaluation. However, they are not appropriate for test case prioritization. The average percentage of fault detected (APFD) [33] is a widely accepted metric for

Project	Time Period (days)	CI Cycles	#Tests	#Test Executions	#Test Failures	#Test Failure Rate
SonarQube	532	4286	3122	6696.0K	2,156	0.03%
Open_liberty	1168	6124	1158	4571.5K	17583	0.38%
IVU_Cpp	267	3996	1240	3608.4K	25,973	0.72%
IVU_Java_2	699	3209	1278	3526.3K	7,603	0.22%
graylog2-server	1381	3891	250	798.5K	403	0.05%
buck	307	846	864	586.1K	1,511	0.26%
jcabi-github	872	809	201	398.1K	740	0.19%
cloudify	909	4973	116	283.6K	602	0.21%
sling	213	1403	304	268.1K	1,158	0.43%
okhttp	1423	3412	266	236.5K	939	0.40%
IVU_Java_1	313	943	279	178.7K	14,568	8.15%
Achilles	1114	642	627	139.9K	162	0.12%
DSpace	1043	1929	83	122.1K	1,697	1.39%
jsprit	368	267	107	91.8K	123	0.13%
jOOQ	961	1318	51	81.5K	573	0.70%
dynjs	1163	385	83	68.5K	496	0.72%
jetty.project	63	192	787	63.9K	415	0.65%
optiq	395	458	63	55.3K	110	0.20%
HikariCP	661	1575	23	44.0K	383	0.87%
titan	747	384	107	43.3K	551	1.27%
wicket-bootstrap	1245	904	91	41.4K	9007	21.76%
jade4j	1539	358	43	35.9K	1323	3.69%
deeplearning4j	727	982	174	14.6K	908	6.22%
LittleProxy	1580	271	50	11.0K	172	1.56%

TABLE II
OVERVIEW OF THE STUDY SUBJECTS

Test case prioritization evaluation. The information regarding fault-to-failure mapping needs to be included in calculating APFD based on faults. In the real world, we only have failures, i.e., failing tests. Therefore, we entirely focus on detecting failures rather than faults, considering a one-to-one mapping as proposed by previous research works [4], [5]. APFD indicates how quickly a set of prioritized test list T'_s can detect the faults present in the system under test, and its value is measured from the weighted average of the percentage of detected faults. APFD values range from zero to one. Higher values indicate that the faults are detected faster using fewer test cases. In addition, we consider the normalized version of APFD, NAPFD [37], to measure the effectiveness of *TCP_TB* for test case selection (TCS).

V. EXPERIMENTAL RESULTS AND ANALYSIS

To evaluate our proposed approach *TCP_TB*, we conduct several experiments and illustrate the experimental results addressing the following research questions:

- *RQ1:* What should be the criteria for choosing a source dataset?
- *RQ2:* Can internal domain knowledge transfer increase the TCP effectiveness for large-scale datasets?
- *RQ3:* How well *TCP_TB* perform in terms of TCP effectiveness?

A. Study Subjects

To evaluate *TCP_TB*, we utilize the open_liberty and 23 other publicly available datasets (e.g., 20 open source and three industrial). The open_liberty dataset is provided by International Business Machines Corporation (IBM). It is primarily written in Java and provides an open framework for developing cloud-based applications and microservices. The other 23 projects are collected from [4]. The details of all these 24 projects are listed in Table II in descending order based on the number of test execution. We remove *jOOQ* from our experiment as it has no failure in the last seven months of the CI cycle. Table II shows the number of CI cycles, tests, test executions, test failures, test failure rate, and data collection period for each project. The wicket-bootstrap project has the highest test failure rate (21.76%), while the *SonarQube* project has the lowest test failure rate (0.03%) but the highest number of test execution. However, the *open_liberty* dataset has the highest number of CI cycles (6124) with 0.38% test failure rate.

	CI-RTP/S	TCP_ML (F_{new})	TCP_TB (TransBoost, F_{new})
SonarQube	0.6330 ± 0.3348 (F_2, LR)	0.9431 ± 0.1220 (XGB)	$\mathbf{0.9522 \pm 0.1044}$ (S_2)
open_liberty	–	0.9217 ± 0.1304 (XGB)	$\mathbf{0.9270 \pm 0.1169}$ (S_5)
IVU_Cpp	0.9768 ± 0.0783 (F_{all}, RF)	0.9781 ± 0.0697 (XGB)	$\mathbf{0.9823 \pm 0.0588}$ (S_{10})
IVU_Java_2	0.9606 ± 0.1384 (F_{all}, RF)	0.9609 ± 0.1217 (XGB)	$\mathbf{0.9676 \pm 0.1099}$ (S_3)
graylog2-server	$\mathbf{0.9832 \pm 0.0003}$ ($f_{3,3}$, H)	0.9618 ± 0.0329 (XGB)	$\mathbf{0.9832 \pm 0.0003}$ (S_{all})
buck	0.5981 ± 0.1916 ($f_{1,5}$, H)	0.9806 ± 0.0807 (RF)	$\mathbf{0.9834 \pm 0.0800}$ (S_2)
jcabi-github	0.8640 ± 0.2056 (F_1, LR)	0.8533 ± 0.2118 (XGB)	$\mathbf{0.8935 \pm 0.1743}$ (S_{10})
cloudify	$\mathbf{0.9921 \pm 0.0000}$ (F_{all}, GBM)	0.9286 ± 0.0000 (RF)	0.9286 ± 0.0000 (S_4)
sling	0.9243 ± 0.2333 (F_{all}, GBM)	0.9845 ± 0.0450 (XGB)	$\mathbf{0.9885 \pm 0.0457}$ (S_4)
okhttp	0.8571 ± 0.2689 (F_2, MLP)	0.8837 ± 0.2099 (GBM)	$\mathbf{0.9509 \pm 0.0475}$ (S_6)
IVU_Java_1	0.8720 ± 0.1646 ($f_{1,2}$, H)	0.8785 ± 0.1747 (Light-GBM)	$\mathbf{0.8882 \pm 0.1642}$ (S_2)
Achilles	0.8452 ± 0.0000 ($f_{3,1}$, H)	0.7024 ± 0.0000 (GBM)	$\mathbf{0.9643 \pm 0.0000}$ (S_9)
DSpace	0.4186 ± 0.2956 (F_1, GBM)	0.7565 ± 0.2778 (GBM)	$\mathbf{0.7653 \pm 0.2225}$ (S_{13})
jsprit	0.7259 ± 0.2489 (F_{all}, LR)	0.6736 ± 0.2386 (NB)	$\mathbf{0.7649 \pm 0.1679}$ (S_{10})
dynjs	0.0094 ± 0.0000 ($f_{3,3}$, H)	$\mathbf{0.9762 \pm 0.00}$ (GBM)	0.9733 ± 0.00 (S_{11})
jetty.project	0.8308 ± 0.2556 (F_{all}, LR)	0.9715 ± 0.0765 (XGB)	$\mathbf{0.9867 \pm 0.0267}$ (S_{11})
optiq	0.6615 ± 0.1813 ($f_{1,4}$, H)	$\mathbf{0.8885 \pm 0.1684}$ (RF)	0.8815 ± 0.2061 (S_{11})
HikariCP	0.4232 ± 0.1563 (F_4, GBM)	0.5770 ± 0.2483 (RF)	$\mathbf{0.6520 \pm 0.2342}$ (S_7)
titan	0.7838 ± 0.2643 (F_1, LR)	0.8856 ± 0.1379 (GBM)	$\mathbf{0.9327 \pm 0.0776}$ (S_2)
wicket-bootstrap	0.8032 ± 0.2666 ($f_{1,2}$, H)	0.8607 ± 0.1855 (XGB)	$\mathbf{0.9754 \pm 0.0232}$ (S_{13})
jade4j	0.7093 ± 0.2146 ($f_{3,3}$, H)	0.6598 ± 0.2394 (DT)	$\mathbf{0.7791 \pm 0.2422}$ (S_4)
deeplearning4j	0.8139 ± 0.1924 ($f_{1,1}$, H)	0.8669 ± 0.1769 (RF)	$\mathbf{0.8969 \pm 0.1503}$ (S_3)
LittleProxy	0.6858 ± 0.2874 (F_2, RF)	0.6708 ± 0.1804 (NB)	$\mathbf{0.8092 \pm 0.2365}$ (S_{13})

TABLE III
MEAN AND STANDARD DEVIATION APFD VALUES OF *CI-RTP/S*, *TCP_ML*, AND *TCP_TB*. THE HIGHEST VALUES ARE HIGHLIGHTED IN BOLD

B. Data Analysis

This paper works with homogeneous transfer learning; thus, it is essential to format all 24 datasets, in the same way to maintain consistency in the number and type of features across all the study subjects. We extract 12 features from the CI test execution history and VCS metadata of *open_liberty* project. The descriptions of these features are provided in the section IV-A. Seven (i.e., Test_Identifier, Execution_Count, Last_Failure, Failure_Rate, Test_Total, #Lines_Inserted, #Lines_Deleted) of these features are missing in the other datasets. The Last_Failure feature in other datasets calculates the number of CI cycles since the test last failed. We mention this feature as Last_Failure_Age. Using the Test_Identifier_string, run_id, run_timestamp, and Failed columns, we are able to extract Test_Identifier, Execution_Count, Last_Failure, Failure_Rate features. However, no information is available regarding the number of test cases in each test suite and the number of code lines added or removed from the changeset. Thus, there is no way to get pre-

cise information regarding these Test_Total, #Lines_Inserted, #Lines_Deleted. From our observation on the open_liberty dataset, we find that these features significantly impact ML model performance. We create random synthetic data points for Test_Total (range: 1-1500), #Lines_Inserted (range: 0-2000), #Lines_Deleted (range: 0-2000) features using *randint* from *numpy* library to balance the number of features among all the datasets.

C. Selection of Efficient Source Datasets for TransBoost

In this experiment, we determine the best source datasets to develop a pre-trained source model for TransBoost. In practice, the common assumption is that the source model trained with a large-scale dataset will be the best source model for transferring knowledge. This paper selects the top 13 projects from Table II with more than 100k test execution volume as potential source datasets. A source dataset will be efficient if it improves the TCP performance of *Transboost* compared to ML models. Therefore, we evaluate ten popular ML models (i.e., decision tree (DT), random forest (RF), gradient boosting tree(GBT), Light-GBM, XGBoost (XGB), logistic regression(LR), naive bayes (NB), k-nearest neighbors, multilayer perceptron (MLP) and support vector machine (SVC)) on each study subjects with the feature set (f_{new}) mentioned in Section IV-A. These ML models have been employed by previous studies at least once. We use the default hyperparameter for running these algorithms, similar to [4]. The best ML results for each study subject are stored in Table III. Figure 4 shows the performance of source datasets for 23 study subjects. We calculate the difference of mean APFD values of *TCP_TB* model with the best ML model (*TCP_ML*). The green color represents the number of study subjects who received an increment, while the orange indicates a decrement in the mean APFD value compared to *TCP_ML*. The results with no change are represented with gray. *SonarQube* is the largest dataset among all the study subjects; it has a 6.6M test execution history. According to the common assumption, a model pre-trained on *SonarQube* should perform better. However, the source model trained with *SonarQube* improves the mean APFD of 13 projects while decreasing for nine others. We are assuming that even if the *SonarQube* has the highest volume of test execution data, it could not be a good data source because of its low test failure rate. The subsequent dataset (i.e., *open_liberty*, achieves the highest performance by enhancing the TCP performance of 16 projects. However, it diminishes the TCP performance of six projects. Six source datasets achieve the second-best position by improving the TCP performance of 15 projects. All these projects have a comparatively high test failure rate and a good volume of data. *Jacabi-github* and *achilles* both these datasets performed poorly as source. The test failure rates of these two datasets are 0.19% and 0.12%, respectively, which is low compared to other sources. Based on the above discussion, we can assume that a balance between test execution volume and the failure rate is required to be a promising source dataset. This answers our first research question *RQ1*, where test failure rate and volume are equally crucial for choosing a source dataset. Thus, we characterize it as a general guideline for selecting a source dataset. However, one can analyze the data distribution similarity of the source and target dataset for a more specific reference.

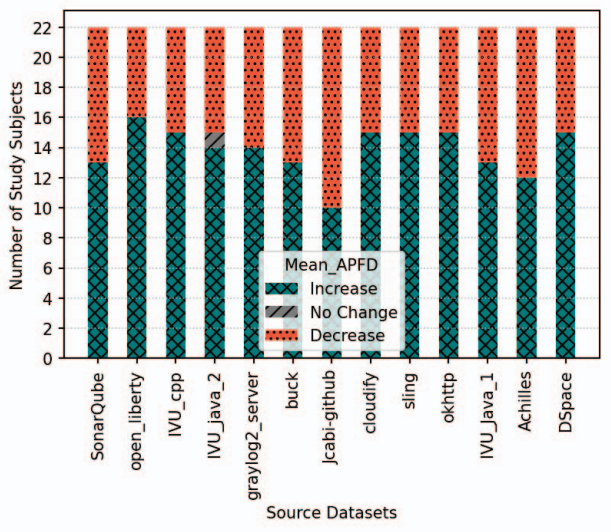

Fig. 4. The performance of source datasets for 23 study subjects compared to *TCP_ML*

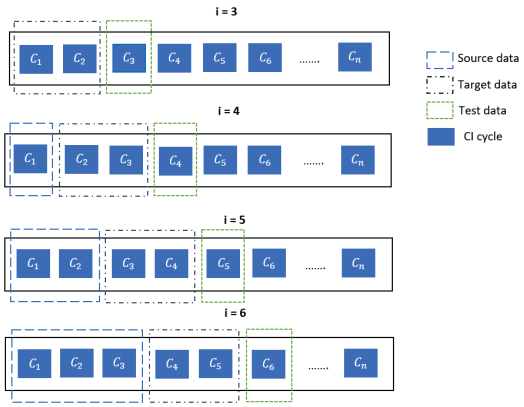

Fig. 5. Spliting dataset, D for internal domain knowledge transfer using *sliding window* (window_size=2) and *forward chaining*

D. Internal Domain Knowledge Transfer for Improving TCP Performance in Large-Scale Projects

The typical transfer learning strategy in industry leverages knowledge from a mature product (large-scale source domain) to improve the performance of new products (small-scale target domain). However, in this experiment, we look for a way to improve the TCP performance of large-scale projects. The large-scale datasets have enough data for constructing efficient ML models; however, the low test failure rate also makes it challenging for large-scale datasets. Some studies [4], [38] consider only the most recent data for building ML models because of test case volatility. The assumption is that some old test cases may be removed and new test cases added to

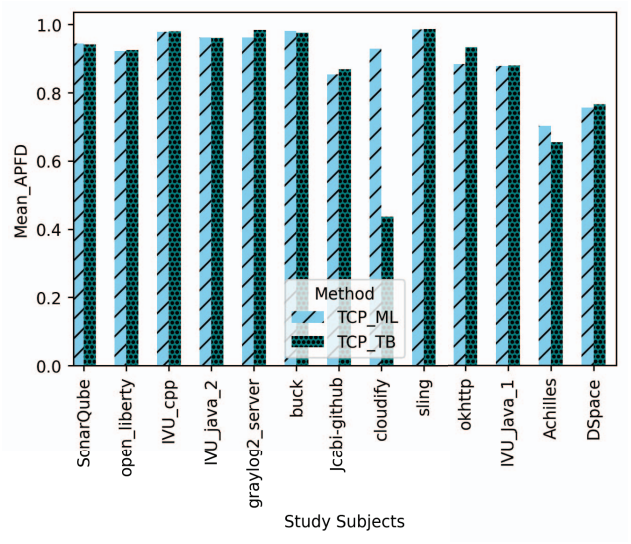

Fig. 6. Mean APFD improvement of *TCP_TB* using internal domain knowledge tranfer

Feature Set	Features
$F1$	Failure count ($f_{1,1}$), Last failure ($f_{1,2}$), Transition count ($f_{1,3}$), Last transition ($f_{1,4}$), Avg. test duration ($f_{1,5}$)
$F2$	Max. (test, file)-failure freq. ($f_{2,1}$), Max. (test, file)-failure freq. (rel.) ($f_{2,2}$), Max. (test, file)-transition freq. ($f_{2,3}$), Max. (test, file)-transition freq. (rel.) ($f_{2,4}$)
$F3$	Min. file path distance ($f_{3,1}$), Max. file path token similarity($f_{3,2}$), Min. file name distance($f_{3,3}$)
$F4$	Distinct authors ($f_{4,1}$), Changeset cardinality ($f_{4,2}$), Amount of commits ($f_{4,3}$), Distinct file extensions($f_{4,4}$)
F_{all}	$F1$, $F2$, $F3$, $F4$

TABLE IV
FEATURES OF CI-RTP/S APPROACHES [4]

the system. Thus, running the ML model with old data can decrease the model's performance. Roza et al. [38] proposed a sliding window-based ML model for TCP. The process discards the old data outside of the sliding window, and if test case information is not present in the window, the test is considered a new test. However, this new test case can be an old one that may run after a long time; thus, discarding the old data can negatively impact the model prediction performance. Therefore, instead of discarding old data, we consider them as the source and refine them on recent data to build a more efficient prediction model. We can follow the same sliding window concept; the data inside the window size will be considered target domain data, and the information before the target data will be considered as source domain data. We name this process internal domain knowledge transfer. An example of creating a source and target dataset within the same domain is shown in Figure 5. The window size of target data can vary depending on the amount of historical data available. In our experiments, the window size varies from six to twelve months.

Figure 6 shows the performance of *TCP_TB* considering internal domain knowledge transfer. This process improves TCP performance for *open_liberty*, *IVU_Cpp*, *graylog2-server*, *jcabi-github*, *sling*, *okhttp*, *IVU_Java_1*, *DSpace* projects by an average of 1.26% compared to *TCP_ML*. For *SonarQube*, *IVU_Cpp*, *IVU_Java_1* and *graylog2-server*, the internal domain knowledge transfer achieves nearly the same result as *TCP_ML* with an average 0.29% decline in mean APFD. However, it performed worse for *cloudify* and *Achilles*, and both these datasets have only one test failure in the last seven months. Thus, we cannot conclude that internal domain knowledge transfer failed for these two datasets. Most large-scale datasets received better performance with internal domain knowledge domain transfer, which is easier to implement. This discussion answers our second research question *RQ2* and indicates that internal domain knowledge transfer can benefit TCP for large-scale projects.

E. Effectiveness of TCP_TB

In this experiment, we compare our approach *TCP_TB* with existing work *CI-RTP/S* [4] to understand its effectiveness. The same 22 study subjects are considered to compare in both approaches. We collect each study subject's optimal settings (feature set, method) and recreate the results for our training testing split. Features used in [4] are listed in Table IV.

.

Moreover, we compare *TCP_TB* with ten popular ML algorithms. Among the ML algorithms, XGBoost achieves the best performance. Table III shows a comparison among *TCP_TB*, *TCP_ML* and *CI-RTP/S* [4] results. The best TCP performances are highlighted in bold. The *CI-RTP/S* column indicates the results calculated based on the optimal setting defined in [4]. The *TCP_ML* column indicates the best ML results with the new feature set f_{new} discussed in Section IV-A.

TCP_TB improves the TCP performance of 19 projects compared to both *CI-RTP/S* and *TCP_ML*. Figure 8 and 7 show the performance of *TCP_TB* in terms of mean APFD on each study subjects compared to *TCP_ML* and *CI-RTP/S* respectively. For *Couldify*, the result of *TCP_TB* remains the same as *TCP_ML* and decreases by 6.35% compared to *CI-RTP/S*. This project has only one failure in the last seven months, and the feature sets are also different. We assume this occurs because of the feature set, and it is also not sufficient to measure the effectiveness based on one failure. In the future, we plan to train *TCP_TB* with different feature sets to identify the optimal feature set for each study subject. However, the mean APFD value of *TCP_TB* for *grayloglog2-server* remains the same and improves 2.14% than *CI-RTP/S* and *TCP_ML* respectively. The TCP performance of *TCP_TB* decreases by 0.29% and 0.70% for *dynjs* and *optiq* respectively compared to *TCP_ML*. We observe that the performance of *TCP_TB*

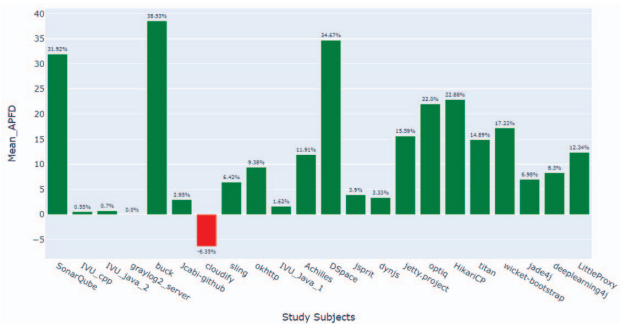

Fig. 7. The percentage of mean APFD improvement of *TCP_TB* for each study subject compared to *CI-RTP/S*

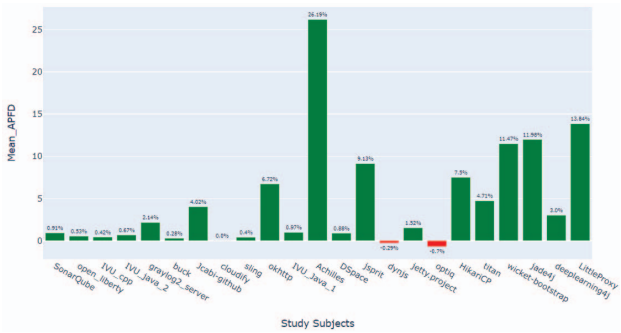

Fig. 8. The percentage of mean APFD improvement of *TCP_TB* for each study subject compared to *TCP_ML*

Fig. 9. Mean NAPFD across study subjects

for these two projects is slightly decreased than *TCP_ML*. This may happen because of the different data distribution of these two projects compared to their source datasets. For the remaining projects *TCP_TB* performed better than *CI-RTP/S* and *TCP_ML*. The performance improvement for the most large-scale dataset is around 1% compared to *TCP_ML* based on the mean APFD value. Project *Achilles* receives the most significant gain, a 26.19% increase in the mean APFD score. For the smaller datasets, the *TCP_TB* achieves significant improvement, which is expected as they have fewer data for ML model training. We find an average of 6.1% improvement in the mean APFD value than *TCP_ML*.

We further analyze the effectiveness of *TCP_TB* for test case selection (TCS). Figure 9 shows the performance of *CI-RTP/S*, *TCP_ML* and *TCP_TB* for there different test budget. For all three test budgets, *TCP_TB* performed better than the other two methods. For the small 10% test execution budget, *CI-RTP/S* and *TCP_ML* perform very poorly. From the above discussion, we can conclude *TCP_TB* can perform better than ML models with the same feature set for TCP and TCS.

VI. Threats to Validity

This section discusses the possible threats to our experimental results with internal and external validity.

- Internal Validity: In our experiment, we use the default execution environments and parameter settings of different machine-learning techniques to evaluate the study subjects. However, hyperparameter tuning and feature selection may improve the performance from one project to another. In our future work, we plan to tune the hyperparameters to see the impact on different scenarios.
- External Validity: The experimental results are discussed by evaluating 24 specific datasets primarily written in Java. Our experiment considers selected features that are extracted from CI and GitHub repositories. The result may vary if different feature sets are selected. However, our study can be reproduced using the same model configurations and dataset available in the repository [39].

VII. Conclusion

This paper introduces *TCP_TB*, a transfer learning-based test case prioritization technique that uses test execution history and VCS changeset features to reorder test suites efficiently. The motivation is to reuse the knowledge of a pre-trained model to improve the TCP performance of similar projects. We evaluate our approach with 24 software projects, and it enhances the TCP performance of 19 projects. *jOOQ* has no test failures in the last seven months of test execution; thus, we exclude it from the evaluation. We use the APFD metric to evaluate our proposed approach. The large-scale projects (#test execution $>$ 100K) receive mostly a 2.82% increase in the APFD value compared to *CI-RTP/S* and the *TCP_ML*. *open_liberty* seem to be the most relevant source dataset for creating transfer learning models as it enhances the mean APFD value of 16 projects. Overall, *TCP_TB* improves the TCP and TCS performance in 82.61% and 79.71% cases, respectively. Therefore, we can conclude that TL benefits test case prioritization and selection. In the future, we plan to further improve *TCP_TB* by exploring the optimal feature set and conducting a cost-benefit analysis of *TCP_TB* compared to other ML-based models. Moreover, we aim to evaluate the effectiveness of other transfer learning models in conjunction with *TCP_TB*.

ACKNOWLEDGEMENTS

This research was supported in part by IBM Center for Advanced Studies (CAS) and Natural Sciences and Engineering Research Council of Canada (NSERC) grants.

REFERENCES

[1] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 235–245, 2014.

[2] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[3] C. Indumathi and K. Selvamani, "Test cases prioritization using open dependency structure algorithm," *Procedia Computer Science*, vol. 48, pp. 250–255, 2015.

[4] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 491–504, 2021.

[5] T. Mattis, P. Rein, F. Dürsch, and R. Hirschfeld, "Rtptorrent: An open-source dataset for evaluating regression test prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 385–396, 2020.

[6] L. Rosenbauer, D. Pätzel, A. Stein, and J. Hähner, "Transfer learning for automated test case prioritization using xcsf," in *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pp. 681–696, Springer, 2021.

[7] Y. Zhang, W. Dai, and S. Pan, "Transfer learning," 2020.

[8] Y. Sun, T. Lu, C. Wang, Y. Li, H. Fu, J. Dong, and Y. Xu, "Transboost: A boosting-tree kernel transfer learning algorithm for improving financial inclusion," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, pp. 12181–12190, 2022.

[9] H. Li, "Learning to rank for information retrieval and natural language processing," *Synthesis lectures on human language technologies*, vol. 7, no. 3, pp. 1–121, 2014.

[10] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1–12, 2020.

[11] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 361–368, IEEE, 2016.

[12] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *2006 22nd IEEE international conference on software maintenance*, pp. 123–133, IEEE, 2006.

[13] T. Joachims, "Optimizing search engines using clickthrough data," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 133–142, 2002.

[14] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering*, pp. 975–980, 2016.

[15] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 656–667, 2018.

[16] M. Hasnain, M. F. Pasha, C. H. Lim, and I. Ghan, "Recurrent neural network for web services performance forecasting, ranking and regression testing," in *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 96–105, IEEE, 2019.

[17] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, and S. Jalali, "Incorporating fault-proneness estimations into coverage-based test case prioritization methods," *Information and Software Technology*, vol. 121, p. 106269, 2020.

[18] M. M. Sharma and A. Agrawal, "Test case design and test case prioritization using machine learning," *International Journal of Engineering and Advanced Technology*, vol. 9, no. 1, pp. 2742–2748, 2019.

[19] S. Mirarab and L. Tahvildari, "An empirical study on bayesian network-based approach for test case prioritization," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 278–287, IEEE, 2008.

[20] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," in *Proceedings of the 13th international conference on predictive models and data analytics in software engineering*, pp. 2–11, 2017.

[21] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu, "An improvement to test case failure prediction in the context of test case prioritization," in *Proceedings of the 14th international conference on predictive models and data analytics in software engineering*, pp. 80–89, 2018.

[22] A. Singh, R. K. Bhatia, and A. Singhrova, "Machine learning based test case prioritization in object oriented testing," *Int. J. Recent Technol*, vol. 8, pp. 700–707, 2019.

[23] N. Medhat, S. M. Moussa, N. L. Badr, and M. F. Tolba, "A framework for continuous regression and integration testing in iot systems based on deep learning and search-based techniques," *IEEE Access*, vol. 8, pp. 215716–215726, 2020.

[24] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 12–22, 2017.

[25] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, 2021.

[26] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–43, 2022.

[27] Y. Zhu, E. Shihab, and P. C. Rigby, "Test re-prioritization in continuous testing environments," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 69–79, IEEE, 2018.

[28] J. Liang, S. Elbaum, and G. Rothermel, "Redefining prioritization: continuous prioritization for continuous integration," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 688–698, 2018.

[29] C. Pan and M. Pradel, "Continuous test suite failure prediction," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 553–565, 2021.

[30] T. Shi, L. Xiao, and K. Wu, "Reinforcement learning based test case prioritization for enhancing the security of software," in *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 663–672, IEEE, 2020.

[31] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 382–391, IEEE, 2011.

[32] L. Rosenbauer, A. Stein, R. Maier, D. Pätzel, and J. Hähner, "Xcs as a reinforcement learning approach to automatic test case prioritization," in *Proceedings of the 2020 genetic and evolutionary computation conference companion*, pp. 1798–1806, 2020.

[33] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 102–112, 2000.

[34] A. Baevski, S. Edunov, Y. Liu, L. Zettlemoyer, and M. Auli, "Cloze-driven pretraining of self-attention networks," *arXiv preprint arXiv:1903.07785*, 2019.

[35] E. Mohamed, K. Sirlantzis, and G. Howells, "Application of transfer learning for object detection on manually collected data," in *Proceedings of SAI Intelligent Systems Conference*, pp. 919–931, Springer, 2019.

[36] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "Fastlane: Test minimization for rapidly deployed large-scale online services," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 408–418, IEEE, 2019.

[37] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *2007 IEEE International Conference on Software Maintenance*, pp. 255–264, IEEE, 2007.

[38] E. A. Da Roza, J. A. P. Lima, R. C. Silva, and S. R. Vergilio, "Machine learning regression techniques for test case prioritization in continuous integration environment," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 196–206, IEEE, 2022.

[39] https://figshare.com/s/851513ca9907a10dd17d.