Test Case Prioritization using Transfer Learning in Continuous Integration Environments

by

Rezwana Mamata

A thesis submitted to the

School of Graduate and Postdoctoral Studies in partial

fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Department of Electrical, Computer and Software Engineering

Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

April 2023

Thesis Examination Information

Submitted by: **Rezwana Mamata**

Master of Applied Science in **Electrical and Computer Engineering**

Thesis Title: Test Case Prioritization using Transfer Learning in Continuous Integration Environments

An oral defense of this thesis took place on April 5th, 2023 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee: **Dr. Mohamed El-Darieby**

Research Supervisor: **Dr. Akramul Azim**

Committee Member: **Dr. Ramiro Liscano**

Thesis Examiner: **Dr. Masoud Makrehchi, Ontario Tech University**

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

Continuous integration (CI) systems use automated tests to verify software builds and optimize the testing process through test case prioritization (TCP). Recent research studies on TCP in CI environments have employed machine learning (ML) techniques to address the dynamic nature of CI. However, the performance of ML for TCP may decrease because of the low volume of data or less failure rate, whereas using existing data with similar patterns from other domains can be valuable. Therefore, this thesis investigates the potential of transfer learning (TL) algorithms to improve test case failure prediction and prioritization in CI environments.

We conduct a comparative analysis of traditional TL algorithms to improve test failure prediction of large-scale industrial projects. Our experimental results show that parameter-based TL is most effective, and its usefulness is further emphasized by the scarcity of publicly available large-scale CI datasets due to data privacy regulations. We also present a new technique called *TCP_TB* that prioritizes test cases using the prediction probability of test failures generated by a parameter-based transfer learning algorithm called *TransBoost*. We compare *TCP_TB*'s performance with different ML approaches and *CI-RTP/S* on 24 study subjects and demonstrate that *TCP_TB* outperforms them, improving TCP performance in 82.61% of the cases.

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

_____

Rezwana Mamata

Statement of Contributions

The research topic and the work described in this thesis have been accepted to the following events:

- Mamata, R., Azim, A., Liscano, R., Smith, K., Chang, Y. K., Tauseef, Q., & Seferi, G. "Test Case Prioritization using Transfer Learning in Continuous Integration Environments" 4th ACM/IEEE International Conference on Automation of Software Test (Accepted in AST 2023).

- Mamata, R., Azim, A., Liscano, R., Smith, K., Chang, Y. K., Tauseef, Q., & Seferi, G (2022, November). "Failure Prediction using Transfer Learning in Large-scale Continuous Integration Environments." In Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering (pp. 193-198), (Published in CASCON 2022).

I hereby certify that I am the sole author of this thesis. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am the sole source of the creative works and/or inventive knowledge described in this thesis.

Acknowledgements

I am delighted to express my gratitude to everyone who has helped me complete my thesis. Firstly, I would like to extend my heartfelt appreciation to my supervisor, Dr. Akramul Azim, for his exceptional guidance and mentorship. Throughout my research, he has provided invaluable support, dedication, and encouragement, and working with him has allowed me to develop as a researcher. His scientific and practical advice has made a significant contribution to the progress of my master's research. I would also like to thank the members of the RTEMSOFT research group and IBM CAS project group for their support and suggestions throughout my academic journey. I am also grateful to the faculty members at Ontario Tech University for their additional support and guidance.

Finally, I would like to express my special gratitude to my family for their continuous support and encouragement throughout my academic journey. I am truly grateful to my parents for their unconditional love, and to my husband for his unwavering patience and sacrifices. His positivity and love have been a constant source of motivation and inspiration for me, and I will forever be grateful to him for being my rock and providing the stability I needed to succeed. Additionally, I want to express my appreciation to my in-laws for their understanding, love, and support. Their encouragement and belief in me have been instrumental in helping me achieve my goals. Without the unwavering support of my family, this accomplishment would not have been possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CI	Continuous Integration
VCS	Version Control System
TCP	Test Case Prioritization
TCS	Test Case Selection
ML	Machine Learning
SL	Supervised Learning
SSL	Semi Supervised Learning
UL	Unsupervised Learning
RL	Reinforcement Learning
TL	Transfer Learning
SA	Subspace Alignment
CORAL	Correlation Alignment
NNW	Nearest Neighbors Weighting
TrAdaBoost	Transfer AdaBoost
TCP_ML	Test Case Prioritization using Machine Learning
TCP_TB	Test Case Prioritization using TransBoost

Chapter 1

INTRODUCTION

Testing is an essential aspect of software development that can have a significant impact on the overall cost of the project. The Continuous Integration (CI) approach is used in software development to automate the compilation, building, and testing phases of a project. The main objective of this approach is to reduce integration issues and development time [1], [2]. In CI, developers frequently integrate their code changes with the mainline codebase, which triggers a CI cycle. After each cycle is compiled, developers receive feedback on their code [2]. The duration of each cycle depends on the number of tests required, which can vary based on the size of the codebase. A high volume of tests requires significant computational resources and time to execute, which can delay the CI process and prolong the developer feedback time. For example, Microsoft documented a delay of three days for executing 65K tests [3], and Google handles 800K builds and 150 million test suites daily, with developers experiencing wait times of 45 minutes to 9 hours for results [4].

To address the issue of long test execution times and enhance testing efficiency in CI, researchers have developed test optimization methods (e.g.,Test Case Prioritization (TCP) and Test Case Selection (TCS)) [5], [6]. TCP involves reordering test cases based on their

failure probability within a test suite instead of deleting any test cases [5]. On the contrary, TCS selects a test case subset which has a higher probability of failure [5]. Test failure prediction is a preliminary step for both TCP and TCS methods. Test optimization methods can be broadly categorized into heuristic-based and Machine Learning (ML)-based approaches [7]. Heuristic-based techniques typically operate on a single feature, whereas ML-based techniques use multiple features extracted from various resources, allowing them to adapt to new changes through techniques such as retraining or incremental learning.

To effectively address the dynamic and complex nature of CI systems, TCP techniques have utilized various ML techniques that rely on different data sources such as CI test execution history, version control system (VCS) metadata, and code coverage information [8]–[10]. However, collecting test coverage and dependency information through white-box program analysis is often too time-consuming and costly, particularly for large-scale codebases with rapid CI testing. In addition, gathering test dependencies from different programming languages of multi-language software can be infeasible [11]. In contrast, VCS and CI metadata are automatically generated during each CI cycle, making them easily accessible and cost-effective [9]. Moreover these information are not software specific (e.g., type or language). For these reasons, TCP approaches that use CI and VCS metadata are more preferable.

Traditional ML algorithms typically require a large volume of data to learn accurate models and generalize to new, unseen data. However, when it comes to the TCP, traditional ML algorithms may face certain challenges. In industrial projects, test suites undergo thorough testing, which reduces the number of failures, making it challenging to generate enough positive samples for model training. As a result, there is often an imbalance in

the data, where the number of positive samples is relatively small compared to negative samples. Traditional ML algorithms can struggle with imbalanced data since they tend to prioritize accuracy on the majority class, resulting in poor performance on the minority class. In the case of TCP, this means that the model may not effectively identify test cases with a high probability of failure, leading to ineffective prioritization. On the other hand, in new projects, there is often a lack of historical data, which is necessary for the TCP model to learn. The absence of sufficient data affects the model's ability to learn the underlying patterns in the data, leading to poor performance. Additionally, the model may be unable to generalize to new, unseen data, as it may have learned to prioritize only a specific subset of test cases based on the limited training data.

To overcome the challenges of imbalanced data in the test case prioritization task, researchers have proposed various solutions, including data balancing techniques [12] that balance the data and generate additional positive samples. Data balancing involves creating synthetic data by modifying existing samples, such as by perturbing the input features, introducing noise or randomization, and creating new test cases by combining existing ones. However, data balancing techniques may not always be effective, and the quality of the synthetic data may not accurately reflect the original data, leading to a reduction in model performance. Furthermore, while data balancing can help balance imbalanced datasets, it may not address the issue of insufficient data in new projects.

Transfer learning (TL) algorithms offer a potential solution for addressing the problem of training data scarcity in the target domain, particularly for test case prioritization (TCP) tasks in unbalanced low-volume software projects [13]. TL allows models to adapt quickly to unknown situations, environments, and tasks by leveraging knowledge from a related task [14]. Thus, incorporating transfer learning techniques in TCP may help to address the

challenge of imbalanced data and insufficient training data in TCP for large-scale industrial and newer projects.

1.1 MOTIVATION

Applying transfer learning to prioritize test case in a CI environment requires careful consideration of several factors. Identifying an appropriate TL algorithm in terms of cost and effectiveness, as well as a suitable feature set for the model, is critical. TCP models should have generic features that can be shared across projects, while project-specific features may not be useful. Statistical features of test failures (e.g., failure rate) and code changes (e.g., number of files changed) are examples of such generic features. Previous studies have shown that these features are inexpensive and easily accessible [9]. However, a detailed analysis of different TL algorithms' performance for test case prioritization and VCS and CI features impact on TL models is required to suggest a baseline. One of the most challenging tasks in TL-based TCP is identifying suitable source domains for the model. While a large volume of data may be available in some cases, it may not be relevant or useful for the target domain. Additionally, large software companies may be unwilling to share failure information from their projects, which could be used to build efficient testing models for newer projects. To overcome these challenges, we propose an approach that analyzes the criteria for choosing a source dataset and the applicability of model-based TL algorithms for TCP, where the source domain is referential, but data points are not available to maintain privacy. By addressing these challenges, TL-based TCP methods can provide a solution to enhance testing efficiency in CI for a wide range of software projects.

1.2 NOVELTY OF THE THESIS

The novelty of this work lies in the implementation of a transfer learning (TL) approach for test case prioritization (TCP) that leverages both test execution histories (CI metadata) and code change information (VCS metadata) to improve testing efficiency. This study provides a comprehensive comparison of different datasets from both CI and GitHub sources to identify source datasets that have the potential to transfer knowledge to enhance TCP performance. Additionally, the study explores the applicability of internal domain knowledge to prioritize test cases for large-scale projects.

The combination of CI metadata and VCS metadata provides a rich source of information that enables the creation of a more robust TCP model. By leveraging both sources of data, the proposed TL-based approach is able to capture a more complete understanding of the underlying failure patterns, leading to improved TCP performance. The analysis of VCS and CI features in the TL models provides insights into the factors that most significantly affect TCP performance.

Through the comprehensive comparison of different datasets from various sources, this study is able to identify the most suitable source datasets for the TCP model. By selecting the most appropriate source datasets, the proposed TL-based approach can effectively transfer knowledge to the target domain, leading to better TCP performance. Additionally, this study provides a valuable benchmark for future research in this area, allowing for more accurate comparisons of different approaches and datasets.

Moreover, this study provides valuable insights into the internal domain knowledge transfer to prioritize test cases for large-scale projects. While large-scale datasets contain enough data for constructing effective ML models, test case volatility can pose challenges. The assumption is that some old test cases may be removed and new ones added to the

system. Consequently, running the ML model with old data can decrease its performance. A study by Roza et al. [15] addresses this challenge by considering only recent data to build ML models. However, old data may still be relevant if an old test case runs after a long time. Thus, instead of discarding old data, the study considers it as a source and refines it with recent data to create a more effective prediction model. This approach provides a novel way to improve the performance of TCP models for large-scale projects.

These insights can be applied to a wide range of software projects and can help improve the testing process by identifying high-risk test cases that require more attention. Overall, the study contributes to the development of more efficient and effective testing methods by utilizing the power of transfer learning.

1.3 CONTRIBUTIONS

In summary, the main contributions of this thesis are:

- Evaluation and comparison of different TL algorithms: This study systematically evaluates and compares the performance of various TL algorithms against ML algorithms for test failure prediction in CI environments. The findings provide insights into the effectiveness and limitations of different TL algorithms.

- Identification of potential source datasets for TL: The study compares and analyzes different datasets to identify the potential source datasets for transferring knowledge to improve TCP performance of other projects.

- Development of a new TCP approach: We propose a new TCP approach called *TCP_TB*, which uses the prediction probability of test failures generated by a parameter-based TL algorithm called TransBoost. *TCP_TB* outperforms other machine learning

approaches and the existing *CI-RTP/S* technique on 24 study subjects, improving TCP performance in 82.61% of the cases.

- Evaluation of internal domain knowledge transfer for TCP: We explore the applicability of internal domain knowledge to prioritize test cases for large-scale projects. This approach addresses the challenge of test case volatility by utilizing old data of a large project as a source and refines it with recent data to create a more accurate prediction model.

1.4 RESEARCH QUESTIONS

We evaluate the proposed TCP approach, *TCP_TB*, with 24 study subjects and compares its performance with ten supervised ML algorithms and an existing method, *CI-RTP/S*. Through these experiments, we aim to answer the following research questions

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

1.5 ORGANIZATION OF THE THESIS

In this thesis, we present our work in five sections. Chapter 2 provides a literature review of methods for test failure prediction and prioritization, including existing machine learning-based approaches. Chapter 3 proposes a transfer learning approach for test failure

prediction and test case prioritization (TCP). Firstly, we evaluate and compare different TL algorithms for test failure prediction, and analyze the impact of VCS and CI features on TL. Secondly, we compare different datasets to identify potential source datasets for TCP and explore the internal domain knowledge transfer for large-scale datasets to address the challenge of test case volatility.

In chapter 4 of this thesis, we present the experimental results of our proposed approaches. We conducted three case studies to evaluate TL for test failure prediction and 24 case studies to evaluate TL for test case prioritization. Our results indicate that model-based TL approaches are more suitable in terms of time, efficiency, and data privacy. We also generalized the criteria for identifying potential source datasets. Additionally, our experimental results demonstrate that internal domain knowledge transfer can be a potential TCP approach for large-scale datasets. Overall, the experimental results provide evidence for the effectiveness of our proposed approaches and contribute to the existing knowledge on TL-based approaches for test failure prediction and prioritization.

The final chapter of this thesis, Chapter 5, provides a conclusion and outlines possible future research directions related to transfer learning for efficient test case prioritization. This chapter summarizes the main contributions and findings of the thesis

Chapter 2

LITERATURE REVIEW

This chapter offers technical background information on software testing, machine learning, and transfer learning to provide a foundation for the work presented in this thesis. The primary objective of this research is to improve testing efficiency in Continuous Integration (CI) environments by employing transfer learning (TL) techniques that utilize historical test data and code change information.

In the context of software development, version control systems (VCS) and CI tools are crucial for managing the codebase and ensuring that software development projects are organized and streamlined. VCS is used to keep track of code changes and maintain a history of modifications.In contrast, CI tools automate the building, testing, and integration of code changes, thereby preventing new changes from disrupting the existing codebase.

Test case prioritization (TCP) and machine learning (ML) are essential for the testing phase of software development. TCP is used to determine the order in which tests should be executed, while ML is used to predict the likelihood of test case failures based on historical test execution data. By using TCP and ML, developers and testers can ensure that the most critical and error-prone test cases are executed first, thereby saving time and resources. Additionally, the chapter discusses existing work on test failure prediction and prioritization

in CI, which will help provide a context for the research carried out in this thesis.

2.1 CONTINUOUS INTEGRATION (CI) AND VERSION CONTROL SYSTEM (VCS)

CI is a common software development approach that entails regularly merging code changes from multiple developers into a central repository throughout the day. This practice enables developers to detect and resolve code conflicts and issues earlier in the development process, which can significantly reduce the cost of fixing bugs and improve software quality [2]. CI involves the use of various tools and techniques, including automated testing, VCSs, and build automation tools. The process starts with developers committing their code changes to a shared repository, which then initiates an automated build and testing process. If the build and tests are successful, the changes are integrated into the main codebase. If there are any issues, the developers are notified, and the build process stops until the issues are resolved. CI has become an essential practice in modern software development, particularly in Agile and DevOps environments [16]. It provides several benefits, such as faster time-to-market, improved code quality, and increased team productivity [17].

Large organizations like Google and Microsoft frequently update their software products to meet user demands and have adopted the CI environment for faster delivery and transparency. Open-source projects have also begun to adopt this practice using available CI tools like Travis CI [18], Circle CI [19], and Jenkins [20]. However, effective use of CI poses additional challenges, requiring organizations to make their tasks executable without human intervention, and control the duration of CI cycles to ensure timely feedback to developers and optimal utilization of time and resources. Testing is a critical step in the CI process, and optimizing test suites can help address these challenges, the details will be discussed in section 2.4.

VCS is a software tool to assists software developers in managing source code changes over time. It enables teams to work collaboratively on the same codebase, track and manage changes, and maintain multiple versions of the code [21]. VCS is an essential tool for any software development project and has become a standard practice in the industry [22]. Some popular VCS tools are Git, SVN, Mercurial, and Perforce. By utilizing these tools, developers can collaborate on the same codebase simultaneously, merge changes, and resolve conflicts as needed.

In addition to tracking changes to source code, VCS also helps with other aspects of software development, such as issue tracking, bug reporting, and code reviews. It helps maintain the integrity of the codebase, reduces the risk of code loss, and makes it easier to collaborate with other developers. They also offer capabilities for code review and rollback to previous versions in case of issues. In software testing, VCSs are often used in conjunction with CI practices to automate the building, testing, and deployment of software. VCSs are crucial in implementing CI as they provide the means to track changes, manage versions, and facilitate collaboration between team members.

In summary VCS and CI are two critical practices in software development that work hand in hand to improve software development efficiency and quality. VCSs allow developers to keep track of changes to the source code, including author information, commit timestamp, and changeset, among others. The VCS log is essential for tracking code changes and ensuring code stability. On the other hand, a CI build can be triggered after a code change or after the previous build or depending on available resources. The code change that caused the CI build will contain at least one commit. The build log will contain information such as build identifier, test suites, build result, build duration, test suite result, test duration, etc. Most testing frameworks and CI systems provide structured build logs.

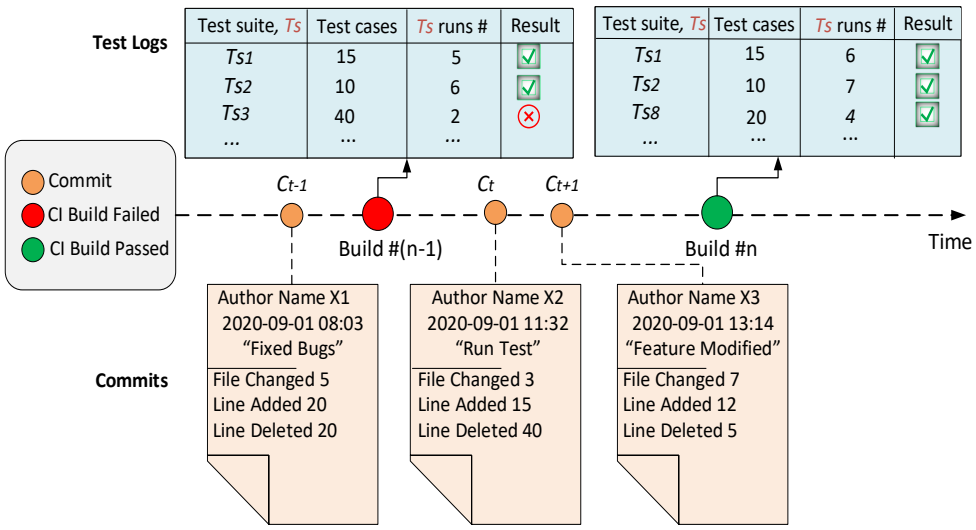

Figure 2.1: Relationship between the CI environment and VCS

However, the build information can also be parsed from raw textual log data using regular expressions [23].

Figure 2.1 illustrates the relationship between the VCS and CI environment. The diagram illustrates two parallel timelines: the VCS commits at the bottom, and the CI builds initiated by these commits at the top. The CI build and VCS log data are essential for tracking the development process, and they can be used to improve software quality. By comparing changes in the VCS and CI build logs, developers can identify which code changes have caused test failures or build issues. This information helps developers to fix issues more quickly and improve the software development process overall. Therefore, VCS and CI logs provide valuable information for software development and continuous improvement.

2.2 MACHINE LEARNING (ML)

ML is a subfield of artificial intelligence that seeks to impart machines with the ability to mimic intelligent human behavior [24]. ML algorithms employ statistical methods and utilize data to derive rules that enable machines to learn and improve performance over time. The structure of the data and the type of rules derived vary based on the category and type of ML algorithm employed. There are three primary categories of ML algorithms: supervised, unsupervised, and reinforcement learning.

Supervised learning (SL) involves training a model with a set of input and output examples, during which the model learns to approximate the relationship between the inputs and expected outputs [25]. SL models can be used for various tasks such as image classification [26] and predicting the price of a house based on a set of attributes [27]. On the other hand, unsupervised learning (UL) involves providing only input data without any expected output labeling. These types of models are often used for clustering data points into separate groups or compressing data for dimensionality reduction [28]. Finally, reinforcement learning enables a model to interact with an external environment by observing the environment's state and issuing corresponding actions [29]. Examples of reinforcement learning applications include game-playing agents [30], learning robots [31], and content placement agents that learn which articles or advertisements to suggest to particular users.

Transfer learning (TL) is a subfield of ML that involves applying previously acquired knowledge to improve learning or performance in a related domain or problem. This approach allows for the transfer of knowledge learned from a source domain to a target domain, where it can be useful for solving a related problem. Transfer learning has been applied in various domains, including computer vision, natural language processing, and speech recognition, and can be utilized in supervised, unsupervised, and reinforcement

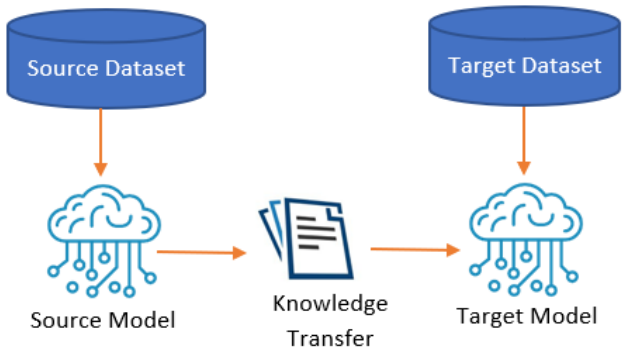

Figure 2.2: Transfer Learning

learning [32]. By improving model performance in the target domain while reducing the amount of labeled data required for training, transfer learning has significant potential for enhancing machine learning applications. In this thesis, we will focus on transfer learning as it relates to our objective of enhancing testing efficiency in continuous integration environments. Figure 2.2 illustrates the basic diagram of the transfer learning approach. We will discuss transfer learning in more detail in the following section.

2.3 TRANSFER LEARNING (TL)

Transfer learning techniques aim to enhance the generalization of models and reduce the training time by leveraging previously learned knowledge. It involves transferring knowledge from a source domain to a target domain, where the feature spaces, distributions, and output spaces may be different [32]. Let $D_S = X_S, Y_S$ be the source domain, where X_S is the input space and Y_S is the output space, and $D_T = X_T, Y_T$ be the target domain, where X_T is the input space and Y_T is the output space.

In transfer learning, a mapping function $f(X_S)$ is learned in the source domain, and the knowledge gained from this mapping function is transferred to the target domain to learn

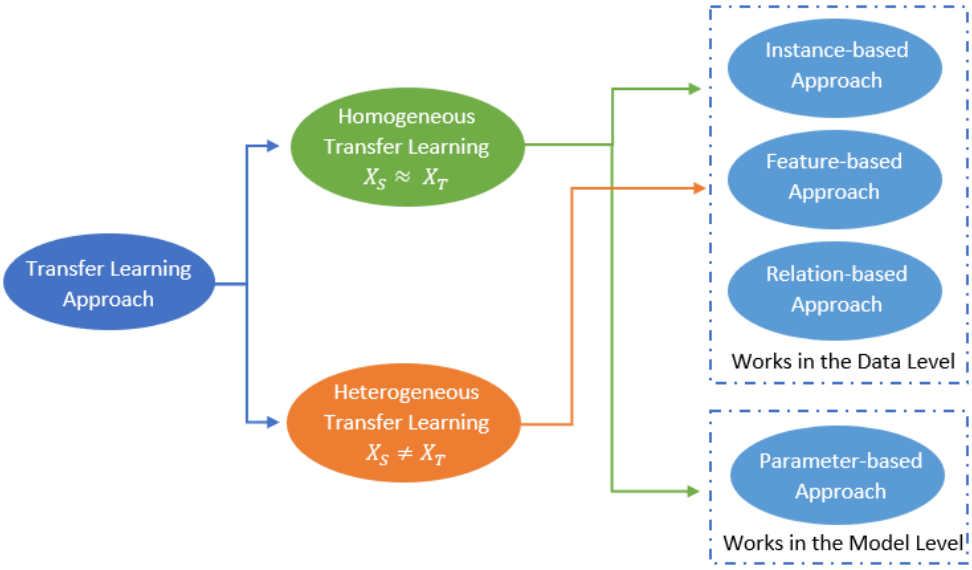

Figure 2.3: Classification of Transfer Learning (TL) Approach

a new mapping function $f(X_T)$ to improve performance [33]. The transfer of knowledge can be achieved through different methods, such as reusing the parameters or intermediate representations of the source data.

Formally, transfer learning can be defined as minimizing the risk in the target domain while utilizing the knowledge gained from the source domain [32]:

$$\min_{f_T} R(f_T) + \lambda \cdot \mathcal{L}(f_T, D_S)$$

where $R(f_T)$ is the risk in the target domain, $\mathcal{L}(f_T, D_S)$ is the loss incurred by applying f_T to the source domain, and λ is a trade-off parameter.

By minimizing the above objective function, the mapping function f_T is learned in the target domain while utilizing the knowledge learned in the source domain to improve its performance.

Transfer learning algorithms can be classified into two main categories based on the

similarity between the source and target domains: homogeneous transfer learning and heterogeneous transfer learning [33]. Homogeneous transfer learning algorithms assume that the source and target domains have the same feature space but differ in data distribution, with the goal of minimizing this difference. In contrast, heterogeneous transfer learning algorithms assume that the source and target domains have different feature spaces. Figure 2.3 displays the classification of transfer learning approaches.

To predict test suite failures, we assume that the feature spaces of both the source and target domains are similar since they perform the same task and follow the same continuous integration practices. Thus, we have chosen to use homogeneous transfer learning algorithms to achieve our goal. Homogeneous transfer learning algorithms can be divided into four subcategories: feature-based, instance-based, parameter-based, and relation-based methods. In the following sections, we will provide a detailed discussion of each of these approaches.

- Instance-based TL: This is an TL approach that focuses on transferring knowledge from labeled instances in the source domain to the target domain. Rather than considering the general structure of the data, this method emphasizes individual instances of data [34], [35]. The main idea behind instance-based transfer learning is to re-weight the samples in the source domain to correct for marginal distribution differences and use the re-weighted instances directly in the target domain for training. Instance-based transfer learning methods work best when the conditional distribution is the same in both domains.

  Various approaches have been proposed to weight the source samples, such as matching the mean of the target and source domains or training a binary classifier that separates source samples from target samples. This method assigns higher weight to

Figure 2.4: An example of source instance weighting using a binary classifier

the source samples that are more similar to target samples [33]. Figure 2.4 provides a simple example of source instance weighting using a binary classifier. Instance-based transfer learning has been successfully applied in computer vision, natural language processing, and speech recognition domains, and it can significantly reduce the amount of labeled data required to train models while improving model performance in the target domain. However, it has limitations when marginal distribution differences between the source and target domains are significant or when there are not enough labeled target samples [33].

- Feature-based TL: This methods focus on creating a new feature representation to reduce the differences in feature distributions between the source and target domains. This approach can be further divided into two subcategories: asymmetric and symmetric feature-based transfer learning [33].In asymmetric feature-based transfer learning, the features of the source domain are transformed to match the target feature

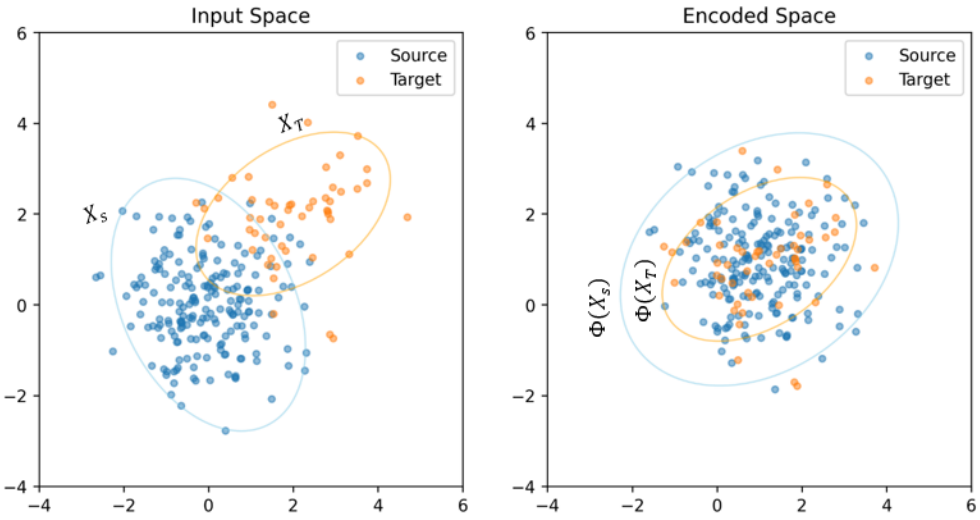

Figure 2.5: An example of symmetric feature transformation

space, which can result in information loss due to marginal differences in feature distribution [36], [37]. This technique has been used in various domains, such as image classification and sentiment analysis.

In contrast, symmetric feature-based transfer learning involves creating a common latent feature space, and both the source and target features are transformed into this new feature representation. This approach helps to reduce the feature distribution gap between the source and target domains and can improve model performance [38], [39]. Symmetric feature-based transfer learning has been used in several domains, such as computer vision and natural language processing. Both asymmetric and symmetric feature-based transfer learning approaches have been successfully applied in various domains, and their effectiveness depends on the characteristics of the source and target domains [33].

- Relation-based TL: This thechniques utilize the underlying relational structure of the

data to transfer knowledge from the source to the target domain [40], [41]. The approach assumes that the source and target domains share some relational structures or dependencies, which can be used to improve learning in the target domain. These methods are used in various applications, such as social network analysis, recommender systems, and link prediction.

Relation-based transfer learning typically represents the data as a graph or network, where the nodes represent the entities, and the edges represent the relationships between these entities. This relational structure can be used to identify the similarities and differences between the source and target domains and to guide the transfer of knowledge. However, this approach can be computationally expensive and may require significant domain-specific knowledge.

Relation-based transfer learning has shown promising results in improving model performance in target domains, especially when the data has a clear relational structure. Nevertheless, there is a need for further research to develop efficient and effective algorithms that can handle larger and more complex relational structures [42].

- Parameter-based TL: This methods that transfers knowledge at the model level, in which the source and target tasks share some model parameters or prior distributions of the hyper-parameters of the models. The previous two transfer learning solutions, namely instance-based and feature-based methods, transfer knowledge at the data level [33]. Parameter-based transfer learning is particularly useful when the feature space of the source and target domains differs, but the models learned from them share a similar structure. This method is commonly used in deep learning models, where the parameters are learned using a large amount of labeled data. Parameter-based transfer learning can be further classified into three subcategories: fine-tuning,

pre-training, and model adaptation [32].

In fine-tuning, a pre-trained model is further trained on the target domain data to adapt to the new task. Pre-training involves training a model on a large, labeled dataset to learn general representations, followed by fine-tuning on the target dataset. Model adaptation aims to modify the source model to suit the target task. Parameter-based transfer learning has shown success in various domains, including computer vision and natural language processing [33].

Hybrid-based transfer learning combines instance-based and parameter-based transfer learning approaches to transfer knowledge through both instances and shared parameters. This approach has shown success in various domains, including image classification, natural language processing, and speech recognition [35]. The main goal of hybrid transfer learning is to find a set of parameters optimized for the source task that can be fine-tuned for the target task. This technique can significantly reduce the amount of labeled data required for the target task and lead to improved model performance.

However, finding the optimal balance between the instance-based and parameter-based transfer learning methods is challenging, and the best approach depends on the specific application and the availability of labeled data [33]. Hybrid transfer learning methods can be used with various machine learning algorithms, such as decision trees, support vector machines, and neural networks, and can be customized to fit the specific requirements of each application.

2.4 TEST CASE FAILURE PREDICTION AND PRIORITIZATION

Test failure prediction and prioritization are critical components of software testing. Test failure prediction involves predicting which tests are likely to fail before they are executed,

while test prioritization involves determining the order in which tests should be executed based on their relative importance.

Test failure prediction can help identify problematic tests early in the testing process, which can save time and resources. Predictive models are built using historical test execution data, and these models can be used to predict which tests are likely to fail based on factors such as the code changes made since the last test execution and the execution history of the tests. By identifying tests that are likely to fail early on, developers and testers can focus on fixing these tests before executing them, thereby reducing the overall test execution time.

Rothermel et al. [5] provided a formal definition of the test case prioritization (TCP) problem as follows:

Definition: for a given test suite T, set of permutations of T as PT, and a function from PT to the real numbers as f.

$Find : T' \in PT \; such \; that \; (\forall T'')(T' \in PT)(T' \neq T'')[f(T') \geq f(T'')]$

According to the author's definition, PT represents the set of all possible prioritizations (orderings) of T, and f is a function that determines a score for every possible test ordering. In other words, TCP helps ensure that the most important tests are executed first, which can lead to earlier detection of defects and better overall quality. Test prioritization techniques are based on different criteria, such as the likelihood of a test case failing, the criticality of the code being tested, and the cost of fixing defects. By prioritizing tests in this way, developers and testers can ensure that the most critical tests are executed first, which can help identify defects earlier in the testing process and ultimately lead to better quality software.

Previous studies on test failure prediction prioritization in the CI context can be divided into two subcategories Heuristic-based TCP and ML-based TCP.

2.5 APPLICABILITY OF TRANSFER LEARNING (TL) IN TEST CASE PRIORITIZATION (TCP)

This section discusses the application of transfer learning (TL) in the context of test failure prediction and prioritization in the continuous integration (CI) environment. In CI, automated regression test cases are run to ensure software quality. After predicting the probability of test failures, test case prioritization (TCP) is used to reorder tests to locate faults as early as possible. ML-based approaches have shown promising results in solving TCP problems in CI due to their dynamic adaptability. However, these approaches can face challenges in creating usable models due to unbalanced data and limited available data for newer projects. Transfer learning (TL), a type of ML algorithm, addresses these challenges by leveraging knowledge from a source domain with abundant data to create a usable model for the target domain.

For most software companies, the tests run in CI to examine code modifications are different, but they share some common failure characteristics due to the same testing environment. For example, if a test has been failing in recent CI cycles, it is likely to fail in the next cycle. These statistical features of test failures are independent of the software type (e.g., web application or car application) and programming language (e.g., Java or C++). Therefore, these features can be shared for TCP across different software projects to some extent.

However, large software companies are often concerned with keeping their failure information private. Failure information from these projects can be valuable for building efficient testing models for newer projects. In this context, model-based transfer learning algorithms are particularly significant due to the data privacy concerns of giant companies

where the source domain is referential, but the data points are not available. One such algorithm is TransBoost [43], a boosting tree kernel-based transfer learning algorithm that has been used in this thesis to predict and prioritize test failures. The details of this algorithm discussed in the following section.

2.6 TransBoost

Homogeneous transfer learning algorithms assume that the source and target domains have the same feature space, denoted by $X_S = X_T$, but different data distributions, which can be marginal or conditional. For instance, the number of test runs in one project for each CI cycle may vary from others, leading to different marginal distributions. Therefore, minimizing the distribution discrepancies between domains is a crucial task for transfer learning.

This thesis employs TransBoost [43], a tree-kernel-based transfer learning algorithm that is specifically designed for commercial tabular datasets and has proven to be efficient, robust, and interpretable for real-world transfer learning applications. TransBoost leverages the instances of developed products to enhance the performance of the newer target domain.

The working process of the TransBoost algorithm is illustrated in the Figure 2.6. It builds two parallel boosting-tree models with identical tree structures but different node weights.

The algorithm starts with initializing the weight vectors $\alpha_i^S = \frac{1}{n_S}$ and $\alpha_i^T = \frac{1}{n_T}$ for the source and target instances, respectively. It then generates the two models $h_S(x)$ and $h_T(x)$, and updates them simultaneously by minimizing the transfer loss function, which is defined as:

$$TL(F_S, F_T) = E_{(x,y)\sim P_S}[\frac{P_T(x,y)}{P_S(x,y)} L(x, y; F_T(x)) - L(x, y; F_S(x))]$$

where $L(x, y; F_T(x))$ and $L(x, y; F_S(x))$ are the classification errors of the target and source models, respectively, and $P_S(x, y)$ and $P_T(x, y)$ are the joint distributions of the source and target domains, respectively.

The weight vectors are then updated using the proximal gradient descent algorithm, which is a first-order optimization algorithm that is efficient for solving large-scale optimization problems. The two models are combined using a linear combination:

$f_T(x) = \gamma h_S(x) + (1 - \gamma) h_T(x)$

where γ is a weighting parameter. The resulting model $f_T(x)$ is a boosting tree kernel-based transfer learning algorithm that can be used for test failure prediction and prioritization in the CI environment.

In summary, TransBoost is a tree-kernel-based transfer learning algorithm that is efficient, robust, and interpretable for real-world transfer learning applications. It leverages the instances of developed products to enhance the performance of the newer target domain and simultaneously reduces the distribution discrepancy between domains.

2.7 RELATED WORK

Prior research on test failure prediction and prioritization in the context of continuous integration (CI) can be broadly categorized into two subcategories: heuristic-based and machine learning (ML)-based methods.

Heuristic-based methods utilize rule-based heuristics or statistical methods to prioritize test cases based on factors such as code coverage, execution time, and past failures. While these methods are easy to implement and interpret, they may not always be effective in accurately predicting test failures or optimizing testing resources.

ML-based methods, on the other hand, utilize machine learning algorithms to analyze

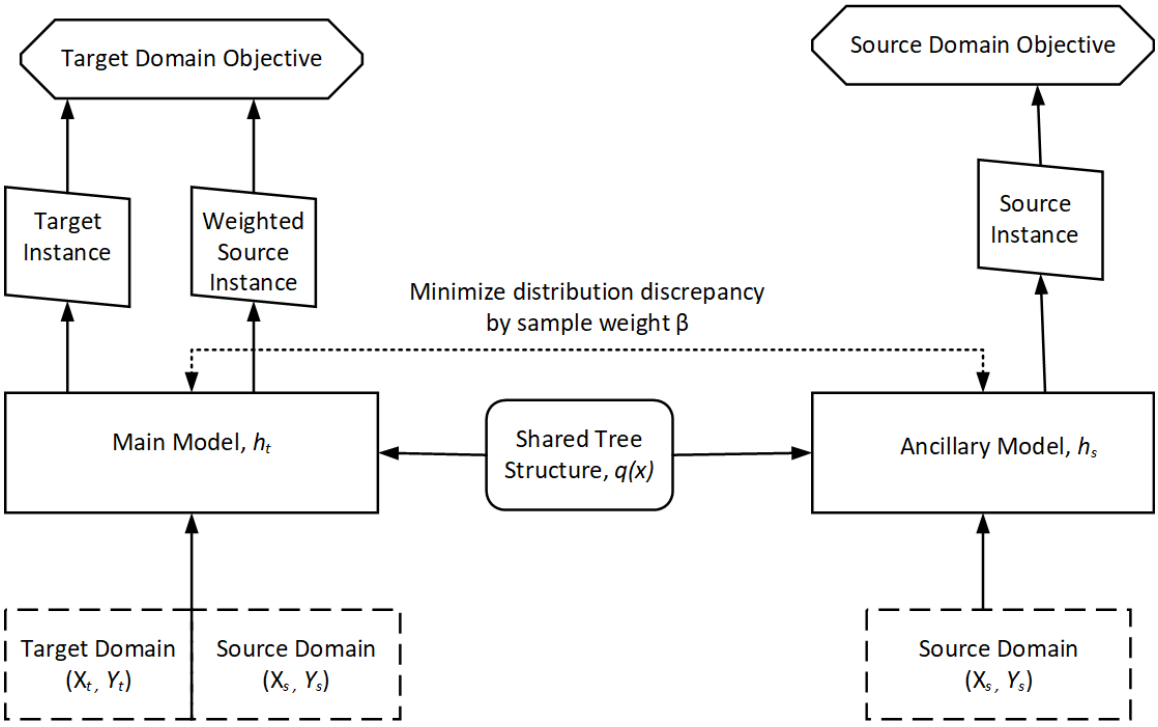

Figure 2.6: TranBoost

historical test execution data and predict the likelihood of test case failures. These methods can be more accurate and effective in prioritizing tests, as they can take into account a larger number of variables and analyze complex patterns in the data. However, ML-based methods may require significant amounts of data to train and may be more complex to implement.

Recent works on these approach are discussed in the below subsections.

2.7.1 HEURISTIC-BASED TCP METHODS

Previous studies on heuristic-based test case prioritization (TCP) mainly focused on code coverage and test case execution history. The Coverage-based heuristics assume that higher structural test coverage, such as function or statement coverage, increases the likelihood

of detecting faults in the System Under Test (SUT) [44]. However, the complexity of extracting exact coverage information is a major limitation. Two ways to collect coverage information are static and dynamic analysis. The static technique is easier to implement but less accurate, while dynamic analysis is more precise but time-consuming for large codebases, making it unsuitable for the Continuous Integration (CI) environment[8].

Execution history-based heuristics assume that test cases that failed in the previous build have a higher probability of failing in the next build. Kim and Porter [44] proposed an execution history-based TCP to compute ranking scores based on the average failure rate of test cases. Elbaum et al.[8] were the first to address the problem of multiple test requests and the implication of using historical failure data in the CI context. They proposed GoogleTCP, an algorithm that prioritizes test cases based on the time since a test's last failure or execution or when it was first introduced. Zhu et al.[45] extended this work by re-prioritizing test cases based on the co-failure probability distribution. The execution history of CI systems is easily accessible and inexpensive [9]. Recent studies [9], [46] reported that execution history-based heuristics could achieve nearly the same accuracy as ML-based TCP.

2.7.2 ML-BASED TCP METHODS

Prior studies describe test case prioritization (TCP) as a ranking problem. To solve this problem, three different ranking models [47] (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 other test cases.

Bertolino et al. [48], Lachmann et al. [49], and Tonella et al. [50] use pairwise ranking models for TCP. Bertolino et al. [48] 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. [49] consider SVM Rank [51], capable of handling large input vectors and, based on training inputs, returns a ranked classification function. Tonella et al. [50] apply the Rankboost algorithm for TCP and utilize statement coverage, cyclomatic complexity, and developers' ranking scores for model training.

On the othe hand, Busjaeger and Xie [52] use the listwise ranking model SVM MAP for TCP. Many papers consider a pointwise ranking algorithm mainly XGBoost [53], Recurrent Neural Network (RNN) [54], Neural Network (NN) [55], K Nearest Neighbour (KNN) [56], Bayesian Network [57], Logistic Regression [58], [59], and SVM [60] for TCP. In another work [61], the authors employ natural language processing (NLP) for software requirement analysis concentrating on test case prioritization.

Some recent studies [46], [62] 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 [13] 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 only CI metadata.

	VCS Metadata	CI Metadata	Study Subject	Approach
Elbaum et al. [8]		✓	1 industrial project	Heuristic
Zhu et al. [45]		✓	2 industrial projects	Heuristic
Liang et al. [63]		✓	2 industrial projects	Heuristic
Mattis et al. [23]	✓	✓	20 open-source projects	Heuristic
Busjaeger et al. [52]		✓	1 open-source project	SL & NLP
Lachmann et al. [49]		✓	1 academic and 1 industrial projects	SL & NLP
Bertolino et al. [48]		✓	6 open-source projects	RL & SL
Noor et al. [58]		✓	5 open-source projects	SL
Chen et al. [53]		✓	50 open-source projects	SL
Mahdieh et al. [55]		✓	5 open-source projects	SL
Hasnain et al. [54]		✓	5 open-source projects	SL
Palma et al. [59]		✓	5 open-source projects	SL
Elsner et al. [9]	✓	✓	20 open-source and 3 industrial projects	Heuristic & SL
Pan et al. [64]	✓	✓	242 open-source projects	SL
Spieker et al. [46]		✓	3 industrial projects	RL
Bagherzadeh et al. [62]	✓	✓	6 open-source and 2 industrial projects	RL
Shi et al. [65]		✓	3 industrial projects	RL
Carlson et al. [66]		✓	1 industrial project	UL
Rosenbauer et al. [67]		✓	3 industrial projects	RL
Rosenbauer et al. [13]		✓	4 industrial projects	TL

Table 2.1: Comparison of existing studies related to test case prioritization (TCP)

Table 2.1 provides a summary of existing techniques used for test case prioritization. From the table, it is apparent that most studies have utilized supervised learning (SL) techniques, while only a few have employed unsupervised (UL) or semi-supervised learning (SSL) algorithms for TCP. In general, UL or SSL algorithms can be more suitable when there is insufficient labeled data available. However, for TCP problems where there is sufficient labeled data, supervised learning techniques may be more appropriate due to their ability to leverage the labeled data to learn and make accurate predictions.

Both RL and 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. [68] deliver a systematic literature review of ML-based TCP and 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 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. [9] reports that CI and GitHub data are easily accessible and inexpensive. Therefore, we select *TransBoost* [43], a transfer learning algorithm that integrates the XGBoost classifier (SP algorithm) for TCP, considering VCS and CI metadata.

Chapter 3

METHODOLOGY

In this chapter, we present the methodology and workflow of our study, which is divided into two main parts: test failure prediction using transfer learning (TL) and test case prioritization (TCP) using TransBoost. The first part of our study investigates the applicability of TL for test failure prediction and examines the impact of version control system (VCS) features on TL. To achieve our research goals, we have designed a workflow as shown in Figure 3.1. At the first step, we compare traditional machine learning (ML) and TL algorithms based on their accuracy and F-measure score. Next, we compute the influence of VCS features on the best TL model. Figure 3.2 shows the system model of TL for test failure prediction, which involves selecting common features from the source and target domains, training models for each domain, and predicting test failures in the new build.

In addition to test failure prediction, we also extend our work to prioritize test cases using transfer learning, as discussed in Section 3.2. The system model for test case prioritization using TL is shown in Figure 3.3, and it consists of three successive steps: feature extraction, predictive modeling, and evaluation. In the first step, we extract VCS and CI metadata and apply feature engineering to generate useful information. Then, we feed this feature vector to a TL model, which provides test failure prediction probability. We sort

the test cases in descending order based on the predicted probability and evaluate the test case prioritization using the average percentage of fault detected (APFD) metric, a well-known metric for evaluating TCP. Data privacy is a significant concern in large software companies, and to address this issue, we have evaluated a model-based transfer learning approach for TCP. The proposed approach utilizes a kernel-based transfer learning algorithm, called TransBoost [43], and we refer to this technique as TCP_TB. The working algorithm of TCP_TB is outlined in Algorithm 1. By applying this method, we aim to improve the accuracy of TCP while maintaining data privacy.

3.1 TEST FAILURE PREDICTION USING TL

This section focuses on the practical application of transfer learning algorithms for test failure prediction, which involves two consecutive steps: building TL-based predictive models and evaluating their performance. In this regard, we evaluate a set of transfer learning algorithms used for test failure prediction and select the best-performing algorithm based on evaluation metrics. The workflow of our proposed approach is illustrated in Figure 3.1.

In the first step, we evaluate various machine learning (ML) and transfer learning (TL) algorithms on our dataset and select the best-performing models. This step involves pre-processing the datasets and training and validating the models on the pre-processed data. In the second step, we compare the best-performing ML and TL models based on their recall and F-measure scores. This comparison helps us determine the effectiveness of TL algorithms in improving test failure prediction performance compared to traditional ML algorithms.

In the final step, we assess the impact of the continuous integration (CI) and version control systems (VCs) feature sets on the performance of the TL model. This step involves

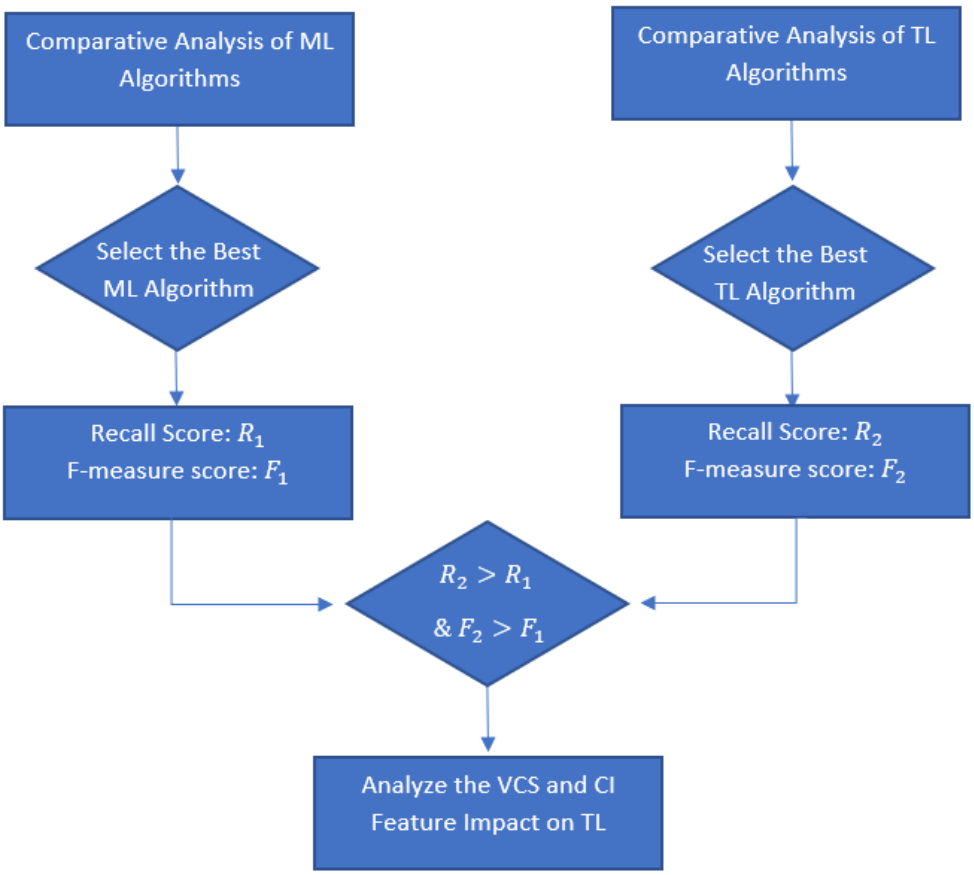

Figure 3.1: Flow chart to evaluate the applicability of TL approaches for test failure prediction

evaluating the model's performance on the dataset with and without these features to determine their impact on the model's recall and F-measure scores.

3.1.1 TL-BASED PREDICTIVE MODEL

To build an effective predictive model using transfer learning, it is necessary to have two datasets: one for creating the source model and another for building the target model. The data in these datasets should be generic, as project-specific features may not be applicable to other projects. Generic features, such as statistical features of test failures (e.g., failure

rate) and code changes (e.g., number of files changed), are examples of features that can be used.

The data log created by continuous integration (CI) and version control system (VCS) tools follows the same format, regardless of the specific tool being used. Data extraction from these sources requires little time since they are automatically generated during the CI build process. As a result, there is no time overhead in the test failure prediction process. Therefore, this thesis selects this information to feed into the predictive transfer learning model and assumes that the feature space of the source and target domains are similar. Figure 3.2 shows the system model for test failure prediction using TL.

As a result, homogeneous transfer learning algorithms seem more suitable for test suite failure prediction. Homogeneous transfer learning algorithms can be classified into four subcategories: feature-based, instance-based, parameter-based, and relation-based. The details of these approaches are discussed in section 2.3.

Among these approaches, the relation-based approach performs well on small-scale datasets with thousands of samples, but it is not appropriate for large volumes of data with millions of instances [32]. The case studies evaluated in this thesis mostly involve large-scale datasets. Therefore, we do not consider any transfer learning algorithms from this category. Instead, we focus on the other three subcategories of homogeneous transfer learning algorithms to build our predictive models. These algorithms are effective in dealing with the data volume and class imbalance issues that are commonly encountered in the context of test failure prediction in the CI environment.

Feature-based transfer learning algorithms focus on finding common features between the source and target domains that have similar behavior concerning the task at hand, resulting in similar data distribution. In this category, the following two algorithms are selected

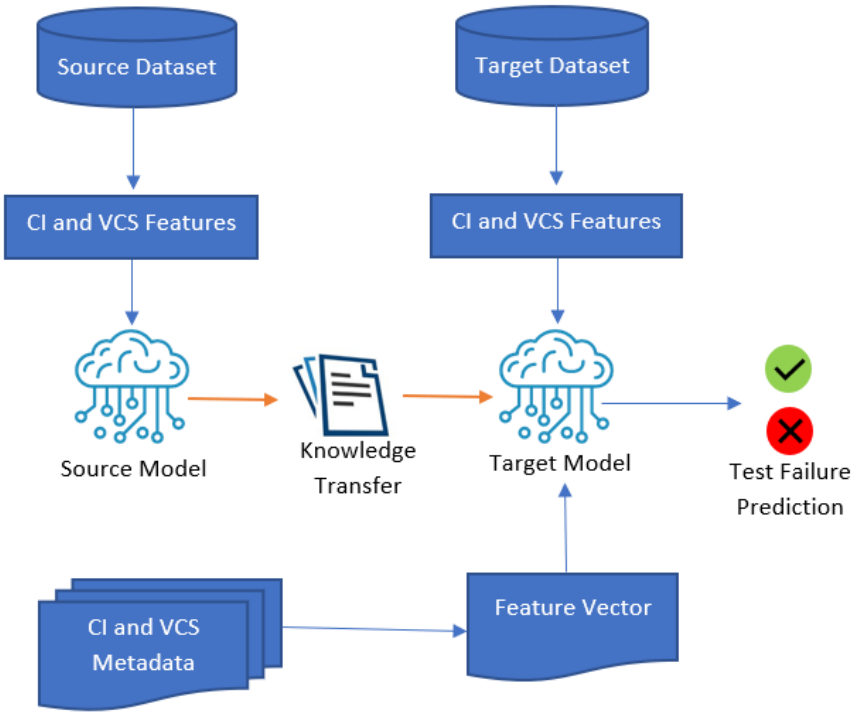

Figure 3.2: System model to predict test failures using TL

for evaluation:

- Subspace Alignment (SA) [69]: This is an unsupervised transfer learning algorithm that utilizes the PCA (Principal Component Analysis) subspace to align the source and target domains linearly. The algorithm projects the data in both domains onto a shared subspace that retains the most significant variance of the original data. This subspace is chosen to maximize the similarity between the source and target distributions. The algorithm is unsupervised, meaning that it does not require labeled data to operate.

- Correlation Alignment (CORAL) [70]: This approach utilizes second-order statistics to correct the difference between source and target distribution. Specifically, CORAL reduces the discrepancy between the source and target domain by aligning

their second-order statistics, namely, their covariance matrices. The algorithm maps the source and target data onto a common subspace to align their covariance matrices. This alignment results in a better match between the source and target distributions, which is critical for transfer learning.

Both SA and CORAL are unsupervised feature-based transfer learning algorithms that do not require labeled data to operate. They are effective in transferring knowledge from the source domain to the target domain by aligning their data distribution.

Instance-based transfer learning algorithms aim to reweight the labeled training data to minimize the domain shift between source and target domain data distributions. This thesis selects the following two instance-based TL algorithms.

- Nearest Neighbors Weighting (NNW) [71]: As the name suggests, this approach relies on the nearest neighbor algorithm to reweight the source instances based on their number of neighbors in the target dataset. The reweighted source data is then used to train the target model. NNW is effective for scenarios where there is a limited amount of labeled data in the target domain, as it can leverage the labeled data in the source domain to improve the predictive performance on the target domain.

- Transfer AdaBoost (TrAdaBoost) [72]: This is a supervised transfer learning algorithm that relies on a ”reverse boosting” principle where at each boosting iteration, the weights of the source instances decrease, which causes poor predictions. The target model is then trained on the weighted source data, which allows it to perform better on the target dataset.

These algorithms are particularly effective for scenarios where there is a limited amount of labeled data in the target domain, as they can leverage the labeled data in the source

domain to improve the predictive performance on the target domain. However, these algorithms are sensitive to the distribution shift between the source and target domains, and they may not work well when the data distributions are significantly different.

Parameter-based transfer learning algorithms assume that if two tasks are related, then the structure of a well-trained model on the source domain can be transferred to the target model. This approach can be particularly useful when the feature spaces of the source and target domains are similar but their data distributions are different. The following algorithms are examples of parameter-based transfer learning algorithms that are commonly used in the context of test failure prediction:

- TransferForestClassifier [73]: This algorithm modifies a random forest model learned on source domain data using sampled data from the target domain. By adapting the decision boundaries of the source domain model to the target domain, the algorithm is able to improve the predictive performance on the target domain.

- TransferTreeClassifier [74]: This algorithm modifies a decision tree learned on the source domain using a training set of sampled data collected from the target domain. By adjusting the splitting criteria of the tree based on the target domain data, the algorithm is able to improve the predictive performance on the target domain.

- TransBoost [75]: This is a tree-kernel-based transfer learning algorithm based on the XGBoost algorithm. It considers two parallel boosting-tree models with similar tree structures but distinct node weights. This specific design ensures the robustness and interpretability of the tree-based models. The algorithm trains models and reduces the discrepancy between domains simultaneously within $O(n)$ complexity compared to the traditional kernel method. The details of this algorithm is discussed in subsection 2.6.

These algorithms are effective in scenarios where the source and target domains share similar feature spaces but have different data distributions. By leveraging the knowledge gained from the source domain, they can improve the predictive performance on the target domain.

3.1.2 EVALUATION SETUP FOR TEST FAILURE PREDICTION

In the context of test failure prediction, most real-world datasets are imbalanced, with the number of failed tests being relatively small compared to the number of successful tests. Therefore, accuracy alone is not an efficient way to measure the performance of a predictive model. Instead, this thesis focuses on Recall and F-measure metrics for evaluation.

Recall, also known as sensitivity or True Positive Rate (TPR), measures the model's ability to detect the positive class by calculating the ratio between correct positive predictions and all possible positive predictions. A high recall score means that the model is good at detecting the relevant test cases that are likely to fail in the next CI cycle.

F-measure is the harmonic mean of precision and recall. It takes into account both precision and recall and is a more robust metric for imbalanced datasets. F-measure considers both the false positive rate (FPR) and the false negative rate (FNR) and provides a balance between the two. A higher F-measure score indicates a better balance between precision and recall, which is desirable for test failure prediction.

Therefore, using Recall and F-measure metrics is suitable for evaluating the performance of predictive models in the context of test failure prediction, especially for imbalanced datasets.

The following equations are used to compute the above-mentioned metrics:

$$precision = \frac{TP}{TP + FP}$$
$$recall = \frac{TP}{TP + FN}$$
$$F - measure = \frac{2 * precision * recall}{precision + recall}$$

To calculate these metrics,it is necessary to identify the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) from the prediction results. For this scenario,

- TP (True Positive): indicates the number of failed tests correctly identified correctly identified,
- TN (True Negative): represents the number of passed tests correctly identified,
- FP (False Positive): points out the number of passed tests misclassified as failed tests, and
- FN (False Negative): indicates the number of failed tests misclassified as passed.

3.2 TEST CASE PRIORITIZATION USING TL

The second part of the proposed approach involves using transfer learning (TL) models to prioritize test cases based on their failure probability. The approach takes into consideration both VCS and CI metadata. To define the problem and notations, let S be a system under test, C be a code change that 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 based on a feature vector F. The probability values are then sorted in descending order to rank the test suites in T_s. Depending on the test budget, either the

complete prioritized list (TCP), T'_s, or a subset of it (TCS) will be executed to evaluate S' [9], [52], [76]. After evaluating S', the feature vector and test results are added to the dataset as historical data. This thesis utilizes *TransBoost* TL for TCP and names this approach as *TCP_TB*.

The system model of the proposed approach is shown in Figure 3.3, and it consists of three successive steps: feature extraction, predictive modeling, and evaluation. The details of these steps are discussed in the following subsections.

3.2.1 FEATURE EXTRACTION

This thesis utilizes information from both the VCS and CI log to generate feature vectors. Based on the configuration, a CI cycle can be triggered either after a code change push to VCS or after the previous cycle. The CI features are closely related to predicting the outcome of a specific test in a CI build. They provide information about the test suite's execution, its history, and its failure rate. These features are useful in understanding the behavior of individual test suites and their impact on the overall build result. On the other hand, the VCS features define how the introduced commits affect the likelihood of failure for all test suites in a CI build. They provide information about the changes made to the source code repository and how they affect the build's outcome. These features are helpful in understanding the impact of code changes on the overall build result and can aid in identifying potential issues and improving the quality of the software.

The details of VCS and CI, as well as the relationship between them, are discussed in section 2.1. The metadata from these resources contains various information, including commit identifier, author, commit timestamp, message, change set (i.e., the number of

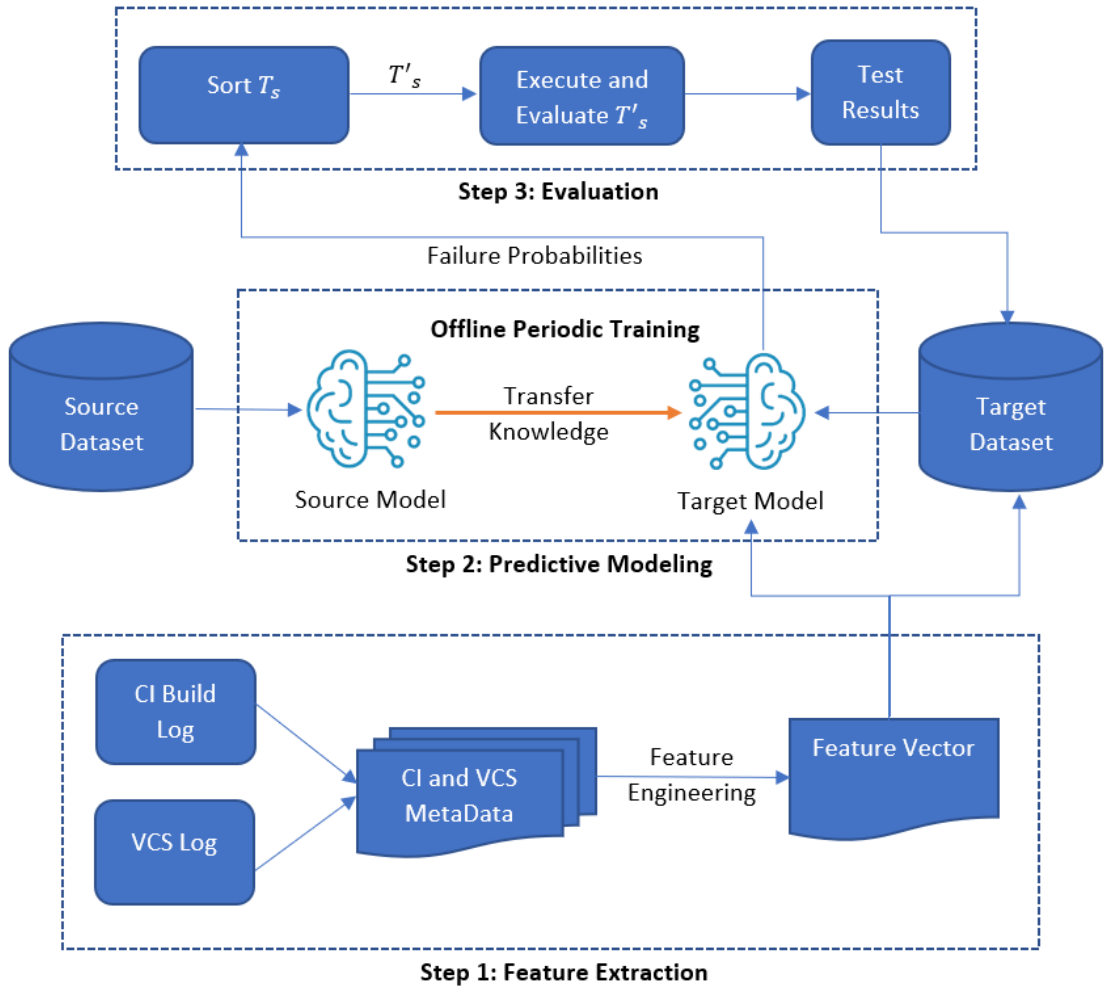

Figure 3.3: System model for test case prioritization using TL

files added, modified, or deleted), build identifier, build result, build duration, test identifier, number of assertions in a test, test result, and test duration. However, some of these information are stored as *object* types and cannot be used directly. For example, the 'test identifier' contains the name of each test suite, and since its data type is *object*, it needs to be mapped to an integer value to be used as a feature.

Most of the prior studies [68] on TCP considered the 'last execution time' or 'average

test execution time' feature for model training and achieved good results, yet it can be misleading. The hypothesis behind considering this feature is that the test suite with higher execution time may cover more significant parts of the code and thus may contain more faults. However, many other factors can influence the test execution time, such as machine speed. In large-scale CI projects, the test runs parallel on different machines, which do not necessarily have the same configuration. A test suite run on a low-configuration machine can have a higher execution time which contradicts the above hypothesis. Therefore, this thesis paper does not consider any features that are solely dependent on the test execution time.

The extracted metadata from both VCS and CI is used to extract nine CI and three VCS features for the *TCP_TB* input. In line with previous studies [9], we provide the definition of each of these features below:

CI Features:

- Test identifier: A unique integer value that represents each test suite.
- Test suite runs [63]: The number of times a test suite has been executed.
- Test suite assertions [63]: The total number of assertions in the test suite.
- Failure count [3], [9], [58], [77], [78]: The total number of times a test suite has failed.
- Last failure age [3], [8], [9], [79]: The number of CI builds since the last failure.
- Last failure: A Boolean value that indicates whether the test suite failed in the last build.
- Failure rate [7]: The percentage of test suite failure in the previous run.

- Transition count [80]: The total number of times a test suite changed its state (e.g., from pass to fail or fail to pass).
- Last transition [80]: The number of CI builds since the last transition.

VCS Features:

- *#Files_Changed* [3], [77], [81]: The number of files changed as a result of a specific change.
- *#Lines_Inserted*: The number of lines of code added in all modified files as a result of a specific change.
- *#Lines_Deleted*: The number of lines of code deleted in all modified files as a result of a specific change.

3.2.2 Predictive Modeling

The most straightforward way to create a predictive transfer model is by training a model on the source dataset and then transferring it to the target domain for further training with the target dataset. The resulting model inherits informative patterns from the source domain and fine-tunes them with the target data points. However, if the data distribution in the source and target domain is different, directly sharing the model can be ineffective due to data drifting in previous analyses. To overcome this issue, this paper selects *TransBoost*, which uses XGBoost to generate two parallel boosting-tree models (source and target) with identical tree structures but different node weights. The details of *TransBoost* are discussed in Section 2.6.

The source model is trained on a source dataset with a relatively large data volume, a

high test failure rate, or both, and then shares its knowledge with the target model to improve the test case prioritization (TCP). The performance of the predictive transfer learning (TL) model decreases over time, but retraining them in each cycle can cause computational overhead on the CI build. Therefore, the TL model is periodically retrained in an offline environment to maintain its performance and avoid computational overhead.

A feature vector generated from the historical CI test data and VCS changeset is passed 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. The list of probability scores is then sorted in descending order. The algorithm 1 represents the working algorithm of *TCP_TB*.

When training a predictive transfer learning (TL) model, selecting the appropriate source dataset is a significant challenge. In natural language processing and computer vision, it is common practice to choose a pre-trained model that performs well on a large dataset, which can help improve predictions for small projects [82], [83]. However, with large-scale datasets, sufficient data is available to construct efficient machine learning (ML) models, but data volatility poses a challenge. Some studies [15] only consider the most recent data when building ML models to address test case volatility, as old tests may be removed, and new ones may be added to the system. Running the ML model with outdated data can decrease its performance.

Roza et al. [15] proposed a sliding window-based ML model for TCP. They discard old data outside the sliding window, and if the test case information is not present in the window, they consider that test as a new one. However, this approach may negatively impact the prediction model's performance as some old tests may still be relevant. Therefore, TL can help large-scale projects by considering the old data as a source and refining it with

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 data = $\{D_S, D_T\}$;
Function `feature_extraction` (T_s, E_T, C) **:**
    for $i = 0$; $i < length(T_s)$; i++ **do**
        $f_s[i] \leftarrow E_{Ti}$;
    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;

recent data to build a more efficient prediction model.

This thesis follows the sliding window concept of Roza et al.[15]. The data inside the window range is considered as the target domain data, and the data before the window range is considered as the source domain data and name this process internal domain knowledge transfer. Figure3.4 shows an example of creating a source and target dataset within the same domain.

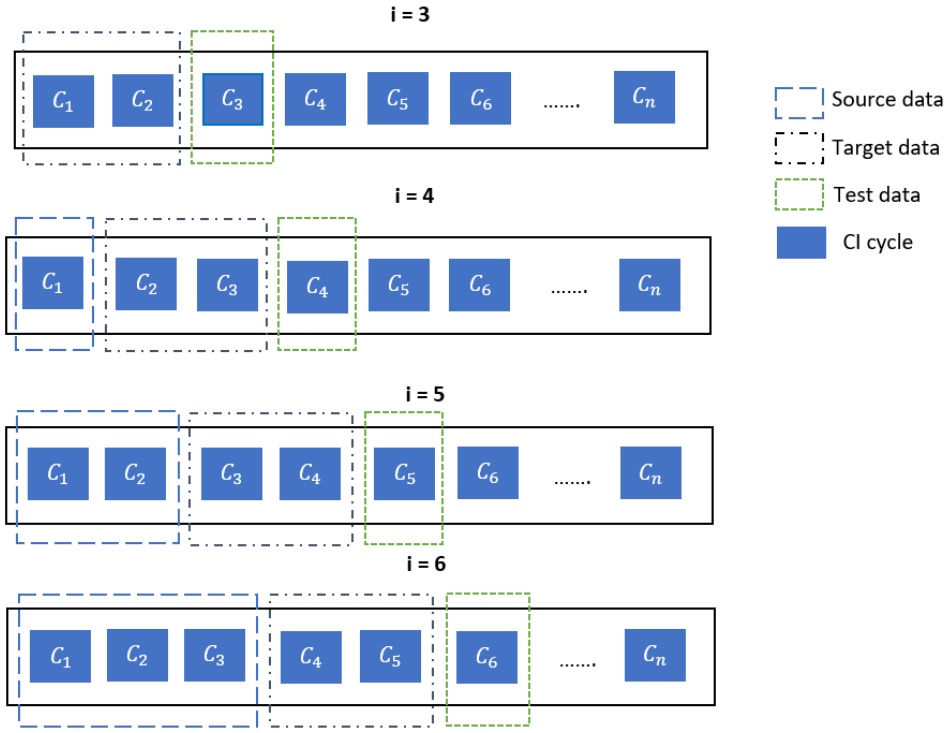

Figure 3.4: Splitting dataset, D for internal domain knowledge transfer using *sliding window* (window size=2) and *forward chaining*

3.2.3 EVALUATION SETUP FOR TCP

Training and Testing Splits

Determining the appropriate training-testing split ratio for a dataset or the volume of training data needed to prepare a suitable model is not a straightforward task. For instance, one study [81] used one year of data for training the model and tested it with the last two months of data. As the performance of the predictive model decays with time, it requires periodic updates. In this paper, we perform periodic offline training and use all available historical data for training. Due to the temporal dependency between CI cycles, it is not reasonable to perform k-fold cross-validation. Thus, it is more realistic to train models on past data and test on more recent data [81].

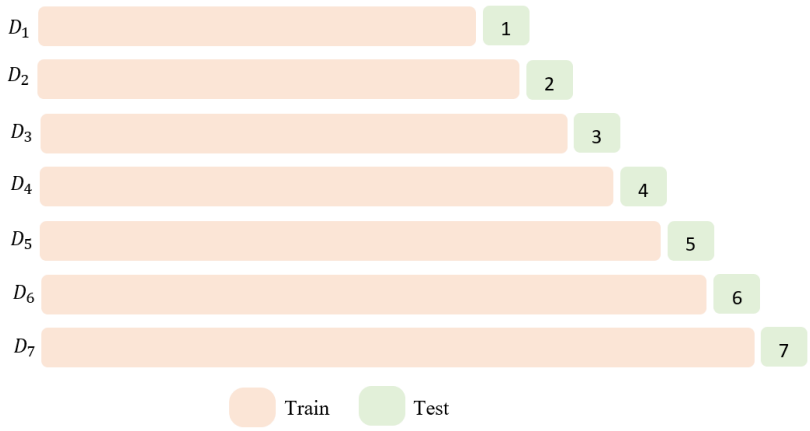

Figure 3.5: Split dataset, D using *forward-chaining*

Figure 3.5 shows the training-testing splits used in this thesis to divide a time-series target dataset, D. We follow the *forward chaining* method for splitting, which performs cross-validation on a rolling basis. First, we take a small subset of data for training and predict the probability of failure for the later data points. We then incorporate the same predicted data points into the subsequent training dataset and test the subsequent data points.

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 the evaluation of test case prioritization as these metrics do not consider position in the equation. The average percentage of fault detected (APFD) [76] is a widely accepted metric for TCP 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 [9]. We assume each failed test cases identifies

	t_A	t_B	t_C	t_D	t_E	t_F	t_G	t_H	t_I	t_J
F_1		x								
F_2			x							
F_3				x						
F_4								x		
F_5									x	
F_6										x
F_7					x					

Table 3.1: Test order: T_1 $\{t_A, t_B, t_C, t_D, t_E, t_F, t_G, t_H, t_I, t_J\}$

one unique failure. APFD indicates how quickly a set of prioritized test list T'_s can detect the failures present in the system under test, and its value is measured from the weighted average of the percentage of detected failures. The APFD ranges from 0 to 1, with higher values indicating that the failures are detected more quickly.

Given a set of prioritized test T'_s and a set of test failures F, the APFD is calculated as:

$$APFD = 1 - \frac{\sum_{i=1}^{|T'_s|}(TF1 + TF2 + ... + TFm)}{m \times n} + \frac{1}{2n} \tag{3.1}$$

where $TF1$ is the position of the first test failure in T'_s and n and m are the total number of test and failures, respectively.

In addition, we consider the normalized version of APFD, NAPFD [84], to measure the effectiveness of *TCP_TB* for TCS. The NAPFD can be calculated as follow,

$$NAPFD = P - \frac{\sum_{i=1}^{|T'_s|}(TF1 + TF2 + ... + TFm)}{m \times n} + \frac{P}{2n} \tag{3.2}$$

where, p is the ratio between the number of failures detected for a specific test budget and the total number of test failures.

	t_I	t_J	t_E	t_B	t_C	t_D	t_F	t_G	t_H	t_A
F_1				x						
F_2					x					
F_3						x				
F_4									x	
F_5	x									
F_6		x								
F_7			x							

Table 3.2: Test order: T_2 $\{t_I, t_J, t_E, t_B, t_C, t_D, t_F, t_G, t_H, t_A\}$

Let's walk through an example of calculating APFD and NAPFD. Suppose we have two different test case prioritization orders, T_1 and T_2, shown in Tables 3.1 and 3.2, respectively. According to Equation 3.1, we can determine the APFD values for T_1 and T_2, where n (the total number of tests) is 10 and m (the total number of failures) is 7.

$$APFD(T_1) = 1 - \frac{2+3+4+8+9+10+5}{7 \times 10} + \frac{1}{2 \times 10} = 0.4643$$

$$APFD(T_2) = 1 - \frac{4+5+6+9+1+2+3}{7 \times 10} + \frac{1}{2 \times 10} = 0.6214$$

From the above APFD values, we can see that prioritization order T_2 is better than T_1.

Now, let's consider that we have only a 50% test budget. Then, T_1 and T_2 will be able to detect four and five failures, respectively. So, the P value of T_1 and T_2 will be (4/7) and (5/7), respectively. Finally, we can calculate the NAPFD values based on Equation 3.2 for T_1 and T_2 as follows:

$$NAPFD(T_1) = 0.5714 - \frac{2+3+4+5}{7 \times 10} + \frac{0.5714}{2 \times 10} = 0.4000$$

$$NAPFD(T_2) = 0.7143 - \frac{4+5+1+2+3}{7 \times 10} + \frac{0.7143}{2 \times 10} = 0.5357$$

For the case selection, the test order T_2 is also better than T_1.

Chapter 4

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter presents our experimental setup and results for evaluating transfer learning (TL) in test failure prediction and test case prioritization (TCP). We divided this evaluation process into two sections, each focused on achieving specific goals.

The first section evaluates various TL algorithms for predicting test failures in three case studies and compares the results with three traditional tree-based machine learning (ML) algorithms. We also analyze the impact of the features used for TL to validate our choice of information selection. The experimental setup and results of test failure prediction are presented in section 4.1.

The second section proposes a novel transfer learning-based test case prioritization method called *TCP_TB*. This method uses test execution history and version control system (VCS) changeset features to efficiently reorder test suites. We evaluate the performance of *TCP_TB* against the state-of-the-art ML approach, *CI-RTP/S*, on 24 study subjects. We also analyze the potential of internal domain knowledge transfer for large-scale projects. The experimental setup and results of test case prioritization are presented in section 4.2.

To ensure the reliability of our experimental results, we also discuss the possible threats to our findings in terms of internal and external validity. These are presented in the final

section of this chapter

4.1 TEST FAILURE PREDICTION EVALUATION

This section presents the experimental setup and results to evaluate the performance of various transfer learning algorithms in predicting test suite failures for large-scale industrial datasets. The main objectives of these experiments are:

- To compare different transfer learning algorithms for predicting test suite failures.
- To analyze the impact of version control system (VCS) features on predicting test suite failures using transfer learning.

We first provide a brief overview of the case studies and then analyze the experimental results.

4.1.1 Case Studies

Table 4.1 presents the six software projects used in our experiments, along with their descriptive statistics such as the number of builds, the number of failing builds, and the percentage of test failure. Five of these projects (three industrial and two open-source) are collected from [9], while the remaining project (*Open_liberty*) is obtained from IBM. The test execution results of industrial projects [9] comprise both unit and integration testing and are obtained from IVU Traffic Technologies. One of the industrial projects, *IVU_Cpp*, is primarily written in C/C++, while the other two (*IVU_Java_1* and *IVU_Java_2*) are written in Java. The *Open_liberty* project is also written in Java and contains both unit and integration testing.

Project	Time Period (days)	CI Cycles	#Tests	#Test Executions	#Test Failures	Test Failure Rate
IVU_Cpp	267	3996	1240	3608.4K	25973	0.72%
IVU_Java_2	699	3209	1278	3526.4K	7603	0.22%
IVU_Java_1	313	943	279	178.7K	14568	8.15%
buck	307	846	864	586.1K	1511	0.26%
sling	213	1403	304	268.1K	1158	0.43%
Open_liberty	803	3421	1051	2408.8K	20639	0.86%

Table 4.1: Description of case studies

We consider *IVU_Cpp*, *IVU_Java_2*, and *Open_liberty* as the target domain for evaluating the transfer learning performance on large-scale projects, as they are representative of large-scale projects and require a significant amount of time for testing. To evaluate the transfer learning performance on these target domain projects, we use the following industrial projects (i.e., *IVU_Cpp*, *IVU_Java_2* and *IVU_Java_1*) along with two large open-source projects, *buck* and *sling*, which have comparatively higher test failure rates, as the source.

The datasets collected from the Elsner et al. [9] paper contain sixteen features extracted from VCS and CI data, as shown in Table 4.2. Among these, $F1$ and $F4$ consist of only CI and VCS features, respectively, while $F2$ and $F3$ include features mapped from both CI and VCS metadata. However, one of the features named 'average test execution time' can be misleading for projects that run in parallel environments. The hypothesis behind this feature is that the test suite with higher execution time may cover a larger area of the code and thus may contain more faults. However, many other factors, such as machine speed, can influence test execution time. In large-scale CI projects, tests run in parallel on different machines, which do not necessarily have the same configuration. As a result, a test suite run on a low-configuration machine can have a higher execution time, which

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 4.2: Features of CI-RTP/S approaches [9]

contradicts the above hypothesis. Therefore, this paper does not consider the test execution time feature because our case studies are large-scale industrial projects that run their tests in a parallel environment.

On the other hand, the dataset provided by the IBM *Open_liberty* project has six features, including *Test_Identifier*, *Test_Total*, Max. file path token similarity, and Changeset cardinality (*#Files_Changed*, *#Line_Inserted*, and *#Line_Deleted*). Only four of these features (i.e., Max. file path token similarity and Changeset cardinality (*#Files_Changed*, *#Line_Inserted*, and *#Line_Deleted*) match with the features of the other datasets. Therefore, knowledge transfer from the other datasets to the *Open_liberty* dataset is challenging due to the limited number of matching features.

4.1.2 COMPARISON OF DIFFERENT TL ALGORITHMS

This paper evaluates popular TL algorithms used in classification problems, as described in subsection 3.1.1. The performance of each model is assessed based on two evaluation metrics: recall and F-measure. Additionally, this paper compares the performance of TL algorithms with the three most popular tree-based classification algorithms: decision tree (DT), random forest (RF), and XGBoost, which are known for their fast performance compared to neural network-based ML algorithms. However, a cost analysis model would be necessary for a complete analysis. All algorithms are implemented using the scikit-learn and adapt python libraries, and default hyperparameters are used for each model. The *IVU_Java_2*, *IVU_Cpp*, and *Open_liberty* datasets include test execution histories from 2019 to 2020, with the last month of testing data used to predict failure and the rest used for ML training.

Table 4.3 compares the performance of various ML models, where we find that RF performs better for *IVU_Cpp* and *IVU_Java_2* datasets in terms of both Recall and F-measure (highlighted in bold). Elsner et al. [9] also reported that RF is the best performing algorithm for these two datasets. We evaluated the performance of TL algorithms on the same testing set, considering the training set of the ML model as the target domain and data from other projects as the source domain.

Table 4.4 presents the TL algorithms' performance on the *IVU_Java_2* dataset, with the best-performing algorithms highlighted in bold. The results show that parameter-based TL algorithms outperformed the instance and feature-based approaches. *TransferForestClassifer* (source: *IVU_Cpp*) performed the best among the TL and ML algorithms, boosting the recall and F-measure values by 0.64% and 0.21%, respectively, compared to RF. These results suggest that the *IVU_Cpp* dataset can be more beneficial as a source domain for the *IVU_Java_2* dataset.

	DT			RF			XGBoost		
	recall	precision	F-measure	recall	precision	F-measure	recall	precision	F-measure
IVU_Cpp	0.8348	0.5882	0.6901	**0.8748**	**0.8895**	**0.8821**	0.8679	0.8893	0.8785
IVU_Java_2	0.8585	0.1501	0.2555	**0.8907**	**0.8993**	**0.8950**	0.8360	0.8755	0.8553
Open_liberty	**0.7208**	0.8193	**0.7669**	0.6738	0.8520	0.7525	0.4707	**0.9703**	0.6339

Table 4.3: Performance of ML algorithms on IVU_cpp, IVU_Java2, and Ibm_openliberty dataset and highest values are highlighted in bold

Algorithm	IVU_Cpp			IVU_Java_1			buck			sling		
	recall	precision	F-measure	recall	precision	F-measure	recall	precision	F-measure	recall	precision	F-measure
SA	0.1222	0.0077	0.0145	0.4437	0.0555	0.0986	0.0193	0.0142	0.0164	0.0836	0.0017	0.0033
Coral	0.4823	0.1670	0.2481	0.3826	0.0276	0.0515	0.7267	0.0020	0.0039	0.4469	0.0376	0.0693
NNW	0.8296	0.1301	0.2249	0.8778	0.0244	0.0474	0.8103	0.0854	0.1545	0.7331	0.0055	0.0110
TrAdaBoost	0.8585	0.7437	0.7970	0.8392	0.4350	0.5730	0.8521	0.6065	0.7086	0.8424	0.6023	0.7024
TransferTreeClassifier	0.7203	0.9143	0.8058	0.8746	0.5562	0.6800	0.8939	0.8967	0.8953	0.7942	0.7694	0.7816
TransferForestClassifier	**0.8971**	0.8971	**0.8971**	0.8939	0.8967	0.8953	0.8071	0.9128	0.8567	0.3698	**0.9502**	0.5324
TransBoost	0.8296	0.8897	0.8586	0.8842	0.8959	0.8900	0.8907	0.8963	0.8935	0.8810	0.8868	0.8839

Table 4.4: Performance of TL algorithms on IVU_Java2 dataset for different sources and highest values are highlighted in bold

Table 4.5 presents the performance of TL algorithms on the *IVU_cpp* dataset, with the best performances highlighted in red. Similar to the *IVU_java_2* dataset, *TransferTreeClassifier* (source: *IVU_java_2*) achieved better recall (0.8782) and F-measure values (0.8830) than other TL and ML algorithms. Based on these results, we observe that there is an opportunity to improve test failure prediction in large-scale industrial projects if the source domain has a large volume of data and a higher test failure rate.

Algorithm	IVU_Java_2			IVU_Java_1			buck			sling		
	recall	precision	F-measure	recall	precision	F-measure	recall	precision	F-measure	recall	precision	F-measure
SA	0.0715	0.0138	0.0231	0.4368	0.0064	0.0126	0.5563	0.0066	0.0130	0.5186	0.0129	0.0252
Coral	0.6535	0.0778	0.1390	0.5483	0.0093	0.0183	0.0663	0.0073	0.0132	0.5546	0.0155	0.0301
NNW	0.7782	0.3117	0.4451	0.8365	0.0098	0.0194	0.8222	0.0489	0.0923	0.6421	0.0047	0.0094
TrAdaBoost	0.8210	0.7925	0.8065	0.8285	0.7667	0.7964	0.8113	0.7581	0.7838	0.8182	0.7392	0.7767
TransferTreeClassifier	0.8634	0.8851	0.8741	0.8645	0.9004	0.8821	0.8782	0.8788	0.8785	0.8651	0.7704	0.8150
TransferForestClassifier	**0.8782**	0.8879	**0.8830**	0.8780	0.8800	0.8790	0.8719	0.8905	0.8811	0.8130	**0.9087**	0.8582
TransBoost	0.8679	0.8877	0.8777	0.8765	0.8795	0.8780	0.8776	0.8792	0.8784	0.8771	0.8795	0.8783

Table 4.5: Performance of TL algorithms on IVU_Cpp dataset for different sources and highest values are highlighted in bold

Algorithm	IVU_Cpp			IVU_Java_2			IVU_Java_1		
	Recall	Precision	F-measure	Recall	Precision	F-measure	Recall	Precision	F-measure
SA	0.4753	0.0094	0.0184	0.5908	0.0085	0.0168	**0.8684**	0.0086	0.0170
Coral	0.1274	0.0065	0.0123	0.1324	0.0098	0.0183	0.1581	0.0089	0.0168
NNW	0.1340	0.0090	0.0168	0.2460	0.0077	0.0149	0.3017	0.0082	0.0160
TrAdaBoost	0.4178	0.0326	0.0604	0.4211	0.0324	0.0602	0.4348	0.0326	0.0607
TransferTreeClassifier	0.0007	0.0042	0.0012	0.0009	0.0085	0.0017	0.0007	0.0042	0.0012
TransferForestClassifier	0.0000	0.0000	0.0000	0.0005	0.0012	0.0007	0.0004	0.0020	0.0007
TransBoost	0.0000	0.0000	0.0000	0.0003	0.0015	0.0005	0.0007	0.0042	0.0012

Table 4.6: Performance of TL algorithms on Open_liberty dataset for different sources and highest values are highlighted in bold

Table 4.6 displays the performance of TL algorithms on the *Open_liberty* dataset, with the best performance highlighted in red. The feature-based TL algorithm, *SA* outperforms the decision tree in terms of recall value (0.8684). However, the F-measure value is quite low, at 0.0170. This result indicates that the model is making random predictions and thus is not efficient. The TL algorithms perform poorly for the *Open_liberty* dataset as it has been trained with fewer features than the ML algorithms, with all the features being VCS features. To attain good TL performance, both VCS and CI features are required.

4.1.3 CI AND VCS FEATURES IMPACT ON TL MODEL PREDICTION

The *TransferForestClassifier* has demonstrated the best performance among the TL algorithms for the *IVU_Java_2* and *IVU_Cpp* datasets. According to Elsner et al.[9], VCS metadata is inexpensive and easily accessible. Also does not have any language dependencies. Additionally, we have found that ML models trained with both CI and VCS metadata achieve better performance, indicating that VCS metadata greatly influences ML model training. To investigate the impact of VCS metadata on TL performance, we have considered two feature sets: one including all the features (All') and the other containing only the CI features (CI'). Figures 4.1 and 4.2 display the *TransferForestClassifier's* performance on the *IVU_Java_2* and *IVU_Cpp* datasets, respectively, for these two feature sets.

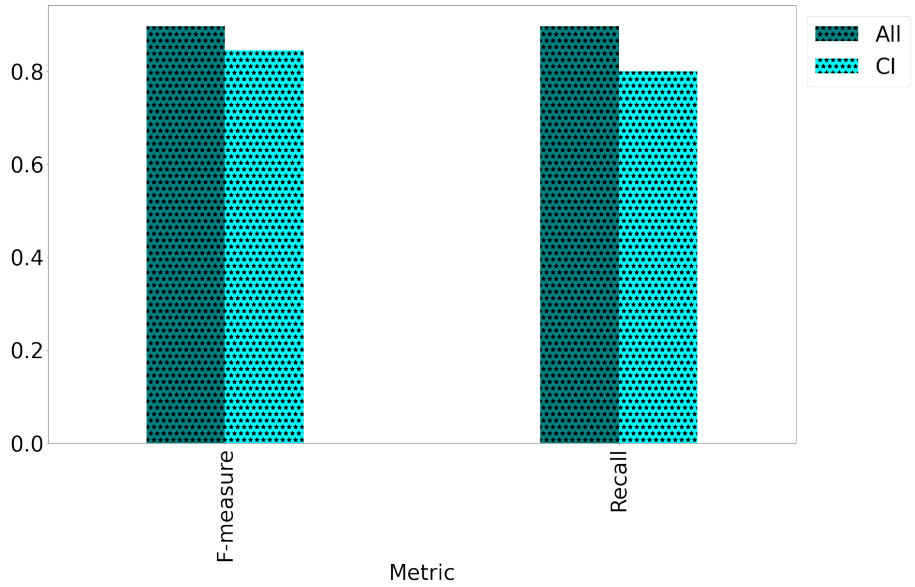

Figure 4.1: 'All' and 'CI' feature sets impact on TransferForestClassifier of IVU_Java2 dataset

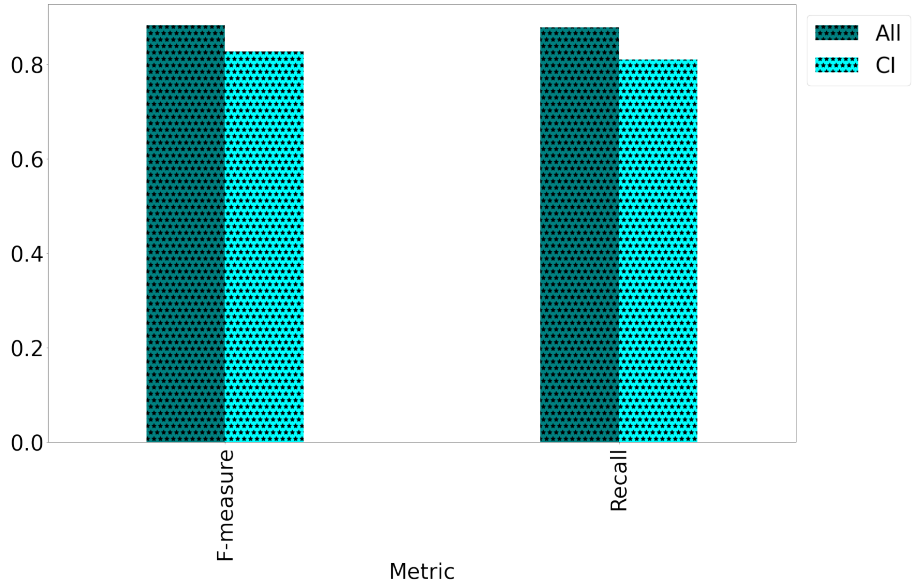

Figure 4.2: 'All' and 'CI' feature sets impact on TransferForestClassifier of IVU_Cpp dataset

The results of the experiment show a significant drop in the recall and F-measure values of the TransferForestClassifier for both the *IVU_Java_2* and *IVU_Cpp* datasets when VCS features are removed. As shown in Figure 4.1, the recall and F-measure values for the *IVU_Java_2* dataset decreased by 9.65% and 5.16%, respectively. Similarly, the recall and F-measure values for the *IVU_Cpp* decreased by 6.82% and 5.53%, respectively, as shown in Figure 4.2. This experiment highlights the importance of VCS metadata in the performance of TL algorithms. Therefore, the findings suggest that for better prediction by transfer learning algorithms, both CI and VCS metadata are necessary.

4.1.4 CONCLUSION

This experiment addresses the challenge of predicting test suite failures, particularly for large-scale datasets with imbalanced class distributions. The results show that TL algorithms can improve the test failure prediction rate for such datasets with low failure rates. Among the various TL algorithms, *TransferForestClassifier* (parameter-based) outperforms Random Forest by boosting the recall value of *IVU_Java_2* and *IVU_Cpp* datasets by 0.64% and 0.34%, respectively. However, it is worth noting that *TransferForestClassifier* requires both CI and VCS features to achieve better predictions.

4.2 TCP_TB EVALUATION

We conduct several experiments to evaluate the effectiveness of our proposed approach, *TCP_TB*, and present the experimental results addressing the research questions outlined in Section 1.4.

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 4.7: Overview of the study subjects

4.2.1 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 [9]. The details of all these 24 projects are listed in Table 4.7 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 4.7 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.

4.2.2 BASELINE METHOD

In order to determine the efficiency of transfer learning approach for improving TCP performance, it is important to compare it with other popular ML models. Therefore, we evaluated ten commonly used ML models, including 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 (SVM), on each study subject using the feature set (f_{new}) described in Section 3.2.1. These ML models have been employed in previous studies at least once, and we used the default hyperparameters for running these algorithms, similar to [9]. The best ML results for each study subject are stored in Table 4.8, and we refer to this baseline method as *TCP_ML*. The goal is to compare the performance of *TCP_TB* with that of *TCP_ML* and assess whether the former outperforms the latter.

	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 4.8: Mean and standard deviation APFD values of *CI-RTP/S*, *TCP_ML*, and *TCP_TB*. The highest values are highlighted in bold

4.2.3 IDENTIFICATION OF POTENTIAL SOURCE DATASET

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 4.7 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 *TCP_ML*. Figure 4.3 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 subject which 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 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.

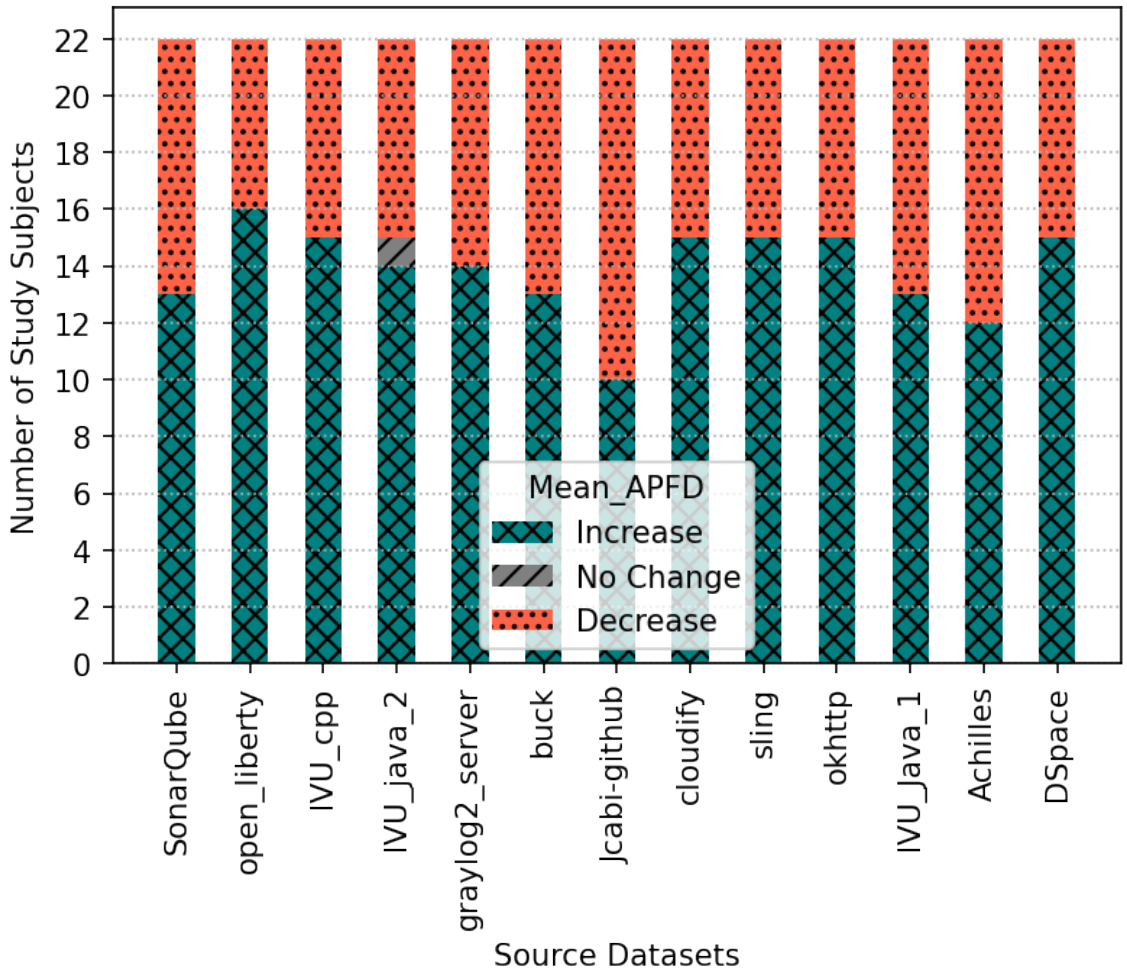

Figure 4.3: The performance of source datasets for 23 study subjects compared to *TCP_ML*

4.2.4 INTERNAL DOMAIN KNOWLEDGE TRANSFER

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 [9], [15] 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 the system. Thus, running the ML model with old data can decrease the model's performance. Roza et al. [15] 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 range will be considered target domain data, and the data before the target window will be considered source domain data. We name this process internal domain knowledge transfer. The example of creating a source and target dataset within the same domain is shown in Figure 3.4.

We evaluate the performance of internal domain knowledge transfer with the same 13 source datasets. We set each project's window size with a minimum of six months of previous data. Figure 4.4 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.

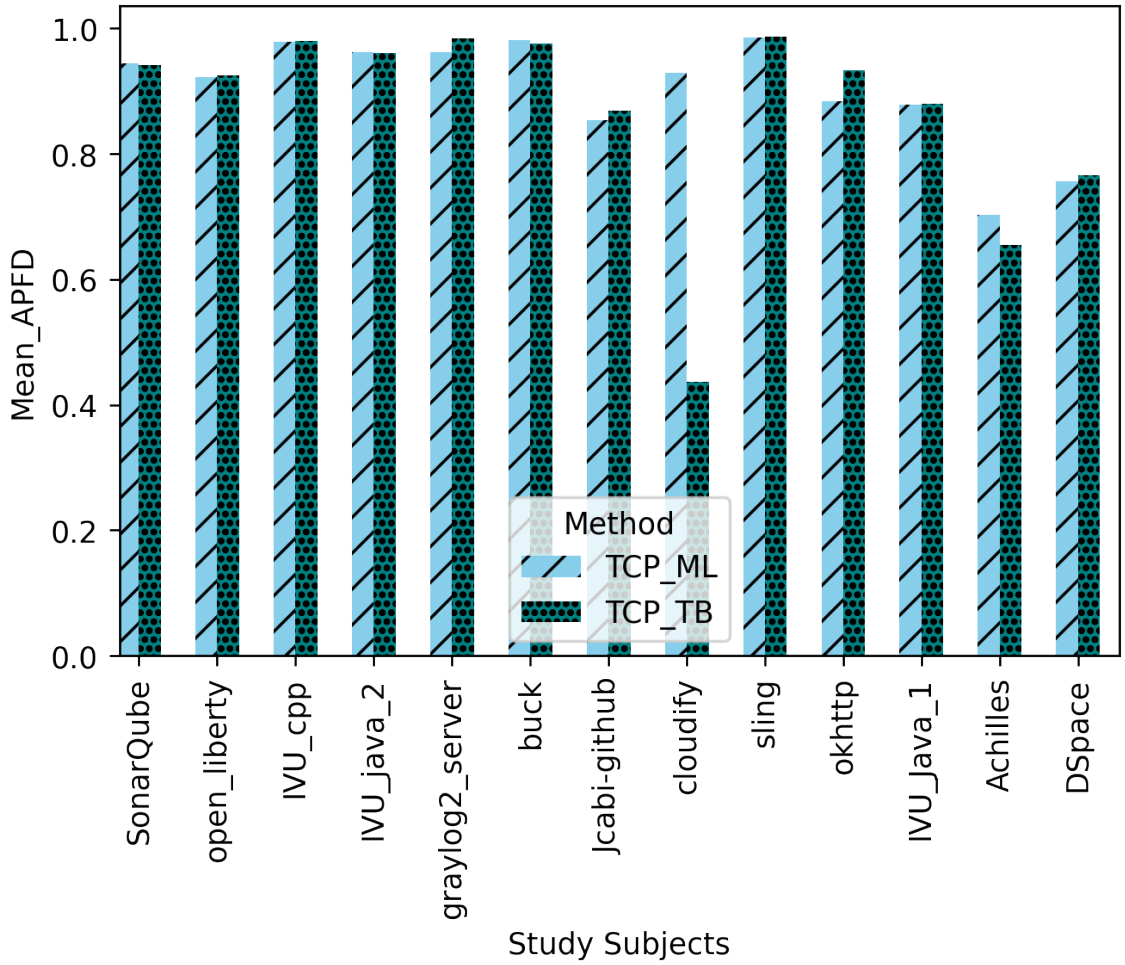

Figure 4.4: Mean APFD improvement of *TCP_TB* using internal domain knowledge tranfer

Thus, we cannot conclude that internal domain knowledge transfer failed for these two datasets. Based on the results shown in Figure 4.4, we can conclude that most large-scale datasets achieved better or similar performance with internal domain knowledge transfer compared to the baseline approach, *TCP_ML*. These findings suggest that when all old data is still relevant, ML and TL will have similar performance because the old data does not negatively affect the performance of the ML model This discussion answers our second research question *RQ2* and indicates that internal domain knowledge transfer can benefit

TCP for large-scale projects.

4.2.5 EFFECTIVENESS OF TCP_TB

In this experiment, we compare our approach *TCP_TB* with existing work *CI-RTP/S* [9] 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 [9] are listed in Table 4.2. Moreover, we compare *TCP_TB* with ten popular ML algorithms. Among the ML algorithms, XGBoost achieves the best performance. Table 4.8 shows a comparison among *TCP_TB*, *TCP_ML* and *CI-RTP/S* [9] 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 [9]. The *TCP_ML* column indicates the best ML results with the new feature set f_{new} discussed in Section 3.2.1.

TCP_TB improves the TCP performance of 19 projects compared to both *CI-RTP/S* and *TCP_ML*. Figure 4.6 and 4.5 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 is 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

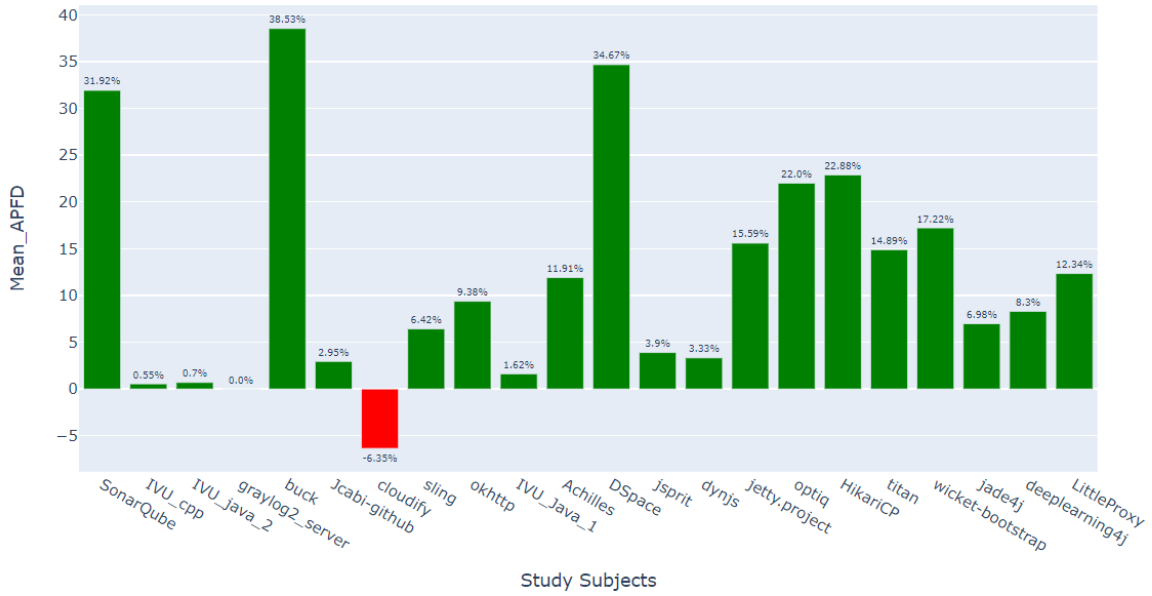

Figure 4.5: The percentage of mean APFD improvement of *TCP_TB* for each study subject compared to *CI-RTP/S*

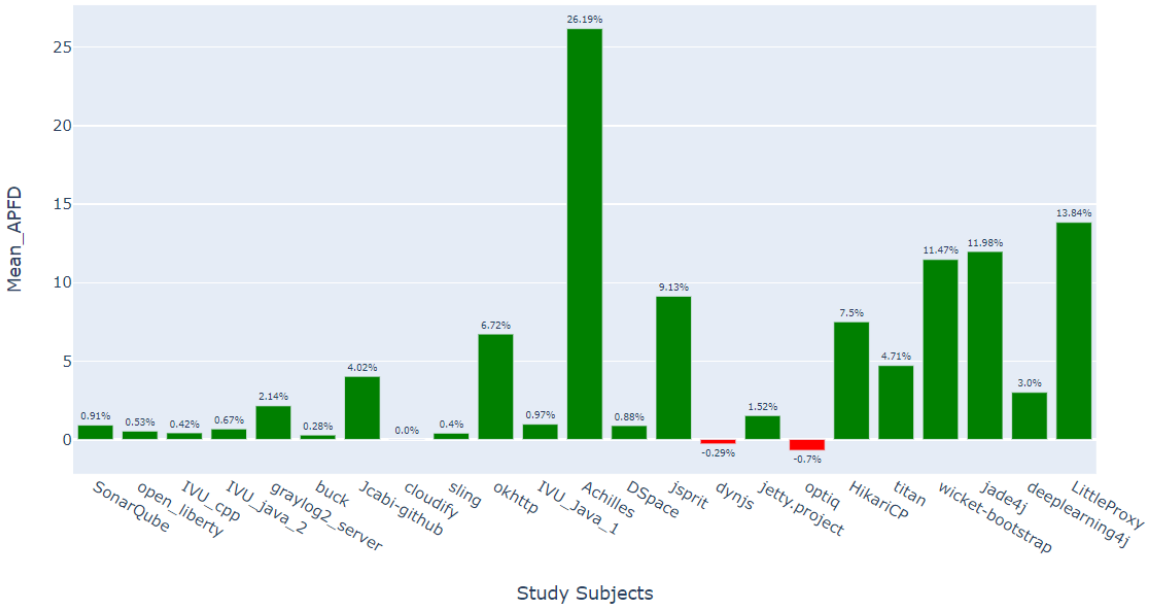

Figure 4.6: The percentage of mean APFD improvement of *TCP_TB* for each study subject compared to *TCP_ML*

observe that the performance of *TCP_TB* 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*. For the most large-scale dataset, the performance improvement 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 conducted a comprehensive analysis to evaluate the effectiveness of the *TCP_TB* approach for test case selection (TCS). The performance of *CI-RTP/S*, *TCP_ML*, and *TCP_TB* were compared for three different test budgets (10%, 50%, and 80%), and the best performances were highlighted in bold in Table 4.9,4.10, and4.11. Overall, *TCP_TB* outperformed the other two methods for all three test budgets. However, when the test budget was limited to 10%, *TCP_ML* performed well for *DSpace, dynjs,* and *optiq*, while *CI-RTP/S* performed well for *IVU_CPP* and *Cloudify*. For both 50% and 80%, *TCP_ML* performed better for two study subjects (i.e., *DSpace, dynjs,* and *optiq*), and *CI-RTP/S* performed better for one study subject (i.e., *Cloudify*). Our results demonstrate that *TCP_TB* is an effective approach for TCS across a range of test budgets and study subjects.

Figure 4.7 presents a box plot that shows the overall performance of the *TCP_TB* approach in terms of NAPFD value. For the small 10% test execution budget, both the *CI-RTP/S* and *TCP_ML* approaches perform very poorly, with median NAPFD values of around 0.58 and 0.78, respectively. In contrast, *TCP_TB* achieves a median NAPFD value of 0.88 across all study subjects. For the larger test budgets of 50% and 80%, the median

	Test Execution Rate 10%		
	CI-RTP/S	TCP_ML	TCP_TB
SonarQube	0.2837 ± 0.4472	0.8268 ± 0.3584	$\mathbf{0.8301 \pm 0.3668}$
open_liberty	-	0.7849 ± 0.3327	$\mathbf{0.7938 \pm 0.3279}$
IVU_Cpp	$\mathbf{0.9464 \pm 0.1668}$	0.9299 ± 0.1969	0.9431 ± 0.1785
IVU_Java2	0.9180 ± 0.2521	0.9080 ± 0.2627	$\mathbf{0.9274 \pm 0.2380}$
graylog2-server	0.9832 ± 0.0003	0.8573 ± 0.2412	$\mathbf{0.9832 \pm 0.0003}$
buck	0.0697 ± 0.2237	0.9568 ± 0.1696	$\mathbf{0.9690 \pm 0.1423}$
jcabi-github	0.6995 ± 0.4230	0.6858 ± 0.4276	$\mathbf{0.7507 \pm 0.3936}$
cloudify	$\mathbf{0.9921 \pm 0.0000}$	0.9286 ± 0.0000	0.9286 ± 0.0000
sling	0.9052 ± 0.2711	0.9666 ± 0.1484	$\mathbf{0.9791 \pm 0.1156}$
okhttp	0.6965 ± 0.4437	0.7319 ± 0.4204	$\mathbf{0.8503 \pm 0.3215}$
IVU_Java_1	0.7359 ± 0.3385	0.7601 ± 0.3241	$\mathbf{0.7807 \pm 0.3160}$
Achilles	0.0000 ± 0.0000	0.0000 ± 0.0000	$\mathbf{0.9643 \pm 0.0000}$
DSpace	0.0380 ± 0.0849	$\mathbf{0.4877 \pm 0.4695}$	0.2307 ± 0.3971
jsprit	0.2137 ± 0.3626	0.2452 ± 0.3804	$\mathbf{0.3376 \pm 0.4427}$
dynjs	0.9416 ± 0.0000	$\mathbf{0.9762 \pm 0.0000}$	0.9661 ± 0.0000
jetty.project	0.5981 ± 0.4654	0.9438 ± 0.1363	$\mathbf{0.9587 \pm 0.1218}$
optiq	0.0505 ± 0.1339	$\mathbf{0.7395 \pm 0.4006}$	0.7235 ± 0.4391
HikariCP	0.0320 ± 0.0519	0.2107 ± 0.3047	$\mathbf{0.2403 \pm 0.3478}$
titan	0.6261 ± 0.4696	0.6508 ± 0.4882	$\mathbf{0.8027 \pm 0.3437}$
wicket-bootstrap	0.4959 ± 0.7013	0.4959 ± 0.7013	$\mathbf{0.9754 \pm 0.0232}$
jade4j	0.1770 ± 0.3464	0.1154 ± 0.1477	$\mathbf{0.4512 \pm 0.4190}$
deeplearning4j	0.4369 ± 0.4509	0.7072 ± 0.3924	$\mathbf{0.7900 \pm 0.3300}$
Little Proxy	0.1942 ± 0.4342	0.1942 ± 0.4342	$\mathbf{0.3833 \pm 0.5251}$

Table 4.9: Mean and standard deviation NAPFD values for CI-RTP/S, TCP_ML, and TCP_TB considering test budget 10%

NAPFD values are positively skewed for all three approaches. Despite this, *TCP_TB* consistently achieves better performance than the other two methods across all study subjects. From the above discussion, we can conclude *TCP_TB* can perform better than ML models with the same feature set for TCP and TCS.

	Test Execution Rate 50%		
	CI-RTP/S	TCP_ML	TCP_TB
SonarQube	0.6013 ± 0.3757	0.9309 ± 0.1773	$\mathbf{0.9485 \pm 0.1204}$
open_liberty	-	0.9114 ± 0.1693	$\mathbf{0.9198 \pm 0.1460}$
IVU_Cpp	0.9724 ± 0.1010	0.9716 ± 0.0939	$\mathbf{0.9794 \pm 0.0761}$
IVU_Java2	0.9557 ± 0.1600	0.9497 ± 0.1714	$\mathbf{0.9597 \pm 0.1481}$
graylog2-server	0.9832 ± 0.0003	0.9618 ± 0.0329	$\mathbf{0.9832 \pm 0.0003}$
buck	0.4998 ± 0.3104	0.9700 ± 0.1363	$\mathbf{0.9793 \pm 0.1055}$
jcabi-github	0.8253 ± 0.2893	0.8148 ± 0.2892	$\mathbf{0.8681 \pm 0.2389}$
cloudify	$\mathbf{0.9921 \pm 0.0000}$	0.9286 ± 0.0000	0.9286 ± 0.0000
sling	0.9227 ± 0.2381	0.9843 ± 0.0496	$\mathbf{0.9864 \pm 0.0715}$
okhttp	0.8381 ± 0.3179	0.8701 ± 0.2655	$\mathbf{0.9491 \pm 0.0573}$
IVU_Java_1	0.8492 ± 0.2130	0.8504 ± 0.2260	$\mathbf{0.8644 \pm 0.2141}$
Achilles	0.8452 ± 0.0000	0.7024 ± 0.0000	$\mathbf{0.9643 \pm 0.0000}$
DSpace	0.3207 ± 0.3599	0.6762 ± 0.4076	$\mathbf{0.7321 \pm 0.2948}$
jsprit	0.6715 ± 0.3423	0.6066 ± 0.3349	$\mathbf{0.7041 \pm 0.2454}$
dynjs	0.9416 ± 0.0000	$\mathbf{0.9762 \pm 0.0000}$	0.9661 ± 0.0000
jetty.project	0.8185 ± 0.2911	0.9654 ± 0.1048	$\mathbf{0.9867 \pm 0.0267}$
optiq	0.6028 ± 0.2999	$\mathbf{0.8885 \pm 0.1684}$	0.8538 ± 0.2913
HikariCP	0.2069 ± 0.2336	0.4795 ± 0.3323	$\mathbf{0.5367 \pm 0.3325}$
titan	0.7096 ± 0.4070	0.8856 ± 0.1379	$\mathbf{0.9327 \pm 0.0776}$
wicket-bootstrap	0.8033 ± 0.2666	0.8607 ± 0.1855	$\mathbf{0.9754 \pm 0.0232}$
jade4j	0.6551 ± 0.2813	0.6114 ± 0.2827	$\mathbf{0.7408 \pm 0.2748}$
deeplearning4j	0.7877 ± 0.2483	0.8416 ± 0.2431	$\mathbf{0.8831 \pm 0.1882}$
Little Proxy	0.5342 ± 0.4902	0.6708 ± 0.1804	$\mathbf{0.7300 \pm 0.4112}$

Table 4.10: Mean and standard deviation NAPFD values for CI-RTP/S, TCP_ML, and TCP_TB considering test budget 50%

4.3 THREATS TO VALIDITY

In this section, we discuss the possible threats to the internal and external validity of our experimental results.

- Internal validity refers to the accuracy and reliability of the experimental results obtained in a specific study. In our experiment, we use the default execution environments and parameter settings of different machine learning techniques to evaluate the

	Test Execution Rate 80%		
	CI-RTP/S	TCP_ML	TCP_TB
SonarQube	0.6310 ± 0.3384	0.9431 ± 0.1220	$\mathbf{0.9522 \pm 0.1044}$
open_liberty	-	0.9211 ± 1331	$\mathbf{0.9265 \pm 0.1195}$
IVU_Cpp	0.9762 ± 0.0813	0.9746 ± 0.0778	$\mathbf{0.9813 \pm 0.0627}$
IVU_Java2	0.9586 ± 0.1482	0.9601 ± 0.1259	$\mathbf{0.9669 \pm 0.1143}$
graylog2-server	0.9832 ± 0.0003	0.9618 ± 0.0329	$\mathbf{0.9832 \pm 0.0003}$
buck	0.5918 ± 0.2059	0.9756 ± 0.1096	$\mathbf{0.9833 \pm 0.0810}$
jcabi-github	0.8620 ± 0.2093	0.8496 ± 0.2208	$\mathbf{0.8914 \pm 0.1791}$
cloudify	$\mathbf{0.9921 \pm 0.0000}$	0.9286 ± 0.0000	0.9286 ± 0.0000
sling	0.9241 ± 0.2341	0.9845 ± 0.0458	$\mathbf{0.9884 \pm 0.0478}$
okhttp	0.8510 ± 0.2874	0.8829 ± 0.2269	$\mathbf{0.9491 \pm 0.0573}$
IVU_Java_1	0.8689 ± 0.1717	0.8751 ± 0.1823	$\mathbf{0.8849 \pm 0.1757}$
Achilles	0.8452 ± 0.0000	0.7024 ± 0.0000	$\mathbf{0.9643 \pm 0.0000}$
DSpace	0.4125 ± 0.2995	0.7565 ± 0.2778	$\mathbf{0.7596 \pm 0.2350}$
jsprit	0.7217 ± 0.2548	0.6569 ± 0.2678	$\mathbf{0.7625 \pm 0.1707}$
dynjs	0.9416 ± 0.0000	$\mathbf{0.9762 \pm 0.0000}$	0.9661 ± 0.0000
jetty.project	0.8185 ± 0.2911	0.9715 ± 0.0765	$\mathbf{0.9867 \pm 0.0267}$
optiq	0.6591 ± 0.1821	$\mathbf{0.8885 \pm 0.1684}$	0.8815 ± 0.2061
HikariCP	0.4004 ± 0.1802	0.5632 ± 0.2600	$\mathbf{0.6406 \pm 0.2507}$
titan	0.7838 ± 0.2643	0.8856 ± 0.1379	$\mathbf{0.9327 \pm 0.0776}$
wicket-bootstrap	0.8033 ± 0.2666	0.8607 ± 0.1855	$\mathbf{0.9754 \pm 0.0232}$
jade4j	0.7015 ± 0.2253	0.6475 ± 0.2644	$\mathbf{0.7734 \pm 0.2591}$
deeplearning4j	0.8115 ± 0.1991	0.8645 ± 0.1845	$\mathbf{0.8945 \pm 0.1601}$
Little Proxy	0.6858 ± 0.2874	0.6708 ± 0.1804	$\mathbf{0.8092 \pm 0.2365}$

Table 4.11: Mean and standard deviation NAPFD values for CI-RTP/S, TCP_ML, and TCP_TB considering test budget 80%

study subjects. However, hyperparameter tuning and feature selection may improve the performance in different scenarios. Therefore, in our future work, we plan to tune the hyperparameters and perform feature selection to determine their impact on different scenarios. By doing so, we can ensure that our results are not affected by suboptimal hyperparameters or irrelevant features.

Figure 4.7: Mean NAPFD across study subjects

- External validity refers to the generalizability of the results to other contexts or populations. Our experimental results are based on the evaluation of 24 datasets written primarily in Java, and the selected features are extracted from CI and GitHub repositories. The results may vary if different feature sets are selected or if the datasets are written in different programming languages. However, we believe that our study can be reproduced using the same model configurations and datasets available in the repository [85]. Moreover, we have provided detailed descriptions of our methodology, making it possible for other researchers to reproduce our results and apply our approach to other datasets.

4.3.1 CONCLUSION

In this experiment, we evaluate a transfer learning-based test case prioritization technique called *TCP_TB*. This approach utilizes both test execution history and VCS changeset features to efficiently reorder test suites. We test our approach on 24 software projects and observe an enhancement in TCP performance for 19 of them. The APFD metric is used to evaluate the performance of *TCP_TB*. For large-scale projects with a test execution history greater than 100K, the APFD value is mostly increased by 2.82% when compared to the performance of *CI-RTP/S* and *TCP_ML*. Moreover, we find that *Open_liberty* is the most effective source dataset for creating transfer learning models as it improves the mean APFD value for 16 projects. Overall, *TCP_TB* demonstrates improved performance in TCP and TCS in 82.61% and 79.71% of cases, respectively. These results suggest that TL can indeed benefit test case prioritization and selection.

Chapter 5

CONCLUSION AND FUTURE WORK

In conclusion, this paper tackles the challenges of test failure prediction and test case prioritization (TCP) in continuous integration (CI) systems, which are critical to effective software development. While machine learning (ML) techniques have been widely applied to TCP, their performance can be limited when dealing with imbalanced or low-failure-rate data. Transfer learning (TL) algorithms have been proposed to address these challenges and enhance test failure prediction. This thesis presents a comparative analysis of various TL algorithms and proposes a new test case prioritization method, *TCP_TB*, which leverages *TransBoost*, a model-based transfer learning algorithm. The proposed method uses test execution history and VCS changeset features to predict test failure probability and reorder test cases accordingly.

The experimental results demonstrate that TL algorithms can improve test failure prediction rates for software projects, including large-scale industrial projects, and *TCP_TB* enhances TCP and TCS performance for most projects compared to the traditional ML and baseline model *CI-RTS/S*. Moreover, the proposed approach *TCP_TB* can facilitate knowledge sharing between similar projects without compromising data confidentiality. This thesis also explores internal domain knowledge transfer for large-scale projects using TL,

which mitigates the challenge of searching for potential source datasets, one of the primary issues in TL.

In summary, this paper provides valuable insights into the use of TL algorithms for improving TCP in CI systems, and proposes a new test case prioritization method that effectively utilizes transfer learning. The results suggest that TL can enhance test failure prediction and TCP performance, especially for large-scale industrial projects. This paper demonstrates the potential of TL in addressing imbalanced and low-failure-rate data issues and improving the efficiency of software development.

One potential avenue for future research is to explore the optimal feature set for TCP_TB. While our approach utilizes test execution history and VCS changeset features, there may be additional features that could improve the test failure prediction and prioritize test cases more effectively. Further investigation could help identify these features and lead to more accurate predictions.

Another research idea is to adapt real-time training methods. In our approach, we build our model in an offline environment and periodically update it. However, as new test results become available, it may be possible to incorporate this information into the model to improve its accuracy. By updating the model with new data in real-time, we can adapt to changes in the software development process and improve TCP performance over time.

However, in this study, we did not analyze the costs and benefits of the TL approach, as we updated our model offline periodically. Therefore, a future direction could be to investigate the costs and benefits of TCP with TL using real-time training methods. This could help to determine the trade-offs between improved performance and the computational costs of real-time training with TL algorithms.

Finally, exploring multi-domain knowledge transfer could also be an interesting avenue

for future research. Our approach focused on transferring knowledge from one software project to another, but it may also be possible to transfer knowledge across multiple domains. By utilizing knowledge from multiple domains, we may be able to improve the accuracy of our TCP approach and prioritize test cases more effectively. This could be particularly useful in cases where there is limited training data available for a specific project, as we could leverage knowledge from other domains to improve test failure prediction and prioritize test cases more accurately.

Bibliography

[1] M. Hilton, "Understanding and improving continuous integration," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1066–1067.

[2] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[3] J. Anderson, S. Salem, and H. Do, "Striving for failure: An industrial case study about test failure prediction," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 2, 2015, pp. 49–58.

[4] A. Memon, Z. Gao, B. Nguyen, *et al.*, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, IEEE, 2017, pp. 233–242.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[7] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, 2022.

[8] 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*, 2014, pp. 235–245.

[9] 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*, 2021, pp. 491–504.

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

[11] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across jvm boundaries," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 809–820.

[12] J. Mendoza, J. Mycroft, L. Milbury, N. Kahani, and J. Jaskolka, "On the effectiveness of data balancing techniques in the context of ml-based test case prioritization," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 72–81.

[13] 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)*, Springer, 2021, pp. 681–696.

[14] Y. Zhang, W. Dai, and S. Pan, *Transfer learning*, 2020.

[15] 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)*, IEEE, 2022, pp. 196–206.

[16] M. Brandtner, E. Giger, and H. Gall, "Supporting continuous integration by mashing-up software quality information," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, IEEE, 2014, pp. 184–193.

[17] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE access*, vol. 5, pp. 3909–3943, 2017.

[18] C. Travis and M. Hroncok, "Travis ci," *Source: https://travis-ci. org*, vol. 17, 2018.

[19] E. E. Romero, C. D. Camacho, C. E. Montenegro, *et al.*, "Integration of devops practices on a noise monitor system with circleci and terraform," *ACM Transactions on Management Information Systems (TMIS)*, vol. 13, no. 4, pp. 1–24, 2022.

[20] V. Armenise, "Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery," in *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, IEEE, 2015, pp. 24–27.

[21] N. N. Zolkifli, A. Ngah, and A. Deraman, "Version control system: A review," *Procedia Computer Science*, vol. 135, pp. 408–415, 2018.

[22] J. Mejia, M. Muñoz, Á. Rocha, and Y. Quiñonez, *New Perspectives in Software Engineering*. Springer, 2022.

[23] 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*, 2020, pp. 385–396.

[24] A. L. Samuel, "Machine learning," *The Technology Review*, vol. 62, no. 1, pp. 42–45, 1959.

[25] A. Ng, "Supervised learning, discriminative algorithms," *ML CS229 Lecture Notes*, pp. 1–30, 2017.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[27] O. Bin, "A prediction comparison of housing sales prices by parametric versus semi-parametric regressions," *Journal of Housing Economics*, vol. 13, no. 1, pp. 68–84, 2004.

[28] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[29] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[31] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[32] S. Pan and Q. Yang, *A survey on transfer learning. ieee transaction on knowledge discovery and data engineering, 22 (10)*, 2010.

[33] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, no. 1, pp. 1–40, 2016.

[34] Y. Yao and G. Doretto, "Boosting for transfer learning with multiple sources," in *2010 IEEE computer society conference on computer vision and pattern recognition*, IEEE, 2010, pp. 1855–1862.

[35] A. Asgarian, P. Sobhani, J. C. Zhang, *et al.*, "A hybrid instance-based transfer learning method," *arXiv preprint arXiv:1812.01063*, 2018.

[36] L. Alzubaidi, M. A. Fadhel, O. Al-Shamma, *et al.*, "Towards a better understanding of transfer learning for medical imaging: A case study," *Applied Sciences*, vol. 10, no. 13, p. 4523, 2020.

[37] R. Liu, Y. Shi, C. Ji, and M. Jia, "A survey of sentiment analysis based on transfer learning," *IEEE access*, vol. 7, pp. 85 401–85 412, 2019.

[38] M. Long, Y. Cao, J. Wang, and M. Jordan, "Learning transferable features with deep adaptation networks," in *International conference on machine learning*, PMLR, 2015, pp. 97–105.

[39] M.-T. Luong, Q. V. Le, I. Sutskever, O. Vinyals, and L. Kaiser, "Multi-task sequence to sequence learning," *arXiv preprint arXiv:1511.06114*, 2015.

[40] F. Li, S. J. Pan, O. Jin, Q. Yang, and X. Zhu, "Cross-domain co-extraction of sentiment and topic lexicons," in *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2012, pp. 410–419.

[41] Z. Yang, J. Zhao, B. Dhingra, *et al.*, "Glomo: Unsupervisedly learned relational graphs as transferable representations," *arXiv preprint arXiv:1806.05662*, 2018.

[42] Y. Wang and W. Li, "Transfer-based deep neural network for fault diagnosis of new energy vehicles," *Frontiers in Energy Research*, vol. 9, p. 796 528, 2021.

[43] Y. Sun, T. Lu, C. Wang, *et al.*, "Transboost: A boosting-tree kernel transfer learning algorithm for improving financial inclusion," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, 2022, pp. 12 181–12 190.

[44] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 119–129.

[45] 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)*, IEEE, 2018, pp. 69–79.

[46] 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*, 2017, pp. 12–22.

[47] 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.

[48] 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*, 2020, pp. 1–12.

[49] 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)*, IEEE, 2016, pp. 361–368.

[50] 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*, IEEE, 2006, pp. 123–133.

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

[52] 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*, 2016, pp. 975–980.

[53] J. Chen, Y. Lou, L. Zhang, *et al.*, "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*, 2018, pp. 656–667.

[54] 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)*, IEEE, 2019, pp. 96–105.

[55] 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. 106 269, 2020.

[56] 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.

[57] 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*, IEEE, 2008, pp. 278–287.

[58] 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*, 2017, pp. 2–11.

[59] 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*, 2018, pp. 80–89.

[60] A. Singh, R. K. Bhatia, and A. Singhrova, "Machine learning based test case prioritization in object oriented testing," *International Journal of Recent Technology and Engineering*, vol. 8, no. 3, pp. 700–707, 2019.

[61] 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. 215 716–215 726, 2020.

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

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

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

[65] 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)*, IEEE, 2020, pp. 663–672.

[66] 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)*, IEEE, 2011, pp. 382–391.

[67] 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*, 2020, pp. 1798–1806.

[68] 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.

[69] B. Fernando, A. Habrard, M. Sebban, and T. Tuytelaars, "Unsupervised visual domain adaptation using subspace alignment," in *Proceedings of the IEEE international conference on computer vision*, 2013, pp. 2960–2967.

[70] B. Sun, J. Feng, and K. Saenko, "Return of frustratingly easy domain adaptation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[71] M. Loog, "Nearest neighbor-based importance weighting," in *2012 IEEE International Workshop on Machine Learning for Signal Processing*, IEEE, 2012, pp. 1–6.

[72] Y. Q. Dai Wenyuan, X. Guirong, and Y. Yong, "Boosting for transfer learning," in *Proceedings of the 24th International Conference on Machine Learning, Corvallis, USA*, 2007, pp. 193–200.

[73] N. Segev, M. Harel, S. Mannor, K. Crammer, and R. El-Yaniv, "Learn on source, refine on target: A model transfer learning framework with random forests," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 9, pp. 1811–1824, 2016.

[74] L. Minvielle, M. Atiq, S. Peignier, and M. Mougeot, "Transfer learning on decision tree with class imbalance," in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE, 2019, pp. 1003–1010.

[75] Y. Sun, T. Lu, C. Wang, *et al.*, "Transboost: A boosting-tree kernel transfer learning algorithm for improving financial inclusion," *arXiv preprint arXiv:2112.02365*, 2021.

[76] 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*, 2000, pp. 102–112.

[77] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2019, pp. 91–100.

[78] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2015, pp. 58–68.

[79] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 13–25.

[80] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2019, pp. 101–110.

[81] 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)*, IEEE, 2019, pp. 408–418.

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

[83] 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*, Springer, 2019, pp. 919–931.

[84] 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*, IEEE, 2007, pp. 255–264.

[85] `https://figshare.com/s/851513ca9907a10dd17d`.