Automated Software Engineering (2022) 29:50
https://doi.org/10.1007/s10515-022-00344-y

Test case prioritization using test case diversification and fault-proneness estimations

Mostafa Mahdieh[1] · Seyed-Hassan Mirian-Hosseinabadi[1] · Mohsen Mahdieh[1]

Received: 16 October 2021 / Accepted: 8 June 2022 / Published online: 1 August 2022

Abstract
Regression testing activities greatly reduce the risk of faulty software release. However, the size of the test suites grows throughout the development process, resulting in time-consuming execution of the test suite and delayed feedback to the software development team. This has urged the need for approaches such as test case prioritization (TCP) and test-suite reduction to reach better results in case of limited resources. In this regard, proposing approaches that use auxiliary sources of data such as bug history can be interesting. We aim to propose an approach for TCP that takes into account test case coverage data, bug history, and test case diversification. To evaluate this approach we study its performance on real-world open-source projects. The bug history is used to estimate the fault-proneness of source code areas. The diversification of test cases is preserved by incorporating fault-proneness on a clustering-based approach scheme. The proposed methods are evaluated on datasets collected from the development history of five real-world projects including 357 versions in total. The experiments show that the proposed methods are superior to coverage-based TCP methods. The proposed approach shows that improvement of coverage-based and fault-proneness-based methods is possible by using a combination of diversification and fault-proneness incorporation.

Keywords Regression testing · Test case prioritization · Defect prediction · Test case diversification · Bug History

1 Introduction

Modern software systems are continuously changing at a rapid rate throughout the software development process. Changes are made to the software to add new features, improve functionality, and repair identified software bugs. During this

✉ Seyed-Hassan Mirian-Hosseinabadi
hmirian@sharif.edu

[1] Computer Engineering Department, Sharif University of Technology, Azadi Ave, Tehran, Iran

evolution, developers mustn't unintentionally inject new bugs into the software. Software *regression testing* attempts to reduce this risk by running a certain suite of test cases regularly or after each modification. Due to the increasing size of the software codebase and the number of change commits, regression testing has become a resource-intensive procedure for current software projects. Moreover, re-running the test suite can take much time, which results in a large feedback delay for developers. For example, at Google, code modification commits are done at the rate of more than 20 times per minute requiring more than 150 million test executions per day (Kumar 2010; Memon et al. 2017).

There has been a wide range of techniques proposed to improve the cost-effectiveness of regression testing. These techniques can be categorized into three groups: Test suite reduction, Test case selection, and Test case prioritization (TCP). *Test suite reduction* (also referred to as Test suite minimization) techniques speed up regression testing by reducing the size of the test suite (Zhong et al. 2008; Fraser and Wotawa 2007; Chen et al. 2017). These methods try to eliminate repetitive test cases, in hope of creating a smaller test suite with similar fault detection capability. *Test case selection* techniques intend to prevent unnecessary regression testing by choosing test cases that cover the modified code between versions (Grindal et al. 2006; Yoo and Harman 2007; Kazmi et al. 2017). *Test case prioritization (TCP)* techniques aim to reorder test cases such that early fault detection is maximized (Jones and Harrold 2003; Elbaum et al. 2002). This approach has the advantage over test suite reduction and test case selection in that it does not exclude any test cases from execution. TCP methods provide a way to execute test cases with more fault detection ability earlier to provide early feedback to developers. TCP also allows continuing testing to the limit of time or budget, by running the test suite in order obtained by prioritization.

TCP, which is the subject problem of this study, has been highly investigated and many approaches have been proposed for the TCP problem (Lou et al. 2019; Khatibsyarbini et al. 2018; Hemmati 2019). The majority of TCP methods have used structural code coverage as a metric to prioritize test cases (Hao et al. 2014; Yoo and Harman 2012). Some researchers have investigated using other sources of information, such as the project requirements (Hettiarachchi et al. 2016; Srikanth et al. 2016; Salehie et al. 2011), source code changes (Alves et al. 2016; Saha et al. 2015; Panda et al. 2016), or test execution history (Noor and Hemmati 2015; Khalilian et al. 2012; Rahman et al. 2018).

One valuable source of information for TCP is the bug history of the project. Bug history has been recently proposed as a source of information to improve TCP (Wang et al. 2017; Paterson et al. 2019; Eghbali et al. 2019; Mahdieh et al. 2020). Bug history can be utilized to estimate the *fault-proneness* of code units, which is the probability that developers have injected a defect in a code unit. In this line, defect prediction methods have been employed to estimate the fault-proneness of code units based on the source code and bug history of the project (Zimmermann et al. 2007; Ostrand et al. 2005; Menzies et al. 2006). However, there remains a challenge in the strategy of incorporating the fault-proneness estimations obtained by defect prediction to prioritize the test cases. For example, if fault-proneness is naively used to prioritize test cases, the test cases that cover a fault-prone area, although similar and possibly redundant will have a high priority.

It has been intuitively conjectured that test cases that have similar properties, also are probable to have similar fault detection capability (Leon and Podgurski 2003; Yoo et al. 2009). Therefore considering diversification in selecting and prioritizing test cases will lead to appropriate results. Based on this conjecture, various approaches have been proposed for TCP (Jiang et al. 2009; Fang et al. 2014; Ledru et al. 2012). These methods have been shown to empirically improve the fault detection rate of TCP, confirming the mentioned conjecture about similar test cases' fault detection capability in the context of TCP.

In this paper, we propose a novel approach to incorporate fault-proneness estimations for TCP utilizing both fault-proneness and test case diversification. Our approach is based on the idea of grouping similar test cases using clustering methods and prioritizing the clustered test cases. To estimate the fault-proneness of all code units from the bug history and source code, we designed a defect prediction method customized for the regression testing setting. Furthermore, we developed a TCP method based on test case clustering which takes into account both fault-proneness and test case diversification.

Another challenge regarding TCP is the empirical study. Many studies are based on manually injected faults or mutant versions of programs. To measure the fault detection rate of various TCP strategies in a more realistic situation, we evaluate the algorithms on real-world projects containing defects that occurred in the development process. Our experiment is conducted on 357 versions of five real-world projects, included in the Defects4J dataset (Just et al. 2014), comparing the fault detection rate of multiple methods.

To more accurately assess our study, we raise the following research questions:

- **RQ1:** The traditional total and additional TCP strategies have proven to be successful for coverage-based TCP (Rothermel et al. 2002; Hao et al. 2015, 2014). How do the proposed clustering-based TCP methods compare to the traditional coverage-based TCP strategies in terms of fault detection performance?
- **RQ2:** Does incorporating fault-proneness improve the proposed clustering-based TCP algorithm in terms of fault detection performance?
- **RQ3:** What is the influence of the clustering parameters (distance function and the number of clusters) on the effectiveness of the proposed TCP algorithms?

This paper makes the following contributions:

- We provide an approach to leverage existing coverage-based TCP methods in a clustering-based TCP scheme. This approach led to the development of new TCP methods which are based on test case coverage data and take advantage of the diversification of test cases for TCP.
- We propose a novel approach to combine fault history data and test case diversification, in the context of coverage-based TCP.
- We design a customized defect prediction method to estimate the fault-proneness of a code unit. This method is customized to work when only a small set of

recorded bugs are available and utilizes the information from all versions of the source code history.
- We present an empirical evaluation using five open-source projects containing in total 357 versions of the projects. Results show that our proposed approach could improve existing coverage-based TCP techniques.

The rest of the paper is organized as follows: Sect. 2 presents the background material. Section 3 presents our approach to solving the problem and our proposed method. Section 4 presents the setup of our empirical evaluation and Sect. 5 shows the results of our experiments. In Sect. 6 the empirical results and threats to the validity of this study are discussed. Section 7 summarizes the most related work to this paper. Finally, Sect. 8 contains the conclusions and future work of this paper.

2 Background

In this section, we present the formal definition of TCP and briefly introduce some of the classical coverage-based TCP methods. We continue by providing background information on defect prediction, which is employed for fault-proneness estimation in this study. Afterward, we present concepts of test case similarity and diversification-based methods for TCP.

2.1 Test case prioritization

In its essence, TCP seeks to find a permutation of test cases, which optimizes a certain intended goal function. To more formally define TCP, consider a test suite containing the set of test cases $T = \{t_1, t_2, \dots, t_n\}$. The TCP problem is defined as follows Elbaum et al. (2002):

Given: *T*, a test suite; *PT*, the set of permutations of T; *f*, a functionfrom *PT* to the real numbers

Problem: Find $T' \in PT$ such that[1]:

$$\forall T'' \, : \, PT \mid T'' \neq T' \bullet f(T') \geq f(T''). \tag{1}$$

In other words, the TCP problem is finding a permutation T' such that $f(T')$ is maximized. Here *f* is a scoring function that assigns a score value to any permutation selected from *PT*.

The *f* function represents the goal of a TCP activity. Software engineers using TCP methods could have different goals, such as testing business-critical functionality as soon as possible, maximizing code coverage, or detecting faults at a faster rate. Since the ultimate target of regression testing is to detect regression faults, the TCP target function is usually specified as to how fast the regression faults can be detected, which is referred to as *fault detection rate*. One of the highly used

[1] This relation is expressed using Z notation's first order logic (Woodcock and Davies 1996).

measurements for evaluating the fault detection rate is the APFD (average percentage of faults detected) goal function, an area-under-curve metric that measures how quickly a test suite can detect faults. APFD is frequently used in the literature for TCP when the goal of TCP is maximizing the fault detection rate (Yoo and Harman 2012; Engström et al. 2011; Catal and Mishra 2013). Another target function that can be used is the percentage of test cases executed until the first failing test case. We have chosen the first-fail metric for our empirical study and discussed this in Sect. 4.2.

2.2 Coverage-based test case prioritization

For the sake of modeling the system for TCP, the source code can be partitioned into units such as files, methods, or statements. Assuming a chosen level of partitioning, the source is partitioned into units $U = \{u_1, u_2, \dots, u_m\}$. Using this modeling, a broad range of coverage-based TCP methods settle on a level of partitioning (usually statements or methods) and measure coverage of test cases over those units. Considering each test case t_i of the test suite and unit u_j of the source code, $Cover(i, j)$ represents how much test case t_i covers unit u_j. The amount of coverage can be either 0 or 1 if the units of code are statements; however if the units are methods or files, it can also be a real number in the range [0, 1] representing the proportion of code that is covered by the test case execution.

Test case coverage can be collected in different ways. Dynamic coverage information is collected by executing the test case and tracking every unit of code that is executed. On the other hand, static coverage is derived by static analysis of the source code (Mei et al. 2012).

When the coverage of a test case on the code units is known, other concepts such as the total coverage of the test case can be computed. The total coverage of a test case t_i is formally defined as follows (Hao et al. 2014):

$$Cover(i) = \sum_{1 \le j \le m} Cover(i,j). \tag{2}$$

The value of $Cover(i)$ is a real non-negative number and can be larger than 1. Coverage-based TCP methods utilize coverage of test cases to prioritize the test suite. Traditional coverage-based TCP methods will be reviewed in the following subsection.

2.3 Review of traditional TCP methods

In this subsection, we review three traditional TCP strategies that are considered baseline methods in our empirical study.

2.3.1 Random strategy

The obvious and simple method of TCP is the random strategy. Taking the random strategy, all test cases of the test suite are shuffled in random order. The expected first-fail metric and APFD of this strategy are near 50%. This method is usually presented as the first baseline to be compared with other proposed strategies for evaluation (Ashraf et al. 2012; Elbaum et al. 2004).

2.3.2 Total strategy for TCP

The *total prioritization strategy* is based on the intuition that test cases that have more coverage are more likely to uncover bugs. The total strategy, therefore, starts with computing the total coverage of all test cases according to Eq. 2. In the next step, test cases are sorted according to their total coverage and as the result, the first test case in the prioritized order has the highest total coverage. The total prioritization strategy does not consider the fact that some test cases might cover duplicate areas of the code. Therefore, when test cases are prioritized using this strategy, frequently some units of code are executed multiple times before the whole units are covered (Elbaum et al. 2002).

Compared to other non-random existing strategies, the total prioritization strategy is simple and efficient. The time complexity of this algorithm is the sum of the time complexity of computing the total coverage for all test cases and the time complexity of the sorting algorithm. The addition of these values results in the time complexity of the total algorithm which is $\mathcal{O}(nm + n\log n)$, where n is the number of test cases and m is the number of source code units.

2.3.3 Additional strategy for TCP

In contrast to total prioritization, the *additional prioritization strategy* takes into account that executing an uncovered unit of the code is more likely to reveal new faults in the code, and therefore a test case that runs uncovered code must have more priority compared to a test case that runs already covered units. The idea behind the additional strategy is that earlier coverage of uncovered units of the code, results in revealing faults sooner (Elbaum et al. 2002).

The additional strategy begins by computing the total coverage of all test cases. Afterward, in each step, the test case with the highest coverage over the uncovered code area is chosen as the next test case. The selected test case is appended to the end of the list of prioritized test cases and marked not to be chosen in the next steps. The area of the code covered by the selected test case will be marked as a covered area.

With this type of selection, the additional strategy falls in the category of greedy algorithms. This strategy works in n steps where n is the number of test cases. In each step, selecting the next test case and updating the coverage of the remaining test cases is done in time complexity of $\mathcal{O}(nm)$. Therefore, the total time complexity of this algorithm is $\mathcal{O}(n^2m)$.

Due to different implementations in some scenarios, different variations of The additional strategy have been developed. In two situations, this strategy faces different options:

- When there exist at least two non-selected test cases which both have the highest coverage over the uncovered code area. In case of such a tie, one of these test cases should be selected with some criteria. For example, one might select the test case randomly.
- In case there are no uncovered areas of the code left. In this case, the remaining test cases can be ordered with different approaches. A common solution is to consider all the code uncovered again and continue the algorithm with the remaining test cases (Elbaum et al. 2002).

2.4 Defect prediction

Software faults are an inevitable part of the development process. These faults happen for various reasons such as the addition or modification of the software features, lack of tests and documentation, high level of dependence between units, and faulty designs.

Modern software development tools can track and record occurrences of each fault. As the cause behind most code faults is related to a limited set of known or unknown generic fault patterns, it is reasonable to generalize the pattern using the previously recorded samples.

There are usually four major steps to a defect prediction method (Nam 2014):

(1) *Feature extraction* in this step, each unit of code (package, file, class, or method) is analyzed and various metrics are extracted from the unit. The result of this step is a feature vector for each unit plus a label that indicates whether the unit contains bugs or not.

(2) *Data preprocessing* to maximize the quality of defect prediction algorithms, the extracted data should be manipulated in accordance with the machine learning algorithm. This step includes removing unnecessary features, normalization, and sampling.

(3) *Model learning* a machine learning algorithm is selected to predict faulty code based on previous versions. The extracted feature vectors are then fed to the machine learning algorithm. A small portion of the training samples is reserved for validation. The choice of the algorithm is made based on the quality of the predictions made by the model on the validation set. Prediction quality is then evaluated by the model's performance on the test set.

(4) *Prediction* the last step is to predict defects in unseen samples. In this step, each new unit of code is labeled with a fault-proneness score, indicating the plausibility of a defect in the unit.

There have been various features proposed for defect prediction. Static code metrics which mainly capture the complexity and structural aspects of the source code have been proposed, such as McCabe (1976), Halstead metrics (1977), CK features (design metrics from UML) (Chidamber and Kemerer 1994), and object-oriented features (coupling, cohesion, etc.) (Harrison et al. 1998; Bansiya and

Davis 2002; e Abreu and Carapuça 1994). Many studies have used static code metrics for defect prediction (Menzies et al. 2010, 2007; Zimmermann et al. 2007). Other metrics, such as historical and process-related metrics (e.g., number of past bugs Kläs et al. 2010; Ostrand et al. 2005 or the number of changes Pinzger et al. 2008; Meneely et al. 2008; Moser et al. 2008) and organizational metrics (e.g., number of developers Weyuker et al. 2008; Graves et al. 2000), have also been proposed.

Various machine learning techniques have been explored for the prediction step of defect prediction, which can be categorized as supervised learning, unsupervised learning, and semi-supervised learning (Li et al. 2018). Many supervised classification models have been applied for defect prediction, such as decision trees (Menzies et al. 2006), neural networks (Kanmani et al. 2007), support vector machines (Elish and Elish 2008), Naive Bayes (Shivaji et al. 2009), and Bayesian networks (Okutan and Yıldız 2014). Jing et al. (2014) employed cost-sensitive dictionary learning for defect prediction. More recently, ensemble learning methods have shown interesting performance and have gained attention in the area of software defect prediction (Petrić et al. 2016; Aljamaan and Alazba 2020; Matloob et al. 2021; Li et al. 2019). Unsupervised learning has been employed for defect prediction (Li et al. 2020) based on clustering methods (Nam and Kim 2015; Bishnu and Bhattacherjee 2011; Zhang et al. 2016) and other unsupervised approaches (Yang et al. 2016; Fu and Menzies 2017; Yan et al. 2017; Boucher and Badri 2018). Semi-supervised learning methods have been utilized for defect prediction using sparse learning (Wang et al. 2016b) and graph-based label propagation (Zhang et al. 2017). The prediction of bugs at change-level or commit-level, namely Just-In-Time (JIT) software defect prediction was introduced by Kamei et al. (2012).

In recent years deep learning techniques have also been utilized for defect prediction. Yang et al. (2015) utilize a deep belief network to extract a set of expressive metrics from an initial set of change metrics. Using the extracted features their method trains a classifier to predict defects at the change-level. Wang et al. (2016d) and Wang et al. (2018) leveraged deep belief networks to learn semantic features from abstract syntax trees and then used these features to create defect prediction models. Hoang et al. (2019) introduced *DeepJIT* for JIT defect prediction, which utilizes convolutional neural network (CNN) in an end-to-end deep learning framework by extracting features from both commit messages and code changes. Hoang et al. (2020) proposed *CC2Vec* as an improvement to DeepJIT using a hierarchical attention network (HAN) architecture. Pandey et al. (2020) present *BPDET* for defect prediction, by implementing a two-layer ensemble of different classifiers in front of an autoencoder-based deep representation. Popular deep learning methods that have been applied in defect prediction, include long short-term memory (LSTM) (Majd et al. 2020; Deng et al. 2020; Liang et al. 2019), stacked denoising autoencoder (Tong et al. 2018; Zhu et al. 2020), CNN (Li et al. 2017), and deep neural network (DNN; Xu et al. 2019). Despite promising results using deep learning methods, applying presents new challenges which are under investigation. Yedida and Menzies (2021), point out that many researchers applying deep learning in software engineering tasks have not compared the results with other non-deep learning techniques. They also provide experiments showing that class imbalance issues still should be cared for when using deep learning methods for defect prediction. In their study they show that using deep learning methods without applying appropriate preprocessing techniques such as oversampling might significantly decrease effectiveness of these methods.

2.5 TCP based on fault-proneness estimations

In case there is prior knowledge available on the presence of faults in certain areas of the code, this knowledge can be employed to improve TCP. One of the categories of research in this line is based on estimating fault-proneness using defect prediction methods (Wang et al. 2017; Paterson et al. 2019; Eghbali et al. 2019; Mahdieh et al. 2020). In Mahdieh et al. (2020), fault-proneness based coverage is presented, which is defined as:

$$Cover^{FP}(i) = \sum_{1 \le j \le m} Cover(i,j) \times Prob(F_j), \tag{3}$$

where $Cover^{FP}(i)$ denotes the fault-proneness based coverage of test case t_i of the test suite and the estimated probability[2] of existing faults in unit u_j $(1 \le j \le m)$ is shown by $Prob(F_j)$.

This concept of coverage can be incorporated in coverage-based TCP methods such as the ones presented in Sect. 2.3.

2.6 Diversity based TCP

As mentioned in Sect. 1, it is believed that test cases with similar features have similar fault detection capabilities (Leon and Podgurski 2003; Yoo et al. 2009). The general idea behind the diversity-based TCP approach is to rank the test cases in an order that at each point of execution of the test cases, the diversity of the executed test cases at that point is maximized. To do so, these methods attempt to implement the following three steps:

- Encode test cases as a vector of features,
- Computing distance/similarity of test cases according to a distance metric,
- Maximize/minimize the distances/similarities of test cases.

There have been different distance functions proposed for diversity-based TCP such as Euclidean distance, Hamming distance, Jaccard Index, and Edit distance (Hemmati 2019).

After choosing an appropriate distance function, an algorithm must be determined to order test cases such that the diversity of the test cases along the prioritized test cases sequence is maximized. The problem of prioritizing test cases

[2] F_j indicates the event in which jth code unit is faulty and $Prob(F_J)$ represents the probability of this event.

to achieve such maximum diversity is an NP-hard problem (traditional set cover) (Mathur 2010). Therefore a heuristic method must be used to attempt to find a permutation of the test cases which sub-optimally maximizes the diversity function. Various heuristic methods have been proposed for this maximization problem. These methods can be categorized as Greedy, Adaptive Random, Clustering, and Search-based algorithms (Hemmati 2019).

Among these categories, clustering-based methods have been employed by several researchers for TCP (Kandil et al. 2017, Pei et al. 2021; Fu et al. 2017; Shrivathsan et al. 2019; Chen et al. 2018). Clustering methods partition data points into groups or clusters, according to the similarity function between the data points, such that data points in the same group have high similarity. For the TCP application, normally a data point is extracted from each test case, which represents the properties of the test case. Various clustering algorithms can be applied in this scenario, such as hierarchical clustering, centroid-based clustering, clustering based on fuzzy theory, distribution-based clustering, density-based clustering, and clustering based on graph theory (Xu and Tian 2015). After clustering the data points various strategies can be imagined to prioritize the test cases. We review some of the previously proposed strategies in Sect. 7. This paper also leverages clustering and proposes a strategy in this manner.

3 Methodology

Our proposed method consists of four main steps. The block diagram of the proposed method is depicted in Fig. 1. In the first phase, defect prediction is utilized to predict the fault-proneness of code units. In the second phase, the test cases are clustered to similar test cases in groups using a hierarchical clustering algorithm. In the third phase, the test cases in each cluster are internally prioritized using coverage-based TCP methods. In the fourth phase test cases of the clusters are aggregated combining the fault-proneness estimations and traditional TCP methods. To evaluate the prioritization algorithm, the actual test results are used to find the rank of the fault-revealing test cases and to compute the fault detection rate of each of the algorithms.

Two proposed algorithms are derived from this approach:

- The first step can be skipped resulting in the proposed TCP method not utilizing fault-proneness and defect prediction concepts. Algorithm 1 shows the pseudo-code of the TCP method without using fault-proneness. We refer to this algorithm as `CovClustering` in the empirical study (Sect. 4).
- Algorithm 2 specifies the pseudo-code of the method with the incorporation of fault-proneness derived by defect prediction. We refer to this algorithm as `CovClustering+FP` in the empirical study results.

In the following subsections, we explain each of the main steps of the proposed approach in more detail. Explanations of the pseudo-codes follow in

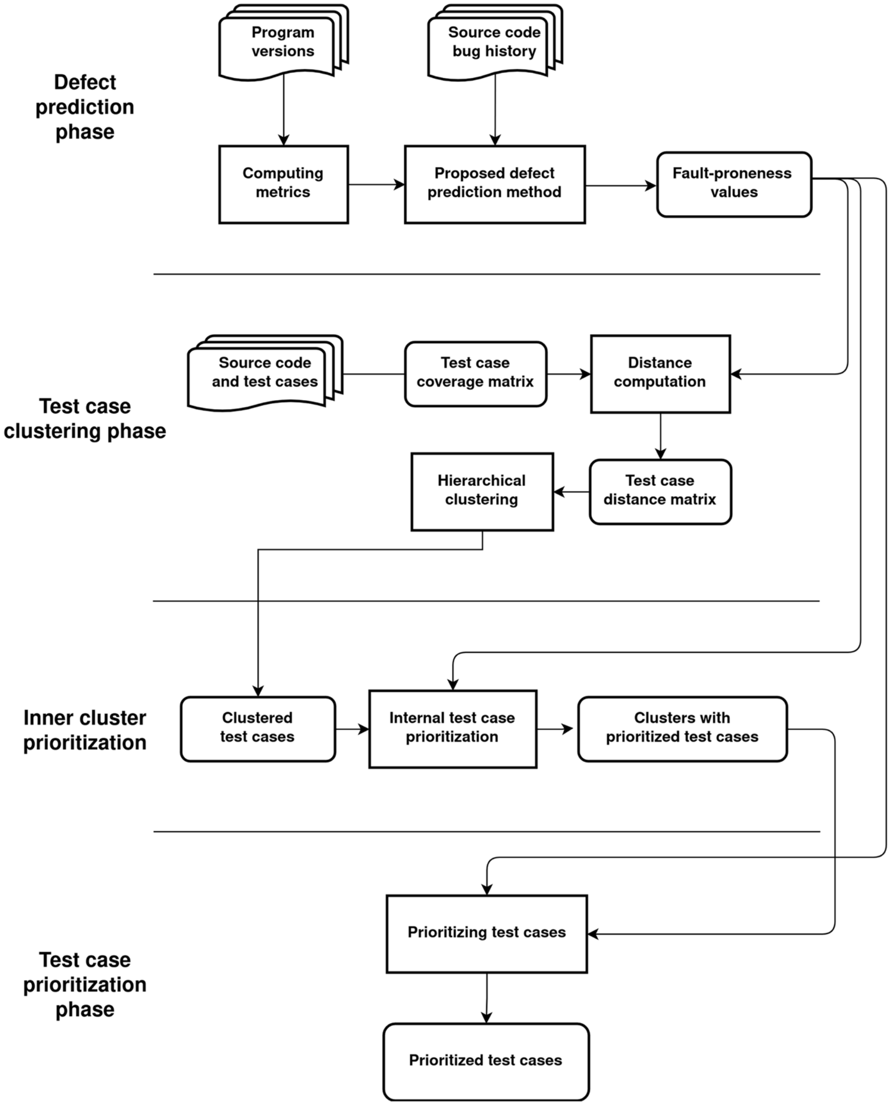

Defect prediction phase
Program versions
Source code bug history
Computing metrics
Proposed defect prediction method
Fault-proneness values
Test case clustering phase
Source code and test cases
Test case coverage matrix
Distance computation
Hierarchical clustering
Test case distance matrix
Inner cluster prioritization
Clustered test cases
Internal test case prioritization
Clusters with prioritized test cases
Test case prioritization phase
Prioritizing test cases
Prioritized test cases

Fig. 1 Overview of the proposed test case prioritization algorithm

Sect. 3.3, where we also give more details on the third and fourth steps of the algorithms.

3.1 Proposed defect prediction method

In this section, the problem of predicting defective codes and the proposed method to obtain an appropriate estimate of the fault-proneness are presented. As mentioned in Sect. 2.5, the goal is to predict $Prob(F_i)$ for each unit u_i of code given a set of features extracted from this unit. One effective method to estimate $Prob(F_i)$ is modeling it through a binary classification model.

In this section, we describe the concepts of the defect prediction model, and the details of implementing this model are described in Sect. 4.5.

3.1.1 Feature set

The project data used for this study, including code features and code coverage data are described in detail in Sect. 4.2. The features are extracted at the class level and consist of 104 input features including static and process features shown in Table 2. Each recorded version of the project in the dataset contains exactly a single bug that can be detected by a few test cases of the test suite. The classes which are buggy will result in data instances marked with the buggy label and the other instances are marked as not-buggy instances.

One helpful step to achieving better performance is to combine features to get new meaningful features. For each version, if there are no changes to a specific unit and other units related to it, it would be highly unlikely that it turns into a buggy unit and vice-versa. Using this idea, a set of features is defined by calculating the difference between two consecutive version metrics of each unit. This set of features can also be seen as the first derivative in discrete time by viewing the unit versions as the time dimension. This additional set of features (known as the churn of source code metrics) is shown to be effective for defect prediction (D'Ambros et al. 2012), therefore we add them to the feature set. Keeping the original features and the churn of source code features together helps to capture both the static and dynamic aspects of the units.

We reviewed defect-prediction methods based on deep learning in Sect. 2.4, however, as mentioned applying these techniques still has difficulties and can be challenging. Our goal is to focus on designing a TCP method, which should successfully work with any good enough defect-prediction method. On the other hand, the volume of source code contained in the subject of empirical study also presents limitations for employing deep learning methods. Therefore we consent to consider more simple defect-prediction models in this paper if their performance is acceptable. Note that using more simple methods has the benefits of less implementation complexity and lower execution resources needed which can be interesting.

3.1.2 Classification model

As mentioned in Sect. 2.4, classification is one of the main components of the defect prediction procedure. Various models have been utilized to be used to predict bugs (Nam 2014). Lessmann et al. (2008) compare many classification models for defect prediction in a standard benchmark and the results show that the classification model has a negligible effect on the performance of defect prediction. Among 22 classification models compared in this study, Random Forest has the best performance which confirms the result of previous studies (Guo et al. 2004). Ghotra et al. (2015) revisit these studies by applying noise-cleaning to the NASA dataset and adding the PROMISE dataset, and observe that tree-based ensemble methods have the best performance. Aljamaan and Alazba (2020) study tree-based ensemble models for defect-prediction and confirm that these methods have promising performance and specifically observe that random forest and XGBoost have notable results. These models are a good fit for our scenario of defect prediction as they have the following properties:

- Our dataset consists of numerous features both originating directly from the codes or derived in the data preparation phase. Although removing irrelevant features using feature selection methods might help with the tree building process, selecting the most effective features manually is a difficult task. An important advantage of tree-based methods is that these models inherently select the most effective features based on the training labels (target values) implicitly.
- Using tree-based models could be helpful since the result only considers the samples which are in the leaf node. Therefore it almost resistant to most of the preprocessing issues which must be considered. However, it's still useful to remove the noisy samples and generate more samples of the class with a smaller size.
- These models are invariant to most scaling methods and require little to no normalization.
- Ensemble models are less prone to over-fitting.

These properties lead to the effectiveness of tree-based ensemble defect prediction models on generally any software project and decrease the dependency of the results on a specific dataset (Hastie et al. 2008). In Sect. 4.5 we present the results of empirically comparing multiple tree-based defect-prediction models on the subject dataset, which leads to the conclusion that XGBoost has the best performance on the subject dataset of study. Therefore XGBoost was chosen as the classification model in the proposed defect prediction method.

An important step is to tune the key parameters of the XGBoost model. The parameter tuning process is usually defined as an optimization problem and the goal is to optimize a certain scoring function. We selected this approach and used it iteratively and the parameters were chosen using validation data. The details of this approach are mentioned in Sect. 4.5.

3.1.3 Data preprocessing

Machine learning models can benefit from data standardization before feeding the data to the model. To standardize the data, it's important to understand the requirements of the selected ML model. In tree-based models (Random Forest, XGBoost, etc.), the final tree structure is invariant to scaling input vectors linearly (Hastie et al. 2008). This is since each split is applied at a given interval. Also, the predicted target value is independent of the input vector because it is calculated using training phase target values.

The dataset consists of over 100 features and after adding the different features, the number of features jumps to 200. Having this many features makes the classification difficult. To address this problem several measures are set to place.

The first strategy is to use the regularization parameters in the classification model. Specifically, L1 regularization is a helpful way to tackle over-fitting when dealing with sparse datasets (Hastie et al. 2008). The XGBoost model has implemented three regularization parameters:

(1) Alpha is the L1 regularization coefficient.
(2) Lambda is the L2 regularization coefficient.
(3) Gamma is the minimum loss reduction of a leaf node to be partitioned.

All three of these parameters are carefully examined and taken into effect.

The second strategy is manually limiting the max depth of the trees. This is a straightforward approach since it limits the choice of dimensions used in the trees and avoids the curse of dimensionality problem.

The third strategy is to sample the number of columns used for each tree in the boosting model. This way, when adding a new tree, the number of features making an effect on it is limited. Hence, it reduces the chance of an over-fitted model.

In real-world projects, it is very common that the project contains a few recorded bugs for each version (Khoshgoftaar et al. 2010). Due to this fact, for each version, there are mostly non-faulty samples. In machine learning terms, this phenomenon is known as imbalanced classification and standard machine learning algorithms struggle in this case and must be implemented with care (Song et al. 2018). Hence, some steps in the data preparation should be manipulated to tackle these problems.

There are several well-known approaches to deal with the imbalanced dataset issue. These include oversampling, undersampling, and class weights. Oversampling attempts to generate samples nearby the existing samples with the same label to increase the samples of the smaller class (Chawla et al. 2002). On the other hand, undersampling methods remove the samples that are considered noisy (i.e. two samples with different labels that are very close in the input vector space) and samples that are insignificant to the results (e.g. duplicates). Furthermore, the class weight approach is a really useful technique in tree-based methods since it emphasizes more on the class with the least samples. We come back to this issue and mention our approach to handling imbalanced data in Sect. 4.5.

3.2 Proposed clustering method

To arrange similar test cases in groups, we use a clustering method. As explained in Sect. 2.6, a standard clustering method receives a set of points, each with a feature vector as input, and returns multiple subsets of points (i.e. the clusters) as output. In our proposed method, the points are the test cases. To create a feature vector for each test case, we use the vector of source code coverage values of that test case. This vector has a size equal to the number of methods in the source code and is created by appending the traced coverage value of each method of the source code after execution of the test case, which is a real number in the range of [0, 1]. The value of 0 represents no coverage of the test case on the method and the value of 1 represents coverage of all statements of the method by executing the test case.

The agglomerative hierarchical clustering method is applied to cluster the test cases. In the agglomerative clustering method, a bottom-up approach is followed. The clustering starts by considering each of the points as a cluster and follows by merging pairs of nearest clusters until the number of clusters reaches the desired number of clusters. Some studies have already reported the successful application of this clustering method in the application of TCP (Carlson et al. 2011; Fu et al. 2017). One of the reasons for using a hierarchical clustering method, in this case, is that the origin of the clusters in the data and the number of clusters are not fully known. Hierarchical clustering methods do not consider any assumption on the number of clusters in the data and have more resistance to such situations.

Clustering methods use a distance function to group similar points (i.e. points with small distance) and any of the normal distance functions can be used for this purpose. Another configuration of the agglomerative clustering methods is the type of distance measurement between two clusters of points. The *average* linkage metric is used in this manner. This metric is computed using the average pairwise distance between points from each cluster.

We also utilize the results of the defect prediction phase to arrange better clusters. As we are partitioning the test cases into groups, it is reasonable to arrange the partitions in a manner that all partitions have a comparable probability of fault-revealing test cases. Therefore we modify the coverage matrix such that test cases that cover areas of code with a high fault-proneness are put into different clusters. More exactly, the coverage of code units is multiplied by their corresponding $Cover^{FP}$ value (presented in Sect. 2.5) for all coverage values and then the distance is computed based on these modified coverage values. As result, test cases covering code areas with high fault-proneness would be grouped with test cases covering the same areas and there would be resistance for them to be merged with other clusters which cover high fault-proneness areas.

Algorithm 1 Proposed test case prioritization algorithm (`CovClustering`)

```
Inputs:
    Cover: the coverage matrix of the test suite
    n: size of the test suite
    clusterNum: the number of clusters for clustering
Outputs:
    Prioritized: the prioritized list of the test suite
$D \leftarrow$ distances($Cover$)
$testClusters \leftarrow$ agglomerativeClustering($D$, $clusterNum$)
for each $c (1 \leq c \leq clusterNum)$ do
    $testClusters[c] \leftarrow$ additionalPrioritization($testClusters[c]$, $Coverage$)
end for
$round \leftarrow 0$
while $|Prioritized| < n$ do
    $tests \leftarrow \emptyset$
    for each $c (1 \leq c \leq clusterNum)$ do
        if $|testClusters[c]| > round$ then
            $tests \leftarrow (\ tests \cup testClusters[c][round])$
        end if
    end for
    $tests' \leftarrow$ totalPrioritization($tests$, $Coverage$)
    $Prioritized \leftarrow Prioritized \parallel tests'$
    $round \leftarrow round + 1$
end while
```

Algorithm 2 Proposed test case prioritization algorithm incorporating fault-proneness (`CovClustering+FP`)

```
Inputs:
    Metrics: computed metrics
    Model: the learned defect prediction model
    Cover: the coverage matrix of the test suite
    n: size of the test suite
    clusterNum: the number of clusters for clustering
Outputs:
    Prioritized: the prioritized order of the test suite
$Prob_{FP} \leftarrow \text{defectPrediction}(Metrics, Model)$    ▷ fault-proneness probability
$D \leftarrow \text{distances}(Cover \times Prob_{FP})$
$testClusters \leftarrow \text{agglomerativeClustering}(D, clusterNum)$
for each $c (1 \leq c \leq clusterNum)$ do
    $testClusters[c] \leftarrow \text{maxPrioritization}(testClusters[c], Coverage, Prob_{FP})$
end for
$round \leftarrow 0$
while $|Prioritized| < n$ do
    $tests \leftarrow \emptyset$
    for each $c (1 \leq c \leq clusterNum)$ do
        if $|testClusters[c]| \geq round$ then
            $tests \leftarrow tests \cup testClusters[c][round]$
        end if
    end for
    $tests' \leftarrow \text{maxPrioritization}(tests, Coverage, Prob_{FP})$
    $Prioritized \leftarrow Prioritized \,||\, tests'$
    $round \leftarrow round + 1$
end while
```

3.3 Proposed test case prioritization method

As mentioned, the proposed TCP method consists of four steps. In the first step (defect prediction phase) which has been described in Sects. 3.1 and 4.5, the fault-proneness of code units is estimated and this estimation is used to prioritize the test cases in the next steps. In the second step (clustering phase) which was explained in Sect. 3.2, test cases are grouped into multiple clusters.

In this section, we will describe the third and fourth steps of the algorithm in detail. In the third step, the test cases of each cluster are prioritized internally, concerning each other. In the fourth step, we use an iterative approach. In each iteration, a test case is selected (according to the internally prioritized order) from each of the clusters. After that, the selected test cases are prioritized using a prioritization strategy.

Algorithm 1 shows the pseudo-code of the proposed TCP method. In lines 1–2, the distances between test cases are computed based on their coverage value and agglomerative clustering is executed to cluster the test cases. Lines 3–5, show the third step which is prioritizing the test cases internally in each cluster. For this purpose, we use the additional prioritization strategy which was mentioned in Sect. 2.3. The additional strategy has been shown to have significant performance among coverage-based TCP strategies (Hao et al. 2015).

Finally lines 6–16, describe the fourth step at which in each iteration of the while loop, a test case is selected from each cluster and added to the *tests* set. After that, the selected test case set is prioritized using the total prioritization strategy. Using the additional prioritization is not necessary for this step, as clustering has already limited duplicate code coverage between clusters.

Algorithm 2 uses the same procedure as Algorithm 1, also adding incorporation of fault-proneness into the method. In line 1 of Algorithm 2, the defect prediction method described in Sect. 3.1 is executed to extract a fault-proneness value for each unit of the code. Line 2 computes the distances between test cases after element-wise multiplication of the coverage matrix into the fault-proneness vector. Lines 3–18 are the same as Algorithm 2, with the difference of using maxPrioritization for prioritizing test sets in lines 5 and 15. We define maxPrioritization as sorting test cases in descending order by the maximum of fault-proneness of units covered by each test case.

4 Empirical study

In this section, we explain our empirical study and discuss the results of our experiments.

4.1 Research questions

In our empirical study, we aim to answer several research questions, presented in the introduction of this paper. These research questions are stated as follows:

- **RQ1:** How does the proposed TCP method (without the usage of the defect prediction), compare to the traditional coverage-based TCP strategies in terms of fault detection performance?
- **RQ2:** Does incorporating fault-proneness improve the proposed clustering-based TCP algorithm in terms of fault detection performance?
- **RQ3:** What is the influence of the distance function and number of clusters on the effectiveness of the proposed TCP algorithms?

Table 1 Projects included in Defects4J initial version

Identifier	Project name	Bugs	Test classes
Chart	JFreechart	26	355
Closure	Closure compiler	133	221
Lang	Apache commons-lang	65	112
Math	Apache commons-math	106	384
Time	Joda-Time	27	122
Sum	**–**	**357**	**1194**

The last row which represents the sum/overall values are marked as bold

4.2 Subjects of study

To evaluate TCP algorithms, the algorithms must be executed on projects with a large test suite. The test suite must reveal at least one bug for the prioritization to be meaningful. Furthermore, the source of the projects and the bug locations must be identifiable so that white box TCP methods can be applied. To apply defect prediction, the bug history of the project throughout development must also be recorded.

The Defects4J collection presented by Just et al. (2014), fulfills the mentioned properties. In its initial published version, Defects4J provided a version history of five well-known open-source Java projects, which contain a considerable number of test cases, alongside a recorded bug history, summarized in Table 1. As these projects represent completely real-world project development, we can hope that the results can be more practically significant.

The Defects4J data set has been collected in a specific standard. For each recorded bug, Defects4J provides a faulty version of the project which contains the bug. In the faulty version, one or more failing test cases identify the bug. This helps us to locate buggy classes in each version of the source code. There is exactly one bug in each version of all projects of the Defects4J dataset, therefore after any failing test case is reached the bug is detected, and executing other failing test cases will not have significant value. Therefore we chose the first failing or first-fail metric to measure the fault detection rate, similar to other studies which have used this metric for evaluating TCP on the Defects4J dataset (Paterson et al. 2019; Palma et al. 2018; Noor and Hemmati 2015; Abou Assi et al. 2021). Furthermore, due to the small number of failing test cases, which is a single test case in some versions, the value of the first failing metric and the value of the APFD metric are near to equivalent in many versions of the dataset.

In order to obtain the coverage of the test cases and also code metrics extracted from the Defects4J source code, we have used the already created and publicly available Defects4J+M dataset[3] (Mahdieh et al. 2020). Defects4J+M is an extension of the Defects4J dataset, containing the measured test coverages and source code metrics for each version of all projects included in Defects4J.

In this dataset, dynamic coverage is used to measure coverage of test case execution on the source code. Dynamic coverage is generally more accurate than static

[3] https://github.com/khesoem/Defects4J-Plus-M.

coverage and can lead to more effective prioritization results. The coverage values in this dataset, represent the amount of coverage of each test case on each unit of the code. The coverage values are measured at the method level with a real value indicating the amount of coverage on each method.

The source code metrics of Defects4J+M are composed of a combination of static and process metrics. These metrics were computed at the class level. Table 2 which is quoted from the article which introduces the Defects4J+M dataset (Mahdieh et al. 2020) contains the details of each feature group contained in Defects4J+M. To use the computed metrics for defect prediction, we stored them in a vector that is used as the input feature vector by the defect prediction algorithm.

4.3 Subject TCP algorithms

To empirically compare our proposed methods with related methods, we selected and implemented notable TCP methods. An important point to consider for selecting these TCP methods is that the information sources used by the methods must be the same as the proposed method. For example, if a TCP method uses both test coverage and software requirements as the information source for prioritization, it is not reasonable to compare this method with methods that only use test coverage for prioritization.

The TCP algorithms used for comparison in our empirical study our summarized in Table 3. These algorithms can be divided into two categories: First, are TCP algorithms that use only coverage as their information source, and second are TCP algorithms using coverage, bug history, and source code metrics as their information source. The algorithms of the first category are the following:

(1) The traditional total and additional prioritization methods (described in Sect. 2.3).
(2) The adaptive random TCP algorithm proposed by Jiang et al. (2009).
(3) The proposed based TCP method of this paper without the usage of the phase of defect prediction, in either of the clustering or prioritization phases (we refer to this method as `CovClustering`).

The algorithms of the second category, which use coverage, bug history, and source code metrics, are as follows:

(1) The total and additional prioritization TCP methods based on fault-proneness coverage proposed in Mahdieh et al. (2020) and presented in Sect. 2.5.
(2) The G-clef proposed by Paterson et al. (2019) in two variants: using either the greedy or the additional strategy as the secondary objective function.
(3) The proposed based TCP method of this paper which was presented in Sect. 3 that utilizes defect prediction (we refer to this method as `CovClustering+FP`).

Note that another TCP algorithm that leverages fault-proneness is the QTEP method proposed by Wang et al. (2017). This algorithm is based on influencing

Table 2 Defect prediction features Mahdieh et al. (2020)

#	Feature type	Category	Definition	Count	General items
1	Input	Source code metrics	Used to quantify different source code characteristics	52	Cohesion Metrics, Complexity Metrics, Coupling Metrics, Documentation Metrics, Inheritance Metrics, Size Metrics
2	Input	Clone metrics	Used to identify the number of type-2 clones (same syntax with different variable names)	8	Clone Classes, Clone Complexity, Clone Coverage, Clone Instances, Clone Line Coverage, Clone Logical Line Coverage, Lines of Duplicated Code, Logical Lines of Duplicated Code
3	Input	Coding rule violations	Used for counting coding violation rules	42	Basic Rules, Brace Rules, Clone Implementation Rules, Controversial Rules, Design Rules, Finalizer Rules, Import Statement Rules, J2EE Rules, JUnit Rules, Jakarta Commons Logging Rules, Java Logging Rules, JavaBean Rules, Naming Rules, Optimization Rules, Security Code Guideline Rules, Strict Exception Rules, String and StringBuffer Rules, Type Resolution Rules, Unnecessary and Unused Code Rules
4	Input	Git metrics	Used to count the number of committers and commits per file (these metrics could not be computed for inner classes)	2	Committers Count, Commit Counts
5	Output	Bug label	Label that shows this file is buggy in this version of the project or not	1	IsBuggy

Table 3 Studied TCP algorithms

Algorithm	Identifier	Information sources	Description
Total strategy	`Total`	Coverage	Total prioritization strategy described in Sect. 2.3
Additional strategy	`Additional`	Coverage	Additional prioritization strategy described in Sect. 2.3
Adaptive random TCP	`ART`	Coverage	Adaptive random test case prioritization proposed in Jiang et al. (2009)
Proposed clustering based TCP method	`CovClustering`	Coverage	The proposed based TCP method without the usage of the phase of defect prediction, in either of the clustering or prioritization phases
Total strategy with fault-proneness based coverage	`Total+FP`	Coverage, Bug history and source code metrics	Total prioritization strategy using fault-proneness based test case prioritization proposed in Mahdieh et al. (2020)
Additional strategy with fault-proneness based coverage	`Additional+FP`	Coverage, Bug history and source code metrics	Additional prioritization strategy using fault-proneness based test case prioritization proposed in Mahdieh et al. (2020)
G-clef with greedy prioritization	`G-clef (Greedy)`	Coverage, Bug history and source code metrics	G-clef prioritization method proposed in Paterson et al. (2019)
G-clef with additional prioritization	`G-clef (Additional)`	Coverage, Bug history and source code metrics	G-clef prioritization method proposed in Paterson et al. (2019)
Proposed TCP method with incorporating fault-proneness	`CovClustering+FP`	Coverage, Bug history and source code metrics	The proposed based TCP method which was presented in Sect. 3

fault-proneness on coverage. As the formulation of QTEP is very similar to the method presented in Mahdieh et al. (2020), we only put the latter in the set of algorithms for comparison.

4.4 Experimental procedure

The main part of the experiment consists of running the algorithms mentioned in Table 3 on the projects of the Defects4J+M dataset. To create the defect prediction model for the *i*th version of the projects, the procedure explained in Sect. 3.1 is performed by aggregating the data of the other projects and the data from the 1st to $(i-1)$th versions of the same project. Since it is reasonable to create the defect prediction model using a minimum number of bugs from the same project, we created the model only for the versions of each project from some version onward. In this regard, the evaluation is done over all versions of the projects, except the oldest 5 versions of each project which are used for defect-prediction hyperparameter tuning. Additionally, other projects were added to the training set of each version to get an advantage in early versions. This addition happens to enhance the model performance even in later versions.

The source code of the methods implemented in this paper and usage instructions are put publicly available on a GitHub Repository.[4] This package contains instructions on the usage of the algorithms and replicating the results of this paper in multiple steps.

The defect prediction model is implemented using Python language and *XGBoost* machine learning libraries. The clustering algorithms are implemented using the Python *scikit-learn* library and the TCP algorithms are also implemented with Python language using *NumPy* and *pandas* libraries. The distance metric used for the agglomerative hierarchical clustering method is the Euclidean distance metric, which is frequently applied when using this clustering method. The number of clusters chosen for our experiments was chosen by observing the Davies–Bouldin index (DBI), which is explained in Sect. 5.3.2.

4.5 Defect prediction implementation

There are several key steps regarding the choice of the proposed classification model and its parameters. These include choosing the best classification algorithm, hyperparameter tuning, and data preparation techniques. We begin with the discussion of the classification algorithm and then the procedure upon which the hyperparameters are chosen is explained. Lastly, a few ideas that were tested regarding the imbalanced nature of data is introduced.

[4] https://github.com/mostafamahdieh/ClusteringFaultPronenessTCP.

Table 4 Comparison of different classification algorithms

Classifier	Run type	MCC	F1-score	Precision	Recall	AUC
Random Forest	Offline	0.71	0.69	0.75	0.71	0.992
Random Forest	Online	0.68	0.67	0.67	0.71	0.994
CatBoost	Offline	0.87	0.87	0.88	0.86	0.992
CatBoost	Online	0.88	0.87	**0.90**	0.85	**0.994**
XGBoost	Offline	0.87	0.87	0.88	0.87	0.992
XGBoost	Online	**0.89**	**0.89**	0.89	**0.90**	0.992

The algorithm with the best value according to each column is marked as bold

4.5.1 Comparison of tree-based models

In Sect. 2.4, tree-based ensemble methods were introduced and in Sect. 3.1 the classification model was further looked into. It is clear that the performance of models varies on different datasets, therefore to choose between tree-based ensemble methods, we selected three major models and compared their performance on our dataset. The selected models are Random Forest, CatBoost, and XGBoost where in this section a comparison between these tree-based models is presented.

The training and evaluation process is repeated separately on every version of each project. For each version of a project, the training set consists of data instances of older versions of that project and data instances of other projects. The idea is to maintain the model's generalization over all projects in the earlier stages and it further improves on average as new versions are added to the training set. We will refer to this type of classification as *online*. The other type of execution of the classification algorithm, denoted by *offline*, is to only use other projects in the training set and does not require iterating over versions. The offline execution type is only used to measure the impact of adding the previous versions into the training set.

We principally evaluate the defect prediction method performance by Matthews correlation coefficient (or MCC in short). MCC is the binary version of the Pearson correlation coefficient that measures the similarity between the predicted labels and the true labels. This evaluation metric works well in the imbalanced dataset cases and has been suggested for usage in defect prediction applications (Yao and Shepperd 2020, 2021), and also used in other fields (Boughorbel et al. 2017; Chicco 2017).

A comparison between the classification algorithms is shown in Table 4. Interestingly, the offline runs have a good enough result without having seen any of the project instances and solely depending of data instances of other projects. This indicates that overfitting has not occurred in our classification models, because the training and evaluation instances are very different in this case.

It is observed that the online models have improvements over the offline models in most cases, which is reasonable due to feeding more training data for the online models. The best model among the six candidate models in terms of the MCC score, is the online XGBoost model, therefore we select this model for our further experiments.

Table 5 Performance of online XGBoost on all projects

Project	Versions	Evaluation versions	MCC	F1-score	Precision	Recall
Chart	26	21	0.88	0.88	0.78	1
Closure	133	128	0.89	0.89	0.90	0.89
Lang	65	60	0.92	0.92	1	0.85
Math	106	101	0.84	0.84	0.86	0.83
Time	27	22	0.91	0.91	0.91	0.91
Overall	**357**	**332**	**0.89**	**0.89**	**0.89**	**0.90**

The last row which represents the sum/overall values are marked as bold

Table 6 Data instances used in the hyperparameter tuning step of each project

Project	Versions	Dataset
Current Project	First 5 versions	Validation
Current Project	Versions higher than 5	Test (untouched in this step)
Other Projects	Last 5 versions	Validation
Other Projects	First version up to the last 5 versions	Training

Table 5 shows a detailed overview of the properties of the learning process of online XGBoost model on all projects. Column *Evaluation versions* show the number of versions that evaluation is done on the project. F1-score is also measured for each project and the results are nearly identical to the MCC score.

The defect prediction method performance can also be measured using the *precision* and *recall* metric. In the context of our setting, recall is the number of bugs identified using the defect prediction model. Also, precision can be evaluated as the proportion of classes that have been correctly labeled among all classes that have been labeled as buggy by the prediction model. A bug is considered to be predicted if the corresponding class to its bug-fix has a fault-proneness higher than the project's computed threshold. This threshold is selected using the validation data to maximize the MCC score.

4.5.2 Classification model hyperparameter tuning

To tune the hyperparameters a randomized search is done according to a distribution for the subjected parameters (Zainab et al. 2020; Sandha et al. 2020). In the XGBoost classifier, the key parameters are Tree count, Max tree depth, L1 and L2 regularization coefficients, and Gamma which is the minimum change needed in the loss function to partition a leaf node. The distributions used in the randomized

Table 7 Selected parameters for each project

Project	Tree count	Tree max depth	Column sampling rate	Alpha	Lambda	Gamma
Lang	400	2	0.83	0	0	0
Math	400	5	0.86	0	5	1
Chart	300	4	0.93	2	0	1
Closure	300	4	1	0.1	5	1
Time	400	2	0.83	0	0	0
Average	**360**	**3.4**	**0.89**	**0.42**	**2**	**0.6**

The last row which represents the sum/overall values are marked as bold

search are mostly uniform. To increase the flexibility of the model, the hyperparameters are tuned separately for each project.

The next step is to make a validation set for the randomized search to select the best hyperparameters. Table 6 summarizes the data instances used in the hyperparameter tuning step of each project as the training, validation, and test sets. The best model is selected in terms of the MCC score (Yao and Shepperd 2021, 2020) on the validation data. In the case of a tie, the model with the highest tree count to max depth ratio is selected.

The usage of the three shrinkage strategies to minimize the risk of the curse of dimensionality discussed in Sect. 3.1.3 can be verified by the parameters chosen in the hyperparameter tuning. The key parameters are Tree count, Tree max depth, Column sampling rate, Alpha, Lambda, and Gamma. Tree count is the number of trees used in the ensemble. Tree max depth is the limit within which each tree in the ensemble can grow. Hence, it is the number of features used in each tree to make the final decision. The column sampling rate is the rate of subset columns used to build each tree in the ensemble. Alpha, Lambda, and Gamma are L1/L2 regularization parameters and the minimum loss reduction of a leaf node respectively. Table 7 shows the values of the aforementioned parameters. The maximum depth for each tree alone has drastically limited the number of features that influence the decision, but in some cases, other parameters also have non-zero values which further limits the curse of dimensionality problem.

4.5.3 Other considerations

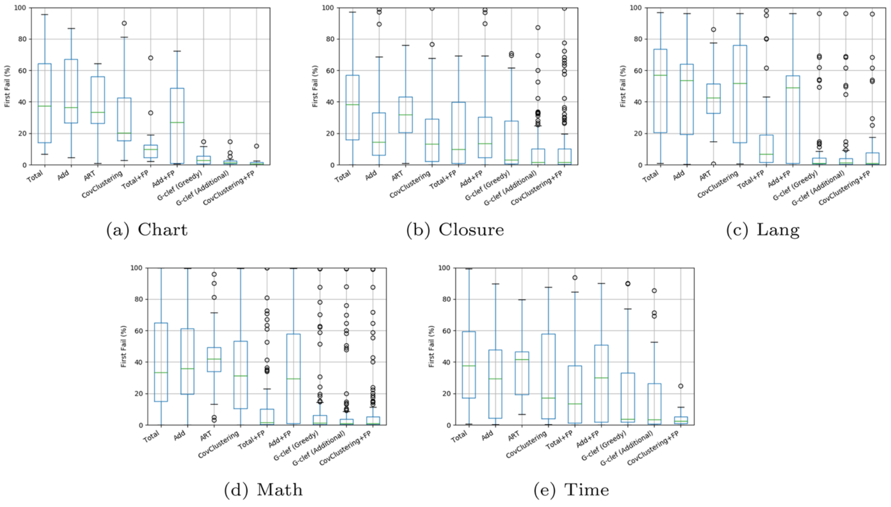
To overcome the imbalanced nature of the dataset, several ideas were tested. Most of the negative samples are unchanged units throughout the versions. These are more likely to be duplicate samples in terms of the final feature instances. The first idea is to remove these duplicates to reduce the negative to positive samples ratio. To further decrease this ratio, a random sub-sampling of the negative instances also takes place. These two combined have improved the final results on average. The idea of generative positive samples using SMOTE was also tested but did not further improve the average result. Therefore, only negative sampling methods were applied to balance the dataset.

Table 8 Average first-fail scores of each TCP strategy

Algorithm	Chart	Closure	Lang	Math	Time	Overall
`Total`	41.14	36.04	50.28	40.32	41.72	41.90
`Additional`	42.69	21.37	45.94	40.80	33.04	36.77
`ART`	38.75	31.92	41.93	42.37	36.16	38.23
`CovClustering`	32.07	19.12	47.44	34.87	31.04	32.91
`Total+FP`	13.01	20.20	15.11	11.32	25.97	17.12
`Additional+FP`	30.79	19.52	37.12	34.24	30.22	30.38
`G-clef Original (Greedy)`	47.9	47.6	33.1	31.8	24.7	37.0
`G-clef Original (Additional)`	41.2	27.1	49.9	36.6	24.3	35.8
`G-clef (Greedy)`	3.81	14.15	9.63	10.00	22.47	12.01
`G-clef (Additional)`	2.40	**9.19**	9.24	9.49	18.97	9.86
`CovClustering+FP`	**1.31**	10.50	**8.79**	**8.98**	**4.31**	**6.78**

In each column, the best algorithm inside the upper part and lower parts of the table are marked with an underline

The algorithm with the best value over both parts is marked as bold



(a) Chart (b) Closure (c) Lang

(d) Math (e) Time

Fig. 2 Evaluation results of all TCP strategies in the subject study

5 Results

In this section, we present and analyze the results of our empirical study. In this regard, we answer the research questions raised in Sect. 4.1 by providing and discussing the corresponding experimental results.

5.1 RQ1: comparing traditional TCP strategies with the CovClustering method (the proposed TCP method without incorporating fault-proneness)

In this research question, we compare the first-fail performance of coverage-based TCP methods with the proposed `CovClustering` TCP method (the proposed clustering-based TCP algorithm without incorporating fault-proneness). The compared methods have been described in Sect. 4.3. For the purpose of comparison, we have implemented and executed these methods on the subjects of study, presented in Sect. 4.2. The average first-fail value of these methods is shown in Table 8. Also, Fig. 2 depicts the boxplot of the first-fail of these methods on the subject versions of each project. Note that lower values of first-fail mean that the TCP algorithm has detected the fault sooner, therefore algorithms with lower values of first-fail are performing better.

The `Additional` algorithm has better performance than the `Total` algorithm, which confirms previous reports (Hao et al. 2015). The `ART` algorithm has interesting performance and performs better than the `Additional` algorithm on the Chart and Lang projects. However, overall their performance is near and the `Additional` algorithm slightly performs better than the `ART` algorithm.

Our research question is related to the rows `Total`, `Additional`, `ART`, and `CovClustering` in Table 8. The first-fail metric of all algorithms on the Lang project are more than 40% which shows that the algorithms do not perform much better than random prioritization (which should have around 50% first-fail). This shows that TCP algorithms based solely on coverage, probably will not have appropriate performance on the Lang project.

As can be seen, the average first-fail metric of the `CovClustering` method is less than the other methods on all projects except the Lang project, where the `ART` method has the best performance. This comparison is also observed in the boxplots of Fig. 2.

The proposed `CovClustering` method has better performance on four of the five projects and is also superior on the overall value on all projects. We performed Wilcoxon signed-rank tests (1992) (p-value < 0.05) to make sure that the overall superiority is statistically significant. The null hypothesis is that there is no significant difference in the first-fail performance of the `CovClustering` TCP method with respect to each of the coverage-based TCP methods. The results of this test demonstrate that there is a statistically significant difference between the `CovClustering` TCP method with respect of other coverage-based methods: `Total` (p-value $= 4.91 \times 10^{-7}$), `Additional` (p-value $= 0.008$), and the `ART` (p-value $= 3.52 \times 10^{-4}$).

Therefore our answer to RQ1 is that the proposed clustering method is superior to the coverage-based TCP methods.

5.2 RQ2: studying the effect of incorporating fault-proneness on the proposed method

We want to know the effect of fault-proneness on the proposed method in this research question. Therefore we compare the `CovClustering` TCP method, with the proposed `CovClustering+FP` method, in terms of the first-fail metric. The

Table 9 The details of the statistical tests of comparison of all TCP algorithms to the CovClustering+FP method

Algorithm	Chart	Closure	Lang	Math	Time	Overall
Total+FP	0.000*	0.000*	0.000*	0.044	0.001	0.000*
Additional+FP	0.001	0.000*	0.000*	0.000*	0.001	0.000*
G-clef (Greedy)	0.001	0.013	0.405	0.012	0.068	0.028
G-clef (Additional)	0.007	0.540	0.587	0.044	0.040	0.062

(0.000*Values less than 0.001 which are typically very small)

average value of the first-fail metric on the proposed method incorporating fault-proneness is shown in row CovClustering+FP of Table 8.

In addition we also compare the CovClustering+FP method with other state-of-the-art fault-proneness based methods which are the TCP using fault-based coverage (Mahdieh et al. 2020) and the G-clef algorithm (Paterson et al. 2019). Rows Total+FP and Additional+FP of Table 8 correspond to the fault-based coverage TCP methods and rows G-clef (Greedy) and G-clef (Additional) correspond to the G-clef algorithm presented by Paterson et al. (2019). To implement the fault-proneness-based algorithms we used the fault-proneness resulting from the defect prediction method proposed in this paper, to have comparable results.

We have also provided the original results of the G-clef algorithm noted in their paper (Paterson et al. 2019) as two rows of Table 8, as they have used the same initial dataset used by our study (the Defects4J dataset). It is observable in Table 8 that the performance of the G-clef algorithm using the fault-proneness results of this paper is significantly better than the G-clef results in the original paper. This observation confirms that the defect prediction method proposed in this paper has remarkable performance.

It is observed that the algorithms based on fault-proneness (the lower part of Table 8) have better performance than the purely coverage-based algorithms (the upper part of Table 8). The performance of Total+FP is also interesting and it is superior to Additional+FP, unlike their purely coverage-based counterparts. This shows that the highly competitive Additional algorithm will not improve enough when naively applying fault-proneness to the coverage formulation.

Comparing the results presented in Table 8, it is observed that the CovClustering+FP method has the best value of average first-fail compared to other TCP algorithms, in all but one case. The only exception to this observation is the comparison of the G-clef (Additional) algorithm on the Closure project.

We performed the Wilcoxon signed-rank again to evaluate the significance of the results. The null hypothesis is that there is no significant difference in the first-fail performance of the CovClustering+FP method with respect to the other TCP algorithms. The results of this test are shown in Table 9. The null hypothesis is rejected in cases where the values are less than 0.05, and in these cases, there is a significant difference between the proposed CovClustering+FP method and other algorithms. The very low *p*-values indicate that significance is confident. For

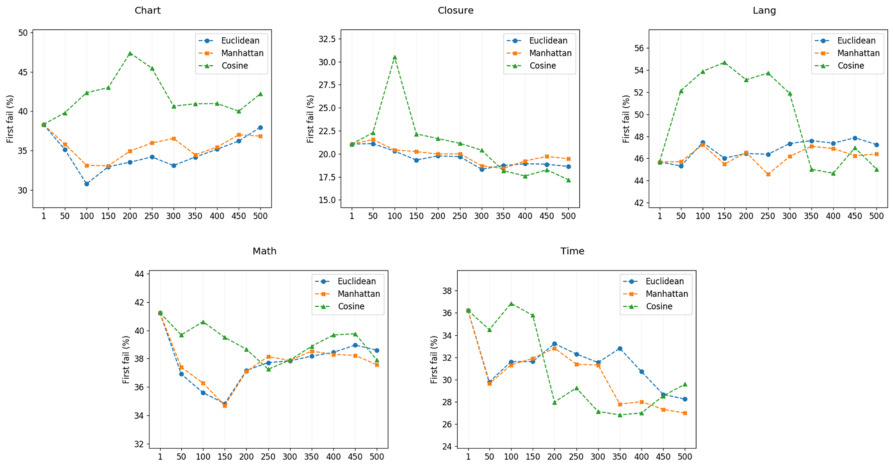


Fig. 3 The performance of the proposed `CovClustering` TCP strategy on the subject study using different distance functions and cluster numbers (RQ3, Distance function)

most of the table, this significance is verified however the statistical test fails for some of the values relating to the G-clef methods, specifically on the Closure and Lang project.

The conclusion is that the `CovClustering+FP` performs better than the `Total+FP` and `Additional+FP` algorithms of Mahdieh et al. (2020) but does not significantly dominate the `G-clef (Additional)` algorithm in all cases. Note that as mentioned and observed in Table 8, `CovClustering+FP` performs much better than the original `G-clef` implementation and the presented results are due to boosting `G-clef` with the defect prediction method presented here.

5.3 RQ3: the effect of the clustering configurations (distance function and number of clusters) on the effectiveness of the proposed TCP strategies

In this research question, we study the effect of the distance function and number of clusters on the effectiveness of the `CovClustering` and `CovClustering+FP` methods. In this manner, we experiment the proposed methods, with different distance functions and vary the number of clusters in a specified range.

5.3.1 Distance function

We experiment using three well-known distance functions which have been also used in previous related research (Pan et al. 2022): Euclidean distance, Manhattan distance, and distance based on Cosine similarity.

Figures 3 and 4 show the performance of executing the `CovClustering` and `CovClustering+FP` method through different distance functions and cluster numbers. For `CovClustering` the Euclidean and Manhattan distance have similar performance and seem to have better performance than cosine similarity on most

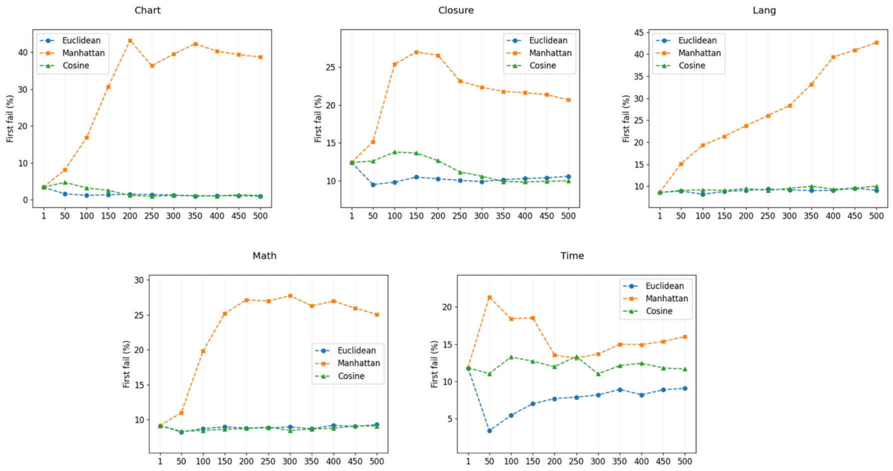


Fig. 4 The performance of the proposed `CovClustering+FP` TCP strategy on the subject study using different distance functions and cluster numbers (RQ3, Distance function)

Table 10 Best numbers of clusters chosen heuristically using the DBI metric

Project	`CovClustering` best cluster number by DBI	`CovClustering+FP` best cluster number by DBI
Chart	100	175
Closure	150	150
Lang	175	150
Math	125	150
Time	150	75

points. However, for the `CovClustering+FP` method, Euclidean distance shows competitive performance compared to other distance functions on all subject projects. The superiority of Euclidean distance for diversity-based TCP algorithms has also been observed by other researchers (Wang et al. 2016a). Therefore we can hope that generally, Euclidean distance can be more appropriate for this application as a first choice, but experimenting with other distance functions can also be considered. This also shows that the accuracy of defect prediction can highly impact the performance of fault-proneness methods.

5.3.2 Number of clusters

To choose the appropriate number of clusters, a well-practiced technique is to employ metrics that evaluate the quality clustering to get a better insight. Among these metrics, we utilize the *Davies–Bouldin index (DBI)* (1979). DBI is defined as the average ratio of within-cluster distances of each cluster to the between-cluster distances to the nearest cluster. Thus, more compact clusters will result in a better

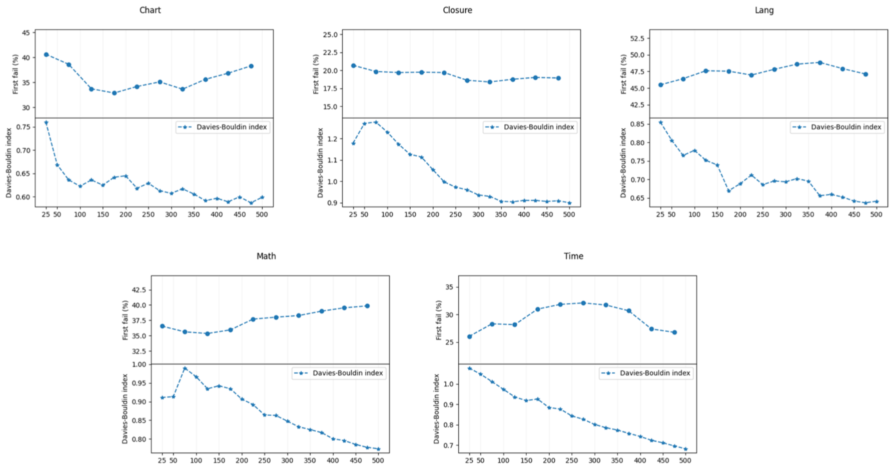


Fig. 5 The DBI value of the clustering of the `CovClustering` method on the subject projects using different cluster numbers (RQ3, Number of clusters)

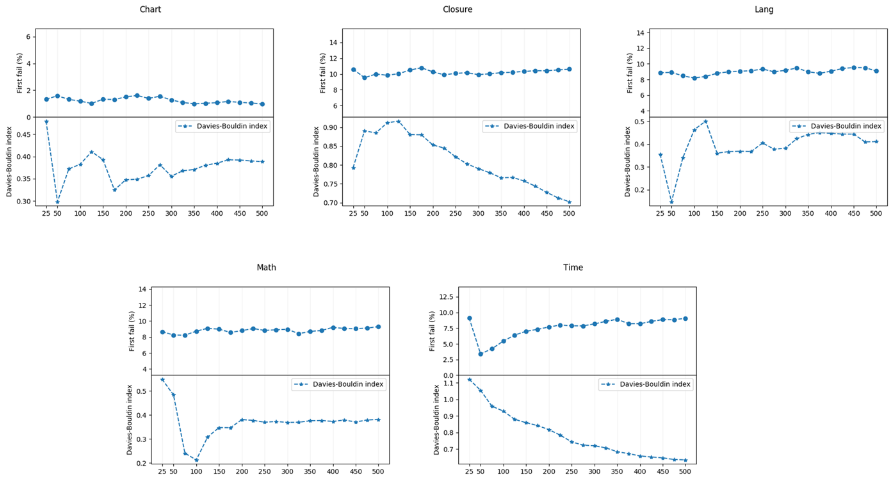


Fig. 6 The DBI value of the clustering of the `CovClustering+FP` method on the subject projects using different cluster numbers (RQ3, Number of clusters)

score. Lower values of DBI indicate better clustering, and therefore the range of points that have low values of DBI are candidates for choosing the number of clusters. However, we must consider some tips in choosing the number of clusters e.g. we shouldn't consider the points in the steady slope at the end of the curve. Using this approach we show an appropriate number of clusters intuitively chosen considering the DBI metric in Table 10.

Additionally, to validate this approach and also observe the effect of the number of clusters on the proposed strategies, we show the result of experimentation of the proposed methods with a varied number of clusters in the range of [25, 500], in Figs. 5 and 6. It is observed that the chosen number of clusters has relatively good TCP performance, confirming the chosen approach. Furthermore, it is observed in Fig. 6 that the number of clusters has less impact on the performance of the `CovClustering+FP` method on the subject projects when compared to `CovClustering`.

6 Discussion

6.1 Practical considerations

To apply the proposed approach, the coverage of the test suite must be measured. This measurement can be done once and used for subsequent changes to the source code until a certain point. This shortcut technique is also applicable for the defect prediction phase and also distance computation. The coverage measurement can be implemented using static analysis methods to speed up this process. The granularity of code units we have experimented with for coverage measurement is method-level, but measuring coverage in statement-level can be even more effective.

There were no theoretical assumptions about the defect prediction phase, therefore it can be replaced with any standard defect prediction method with hopefully appropriate results. Additionally, The proposed classification process can be used with other feature sets extracted from the source code. Applying cross-project methods can help the usage of the proposed method in the early stage of a project.

One important practical advantage of the proposed TCP method is that the proposed method is designed such that using a weak classifier for the defect prediction phase does not deteriorate the whole TCP result. This is due to the clustering phase which is independent of the fault-proneness estimations and provides a stable base platform for test-case diversification.

6.2 Threats to validity

Construct Validity. Construct validity focuses on the relation between the theory behind the experiment and the observed results. One of the main concerns of this threat is related to the evaluation metrics in our experiment. We considered the first-fail metrics as measures of the effectiveness of TCP. The first failing metric has also been previously used (Noor and Hemmati 2015, 2017; Palma et al. 2018; Rahman et al. 2018; Abou Assi et al. 2021) and is also reasonable to be used when only a few test cases fail such as our case.

Internal Validity. Internal validity refers to whether the relationship between the experiment itself and the result obtained is causal rather than the result of other factors. A major part of our experiment concerns the Defects4J and Defects4J+M datasets which we have relied on because of being previously reviewed by other researchers (Paterson et al. 2019; Noor and Hemmati 2015; Luo et al. 2018b; Mahdieh et al. 2020). The main methods of our implementation are also parts of standard libraries which have been thoroughly tested. One concern that can be mentioned is using some parameters in methods, such as the number of clusters. In practice, choosing these parameters can be done using heuristics (which is practiced in this paper) or by checking multiple values and choosing the one with the best performance.

External Validity. The experimental procedure has been performed on projects which are mostly implemented in the Java language; therefore, the results might differ in projects developed using other languages. However, the clustering algorithms are completely language-neutral and the defect prediction procedure is mostly based on language-independent features. Additionally, by using popular open-source projects that contain large test suites in our empirical study, we tried to study a completely real-world scenario. Despite these facts, in the future, we have to evaluate our approaches using projects with different languages and other characteristics to ensure that the results are generalizable.

Conclusion Validity. Conclusion validity focuses on the significance of the treatment specifically the statistical validity of the conclusions. To enhance conclusion validity, we applied the Wilcoxon signed-rank statistical tests to the results of the experiments to validate the significance of the conclusions of comparing the performance of the algorithms with each other. The Wilcoxon signed-rank has been applied in many TCP experimental comparisons (Luo et al. 2016, 2018a; Kwon et al. 2014; Wang et al. 2017). This non-parametric test is chosen since we did not make assumptions that the data under consideration is normally distributed.

7 Related work

Much research has been conducted in the last two decades to study different methods and analyze their performance for TCP in the context of regression testing. From a big-picture point of view, TCP methods can be categorized based on two aspects: different sources of information used for TCP and various heuristics and optimization strategies used for ordering the test cases (Hemmati 2019). We continue by reviewing different TCP algorithms, considering their source of information and optimization strategy.

Considering the first point of view, many TCP studies have used code coverage as a major source of information for prioritization (Khatibsyarbini et al. 2018). These methods are based on the assumption that test cases with larger coverage have a better ability for fault detection. Other sources of information have also been used for TCP such as historical failure data of test cases, formal specifications or requirements, and source code metrics (Hemmati 2019).

Lachmann et al. (2016) propose using machine learning techniques to leverage test case execution history and test case description texts for prioritizing manual system-level test cases. Their method runs in a completely black-box context which implies better applicability of the method in practice. Hettiarachchi et al. (2016) designed a fuzzy expert system which estimates the risks of system requirements and then prioritize test cases based on the risks which they cover. Arafeen and Do (2013) propose a method that first clusters the requirements based on a text-mining technique and then uses the requirements-test cases traceability matrix to cluster the test cases. Afterward, the test cases in each cluster are prioritized using source code metrics such as McCabe cyclomatic complexity and finally, the test cases are prioritized according to the importance of the requirements to the clients.

Noor and Hemmati (2015) propose a similarity-based TCP approach based on historical failure data of test cases. Their method uses the intuition that test cases that are similar to failed test cases in the past are probable of fault detection.

As another source of information, several researchers have been proposed methods to utilize bug history for test case selection and prioritization. Laali et al. (2016) propose an online TCP method that utilizes the locations of faults in the source code revealed by failed test cases to prioritize the non-executed test cases. A method utilizing previously fixed faults to choose a small set of test cases for test selection is proposed by Engström et al. (2010). Some studies such as Anderson et al. (2014) and Engström et al. (2011), suggest the idea of using the failure history of regression test cases to improve future regression testing phases. Kim and Baik (2010) borrowed ideas from fault localization to tackle the TCP problem. By considering the fact that defects are fixed after being detected, they guess that test cases covering previous faults must have lower priority in TCP ordering because they will have lower fault detection possibility. Wang et al. (2017) proposed a quality-aware TCP method (QTEP) that uses static bug finders and unsupervised methods for defect prediction. Paterson et al. (2019) proposed a ranked-based technique to prioritize test cases that estimate the likelihood of Java classes having bugs. Their experiments show that using their TCP method reduces the number of test cases required to find a fault compared with existing coverage-based strategies.

Multi-objective evolutionary techniques have been of interest for TCP and test selection as they can tackle two or more different kinds of objectives (such as code coverage, requirements coverage, etc.) for prioritization (Wang et al. 2016c; Yoo and Harman 2007; Mondal et al. 2015). Pradhan et al. (2019) employ rule mining on the test execution history to extract relationships among test cases and use multi-objective algorithms to prioritize test cases in a black-box setting.

Due to the relatively high computation cost of TCP algorithms, proposing TCP methods with lower computation costs for large-scale test suites has been investigated. Miranda et al. (2018) propose using hashing-based approaches to provide faster TCP algorithms.

Reinforcement learning (RL) based continuous integration (CI) testing (Spieker et al. 2017) was introduced to prioritize test cases based on applying RL techniques to the test case execution history of CI systems. Bagherzadeh et al. (2021) provide RL-based TCP methods for CI, employing both the test execution history and lightweight code features and show that their methods are effective.

We will thoroughly review methods that utilized clustering methods for test case diversification, due to their relation to the approach of this paper. Carlson et al. (2011) proposed a method based on clustering of method coverage for TCP. Their approach works in two steps, in the first step the test cases are clustered using code coverage similarity. The clustering is performed using an agglomerative hierarchical clustering method [Tan et al. 2016). In the second step, the test cases of each cluster are prioritized using multiple metrics such as code coverage, code complexity, fault history, and a combination of these metrics. They empirically investigate their method on a subset of the Microsoft Dynamics AX project. Their results show that using clustering improves TCP compared to prioritizing without clustering. By utilizing the same coverage clustering method and prioritizing test cases according to the previous failure history of test cases, another TCP method is proposed by Fu et al. (2017). Their method also uses estimations of failure rate according to the program line changes.

Chen et al. (2018) employed ideas from adaptive random testing and clustering to propose TCP methods for object-oriented software. Their methods start by clustering the test cases by comparing the number of objects and methods and also the Object and Method Invocation Sequence Similarity (OMISS) metric (Chen et al. 2016). Afterward, the clusters are sorted in an adaptive random sequence and test cases are sampled iteratively according to the order of clusters. Zhao et al. (2015) have combined Bayesian networks with coverage-based clustering for TCP. Their method works by prioritizing test cases in each cluster by the Bayesian network proposed by Mirarab and Tahvildari (2007) which uses change information, software quality metrics, and test coverage as data sources. They conclude that their TCP method has a higher fault detection rate than the plain Bayesian network-based approach of Mirarab and Tahvildari (2007).

Fang et al. (2014) introduced a new test case similarity measure by comparing the ordering of execution count on program entities. They use this similarity measure to prioritize test cases using both an adaptive random testing inspired method and a clustering-based method. These methods are evaluated empirically by creating mutant versions of multiple open-source projects and measuring the fault detection rate.

History of test case failure is used by Hasan et al. (2017) to improve clustering-based TCP. Their proposed methods order test cases in order of similarity to test cases that have failed in previous versions. Their empirical study is based on real-world faults however the number of versions experimented with is very limited (only 2 versions of 3 projects) and the comparison of their methods is done with random TCP.

A test case failure prediction method based on coverage clustering is proposed by Pang et al. (2013). Their approach divides test cases into two categories of effective and ineffective through *k*-means clustering. Their results show that their coverage clustering method is effective in failure prediction. This result confirms our assumption that test cases with similar execution coverage are likely to have similar failure detection capability.

8 Conclusions and future work

In this paper, to address the challenges of TCP, we propose a method that combines the ideas of test case diversification and the incorporation of fault-proneness estimations. Specifically, we leverage defect prediction models to estimate the fault-proneness of source code areas and use agglomerative clustering to diversify the test cases. The difference between the proposed method with other state-of-the-art TCP methods is that it considers fault-proneness and diversification at the same time in a natural composition. The method proposed can also be extended to other scenarios such as other types of information sources for diversification.

We conducted an empirical study on 357 versions of five real-world projects included in the Defects4J dataset to investigate and compare different approaches. Our evaluation shows that the proposed clustering-based TCP methods are a great improvement over traditional coverage-based TCP methods. Also, the proposed combination of clustering and fault-proneness for TCP is superior to the naive fault-proneness-based TCP methods.

In future work, it is possible to apply the proposed techniques for other software testing applications such as automatic test case generation, test suite reduction, and test selection. Also, in the internal and final test case ordering phases we have used specific strategies, but other strategies can also be studied. To further study and evaluate these methods, these approaches can be executed on other subject programming languages and datasets. Furthermore, applying other methods for defect prediction, such as unsupervised methods can be interesting.

Acknowledgements The authors would like to thank Mrs. Monireh Ghasri for her helpful comments and tips and Dr. Hamid Beigi, Mr. Ahmad Khajehnejad, and members of the Intelligent Systems Laboratory (ISL) for kindly providing hardware facilities.

Funding Not applicable.

Declarations

Conflict of interest The author(s) declare(s) that there is no conflict of interest regarding the publication of this manuscript.

References

Abou Assi, R., Masri, W., Trad, C.: How detrimental is coincidental correctness to coverage-based fault detection and localization? An empirical study. Softw. Test. Verif. Reliab. **31**(5), 1762 (2021)

Alves, E.L., Machado, P.D., Massoni, T., Kim, M.: Prioritizing test cases for early detection of refactoring faults. Softw. Test. Verif. Reliab. **26**(5), 402–426 (2016)

Bagherzadeh, M., Kahani, N., Briand, L.: Reinforcement learning for test case prioritization. IEEE Trans. Softw. Eng. (2021). https://doi.org/10.48550/arXiv.2011.01834

Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. **28**(1), 4–17 (2002)

Bishnu, P.S., Bhattacherjee, V.: Software fault prediction using quad tree-based k-means clustering algorithm. IEEE Trans. Knowl. Data Eng. **24**(6), 1146–1150 (2011)

Boucher, A., Badri, M.: Software metrics thresholds calculation techniques to predict fault-proneness: an empirical comparison. Inf. Softw. Technol. **96**, 38–67 (2018)

Boughorbel, S., Jarray, F., El-Anbari, M.: Optimal classifier for imbalanced data using Matthews correlation coefficient metric. PLoS ONE **12**(6), 0177678 (2017)

Catal, C., Mishra, D.: Test case prioritization: a systematic mapping study. Softw. Qual. J. **21**(3), 445–478 (2013)

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: SMOTE: synthetic minority over-sampling technique. J. Artif. Intell. Res. **16**, 321–357 (2002)

Chen, J., Kuo, F.-C., Chen, T.Y., Towey, D., Su, C., Huang, R.: A similarity metric for the inputs of OO programs and its application in adaptive random testing. IEEE Trans. Reliab. **66**(2), 373–402 (2016)

Chen, J., Zhu, L., Chen, T.Y., Towey, D., Kuo, F.-C., Huang, R., Guo, Y.: Test case prioritization for object-oriented software: an adaptive random sequence approach based on clustering. J. Syst. Softw. **135**, 107–125 (2018)

Chicco, D.: Ten quick tips for machine learning in computational biology. BioData Min. **10**(1), 1–17 (2017)

Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)

D'Ambros, M., Lanza, M., Robbes, R.: Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. **17**(4), 531–577 (2012)

Davies, D.L., Bouldin, D.W.: A cluster separation measure. IEEE Trans. Pattern Anal. Mach Intell. **2**, 224–227 (1979)

Deng, J., Lu, L., Qiu, S.: Software defect prediction via LSTM. IET Softw. **14**(4), 443–450 (2020)

e Abreu, F.B., Carapuça, R.: Candidate metrics for object-oriented software within a taxonomy framework. J. Syst. Softw. **26**(1), 87–96 (1994)

Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)

Elbaum, S., Rothermel, G., Kanduri, S., Malishevsky, A.G.: Selecting a cost-effective test case prioritization technique. Softw. Qual. J. **12**(3), 185–210 (2004)

Elish, K.O., Elish, M.O.: Predicting defect-prone software modules using support vector machines. J. Syst. Softw. **81**(5), 649–660 (2008)

Fang, C., Chen, Z., Wu, K., Zhao, Z.: Similarity-based test case prioritization using ordered sequences of program entities. Softw. Qual. J. **22**(2), 335–361 (2014)

Fu, W., Yu, H., Fan, G., Ji, X.: Coverage-based clustering and scheduling approach for test case prioritization. IEICE Trans. Inf. Syst. **100**(6), 1218–1230 (2017)

Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. **26**(7), 653–661 (2000)

Grindal, M., Lindström, B., Offutt, J., Andler, S.F.: An evaluation of combination strategies for test case selection. Empir. Softw. Eng. **11**(4), 583–611 (2006)

Halstead, M.H.: Elements of Software Science. Operating and Programming Systems Series, Elsevier Science, Inc., Amsterdam (1977)

Hao, D., Zhang, L., Zhang, L., Rothermel, G., Mei, H.: A unified test case prioritization approach. ACM Trans. Softw. Eng. Methodol. **24**(2), 1–31 (2014)

Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., Xie, T.: To be optimal or not in test-case prioritization. IEEE Trans. Softw. Eng. **42**(5), 490–505 (2015)

Harrison, R., Counsell, S.J., Nithi, R.V.: An evaluation of the mood set of object-oriented software metrics. IEEE Trans. Softw. Eng. **24**(6), 491–496 (1998)

Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning, 2nd edn. Springer, Berlin (2008). (**Google Scholar**)

Hettiarachchi, C., Do, H., Choi, B.: Risk-based test case prioritization using a fuzzy expert system. Inf. Softw. Technol. **69**, 1–15 (2016)

Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. **29**(3), 195–209 (2003)

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. **39**(6), 757–773 (2012)

Kandil, P., Moussa, S., Badr, N.: Cluster-based test cases prioritization and selection technique for agile regression testing. J. Softw. Evol. Process **29**(6), 1794 (2017)

Kanmani, S., Uthariaraj, V.R., Sankaranarayanan, V., Thambidurai, P.: Object-oriented software fault prediction using neural networks. Inf. Softw. Technol. **49**(5), 483–492 (2007)

Kazmi, R., Jawawi, D.N., Mohamad, R., Ghani, I.: Effective regression test case selection: a systematic literature review. ACM Comput. Surv. (CSUR) **50**(2), 1–32 (2017)

Khalilian, A., Azgomi, M.A., Fazlalizadeh, Y.: An improved method for test case prioritization by incorporating historical test case data. Sci. Comput. Program. **78**(1), 93–116 (2012)

Khatibsyarbini, M., Isa, M.A., Jawawi, D.N., Tumeng, R.: Test case prioritization approaches in regression testing: a systematic literature review. Inf. Softw. Technol. **93**, 74–93 (2018)

Kumar, A.: Development at the Speed and Scale of Google. QCon, San Francisco (2010)

Ledru, Y., Petrenko, A., Boroday, S., Mandran, N.: Prioritizing test cases with string distances. Autom. Softw. Eng. **19**(1), 65–95 (2012)

Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans. Softw. Eng. **34**(4), 485–496 (2008)

Li, Z., Jing, X.-Y., Zhu, X.: Progress on approaches to software defect prediction. IET Softw. **12**(3), 161–175 (2018)

Li, N., Shepperd, M., Guo, Y.: A systematic review of unsupervised learning techniques for software defect prediction. Inf. Softw. Technol. **122**, 106287 (2020)

Liang, H., Yu, Y., Jiang, L., Xie, Z.: Seml: a semantic LSTM model for software defect prediction. IEEE Access **7**, 83812–83824 (2019)

Luo, Q., Moran, K., Zhang, L., Poshyvanyk, D.: How do static and dynamic test case prioritization techniques perform on modern software systems? An extensive study on GitHub projects. IEEE Trans. Softw. Eng. **45**(11), 1054–1080 (2018a)

Mahdieh, M., Mirian-Hosseinabadi, S.-H., Etemadi, K., Nosrati, A., Jalali, S.: Incorporating fault-proneness estimations into coverage-based test case prioritization methods. Inf. Softw. Technol. **121**, 106269 (2020)

Majd, A., Vahidi-Asl, M., Khalilian, A., Poorsarvi-Tehrani, P., Haghighi, H.: SLDeep: statement-level software defect prediction using deep-learning model on static code features. Expert Syst. Appl. **147**, 113156 (2020)

Mathur, A.P.: Foundations of Software Testing. Addison-Wesley Professional 11. Academic Accommodation Policy. Addison-Wesley, Boston (2010)

Matloob, F., Ghazal, T.M., Taleb, N., Aftab, S., Ahmad, M., Khan, M.A., Abbas, S., Soomro, T.R.: Software defect prediction using ensemble learning: a systematic literature review. IEEE Access **9**, 98754–98771 (2021)

McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **4**, 308–320 (1976)

Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., Rothermel, G.: A static approach to prioritizing JUnit test cases. IEEE Trans. Softw. Eng. **38**(6), 1258–1275 (2012)

Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. **33**(1), 2–13 (2006)

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

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: Defect prediction from static code features: current results, limitations, new approaches. Autom. Softw. Eng. **17**(4), 375–407 (2010)

Okutan, A., Yıldız, O.T.: Software defect prediction using Bayesian networks. Empir. Softw. Eng. **19**(1), 154–181 (2014)

Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. IEEE Trans. Softw. Eng. **31**(4), 340–355 (2005)

Pan, R., Bagherzadeh, M., Ghaleb, T.A., Briand, L.: Test case selection and prioritization using machine learning: a systematic literature review. Empir. Softw. Eng. **27**(2), 1–43 (2022)

Panda, S., Munjal, D., Mohapatra, D.P.: A slice-based change impact analysis for regression test case prioritization of object-oriented programs. Adv. Softw. Eng. (2016). https://doi.org/10.1155/2016/7132404

Pandey, S.K., Mishra, R.B., Tripathi, A.K.: BPDET: an effective software bug prediction model using deep representation and ensemble learning techniques. Expert Syst. Appl. **144**, 113085 (2020)

Pei, H., Yin, B., Xie, M., Cai, K.-Y.: Dynamic random testing with test case clustering and distance-based parameter adjustment. Inf. Softw. Technol. **131**, 106470 (2021)

Pradhan, D., Wang, S., Ali, S., Yue, T., Liaaen, M.: Employing rule mining and multi-objective search for dynamic test case prioritization. J. Syst. Softw. **153**, 86–104 (2019)

Rahman, M.A., Hasan, M.A., Siddik, M.S.: Prioritizing dissimilar test cases in regression testing using historical failure data. Int. J. Comput. Appl. **975**, 8887 (2018)

Rothermel, G., Harrold, M.J., Von Ronne, J., Hong, C.: Empirical studies of test-suite reduction. Softw. Test. Verif. Reliab. **12**(4), 219–249 (2002)

Shrivathsan, A., Ravichandran, K., Krishankumar, R., Sangeetha, V., Kar, S., Ziemba, P., Jankowski, J.: Novel fuzzy clustering methods for test case prioritization in software projects. Symmetry **11**(11), 1400 (2019)

Song, Q., Guo, Y., Shepperd, M.: A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans. Softw. Eng. **45**(12), 1253–1269 (2018)

Srikanth, H., Hettiarachchi, C., Do, H.: Requirements based test prioritization using risk factors: an industrial study. Inf. Softw. Technol. **69**, 71–83 (2016)

Tan, P.-N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Pearson Education, Noida (2016)

Tong, H., Liu, B., Wang, S.: Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. Inf. Softw. Technol. **96**, 94–111 (2018)

Wang, R., Jiang, S., Chen, D., Zhang, Y.: Empirical study of the effects of different similarity measures on test case prioritization. Math. Probl. Eng. (2016a). https://doi.org/10.1155/2016/8343910

Wang, T., Zhang, Z., Jing, X., Liu, Y.: Non-negative sparse-based semiboost for software defect prediction. Softw. Test. Verif. Reliab. **26**(7), 498–515 (2016b)

Wang, S., Liu, T., Nam, J., Tan, L.: Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng. **46**(12), 1267–1293 (2018)

Weyuker, E.J., Ostrand, T.J., Bell, R.M.: Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. Empir. Softw. Eng. **13**(5), 539–559 (2008)

Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof, vol. 39. Prentice Hall, Englewood Cliffs (1996)

Xu, D., Tian, Y.: A comprehensive survey of clustering algorithms. Ann. Data Sci. **2**(2), 165–193 (2015)

Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., Zhang, T., Keung, J., Tang, Y.: LDFR: learning deep feature representation for software defect prediction. J. Syst. Softw. **158**, 110402 (2019)

Yao, J., Shepperd, M.: The impact of using biased performance metrics on software defect prediction research. Inf. Softw. Technol. **139**, 106664 (2021)

Yedida, R., Menzies, T.: On the value of oversampling for deep learning in software defect prediction. IEEE Trans. Softw. Eng. (2021). https://doi.org/10.48550/arXiv.2008.03835

Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012)

Zhang, Z.-W., Jing, X.-Y., Wang, T.-J.: Label propagation based semi-supervised learning for software defect prediction. Autom. Softw. Eng. **24**(1), 47–69 (2017)

Zhong, H., Zhang, L., Mei, H.: An experimental study of four typical test suite reduction techniques. Inf. Softw. Technol. **50**(6), 534–546 (2008)

Zhu, K., Zhang, N., Ying, S., Zhu, D.: Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. IET Softw. **14**(3), 185–195 (2020)

Aljamaan, H., Alazba, A.: Software defect prediction using tree-based ensembles. In: Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 1–10 (2020)

Anderson, J., Salem, S., Do, H.: Improving the effectiveness of test suite through mining historical data. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 142–151. ACM (2014)

Arafeen, M.J., Do, H.: Test case prioritization using requirements-based clustering. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 312–321. IEEE (2013)

Ashraf, E., Rauf, A., Mahmood, K.: Value based regression test case prioritization. In: Proceedings of the World Congress on Engineering and Computer Science, vol. 1, pp. 24–26 (2012)

Carlson, R., Do, H., Denton, A.: A clustering approach to improving test case prioritization: an industrial case study. In: ICSM, vol. 11, pp. 382–391 (2011)

Chen, J., Bai, Y., Hao, D., Zhang, L., Zhang, L., Xie, B.: How do assertions impact coverage-based test-suite reduction? In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 418–423. IEEE (2017)

Eghbali, S., Kudva, V., Rothermel, G., Tahvildari, L.: Supervised tie breaking in test case prioritization. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 242–243. IEEE (2019)

Engström, E., Runeson, P., Wikstrand, G.: An empirical evaluation of regression testing based on fix-cache recommendations. In: 2010 Third International Conference on Software Testing, Verification and Validation, pp. 75–78. IEEE (2010)

Engström, E., Runeson, P., Ljung, A.: Improving regression testing transparency and efficiency with history-based prioritization—an industrial case study. In: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, pp. 367–376. IEEE (2011)

Fraser, G., Wotawa, F.: Redundancy based test-suite reduction. In: International Conference on Fundamental Approaches to Software Engineering, pp. 291–305. Springer (2007)

Fu, W., Menzies, T.: Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 72–83 (2017)

Ghotra, B., McIntosh, S., Hassan, A.E.: Revisiting the impact of classification techniques on the performance of defect prediction models. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 789–800. IEEE (2015)

Guo, L., Ma, Y., Cukic, B., Singh, H.: Robust prediction of fault-proneness by random forests. In: 15th International Symposium on Software Reliability Engineering, pp. 417–428. IEEE (2004)

Hasan, M.A., Rahman, M.A., Siddik, M.S.: Test case prioritization based on dissimilarity clustering using historical data analysis. In: International Conference on Information, Communication and Computing Technology, pp. 269–281. Springer (2017)

Hemmati, H.: Advances in techniques for test prioritization. In: Advances in Computers, vol. 112, pp. 185–221. Elsevier, Amsterdam (2019)

Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N.: DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 34–45. IEEE (2019)

Hoang, T., Kang, H.J., Lo, D., Lawall, J.: CC2Vec: distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 518–529 (2020)

Jiang, B., Zhang, Z., Chan, W.K., Tse, T.: Adaptive random test case prioritization. In: 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 233–244. IEEE (2009)

Jing, X.-Y., Ying, S., Zhang, Z.-W., Wu, S.-S., Liu, J.: Dictionary learning based software defect prediction. In: Proceedings of the 36th International Conference on Software Engineering, pp. 414–423 (2014)

Just, R., Jalali, D., Ernst, M.D.: Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440. ACM (2014)

Khoshgoftaar, T.M., Gao, K., Seliya, N.: Attribute selection and imbalanced data: problems in software defect prediction. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, vol. 1, pp. 137–144. IEEE (2010)

Kim, S., Baik, J.: An effective fault aware test case prioritization by incorporating a fault localization technique. In: Proceedings of the 2010 ACM–IEEE International Symposium on Empirical Software Engineering and Measurement, p. 5. ACM (2010)

Kläs, M., Elberzhager, F., Münch, J., Hartjes, K., Von Graevemeyer, O.: Transparent combination of expert and measurement data for defect prediction: an industrial case study. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 2, pp. 119–128. ACM (2010)

Kwon, J.-H., Ko, I.-Y., Rothermel, G., Staats, M.: Test case prioritization based on information retrieval concepts. In: 2014 21st Asia–Pacific Software Engineering Conference, vol. 1, pp. 19–26. IEEE (2014)

Laali, M., Liu, H., Hamilton, M., Spichkova, M., Schmidt, H.W.: Test case prioritization using online fault detection information. In: Ada-Europe International Conference on Reliable Software Technologies, pp. 78–93. Springer (2016)

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

Leon, D., Podgurski, A.: A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In: 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003, pp. 442–453. IEEE (2003)

Li, J., He, P., Zhu, J., Lyu, M.R.: Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 318–328. IEEE (2017)

Li, R., Zhou, L., Zhang, S., Liu, H., Huang, X., Sun, Z.: Software defect prediction based on ensemble learning. In: Proceedings of the 2019 2nd International Conference on Data Science and Information Technology, pp. 1–6 (2019)

Lou, Y., Chen, J., Zhang, L., Hao, D.: A survey on regression test-case prioritization. In: Advances in Computers, vol. 113, pp. 1–46. Elsevier, Amsterdam (2019)

Luo, Q., Moran, K., Poshyvanyk, D.: A large-scale empirical comparison of static and dynamic test case prioritization techniques. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 559–570 (2016)

Luo, Q., Moran, K., Poshyvanyk, D., Di Penta, M.: Assessing test case prioritization on real faults and mutants. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 240–251. IEEE (2018b)

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J.: Taming Google-scale continuous testing. In: 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 233–242. IEEE (2017)

Meneely, A., Williams, L., Snipes, W., Osborne, J.: Predicting failures with developer networks and social network analysis. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 13–23. ACM (2008)

Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A.: Fast approaches to scalable similarity-based test case prioritization. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 222–232. IEEE (2018)

Mirarab, S., Tahvildari, L.: A prioritization approach for software test cases based on Bayesian networks. In: International Conference on Fundamental Approaches to Software Engineering, pp. 276–290. Springer (2007)

Mondal, D., Hemmati, H., Durocher, S.: Exploring test suite diversification and code coverage in multi-objective test case selection. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–10. IEEE (2015)

Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE'08. pp. 181–190. IEEE (2008)

Nam, J.: Survey on Software Defect Prediction. Technical Report. Department of Computer Science and Engineering, The Hong Kong University of Science and Technology (2014)

Nam, J., Kim, S.: CLAMI: defect prediction on unlabeled datasets. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 452–463. IEEE (2015)

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

Noor, T.B., Hemmati, H.: Studying test case failure prediction for test case prioritization. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 2–11 (2017)

Palma, F., Abdou, T., Bener, A., Maidens, J., Liu, S.: An improvement to test case failure prediction in the context of test case prioritization. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 80–89 (2018)

Pang, Y., Xue, X., Namin, A.S.: Identifying effective test cases through k-means clustering for enhancing regression testing. In: 2013 12th International Conference on Machine Learning and Applications, vol. 2, pp. 78–83. IEEE (2013)

Paterson, D., Campos, J., Abreu, R., Kapfhammer, G.M., Fraser, G., McMinn, P.: An empirical study on the use of defect prediction for test case prioritization. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 346–357. IEEE (2019)

Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N.: Building an ensemble for software defect prediction based on diversity selection. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–10 (2016)

Pinzger, M., Nagappan, N., Murphy, B.: Can developer-module networks predict failures? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 2–12. ACM (2008)

Saha, R.K., Zhang, L., Khurshid, S., Perry, D.E.: An information retrieval approach for regression test prioritization based on program changes. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 268–279. IEEE (2015)

Salehie, M., Li, S., Tahvildari, L., Dara, R., Li, S., Moore, M.: Prioritizing requirements-based regression test cases: a goal-driven practice. In: 2011 15th European Conference on Software Maintenance and Reengineering, pp. 329–332. IEEE (2011)

Sandha, S.S., Aggarwal, M., Fedorov, I., Srivastava, M.: Mango: a python library for parallel hyperparameter tuning. In: ICASSP 2020—2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 3987–3991. IEEE (2020)

Shivaji, S., Whitehead, E.J., Akella, R., Kim, S.: Reducing features to improve bug prediction. In: 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 600–604. IEEE (2009)

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

Wang, S., Ali, S., Yue, T., Bakkeli, Ø., Liaaen, M.: Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 182–191 (2016c)

Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 297–308. IEEE (2016d)

Wang, S., Nam, J., Tan, L.: QTEP: quality-aware test case prioritization. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 523–534 (2017)

Wilcoxon, F.: Individual comparisons by ranking methods. In: Breakthroughs in Statistics, pp. 196–202. Springer, New York (1992)

Yan, M., Fang, Y., Lo, D., Xia, X., Zhang, X.: File-level defect prediction: unsupervised vs. supervised models. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 344–353. IEEE (2017)

Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 17–26. IEEE (2015)

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H.: Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 157–168 (2016)

Yao, J., Shepperd, M.: Assessing software defection prediction performance: why using the Matthews correlation coefficient matters. In: Proceedings of the Evaluation and Assessment in Software Engineering, pp. 120–129 (2020)

Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 140–150 (2007)

Yoo, S., Harman, M., Tonella, P., Susi, A.: Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 201–212 (2009)

Zainab, A., Ghrayeb, A., Houchati, M., Refaat, S.S., Abu-Rub, H.: Performance evaluation of tree-based models for big data load forecasting using randomized hyperparameter tuning. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 5332–5339. IEEE (2020)

Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E.: Cross-project defect prediction using a connectivity-based unsupervised classifier. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 309–320. IEEE (2016)

Zhao, X., Wang, Z., Fan, X., Wang, Z.: A clustering-Bayesian network based approach for test case prioritization. In: 2015 IEEE 39th Annual Computer Software and Applications Conference, vol. 3, pp. 542–547. IEEE (2015)

Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007), p. 9. IEEE (2007)