SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2020, Vol. 63 190102:1–190102:16
https://doi.org/10.1007/s11432-019-2859-8

Special Focus on Big Data-Driven Intelligent Software Development for Onsite Programming

Quality assessment of crowdsourced test cases

Yuan ZHAO[1], Yang FENG[1*], Yi WANG[2*], Rui HAO[1],
Chunrong FANG[1*] & Zhenyu CHEN[1]

[1]*State Key Laboratory for Novel Software Technology Nanjing University, Nanjing* 210093*, China;*
[2]*Golisano College of Computing and Information Sciences, Rochester Institute of Technology, Rochester* 14623*, USA*

Received 19 September 2019/Revised 23 January 2020/Accepted 3 April 2020/Published online 10 August 2020

Abstract Various software-engineering problems have been solved by crowdsourcing. In many projects, the software outsourcing process is streamlined on cloud-based platforms. Among software engineering tasks, test-case development is particularly suitable for crowdsourcing, because a large number of test cases can be generated at little monetary cost. However, the numerous test cases harvested from crowdsourcing can be high- or low-quality. Owing to the large volume, distinguishing the high-quality tests by traditional techniques is computationally expensive. Therefore, crowdsourced testing would benefit from an efficient mechanism distinguishes the qualities of the test cases. This paper introduces an automated approach — TCQA — to evaluate the quality of test cases based on the onsite coding history. Quality assessment by TCQA proceeds through three steps: (1) modeling the code history as a time series, (2) extracting the multiple relevant features from the time series, and (3) building a model that classifies the test cases based on their qualities. Step (3) is accomplished by feature-based machine-learning techniques. By leveraging the onsite coding history, TCQA can assess the test-case quality without performing expensive source-code analysis or executing the test cases. Using the data of nine test-development tasks involving more than 400 participants, we evaluated TCQA from multiple perspectives. The TCQA approach assessed the quality of the test cases with higher precision, faster speed, and lower overhead than conventional test-case quality-assessment techniques. Moreover, TCQA provided yield real-time insights on test-case quality before the assessment was finished.

Keywords crowdsourcing, onsite programming, test quality, programming behavior

Citation Zhao Y, Feng Y, Wang Y, et al. Quality assessment of crowdsourced test cases. Sci China Inf Sci, 2020, 63(9): 190102, https://doi.org/10.1007/s11432-019-2859-8

1 Introduction

Crowdsourcing is gaining traction in software engineering research [1]. The crowdsourcing approach engages a large workforce of crowd workers in various software-engineering tasks, such as testing [1], design [2], optimization [3], and development [4]. Crowdsourcing promises many benefits in software development, but harvesting test cases from a large population of crowd workers is especially useful for developers wishing to improving the quality of their products [4]. Writing a test case for a micro-task is more appropriate than implementing features or fixing bugs, because many test cases are independent of the core logic of a software system. Moreover, multiple crowd workers working on the same task generate a large number of test cases. Therefore, the likelihood of obtaining high-quality test cases is boosted by crowdsourcing [4].

On the downside, crowdsourcing development generates many low-quality test cases. As executing all test cases is resource-intensive and may introduce security risks, distinguishing the high-quality test cases

* Corresponding author (email: charles.fengy@gmail.com, yi.wang@rit.edu, fangchunrong@nju.edu.cn)

from the low-quality ones is desired but challenging task. Manually reviewing all test cases is extremely labor-intensive, but is more effective than conventional source-code analysis techniques, which struggle to distinguish the short and similar test cases harvested through crowdsourcing. Meanwhile, static software metrics may provide no useful insights, because the coverage metrics and are often weakly correlated with the quality of a test case [5]. An alternative technique is, mutation testing, which exhaustively executes all artificial buggy versions (mutants), and determines whether the test cases detect these bugs. However, mutation testing requires large time and computational resources [6]. Therefore, to assess the test cases generated by crowdsourcing, we need innovative techniques requiring minimal information.

Current software crowdsourcing is often performed on cloud-based platforms [7] which offer an integrated development environment (IDE) for code writing. Cloud-based platforms are an ideal infrastructure solution for software crowdsourcing [7] (see Section 2); in particular, they capture the fine-grained activities of the crowd workers [7–10]. Using the hypertext transfer protocol (HTTP) communications between the browser and the remote server, one can reconstruct the fine-grained dynamic history of the code development[1], which is represented by the file-size changes of the source-code. The code history provides rich information on the whole problem-solving process of a given code [9, 10, 12].

Moreover, from the fine-grained dynamic code history, we can estimate the quality of the crowdsourced test cases harvested from cloud-based source-code-editing platforms [8, 10]. Here, we develop a novel approach named TestCase quality assessment (TCQA), which analyzes the fine-grained dynamic code history, negating the need for source-code analysis techniques or mutation testing. TCQA first models the dynamic code history as a time series, and then extracts a set of features from the constructed time series [13]. Based on the extracted features, TCQA classifies a given test case into one of three quality categories (high, medium, or low), providing a cost-effective assessment of numerous crowdsourced test cases.

The proposed approach is evaluated from multiple perspectives — effectiveness, efficiency, and real-time insights — on a dataset of 2193 test cases developed by over 400 crowd workers. In the effectiveness evaluation, TCQA classification is tested in three scenarios: within-task, whole-sample, and cross-task. The efficiency evaluation compares the performances (time and computational resource consumption) of TCQA, collecting coverage metrics, and mutation testing. Finally, to assess whether TCQA provides real-time insights, we determine whether TCQA can predict the quality of a test case that is not yet submitted by a crowd worker. Together, the evaluations demonstrate the potential of TCQA in assessing the quality of test cases harvested through crowdsourcing on cloud-based code-editing platforms.

The main contributions of this paper are as follows.

• The proposed approach (TCQA) automatically assesses the quality of crowdsourced test cases using the dynamic code history, without requiring source-code analysis and mutation testing. The approach employs multiple time-series analysis techniques.

• A browser-based extension for the crowdsourced platform, MoocTest[2], enables online programming by users. In our browser-based extension of the programming environment, the onsite coding behaviors of crowd workers can be logged and analyzed on the fly.

• TCQA is comprehensively evaluated from three perspectives on the MoocTest platform. TCQA can precisely and efficiently estimate the qualities of the crowdsourced test cases, without demanding excessive computing resources. Moreover, TCQA provides real-time insights before a test case is submitted by a crowd worker.

1) In this paper, the word fine-grained means that the time unit of the operation history is finer than a revision history that can be captured by a traditional version-control systems (VCS), and which has been sufficiently well studied [11] that every character-level source-code change (insertion and deletion) can be traced.

2) MoocTest. http://www.mooctest.net/.

2 Cloud-based software crowdsourcing

2.1 Supporting software crowdsourcing with cloud-based platforms

Software crowdsourcing is an emerging approach in software engineering approach. Almost all software development tasks requirement engineering, design, implementation, testing, maintenance, and documentation — can be crowdsourced. Crowdsourcing also engages the end-users in the software development process, because crowd workers can participate in multiple development tasks. Software crowdsourcing can potentially shift the traditional development practices of industrial software to peer production software development [4]. To reach its full potential, software crowdsourcing requires a highly scalable infrastructure that supports its entire process.

Software crowdsourcing on cloud-based platforms is rooted in large-scale collective problem in computer science, and is a natural choice for solving such problems [7]. The Dagstuhl Seminar "Cloud-based software crowdsourcing" [14] described crowdsourcing as follows: "Metaphorically, it can be regarded as synergy between two clouds — machine cloud and human cloud, — towards the ultimate goal of developing high-quality and low-cost software products." The heterogeneous "human cloud" requires a computational infrastructure that supports heterogeneity. Cloud-based platforms are significantly more beneficial for software crowdsourcing than the traditional development environments of collaborative software. Many features of cloud infrastructure, such as scalability architecture, automated migration, ubiquitous access, and monitoring, can support software crowdsourcing. For example, cloud-based platforms offer ubiquitous access to crowd workers using any devices. Therefore, workers can perform micro-tasks without worrying about the configurations of the development, build, and run-time environments. On cloud-based platforms, the entire software crowdsourcing process is fully transparent; that is, every action of the crowd workers can be tracked through the HTTP communications between the workers' browsers and the cloud servers. Moreover, this transparency does not interrupt the normal work processes of the crowd workers. Cloud-based platforms facilitate software-engineering tasks, such as software design[3], collaborative coding [15], testing [16], and deployment[4], across distributed and heterogeneous devices.

2.2 Crowdsourced test case development on cloud-based platforms — minimizing the quality assessment costs

Among the various software development tasks, test-case development is particularly suitable for crowdsourcing. However, the effectiveness and efficiency of any unit testing depends on the quality of the test cases [17,18]. Whereas writing a good test case can be difficult and tedious, automatic test-case-generation techniques (e.g., [19]) are beset by the external-method-call problem, the object-creation problem, and other problems [20]. Moreover, even when artificial faults are accurately detected, automatically generated unit tests may not detect real faults in practice, for several reasons [21, 22]. Hence, we posit that human intelligence and efforts are often beneficial (and even necessary) for producing high-quality tests [1]. These demands are fully satisfied by crowdsourced test-case development.

Although crowdsourced test-case development can naturally harvest test cases through human intelligence, assessing the quality of numerous crowdsourced test cases is a challenging task. Processing a large number of test cases by traditional techniques is time- and resource-intensive, but the native features of cloud-based platforms provide new opportunities.

As mentioned above, cloud-based platforms introduce transparency to the entire software crowdsourcing process. Such transparency provides the fine-grained (character-level) history of the source code throughout the crowdsourced test-case development. This history contains rich information of the entire problem-solving process, which is erased in the final source code [12,23]. Hence, it can be exploited in analytics of wide-ranging concerns in software development, such as developers' subjective experience [10], behavioral characteristics [9], expertise evaluation [8], and, of course, the quality of crowdsourced test cases.

3) Gen My Model. https://www.genmymodel.com/.
4) IMB DevOps. https://www.ibm.com/cloud/devops.

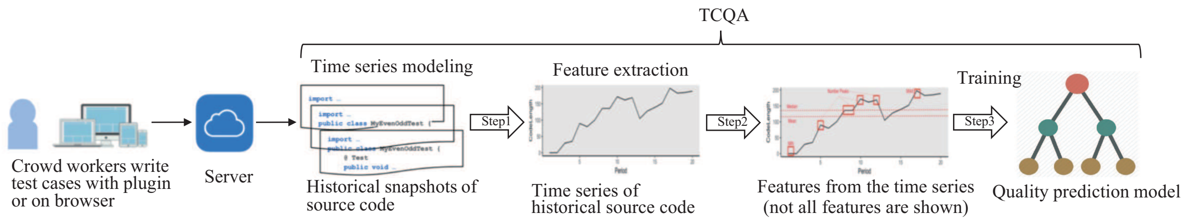
TCQA
Time series modeling
Feature extraction
Training
Crowd workers write test cases with plugin or on browser
Server
Step1
Step2
Step3
Historical snapshots of source code
Time series of historical source code
Features from the time series (not all features are shown)
Quality prediction model

Figure 1 (Color online) Overview of assessing the quality of a test case from the dynamic code history using TCQA.

Cloud-based platforms provide support for highly effective and efficient crowdsourced test-case development processes. The cloud-based platforms capture the fine-grained source code history while a crowd worker develops a piece of code. Once the test cases are submitted (or even before submission), their quality can be automatically assessed. Because the fine-grained source code history requires no mutation testing, the time and computational cost of evaluating its quality is minimal. The requester can then decide whether to include or exclude a worker's submission.

3 Our approach — TCQA

Figure 1 describes the whole process of applying TCQA to assess the quality of test cases written by crowd workers. It contains three major steps.

3.1 Step 1. from dynamic code history to time-series

Following the insights provided in [9,10], this work examines the changing size of the source code induced by a crowd worker's activities. For instance, suppose that a crowd worker performs a "deletion". What is deleted is unimportant, but how much is deleted is crucial. In the first step, TCQA simply records the size of the source code (measured as the number of characters) at the corresponding time point and inserts it into a sequence. Supposing n periods in total, the dynamic code history of a test case after this step is given by the following n-period time series:

$$\langle t_1, s_1\rangle, \langle t_2, s_2\rangle, \ldots, \langle t_i, s_i\rangle, \ldots, \langle t_n, s_n\rangle,$$

where t_i is the current time and s_i is the size of the test code at t_i. Note that TCQA does not require equal lengths of the generated time series because the classification (Step 3) uses the extracted features (Step 2) — and not the raw time series. However, TCQA allows the data prepossessing for generating equal-length time series. With this support, TCQA can be extended to distance-based classification techniques such as dynamic time wrapping (DTW) in future [24].

3.2 Step 2. extracting the time-series features

The prior step creates the time series representing the test code history. In the next step, TCQA extracts the features from these time series. Given the dynamic nature of the time series, a large number of features can potentially be extracted. For example, let us consider the autocorrelation, a statistical feature defining the correlation between the elements of a time series and other elements from the same time series separated by a given interval. If the time interval is varied, Step 2 may produce hundreds of features.

To solve this problem, we extract the relevant features from the time-series data using the off-the-shelf library TSFRESH[5], a Python package that extracts the features and evaluates their relevance by a three-step procedure [25]. First, TSFRESH characterizes the time series by comprehensive and well-established feature mappings, and describes the meta-information in additional features. Second, the significance of each feature vector in the target prediction is individually and independently evaluated. This step

5) TSFRESH. http://tsfresh.readthedocs.io/en/latest/.

Table 1 Extracted features and their meanings

Category	Feature	Meaning
Simple metrics	Maximum	Highest (normalized) value of the time series.
	Mean	Mean of the time series.
	sum_of_reoccurring_values	Sum of reoccurring values in the time series.
Statistical metrics	c3*	Non-linearity of the time series, see [27] for more details.
	abs_energy	Absolute energy of the time series (sum of the squared values).
	agg_linear_trend*	Linear least-squares regression of values of the time series.
Frequency-based metrics	fft_coefficient*	Fourier coefficients of the one-dimensional discrete fast fourier transform for real parameters.
	spkt_welch_density	Cross-power spectral density of the time series at different frequencies.

* Multiple features of this type can result from different input parameters.

generates a vector of multiple p-values. In the last step, TSFRESH evaluates the vector of p-values using the Benjamini-Yekutieli procedure [26], and decides the features to be retained.

Identified features. TSFRESH extracted 87 relevant features and placed them into three broad categories: simple metrics (e.g., maximum), statistical metrics (e.g., agg_linear_trend), and frequency-based metrics (e.g., fft_coefficient). Frequency-based metrics are produced by transforming a time series from the time domain to the frequency domain using the fast fourier transform (FFT) algorithm. Table 1 lists a subset of the extracted features and their meanings. An exhaustive feature list is precluded by space limitations, but examples of the features in each category are provided to clarify their main concepts.

3.3 Step 3. predicting the quality of crowdsourced tests

Machine learning trains predictive models on training data, and makes predictions on test data. Machine learning is widely applied in software-engineering problems [28], and is a natural choice for evaluating the quality of numerous test cases by TCQA. Especially, it learns a classification model on some instances and then classifies new instances into different categories.

3.3.1 *Training*

We seek to automatically classify many crowdsourced test cases into quality categories while minimizing the computational resources. To this end, we require a method that evaluates the quality of the test cases based on a training set of test cases. The quality can be determined by various metrics such as the mutation score and code coverage the metrics used in our experiments are described in Section 4. After evaluating the quality of the training test cases and labeling each case with its relative quality degree, the machine-learning model can be trained to classify future test cases without requiring such expensive execution or analysis.

3.3.2 *Algorithms*

Classification algorithms are many and varied. Examples are naive bayes, decision tree classifiers, neural networks, and support vector machines (SVM). The default classification algorithm in TCQA is random forest [29], an ensemble learning method that improves the robustness of classification models by conferring the power, flexibility, and interpretability of tree-based classifiers. The TCQA classification algorithm was decided after extensive empirical performance comparisons of multiple algorithms on our own empirical data. The random forest-based models consistently yielded the best values of the most important comparison metric, namely, the F-measure. Moreover, random forest can handle imbalanced data [30], which dominate the data collected in real-world practices.

Besides the default algorithm random forest. TCQA admits other algorithms such as naive Bayers, J48, and SVM, which can be implemented with minimal effort. Non-feature-based time-series classification algorithms such as DTW [31] are also easily implemented in TCQA's framework.

Table 2 Statistic information of subjected tasks

Task	No. tests	LOC	No. classes
CMD	134	566	1
Datalog	649	589	9
ITClocks	134	1071	13
JMerkle	370	774	5
LunarCalendar	561	1170	8
QuadTree	345	644	6

4 Experiment design

Our evaluation dataset was prepared from the results of the software-testing contest held at the International Conference on Software Quality, Reliability, and Security (QRS)[6], which simulated crowdsourced testing. On the cloud-based programming platform –MoocTest–, participants could write test cases in any standard web browser. The source-code snapshots were periodically sent to the server through the http `PUT` method. All participants were allocated two hours to write test cases in MoocTest. For evaluation, we collected the data of six subject programs, each belonging to a separate subject task, thus obtaining 2193 test cases. The qualities of the submitted test cases were evaluated by two classic criteria: the coverage score and the mutation score [32]. The coverage score is widely used for evaluating the execution degree of a program's source code. A program with a high test coverage score will less likely contain undetected software bugs than a program with a lower test coverage score. The mutation score is also popular for evaluating the effectiveness of test suites [33]. The mutation score is computed after mutation testing, which involves small modifications of the program. Each modified version is called a mutant. If the test suites detect the modification, the mutant can be killed. The mutation score is the ratio of the number of killed mutants to the total number of mutants.

Here, the branch coverage of each test case was evaluated by Jacoco[7]. The the coverage score defines the percentage of covered branches to all branches. The mutation score of each test case was calculated by the mutation testing tool — PIT[8]. The final score of each test case was determined by averaging the coverage and mutation scores. Based on these scores, the training-test cases were assigned with labels for the machine-learning classifier.

Table 2 summarizes the basic information of the six subject programs in the experiment.

In typical crowdsourced software development, a requester may be interested only in the few best cases. However, a sustainable crowdsourcing practice should offer opportunities for workers to develop their skills, particularly those with middle-level skills. For this purpose, we focus not merely on the top results, but instead categorize these crowdsourced test cases into three levels, i.e., low, medium, and high quality. We then label the top 25% as "high-quality", the bottom 25% as "low-quality", and the remainder as "medium-quality". Note that this method identifies the middle-level workers and provides them with good test-case examples, encouraging them to develop their software testing skills.

4.1 Research questions

The attractiveness of TCQA will depend on whether it accurately estimates the quality of the test cases while minimizing the time cost. To further reduce the computational costs, the requester should also gain real-time insights on the test-case quality. To meet these ends, we formulated the following research questions (RQs):

RQ$_1$: How accurately can TCQA assess the quality of the crowdsourced test cases using from their dynamic code histories? This RQ evaluates the effectiveness of TCQA.

RQ$_2$: How efficiently can TCQA assess the quality of the crowdsourced test cases, relative to the traditional quality assessment techniques? This RQ evaluates the efficiency of TCQA.

6) http://paris.utdallas.edu/qrs17/contest.html.
7) Jacoco. http://www.eclemma.org/jacoco/.
8) P mutation test tool. http://pitest.org/.

RQ_3: Can TCQA assess the quality of the crowdsourced test cases in real-time before they are submitted? This RQ evaluates whether TCQA can determine the quality of a crowdsourced test case from its partial dynamic code history.

4.2 Implementation and environmental setup

The experimental subjects edited their source code in the integrated web IDE of the MoocTest platform. The web IDE is fully functional, enabling users to edit code in their browser and run it on MoocTest's cloud servers. Each crowd worker's source code is automatically saved and is periodically sent to the cloud server through the `PUT` method. By specifying the auto-save time intervals, the dynamic code history can be retrieved at varying granularities. In our experiments, we set the default auto-save time interval.

As described in Section 4, TCQA creates a time series for all test cases in the dataset and extracts the features using TSFRESH. The random forest classifiers were built using the scikit-learn library for Python. In the model training phase, we adopted the default parameter settings of the scikit-learn library, namely, a 10-tree forest, maximum tree depth 10, and minimum number of samples required for internal node splitting 2. All experiments were performed on a platform with an 8-core 3800 MHz CPU, a 64-bit Ubuntu 14.04, and 12 GB of RAM.

4.3 Overview of experiments

RQ_1, RQ_2, and RQ_3 were addressed in three independent experiments. This subsection overviews the high-level experimental design.

4.3.1 *Experiment 1 (RQ_1)*

Given the nature of the data (6 independent tasks), whether a model built using the data of multiple tasks can predict one instance of a specific target task must be comprehensively evaluated. As this problem is akin to transfer learning [34–36], we evaluated the performance of TCQA in three scenarios as follows:

- **Within-task scenario.** The performance evaluation and model training are performed on data from the same task.
- **Whole-sample scenario.** The model is trained on the data of five tasks, and its performance is evaluated on the data of the remaining task.
- **Cross-task scenario.** The model is trained on the data of one task, and its performance is evaluated on the data of the remaining five tasks.

The experiments in these three scenarios require special considerations, which will be described alongside the results for the reader's convenience.

4.3.2 *Experiment 2 (RQ_2)*

The second experiment compares the efficiency of TCQA with those of traditional techniques. The efficiency was evaluated by two traditional techniques coverage-metrics evaluation and mutation testing. We are confident that test evaluations are much faster in TCQA than in mutation testing; however, a precise measure will facilitate a precise performance estimation when deploying TCQA in real-world cloud-based crowdsourcing platforms. Here, the performance of TCQA was compared with that of the method for evaluating MoocTest submissions (see the first paragraph of Section 4).

4.3.3 *Experiment 3 (RQ_3)*

Recall that RQ_3 assesses whether TCQA can generate real-time insights on the quality of the test cases. Therefore, the experimental conditions must ensure that the time series is only partially available. For this experiment, we created nine experimental conditions covering 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% or 100% of the whole time series (the final condition provided the baseline condition). Under each experimental condition $x\%$, the prediction and performance evaluation were performed on the first

$x\%$ of each time series. For example, under the 50% condition, the prediction was made using the first 50% of the whole time series. If the results of under any $x\%$ condition are were comparable to those of the baseline condition (100%), we concluded that TCQA can assess the quality of a test case from the partial time series. That is, TCQA provides real-time insights into the crowdsourced test case before the case is submitted.

4.4 Evaluation metrics

The machine-learning models were evaluated by three common metrics: precision, recall, and F-measure. In the following expressions, TP, FP, FN, and TN denote the numbers of true positives, false positives, false negatives, and true negatives, respectively. Considering the nature of the multi-class classification and the imbalance in the data, we weighted the metrics as proposed in the existing machine-learning literature [37].

Precision. The precision defines the fraction of true positive instances among the instances that are predicted as positive. The precision P_c of class c is calculated as $\frac{\mathrm{TP}_c}{\mathrm{TP}c+\mathrm{FP}c}$. Now suppose that class c contains k_c instances, and that all classes are collected into C. The overall weighted precision is given by Eq. (1). A high precision denotes a low proportion of prediction errors.

Recall. The recall defines the fraction of detected true positive instances among the actual positive instances. The recall R_c of class c is calculated as $\frac{\mathrm{TP}_c}{\mathrm{TP}_c+\mathrm{FN}_c}$. Similarly to the precision, we compute the overall weighted recall by Eq. (2).

$$\text{Precision} = \frac{1}{\sum_{c\in C} k_c} \sum_{c\in C} k_c P_c, \tag{1}$$

$$\text{Recall} = \frac{1}{\sum_{c\in C} k_c} \sum_{c\in C} k_c R_c. \tag{2}$$

***F*-measure.** The F-measure is the harmonic mean between the recall and precision. In class c, the F-measure is $F_c = 2 \times \frac{P_c \times R_c}{P_c + R_c}$. The overall weighted F-measure is given by

$$F\text{-measure} = \frac{1}{\sum_{c\in C} k_c} \sum_{c\in C} k_c F_c. \tag{3}$$

A high F-measure guarantees high levels of both precision and recall. Note that another popular measure, the area-under-curve AUC, was omitted because the TCQA employs a 3-class classifier, so the AUC cannot be calculated.

5 Experimental results

This section experimentally analyzes the three research questions formulated in Subsection 4.1. All models were built with random forest. Because training a random forest model is inherently non-deterministic [29], we report the averages of 30 independent training runs of each random forest model.

5.1 RQ$_1$: How accurately can TCQA assess the quality of crowdsourced test cases from their dynamic code histories?

The results were evaluated in the three scenarios described in Subsection 4.3.1.

5.1.1 *Within-task results*

The within-task scenario is "training on the data of one task, and testing on the data of that task". In this scenario, the training and testing data were not separated, and the evaluation was performed by 10-fold cross-validation. The results are presented in Table 3. We immediately observe that TCQA performed reasonably well on the data of the six tasks. The precision exceeded 0.7 on all tasks except ITClocks, and approached 0.8 on four out of six (66.67%) tasks.

Table 3 Within-task scenario results

Task	Precision	Recall	F-measure
CMD	0.77	0.83	0.78
Datalog	0.80	0.82	0.81
ITClocks	0.65	0.75	0.68
JMerkle	0.76	0.79	0.76
LunarCalendar	0.70	0.76	0.71
QuadTree	0.78	0.79	0.78

Table 4 Whole-sample scenario results

Testing task	Precision	Recall	F-measure
CMD	0.71	0.80	0.74
Datalog	0.67	0.66	0.66
ITClocks	0.70	0.74	0.71
JMerkle	0.60	0.84	0.73
LunarCalendar	0.71	0.81	0.74
QuadTree	0.71	0.80	0.72

5.1.2 *Whole-sample results*

In this scenario, a classification model in TCQA was constructed from the data of any five tasks, and the quality of the test cases was predicted on the data of the remaining subjected task. The training data of tasks with many data instances (e.g., Datalog: 649) can overwhelm those of tasks with very few data instances (e.g., CMD: 134). Following the standard technique in machine learning, we therefore weighted each instance according to the project size when building the classification model. The results are shown in Table 4.

In general, the results of the whole-sample and within-task scenarios were comparable. The precision values in the whole-sample scenario were approximately $\geqslant 0.7$. This result is particularly important for deploying TCQA in production environments, in which hundreds of requesters publish thousands of tasks. Therefore, we consider that TCQA will refine its machine-learning model by the self-growing dataset that is continuously collected from multiple tasks (which may arrive from multiple components or systems being tested). Hence, high robustness of the models is expected.

5.1.3 *Cross-task results*

The third scenario is "training on the data of one task, and testing on the data of other tasks". The cross-task scenario reveals the sensitive sensitivity of the models to different training and testing data. The results are reported in Table 5. In this table, the rows and columns represent the subjected tasks whose data were collected as the training and testing datasets, respectively. For example, a the cell [Datalog, CMD] shows the performance of the classifier trained on the data of Datalog, and tested on the data of CMD's data.

In general, the performance of TCQA was lower in the cross-task scenarios than in the other two scenarios. Therefore, caution is required when deploying TCQA in cross-task scenarios.

Based on the above results of the three experimental scenarios, we answer RQ_1 as follows. TCQA can predict the qualities of crowdsourced test cases. The accuracy error was small in most tasks. TCQA performed better on within-task and whole-sample scenarios than in the cross-task scenario. The high accuracy in the whole-sample scenario enhances our confidence that TCQA can learn the data of multiple tasks, and use them in specific task prediction.

Table 5 Average precisions of 30 runs in the cross-task scenario

Task	Testing task					
	CMD	Datalog	ITClocks	JMerkle	LunarCalendar	QuadTree
CMD	–	0.41	0.34	0.33	0.35	0.52
Datalog	0.62	–	0.61	0.63	0.64	0.60
ITClocks	0.68	0.59	–	0.60	0.65	0.59
JMerkle	0.57	0.57	0.55	–	0.57	0.60
LunarCalendar	0.60	0.58	0.62	0.62	–	0.60
QuadTree	0.58	0.57	0.56	0.60	0.59	–

Table 6 Time-cost comparison of efficiency measures (coverage metrics and mutation testing in seconds)

Task	Traditional scoring	TCQA			TCQA in production environment
		Feature extraction	Training	Prediction	(feature extraction + prediction)
CMD	763.29	29.79	0.02	0.02	29.81 (25.60x)
Datalog	1987.59	103.60	0.04	0.01	103.61 (19.18x)
ITClocks	448.57	26.77	0.02	0.01	26.78 (16.75x)
JMerkle	859.68	46.85	0.03	0.01	46.86 (18.35x)
LunarCalendar	5035.04	89.88	0.04	0.02	89.90 (56.00x)
QuadTree	982.64	46.62	0.02	0.01	46.63 (21.07x)

5.2 RQ$_2$: Can TCQA assess the quality of crowdsourced test cases more efficiently than traditional quality-assessment techniques?

Table 6 compares the time costs of TCQA evaluated by traditional quality-assessment techniques (mutation testing and execution coverage). The most time-consuming step of TCQA was feature-extraction, which ranged from 29 to 103 s (consuming and $> 99\%$ of the total time cost). Column 6, of this table simulates the deployment of TCQA in production environments. As TCQA uses an offline learning strategy, it can build predictive models off-line, ahead of assessment. Thus, when predicting a new upcoming submission, TCQA consumes both a feature-extraction time and a prediction time. When a new test case arrives for evaluation, the training time is omitted. As indicated in Table 6, TCQA outperformed the traditional quality-assessment techniques in terms of efficiency. The final column shows the magnitude of the time saving of TCQA relative to the traditional methods. TCQA executed from 16.75 to 56.00 times faster than the traditional methods.

In fact, our efficiency evaluation is very conservative. First, the performance of the traditional method as determined on only 10 mutants. In mutation-testing practice, a 1 KLOC program is often tested with over 100 mutants. Second, the six subject tasks were run by simple programs, so their execution times were relatively short. Although the time cost of a complex target program increases dramatically, TCQA does not execute the target program in the prediction phase. Its most time-consuming part (feature extraction depends only on the developmental time series of the test case, so its performance is more stable.

Hence, we answer RQ$_2$ as follows. The execution time is much lower in TCQA than in traditional quality-assessment techniques. On the six subjected tasks, TCQA ran 16.75–56.00 times faster than the traditional methods.

5.3 RQ$_3$: Can TCQA yield real-time insights on the quality of the crowdsourced test cases before they are submitted?

Figure 2 shows the dependence of the classification precision values on the data-sequence length in the six tasks. As an example, let us consider Figure 2(b), which describes the dependence of data-sequence length on the precision values of TCQA prediction. The first and second boxes show the precisions of predicting from the first 10% and 20% of the time series, respectively. The remaining eight boxes are the precision results of including 30%–100% of the time-series data at 10% intervals.

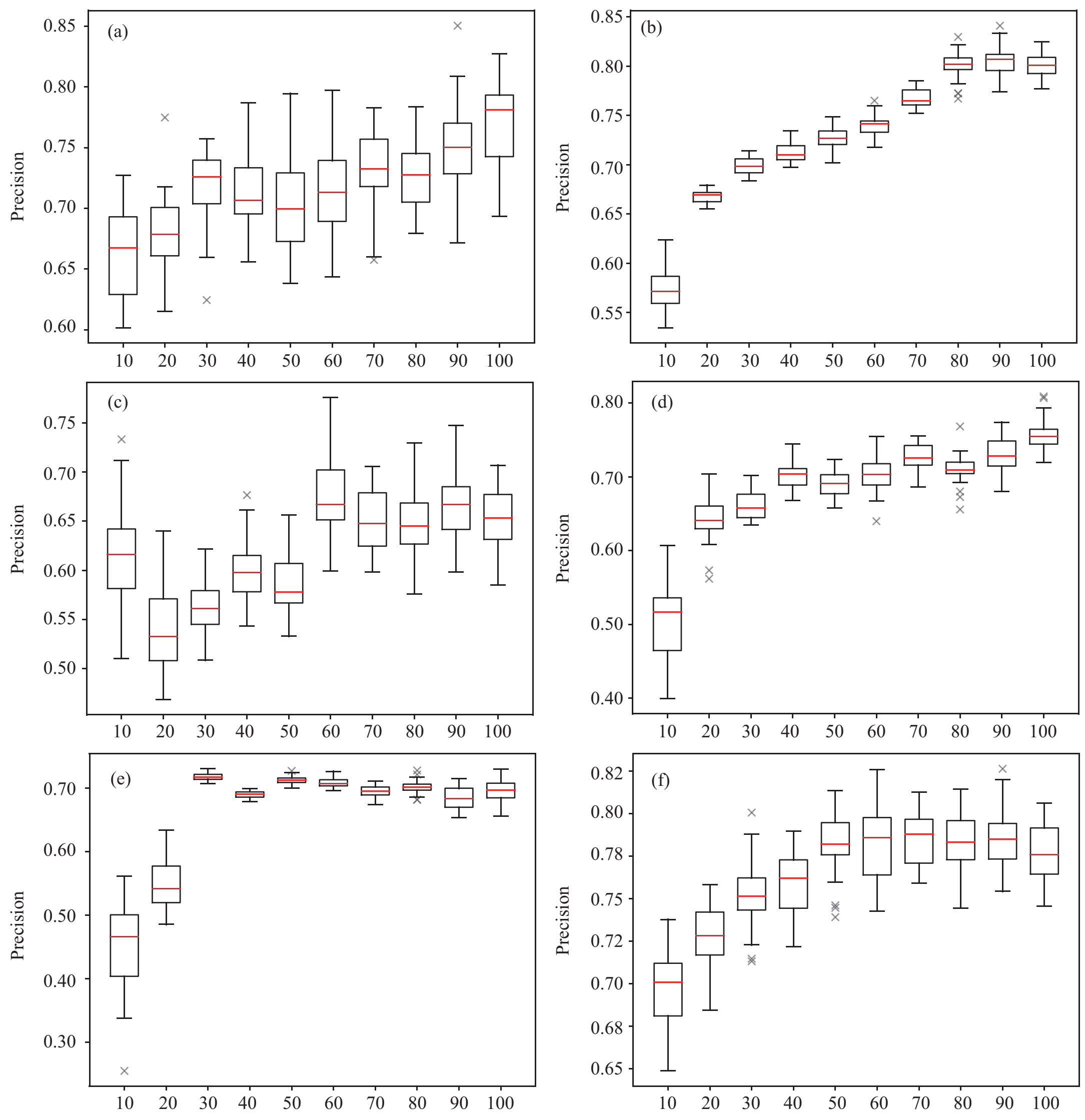

Figure 2 (Color online) Data-volume dependence on the precision performance of each task (in the within-task scenarios). (a) CMD; (b) Datalog; (c) ITClocks; (d) JMerkle; (e) LunarCalendar; (f) QuadTree.

As more of the time series was included in the prediction, the quality of the TCQA prediction generally improved. However, TCQA frequently achieved its highest performance on a partial rather than the whole time series. On average, the prediction results were maximized on 90% of the data in Figure 2(b), and were not significantly degraded on 80% of the data.

This result is not so surprising, because the end part of a time series is often flat. A crowd worker often stops editing and reads the code before submitting it. This behavior (checking the code before submission) is probably universal among the crowd workers, regardless of the quality of their test cases. The checking also covers parts that are unhelpful in classifying the test cases.

Indeed, the results of all six subjected tasks were stabilized when the data volume reached 50%–60%. This pattern, which is clarified in Table 7, evidences that TCQA can estimate the quality of a test case in real-time, before the test case has been submitted by the crowd worker developing it. In the present evaluation, TCQA reasonably predicted the quality of even a half-finished test case.

Such early prediction is a very attractive feature for crowdsourcing test-case development. A requester can exploit this property for real-time planning in microwork publishing. For instance, if a TCQA requester estimates that a crowd worker is producing a poor-quality test case, he or she can immediately reassign the microwork to another worker. The waiting time of the requester is then markedly reduced and

Table 7 The average precision value of 30 runs with the feature from last X percent time series

Task	100%	90%	80%	70%	60%	50%
CMD	0.77	0.75	0.73	0.73	0.72	0.70
Datalog	0.80	0.81	0.80	0.77	0.74	0.73
ITClocks	0.65	0.66	0.65	0.65	0.67	0.59
JMerkle	0.76	0.73	0.71	0.73	0.70	0.69
LunarCalendar	0.70	0.68	0.70	0.70	0.71	0.71
QuadTree	0.78	0.78	0.78	0.78	0.78	0.78

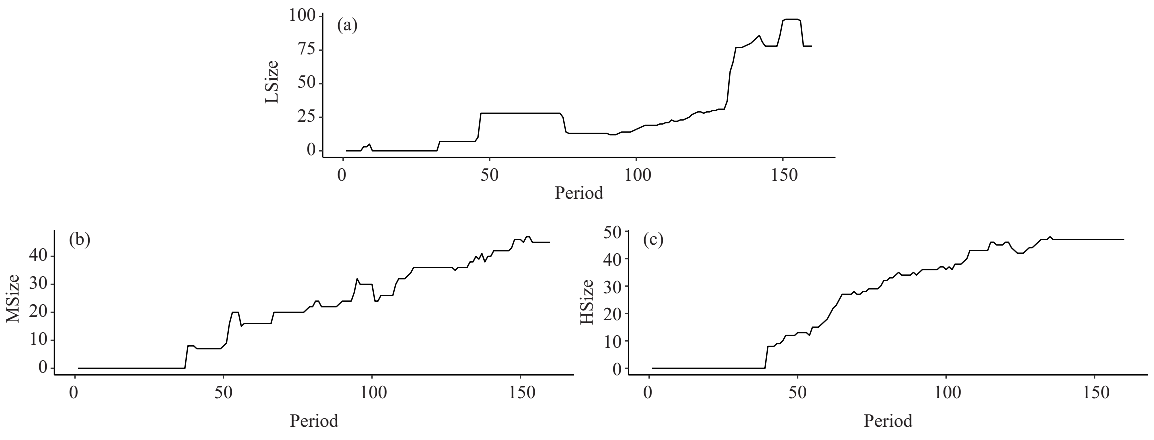

Figure 3 Representative dynamic histories of codes with different quality levels. The x and y axes represent the percentage of the development, time and the size growth of the test-case code, respectively. (a) Low quality tests; (b) medium quality tests; (c) high quality tests.

the efficiency of the crowdsourcing workflow improves. Moreover, this process can be easily automated for further optimization.

Based on the above results, we answer RQ_3 as follows. TCQA can provide real-time insights on the quality of test cases before the cases are submitted by the crowd workers. Such early prediction may help to optimize the process of crowdsourced test case development.

6 Discussion

6.1 Quality-dependent characteristics of test cases

To further understand the factors characterizing high-, medium-, and low-quality test cases, we averaged the time-series of all growing test-case data, and visualized the results. Representative time-series graphs are shown in Figure 3.

Note that in these representative time series, the test cases classified as low-quality (e.g., Figure 3(a)) exhibited multiple sudden and large size drops in their code the growth patterns. These code deletions might characterize uncertainties or mistakes in the coding of the test cases. In contrast, many of the test cases classified as high-quality (e.g., Figure 3(c)) showed a relatively smooth increase in test-code size. Moreover, when beginning the task, the competent crowd workers did not edit their code — (Figure 3(c)), probably because they were assessing and thinking about the solution. Wang et al. [9] also identified the “think first, code second” pattern in the coding behaviors of more qualified developers. Figure 3(b) is the time series of a medium-quality test case. The graph shows some of the characteristics of both high- and low-quality coding (e.g., smooth growth and size drops, respectively), but to a smaller degree. These intuitive patterns suggest that the local shape features of the time series (called shapelets, see [38]) may boost the quality of the prediction. This possibility will be explored in future work.

6.2 Implications to research and practice

Our study has significant implications for both research and practice. From a research perspective, it rediscovers the importance and value of fine-grained dynamic code history. We showed that with proper modeling and analyzing techniques, the fine-grained dynamic history of a source code embeds the quality of the code, negating the need for time-consuming mutation testing. The source code represents the final programming product, but the code history reflects the problem-solving process leading to the product. Combining these complementary results may provide a holistic picture of the programming activity of developers, enrich our knowledge, and ultimately help us to design innovative techniques and software engineering tools. We hope that more software engineering researchers, including ourselves, will continue working in this direction.

From a practice perspective, TCQA will be integrated into MoocTest's production environment and will support future harvesting activities of crowdsourced test-cases. TCQA offers several advantages over mainstream industrial solutions. First, because it requires no specific techniques for testing the frameworks (e.g., JUnit) that execute the test cases, it can support development in multiple languages and platforms. In the present experiments, the test cases were written in Java, but other languages such as Python, Ruby, and C# are equally valid. While guaranteeing high scalability, this feature also minimizes the maintenance cost. The training and predicting phases in TCQA consume very few computing resources, thereby avoiding large hardware investments. Even when applied as an auxiliary method to traditional techniques (such as mutation testing), TCQA benefits traditional techniques by excluding many test cases in advance. Moreover, the real-time insights obtained from TCQA time-series data are well suited to the context of cloud- based crowdsourcing, and can potentially optimize the development of crowdsourced test case.

6.3 Threats to validity

Internal validity. Most of the internal threats are removed by the rigorous procedures of our empirical process (Section 4) and analysis (Section 5). Our study adopts well-established and intensively validated feature extraction and classification algorithms, ensuring a high level of internal validity.

External validity. External threats pose the main challenge to the validity of the present study. TCQA requires a cloud-based architecture, in which the cloud and web-based clients communicate through `HTTP` conversations. If crowd workers write their tests in traditional IDEs, data availability is a potential problem. Although TCQA is a favourable environment for small programming tasks such as writing unit tests, its effectiveness might reduce on large programming tasks. Moreover, as its underlying rationale, TCQA assumes that the dynamic code history reflects and characterizes the problem-solving process of a single developer. Therefore, it may not capture the efforts of multiple persons collaborating on a programming task. When applying TCQA in tasks that require multi-person collaboration settings, we urge much caution because an individual's problem-solving is difficult to characterize [39]. Nevertheless, TCQA promises future research opportunities for modeling and analyzing the dynamic code histories of collective team efforts.

7 Related work

Code coverage is arguably the most widely used methodology for measuring the quality of test cases, in both academia and industry [40–43]. Several researchers have correlated various test coverage metrics with test effectiveness. Gligoric et al. [41] claimed that fault-detection effectiveness is moderately correlated with coverage in general, and also with project size. Namin & Andrews [44] investigated how the effectiveness of a test suite depends on block coverage, decision coverage, and other two data-flow criteria. In contrast, recent evidence supports more complex relationships between coverage and test quality. For example, Inozemtseva & Holmes found a low-to-moderate correlation between coverage and test effectiveness. They reported that stronger forms of coverage do not improve the data for measuring the effectiveness of a test [5]. A few other studies [45, 46] have raised similar concerns. Zhang and Mesbah [47] argued that such low correlations may result from the lack of assertion information in code coverage.

The test criterion in mutation testing is the "mutation adequacy score", which measures the fault-detection effectiveness of a test set [33]. A mutation analysis seeds artificial defects (mutations) into programs, and determines how many of these mutants are killed. If all mutations are successfully killed, the quality of the test suite is high. However, in mutation testing, many mutants must be compiled and executed on one or more test cases, placing unacceptable demands on computing resources and limiting the restricting its practical applicability of this method [48]. Accordingly, several researchers have proposed cost-reduction methods for the three stages of mutation testing (mutation generation, mutant execution, and result analysis). Among these methods, selective mutation is popular because it reduces the number of mutants [49]. A typical cost-reduction technique generates n-order mutants as a set of combined 1-order mutants e.g., [50–52]. Cost reduction can also be achieved by search-based testing. Recently, the high cost of mutation testing has been tackled from a data-science perspective. For instance, Zhang et al. [6] proposed predictive mutation testing, which classifies the mutation test results based on the mutant-related features. This approach avoids the mutant execution. Besides, measuring the diversity of the inputs is another method to assess the effectiveness of the test suites. Shi et al. [53] proposed the entropy-based metric as a diversity measure. Although this metric can indicate the effectiveness of a test suite, it is limited to the input domain and cannot evaluate the whole suite of test cases.

Unlike coverage metrics and mutation testing, TCQA utilizes the dynamic code history, opening a new dimension of test-code assessment. Our evaluations confirmed that TCQA is more efficient than state-of-the-art techniques, and uniquely provides real-time insights that were never previously unrealized in software-engineering research. Therefore, TCQA is eminently suited to the and cloud-based crowdsourcing context.

8 Conclusion and future work

Writing test codes is a difficult task. Crowdsourcing harvests a large number of tests at low cost and will collect good tests with high probability, but distinguishing the high-quality tests from the low-quality ones is a major challenge. This paper introduced our TCQA approach, which automatically assesses the quality of crowdsourced test cases harvested on cloud-based crowdsourcing platforms. Using the fine-grained dynamic history of the code, TCQA can estimate the code's quality without calculating the coverage metrics or performing mutation testing. The proposed approach was implemented and extensively evaluated on 2193 test cases collected from more than 400 participants. The evaluations confirmed that TCQA can accurately predict the quality of test cases. In a comparison with traditional quality-assessment techniques, TCQA achieved superior efficiency. In our third experiment, TCQA additionally yielded real-time insights on the qualities of the crowdsourced test cases, before those codes had been submitted by the crowd workers. In summary, TCQA is a promising technique for assessing the quality of crowdsourced test cases. Owing to its high efficiency and minimal computational overhead, TCQA is particularly attractive for cloud-based crowdsourcing software-development platforms.

We have already planned extensive follow-up work that will enhance TCQA. In the current study, TCQA assessed the overall quality of a test case with high effectiveness and efficiency. In the upcoming design and implementation iterations, we will add features that identify the local characteristics of test-case dynamics (such as a user's deletions, and source code that remains unchanged over a long period), thereby providing further insights. To achieve this goal, we will integrate additional time-series modeling and analysis techniques, such as anomaly detection [54] and shapelets [38], into TCQA. Besides the time-series related features, we plan to incorporate features related to the source code [6], which will further improve TCQA's performance. Finally, the integration of MoocTest with TCQA is ongoing and TCQA will eventually be evaluated in the production environment.

Acknowledgements This work was partly supported by National Key Research and Development Program of China

(Grant No. 2018YFB1403400) and National Natural Science Foundation of China (Grant Nos. 61690201, 61772014).

References

1 Mao K, Capra L, Harman M, et al. A survey of the use of crowdsourcing in software engineering. J Syst Softw, 2017, 126: 57–84
2 LaToza T D, Chen M, Jiang L X, et al. Borrowing from the crowd: a study of recombination in software design competitions. In: Proceedings of the 37th International Conference on Software Engineering, 2015. 551–562
3 Musson R, Richards J, Fisher D, et al. Leveraging the crowd: how 48,000 users helped improve lync performance. IEEE Softw, 2013, 30: 38–45
4 LaToza T D, Towne W B, Adriano C M, et al. Microtask programming: building software with a crowd. In: Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology, 2014. 43–54
5 Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 435–445
6 Zhang J, Wang Z Y, Zhang L M, et al. Predictive mutation testing. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016. 342–353
7 Tsai W T, Wu W, Huhns M N. Cloud-based software crowdsourcing. IEEE Int Comput, 2014, 18: 78–83
8 Park J, Park Y H, Kim S, et al. Eliph: effective visualization of code history for peer assessment in programming education. In: Proceedings of ACM Conference on Computer Supported Cooperative Work and Social Computing, 2017. 458–467
9 Wang Y, Wagstrom P, Duesterwald E, et al. New opportunities for extracting insights from cloud based IDEs. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 408–411
10 Wang Y. Characterizing developer behavior in cloud based IDEs. In: Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017. 48–57
11 Negara S, Vakilian M, Chen N, et al. Is it dangerous to use version control histories to study source code evolution? In: Proceedings of European Conference on Object-Oriented Programming, 2012. 79–103
12 LaToza T D, Myers B A. Hard-to-answer questions about code. In: Proceedings of Evaluation and Usability of Programming Languages and Tools, 2010
13 Christ M, Kempa-Liehr A W, Feindt M. Distributed and parallel time series feature extraction for industrial big data applications. 2016. ArXiv:1610.07717
14 Huhns M N, Li W, Tsai W T. Cloud-based software crowdsourcing (dagstuhl seminar 13362). Dagstuhl Rep, 2013, 3: 34–58
15 Fast E, Steffee D, Wang L, et al. Emergent, crowd-scale programming practice in the IDE. In: Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems, 2014. 2491–2500
16 Bai X Y, Li M Y, Huang X F, et al. Vee@cloud: the virtual test lab on the cloud. In: Proceeding of the 8th International Workshop on Automation of Software Test (AST), 2013. 15–18
17 Zhu H, Hall P A V, May J H R. Software unit test coverage and adequacy. ACM Comput Surv, 1997, 29: 366–427
18 Young M. Software Testing and Analysis: Process, Principles, and Techniques. Hoboken: John Wiley & Sons, 2008
19 Rojas J M, Fraser G, Arcuri A. Automated unit test generation during software development: a controlled experiment and think-aloud observations. In: Proceedings of International Symposium on Software Testing and Analysis, 2015. 338–349
20 Xiao X S, Xie T, Tillmann N, et al. Precise identification of problems for structural test generation. In: Proceedings of the 33rd International Conference on Software Engineering, 2011. 611–620
21 Almasi M M, Hemmati H, Fraser G, et al. An industrial evaluation of unit test generation: finding real faults in a financial application. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, 2017. 263–272
22 Shamshiri S, Just R, Rojas J M, et al. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. 201–211
23 Ying A T T, Murphy G C, Ng R, et al. Predicting source code changes by mining change history. IEEE Trans Softw Eng, 2004, 30: 574–586
24 Keogh E J, Pazzani M J. Scaling up dynamic time warping for datamining applications. In: Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2000. 285–289
25 Christ M, Kempa-Liehr A W, Feindt M. Distributed and parallel time series feature extraction for industrial big data applications. 2016. ArXiv:1610.07717
26 Yekutieli D, Benjamini Y. The control of the false discovery rate in multiple testing under dependency. Ann Statist, 2001, 29: 1165–1188
27 Schreiber T, Schmitz A. Discrimination power of measures for nonlinearity in a time series. Phys Rev E, 1997, 55: 5443–5447
28 Menzies T, Williams L, Zimmermann T. Perspectives on Data Science for Software Engineering. San Francisco: Morgan Kaufmann, 2016
29 Breiman L. Random forests. Mach Learn, 2001, 45: 5–32
30 Khoshgoftaar T M, Golawala M, van Hulse J. An empirical study of learning from imbalanced data using random forest. In: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence, 2007. 310–317
31 Petitjean F, Forestier G, Webb G I, et al. Faster and more accurate classification of time series by exploiting a novel

dynamic time warping averaging algorithm. Knowl Inf Syst, 2016, 47: 1–26

32 Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 435–445

33 Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng, 2011, 37: 649–678

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

35 Pan S J, Tsang I W, Kwok J T, et al. Domain adaptation via transfer component analysis. IEEE Trans Neural Netw, 2011, 22: 199–210

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

37 Sokolova M, Lapalme G. A systematic analysis of performance measures for classification tasks. Inf Process Manage, 2009, 45: 427–437

38 Ye L, Keogh E. Time series shapelets: a new primitive for data mining. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2009. 947–956

39 Rokicki M, Zerr S, Siersdorfer S. Groupsourcing: team competition designs for crowdsourcing. In: Proceedings of the 24th International Conference on World Wide Web, 2015. 906–915

40 Soffa M L, Mathur A P, Gupta N. Generating test data for branch coverage. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, 2000. 219–227

41 Gligoric M, Groce A, Zhang C, et al. Comparing non-adequate test suites using coverage criteria. In: Proceedings of International Symposium on Software Testing and Analysis, 2013. 302–313

42 Gopinath R, Jensen C, Groce A. Code coverage for suite evaluation by developers. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 72–82

43 Perry W E. Effective Methods for Software Testing: Includes Complete Guidelines, Checklists, and Templates. Hoboken: John Wiley & Sons, 2007

44 Namin A S, Andrews J H. The influence of size and coverage on test suite effectiveness. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, 2009. 57–68

45 Briand L, Pfahl D. Using simulation for assessing the real impact of test coverage on defect coverage. In: Proceedings of the 10th International Symposium on Software Reliability Engineering, 1999. 148–157

46 Cai X, Lyu M R. The effect of code coverage on fault detection under different testing profiles. SIGSOFT Softw Eng Notes, 2005, 30: 1–7

47 Zhang Y C, Mesbah A. Assertions are strongly correlated with test suite effectiveness. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, 2015. 214–224

48 Wong W E, Mathur A P. Reducing the cost of mutation testing: an empirical study. J Syst Softw, 1995, 31: 185–196

49 Offutt A J, Lee A, Rothermel G, et al. An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol, 1996, 5: 99–118

50 Polo M, Piattini M, García-Rodríguez I. Decreasing the cost of mutation testing with second-order mutants. Softw Test Verif Reliab, 2009, 19: 111–131

51 Jia Y, Harman M. Higher order mutation testing. Inf Softw Tech, 2009, 51: 1379–1393

52 Harman M, Jia Y, Langdon W B. Strong higher order mutation-based test data generation. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011. 212–222

53 Shi Q K, Chen Z Y, Fang C R, et al. Measuring the diversity of a test set with distance entropy. IEEE Trans Rel, 2016, 65: 19–27

54 Chandola V, Banerjee A, Kumar V. Anomaly detection: a survey. ACM Comput Surv, 2009, 41: 1–58