IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 17, 2019, accepted September 27, 2019, date of publication October 21, 2019, date of current version October 31, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2948640

Amelioration of Teaching Strategies by Exploring Code Quality and Submission Behavior

YU BAI, TAO WANG, AND HUAIMIN WANG
Science and Technology on Parallel and Distributed Laboratory, National University of Defense Technology, Changsha 410073, China
Key Laboratory of Software Engineering for Complex Systems, National University of Defense Technology, Changsha 410073, China

Corresponding author: Tao Wang (taowang2005@nudt.edu.cn)

This work was supported by the National Key Research and Development Program of China under Grant 2018YFB1003903.

ABSTRACT Online learning platforms provide an opportunity to better understand students' weaknesses by tracking both their learning behavior and knowledge. This information can then be used to assist teachers in making instructional decisions and to further guide those who are at risk of failure. In this paper, we tracked student learning data from a C++ programming course over a whole semester of their freshman year via the Trustie platform. A total of 17,854 code submissions were collected. We then used CppCheck, SonarQube and Trustie to capture the quality characteristics and submission characteristics of the code, including lineOfCode, cyclomaticComplexity, codeSmell, syntacticError, averageScore, submission, and logicError, and analyzed the impact of code quality on the assignment work results. Several factors were discovered that we believe can help teachers to develop more effective teaching strategies.

INDEX TERMS Engineering students, code quality, programming course, teaching strategy.

I. INTRODUCTION

Computer science (CS) is a popular college course requiring fundamental practical abilities, such as programming. There are many e-learning platforms that offer students more practical opportunities in order to help them achieve better results, such as CodingBat [1], CloudCoder [2], [3], and Trustie [4], [5].

Programming is a challenging practice for many beginners, particularly the process of determining the meaning of compiler error messages, and they often cannot overcome syntactic and semantic errors [6]–[8]. Moreover, logical errors can present an even greater challenge for novices [9]–[12]. Past research in this area is mainly based on surveys and interviews due to difficulties in data collection. However, with the development of technology and the diversification of educational resources, corresponding tools and platforms have also been enriched. Data collection is now more automated, and analysis of these data would help researchers to understand the shortcomings of students. Commonly mentioned platforms are WebCAT [13], [14] , BlueJ [15], [16], and BlueJ extensions [17], CodeWrite [18] and its plugins, and Athene [19]. For example, Brown *et al.* [20] described how Blackbox is used to collect information submitted by students in the BlueJ environment and store it in a central repository. Keuning *et al.* [21] analyzed code snapshots written by novice programmers using BlueJ IDE and found that there were a considerable number of code quality issues, and they do not seem to get fixed, especially when they are related to modularization. Tabanao et al. 's work [22] focused on syntactic errors. They collected compilation errors from novice programmers and examined the relationship between these errors and student course performance. Then, they built a linear regression model to predict students' scores on a midterm exam, but the predictions were not sufficiently accurate. However, BlueJ is an integrated development environment (IDE) for Java programming languages only. Such limitations on language type thus narrow the scope of observations. More noteworthy is that there is no distinction between user levels in BlueJ. The program was likely written by a novice or more experienced developer. Norris et al. used ClockIt to inspect the behavior of novice programmers in introductory programming courses, typically known as CS1 and CS2 [23]. They found that the performance of students in projects seems to affect the percentage and type of compilation errors. However, this specific type of error was not analyzed. Pettit *et al.* [19] analyzed the quality of code written by novice programmers from a code style perspective and mainly focused on the complexity of the program, including the source lines of code, the cyclomatic complexity,

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

the state space (number of unique variables), and the six Halstead (Vocabulary, Length, Computed Length, Volume, Difficulty, and Effort). This work also did not analyze the specific types of errors.

There seems to be great potential to understand students' learning through analysis of their coding behavior, but most works focus on students' compilation behavior [24]. There is still plenty of scope to explore the correlation between programming behavior and learning outcomes [25]. The exploration of code quality can break the limitation that the compiler can only detect errors in the code, so if it can be combined with the analysis of compilation behavior, students' programming behavior can be more comprehensively understood. Quantifying code quality and compilation behavior will help reveal the rules of programming behavior and better help teachers improve their strategies. Therefore, the contribution of this paper is the analysis of student programming behavior from the perspective of code quality the determination of why beginners often struggle to improve their programming abilities. We examine the practice data in CS1 at the NUDT (National University of Defense Technology, China), a programming-based course for first-year students using the C++ programming language. More specifically, we collected 17,854 code submissions from CS1 from the Trustie platform. SonarQube was then used to perform quality checks on each code submission, which were ultimately analyzed.

The rest of the paper is structured as follows. In Section 2, we present detailed information on the Trustie platform and the code quality characteristics on the SonarQube. We also introduce the data characteristics and data collection in this section. In Sec. 3, we analyze the results of the SonarQube analysis and the submitted behavioral data and determine which features highly affect the pass rate. Based on the analysis in Sec. 3, we provide suggestions for formulating teaching strategies in Sec. 4. These suggestions aim to improve student pass rates, as well as increase self-confidence in the early stages of learning. Finally, in Sec. 5, we summarize the main conclusions, present the limitations of this paper, and propose a future research agenda.

II. MATERIALS AND METHODS

A. E-LEARNING PLATFORM

1) TRUSTIE

TRUSTIE[1] (**Trust**worthy Software tools and **I**ntegration **E**nvironment) is an e-learning platform that not only provides hosting services for school courses but also includes two systems, a container-based online programming system and a Git-based version control system. Thus, Trustie can record both students' learning behavior and their submitted code. It supports several programming languages, including C/C++, Python and Java.

Teachers can create or select assignments from the exercise repository in Trustie and provide them to students as an online coding practice. To create such assignments, teachers must first build a test set for each exercise to verify the logical correctness of the students' programs in the future. Students can code anytime, anywhere, so long as it is before the deadline. Trustie subsequently collects the data generated during the students' practice, as shown in Fig. 1.

[1]https://www.trustie.net

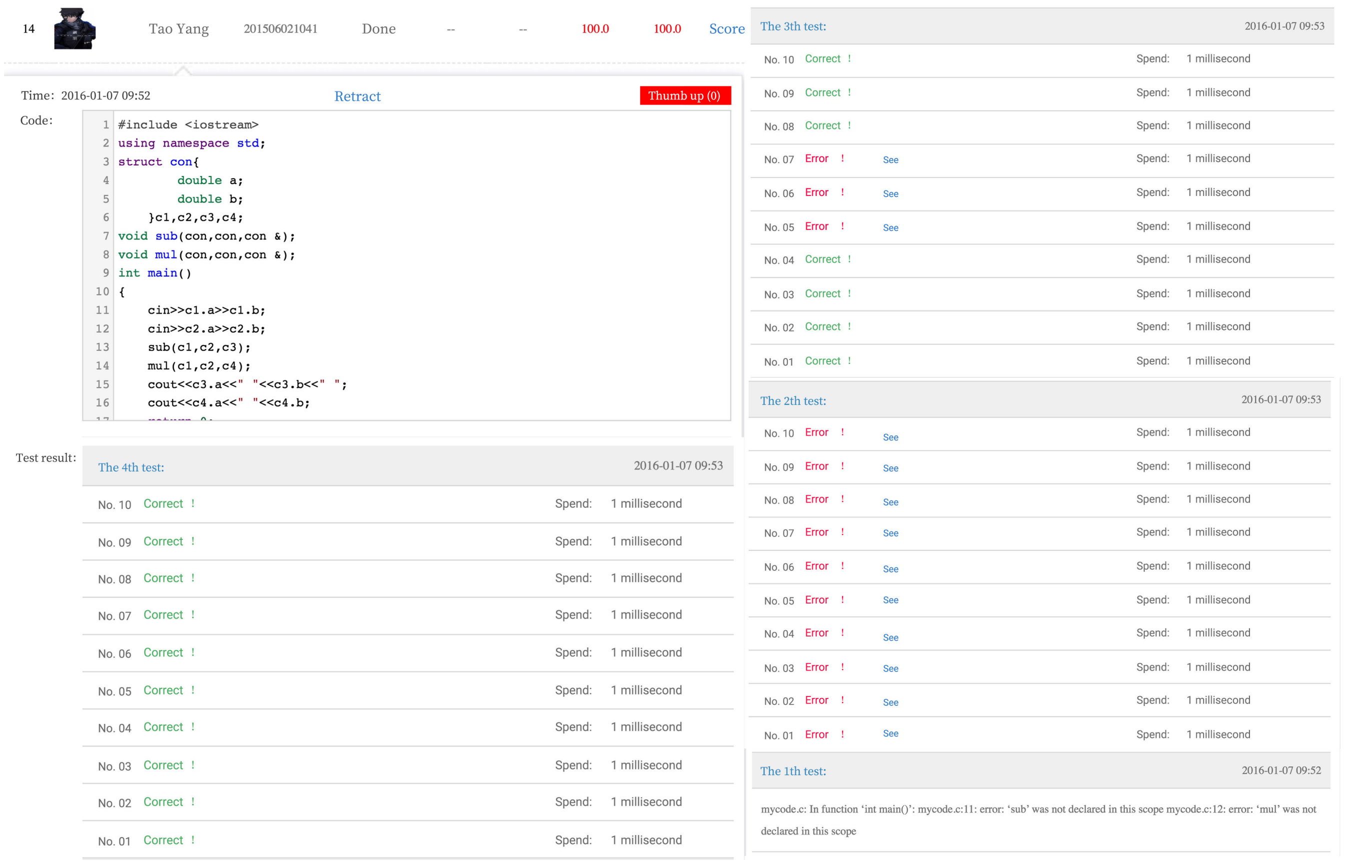
2) SONARQUBE

SonarQube is an open-source platform used for the continuous checking of code quality, with its main detection rules provided by various language plugins. Thus far, it has been able to detect up to 25 programming languages, including Java, C/C++, Objective-C, C, PHP, Flex, Groovy, JavaScript, Python, PL/SQL, COBOL, and Swift. The platform is flexible enough to integrate various test, code analysis, and continuous integration tools. Because the Trustie platform can host both the developer's project and the various courses in the university, the diversity of programming languages is the first thing researchers need to consider, and SonarQube perfectly meets our needs. More importantly, the severity of the quality rules in SonarQube can be adjusted to meet demand. In our scenario, teachers can raise the severity level of rules that students often violate and cultivate students' good code style at the beginning of learning. The version of SonarQube used in this article is 5.6.1.

SonarQube can detect three types of issues: bugs, security vulnerabilities, and code smells. Specifically, **bugs** can track code that is evidently incorrect or highly likely to yield unexpected behavior. **Vulnerabilities** identify code that is potentially vulnerable to the exploitation of hackers. **Code smells** can confuse a developer or maintainer such that they may inadvertently introduce bugs. A code smell is a concept in programming that is also known as a bad smell. Any bad symptoms in the source code may indicate a deeper problem, such as long methods, cyclomatic complexity and duplicated code. Thus, it is important for novice programmers to develop a good programming style, which will lay a solid foundation for writing more complex programs in the future. A good programming style usually means that the code is easy to read and understand and that the code is well organized.

B. PROGRAMMING COURSE PROFILE

The students' programming data for the CS1 programming course at the University of Defense Technology (NUDT) was recorded for analysis. All exercises completed by the students were released through the Trustie platform and required the students to code online. For each exercise, the teacher must provide a set of test cases to evaluate the correctness of the submitted code. The automated scoring system will score the submitted program based on the test cases. When a submitted program passes all test cases, the automatic evaluation system will give it full marks. Students can try to submit multiple times and will always obtain feedback from Trustie's test system. The CS1 programming course at the NUDT is a programming-based course for first-year students using the C++ programming language. Each year, approximately



14 Tao Yang 201506021041 Done -- -- 100.0 100.0 Score

Time: 2016-01-07 09:52 Retract Thumb up (0)

Code:

```
#include <iostream>
using namespace std;
struct con{
        double a;
        double b;
    }c1,c2,c3,c4;
void sub(con,con,con &);
void mul(con,con,con &);
int main()
{
    cin>>c1.a>>c1.b;
    cin>>c2.a>>c2.b;
    sub(c1,c2,c3);
    mul(c1,c2,c4);
    cout<<c3.a<<" "<<c3.b<<" ";
    cout<<c4.a<<" "<<c4.b;
```

Test result:

```
The 4th test:                                   2016-01-07 09:53
No. 10  Correct !                     Spend:  1 millisecond
No. 09  Correct !                     Spend:  1 millisecond
No. 08  Correct !                     Spend:  1 millisecond
No. 07  Correct !                     Spend:  1 millisecond
No. 06  Correct !                     Spend:  1 millisecond
No. 05  Correct !                     Spend:  1 millisecond
No. 04  Correct !                     Spend:  1 millisecond
No. 03  Correct !                     Spend:  1 millisecond
No. 02  Correct !                     Spend:  1 millisecond
No. 01  Correct !                     Spend:  1 millisecond

The 3th test:                                   2016-01-07 09:53
No. 10  Correct !                     Spend:  1 millisecond
No. 09  Correct !                     Spend:  1 millisecond
No. 08  Correct !                     Spend:  1 millisecond
No. 07  Error   !   See               Spend:  1 millisecond
No. 06  Error   !   See               Spend:  1 millisecond
No. 05  Error   !   See               Spend:  1 millisecond
No. 04  Correct !                     Spend:  1 millisecond
No. 03  Correct !                     Spend:  1 millisecond
No. 02  Correct !                     Spend:  1 millisecond
No. 01  Correct !                     Spend:  1 millisecond

The 2th test:                                   2016-01-07 09:53
No. 10  Error   !   See               Spend:  1 millisecond
No. 09  Error   !   See               Spend:  1 millisecond
No. 08  Error   !   See               Spend:  1 millisecond
No. 07  Error   !   See               Spend:  1 millisecond
No. 06  Error   !   See               Spend:  1 millisecond
No. 05  Error   !   See               Spend:  1 millisecond
No. 04  Error   !   See               Spend:  1 millisecond
No. 03  Error   !   See               Spend:  1 millisecond
No. 02  Error   !   See               Spend:  1 millisecond
No. 01  Error   !   See               Spend:  1 millisecond

The 1th test:                                   2016-01-07 09:52
mycode.c: In function 'int main()': mycode.c:11: error: 'sub' was not declared in this scope mycode.c:12: error: 'mul' was not
declared in this scope
```

FIGURE 1. Trustie online programming system. Students can program in the editing area at the top left and run it online after editing. The platform will return the results of each test.

120 first-year computer science students take this course. The objectives of this course are as follows. After completing the course, the students should be able to: 1) correctly use the syntax of the C++ programming language; 2) correctly apply programming skills to solve problems; and 3) identify a good programming style.

C. DATA CHARACTERISTICS

The experimental data used in this work are described as follows.

1) CODE SUBMISSIONS

After programming in the editing area, students submit their program and wait for the results. Regardless of whether the code can be compiled or tested, the database records each submission and submits the behavior. Students can submit as many times as they wish until the deadline. In the following content, the code submission is expressed as the "submission".

2) TEST RESULTS

Following the code submission by the students, the code will be compiled and tested on the server. If the code is successfully compiled, no syntactic error is observed in the code. If the code passes all of the test cases, it has logically taken into account all of the circumstances; otherwise, the system will return a hint for compilation errors or a failed test set. The database will store all submitted code and the feedback from the platform. In the following content, syntactic error and logic error are expressed as "syntacticError" and "logicError."

3) SCORES

Students must pass all test cases to obtain full marks. If the output does not match the correct answer, the score will decrease accordingly. If a program fails to compile, a score of 0 will be given. In the following content, the average score for each assignment is expressed as the "averageScore."

4) QUALITY FEATURES

Following the analysis, the quality features of each submission will be recorded in the SonarQube database. Such features include LOC (lines of code), error, code smell, and cyclomatic complexity. In the following content, lines of code, cyclomatic complexity and code smell are expressed as "lineOfCode," "cyclomaticComplexity" and "codeSmell".

The data collection process is shown in Fig. 2. Note that, although SonarQube is an open-source platform, the C/C++

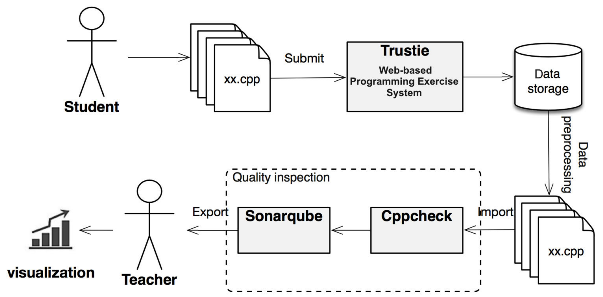


FIGURE 2. The data collection process. Students submit code to Trustie, which checks the correctness of the program in the background and stores the program and corresponding test results in the database. The entire.csv file that recorded all of the students' code was then extracted from the Trustie database and split into multiple.cpp files. CppCheck will perform quality checks on the program according to its own quality rules and then send the results to SonarQube. SonarQube will calculate the results and provide additional analysis of LOC and cyclomatic complexity.

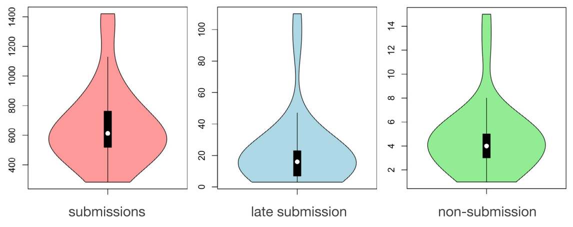
language plugin is not free. Therefore, CppCheck is selected for use. It is a free code quality checker for C/C++, not an official plugin. To prepare the data, the entire .csv file was extracted from the Trustie database, which records all of the students' code and splits it into multiple .cpp files. A quality check is then performed on these files.

III. RESULTS

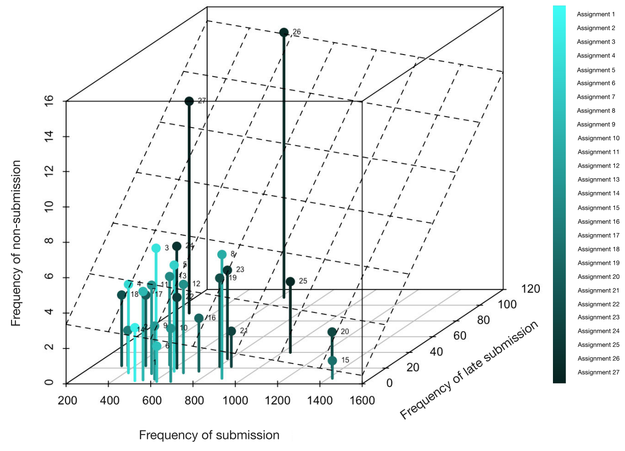
In the following sections, CppCheck and SonarQube are used to check the code quality of the raw data retrieved from Trustie.

To analyze the reasons for the late submission and nonsubmission of assignments, the relationship between the frequency of submissions, late submissions, and non-submissions was evaluated, and the results are shown in Fig. 3. From the distribution of the data in Fig. 3(a), the frequency of late submission and nonsubmission is much lower than that of submission, indicating that the vast majority of students were able to submit their code on time. Outliers are present in all three groups at the top of the violin plots. It is assumed that, the more submissions there are, the more challenging the exercise is. Thus, we assert that the existence of more submissions for an assignment is associated with a higher frequency of late submissions and nonsubmissions. Therefore, it is predicted that the outliers appearing in the three groups belong to the same assignment. However, the data suggest otherwise.

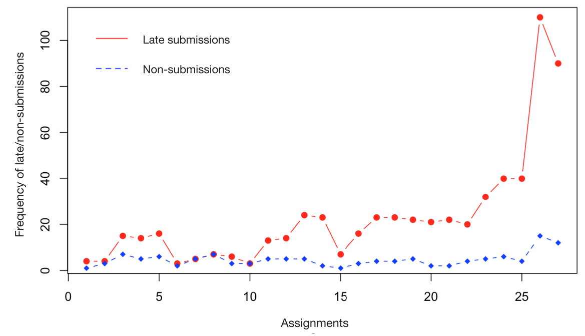
As shown in Fig. 3(b), the frequency of submission for assignments 15 and 20 is much higher than that of other assignments. However, the frequency of late submission and nonsubmission is much lower than the average. Alternately, the frequency of late submission and nonsubmission for assignments 26 and 27 is significantly higher than that for other assignments. Surprisingly, the frequency of submission for the former is close to the median of all assignments, and the frequency of submission for the latter is the smallest among all assignments. Data trends were analyzed to identify the causes for this observation, as shown in Fig. 3(c).



(a) Data density distribution for the code submission, late submission and nonsubmission.



(b) Comparison of three dimensions (submission, late submission and nonsubmission) for each assignment's work



(c) Comparison of late submission and nonsubmission trends.

FIGURE 3. The relationship between the frequency of code submission, late submission, and nonsubmission of assignments. We counted their frequency for each of the 27 assignments to explore their trends and the main reasons for late submissions and nonsubmissions.

We found that the frequency of late submission increased continuously from assignment 23 and peaked at assignment 26, while the frequency of nonsubmission was relatively stable and peaked at assignment 26. After further analysis, it was found that assignments 26 and 27 were released and terminated at the same time towards the end of the semester. This demonstrates that code submission before the deadline is related to workload rather than difficulty of the assignment. We then counted the frequency of syntactic errors, logic errors, and code smells in the submissions. Here, a syntax error refers to a violation of the syntax rules of the programming language. Syntax errors are fatal because the

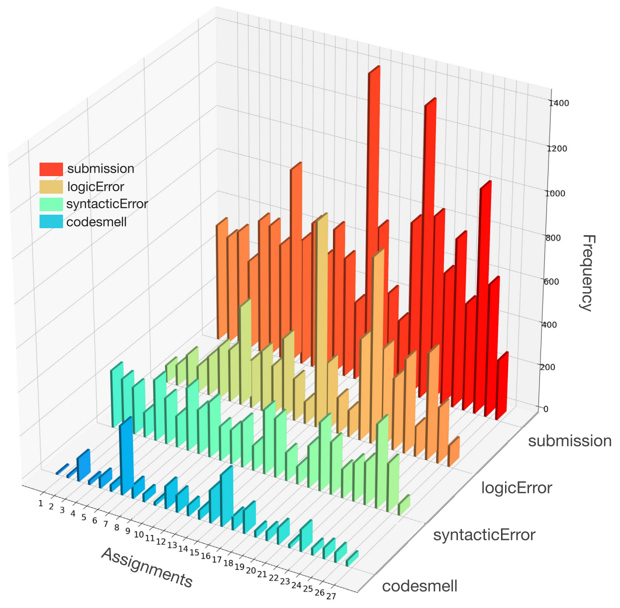


FIGURE 4. A trend comparison chart of syntactic errors, logical errors and code smells in 27 assignments. The red, yellow, green, and blue bars represent submission, logicError, syntacticError, and codesmell, respectively.

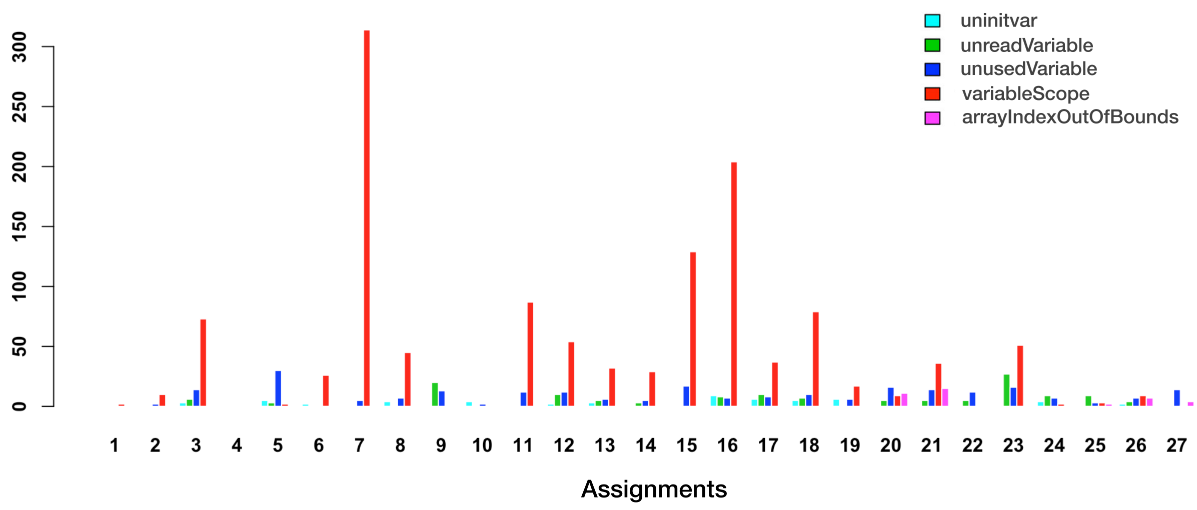
compiler will not know what the coder means. One syntax error is enough to prevent the compilation of the program into executable code. Logical errors refer to violations of the meaning rules of the programming language, including incorrect modeling of objects. In most cases, the compiler is not able to identify logical errors because the code follows the syntax rules. Identifying logical errors can be tricky because it requires the programmer to debug by checking the program's output to determine exactly what has gone wrong. Code smell is used to describe bad programming styles in SonarQube. In the short term, a bad programming style will not affect the execution of a program, but it poses potential risks to the code. However, the compiler will not provide an alarm for a code smell, and it will thus not affect the correctness of the output if no error is produced. In Fig. 4, trends of the syntactic errors, logical errors and code smells are presented to explore which training method has a far-reaching and positive effect on the learners. As seen, the frequency of syntactic error in the first five assignments exceeds the frequency of logical error. This implies that most errors appearing at the beginning of the learning period are due to unfamiliarity with syntax. However, the most obvious trend is that, while logical errors increase significantly as the assignments become more challenging, the frequency of syntactic error remains constant. The data also confirm that logical errors are undoubtedly the most common type of error, and they are also the most difficult to identify and correct. When a student triggers a syntax error, they can quickly resolve the problem if they can follow the compiler's instructions to find the error. Logical errors, however, are influenced by way of logical thinking and thus cannot be corrected quickly [12], [26].

Additionally, contrary to our expectations, the frequency of code smell does not exhibit a clear trend as the assignments became more challenging. This may be limited by file size. Empirical validation by Kessentin and Nascimento found a strong correlation between code smell and bugs when the code smell categories were BlobClass (a class that contains almost all of the functionality in a given application) and FeatureEnvy (a function or method that is more interested in the data of other classes and modules) [27], [28]. However, few of the 27 assignments surveyed involved class-level programming.

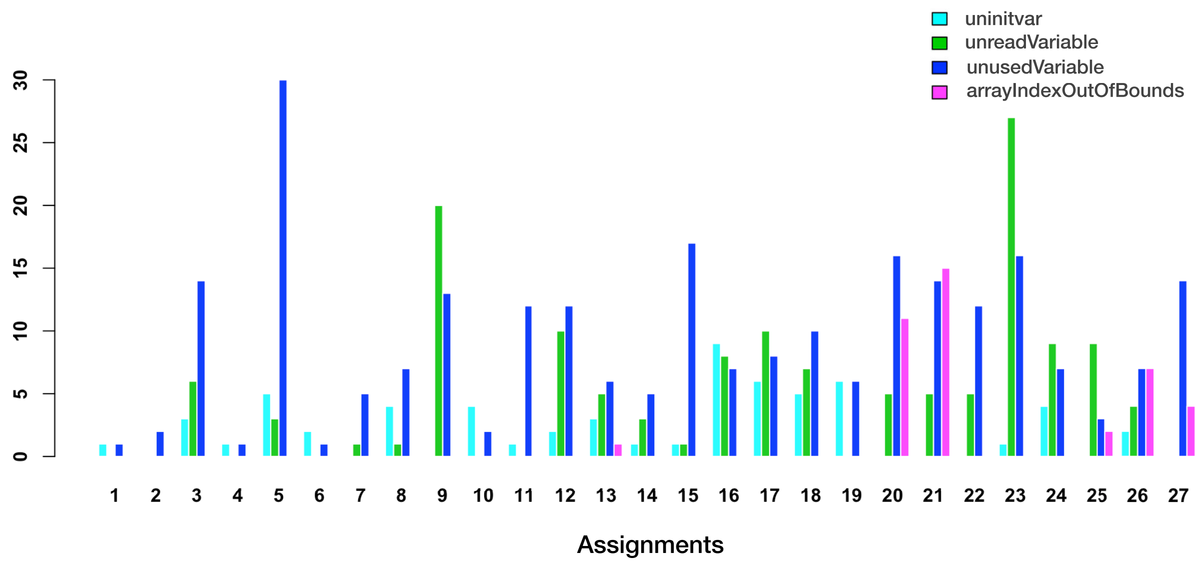
As shown in Fig. 5, in addition to syntactic and logical errors, there are five common rules violated by students that do not cause compilation or run-time errors. These are "uninitvar," "unreadVariable," "unusedVariable," "variableScope," and "arrayIndexOutOfBounds." Three of these rules are programming style issues that are part of the code smell ("unreadVariable," "unusedVariable," and "variableScope"). The remaining two, "uninitvar" and "arrayIndexOutOfBounds," are semantic errors. The presence of semantic errors will lead to unexpected results, which is different from a style problem. However, there were exceptions. For example, uninitializing global variables (or static variables) does not change the results of the program because they are automatically initialized once executed. However, forgetting the initialization of local variables may create unexpected results. An out-of-bounds array may not prevent us from obtaining the correct results when it does not overwrite essential data. Notably, we only calculated the frequency at which the program obtained correct results but violated these two semantic rules. This is different from the logical error that caused a true error in the previous analysis.

"variableScope" in Cppcheck is described as "the scope of variable 'xxx' can be reduced." As shown in Fig. 5(a), the frequency of "variableScope" is much higher than that of the other four rules, and this trend becomes more obvious in the early and middle term. It can be seen that students prefer to use global variables rather than local variables in early and mid-term assignments. Unused variables are considered inoperable code and should be removed for more efficient maintenance. As shown in Fig. 5(b), the frequency of "unusedVariable" is relatively high compared to that of the other three rules and is also relatively stable. In addition to the syntactic and logical errors, there are many hidden risks in the student code, even when the file size is small. Although code smell is not an error, it still has rules to follow. Violation of these rules can be easily avoided if students are told early on to pay more attention to programming style.

In addition to the four features provided by Trustie (averageScore, submissions, logicError and syntacticError), three additional features are available through SonarQube: lineOfCode, cyclomaticComplexity and codeSmell. To determine whether these features are related and whether they can aid students in passing the exams, it is necessary to analyze the correlation between them. For this, we must first decide which method to use for the correlation analysis.



(a) Violation rate of "variableScope" and the remaining four rules, which are uninitvar, unreadVariable, unusedVariable, and arrayIndexOutOfBound.



(b) Violation rate of "unusedVariable" and the other three rules, which are uninitvar, unreadVariable, and arrayIndexOutOfBound.

FIGURE 5. The five most common rules violated by students. The cyan, green, blue, red, and purple bars represent uninitvar, unreadVariable, unusedVariable, variableScope, and arrayIndexOutOfBound, respectively.

Many coefficients are applied to measure the correlation between variables. Among them, the three most popular coefficients are the Pearson coefficient (r), Spearman's rho coefficient (r_s) and Kendall's tau coefficient (τ) [29].

To confirm whether the Pearson coefficient can be used to measure the correlation between any two variables, it is necessary to first confirm: (1) whether the data distribution of the variables conforms to the bivariate normal distribution; and (2) whether there are outliers in the variables [30]. If the data have a multivariate normal distribution, then each variable should have a univariate normal distribution; however, if each variable has a univariate normal distribution, it cannot be ascertained that the data have a multivariate normal distribution [31]–[33]. Thus, the data distribution for each of the features must first be checked to diagnose whether a deviation is observed. It can be seen from Fig. 6 that the features codeSmell, submission and logicError have right-skewed distributions, while the feature averageScore has a left-skewed distribution. Therefore, it is unlikely that the seven features have a bivariate normal joint distribution.

Box plots were then generated to explore the outliers of the features. As seen in Fig. 7, there are clearly many outliers in the features averageScore, codeSmell, submission and logicError. Thus, given that our data violates both assumptions (bivariate normal distribution and no outliers), the Pearson coefficient is excluded and thus not used to measure the correlation between features.

TABLE 1. The ties found in the data of the seven features. Among them, averageScore, logicError and syntacticError have 2 ties each, and codeSmell has 4 ties.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	15	16	17	18	19	20	21	22	23	24	25	26	27	
averageScore	96.54	94.08	95.94	95.52	95.07	97.95	95.65	97.11	92.96	97.27	95.13	94.43	96.53	95.65
	93.54	91.47	97.11	96.64	94.22	97.81	96.13	96.1	97.65	96.12	97.32	96.61	95.44	
lineOfCode	1891	2065	5310	2959	4694	2970	2723	2572	3094	3426	4382	3201	3039	2909
	4804	5609	3157	3675	3584	4590	5831	5068	5743	3680	3900	4341	4069	
cyclomaticComplexity	149	125	787	582	1306	460	393	338	1364	847	679	384	361	383
	1070	1300	434	515	676	865	1323	1112	1324	575	678	837	367	
codeSmell	7	12	107	19	51	32	320	61	38	16	105	80	48	42
	151	237	63	107	31	45	76	18	102	34	51	48	26	
submissions	546	503	542	417	622	612	536	898	585	678	532	677	556	364
	1420	739	449	335	804	1341	859	613	785	501	1038	619	282	
logicError	82	98	161	122	182	242	244	461	224	282	228	371	197	103
	950	315	169	124	472	859	456	335	439	140	497	259	98	
syntacticError	269	244	220	117	284	215	146	293	206	255	159	158	211	117
	298	274	132	83	194	318	243	134	178	197	388	224	52	

TABLE 2. A feature-pair correlation matrix with the significance levels. The p-value of the feature pair (codeSmell, logicError) is less than 0.05, while the p-values of the feature pairs (lineOfCode, cyclomaticComplexity), (submissions, logicError), (submissions, syntacticError) and (logicError, syntacticError) are less than 0.01.

	averageScore	lineOfCode	cyclomaticComplexity	codeSmell	submissions	logicError	syntacticError
averageScore		0.7182	0.4954	0.3843	0.5609	0.3757	0.5480
lineOfCode			0.0002	0.1282	0.8006	0.8366	0.7341
cyclomaticComplexity				0.5702	0.6000	0.9357	0.7593
codeSmell					0.0888	0.0373	0.2377
submissions						0.0000	0.0000
logicError							0.0001
syntacticError							
significance level	$p < 0.01$	$p < 0.05$					
hypothesis	H_0: there is no correlation between two variables(null hypothesis)						

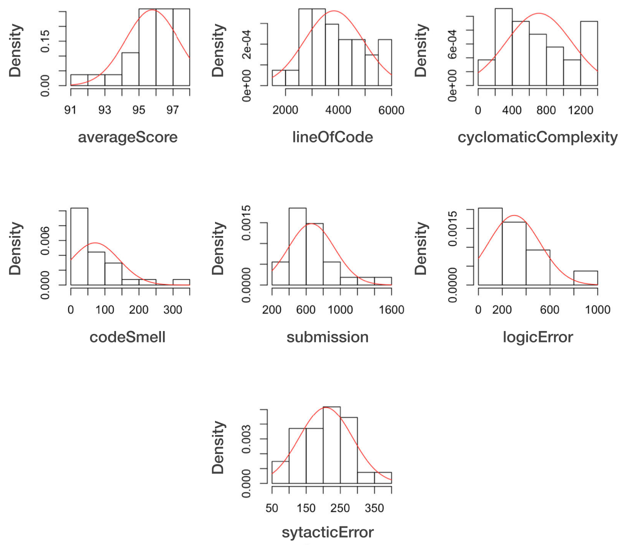


FIGURE 6. The univariate distribution of each feature. The features codeSmell, submission and logicError have right-skewed distributions, while the feature averageScore has a left-skewed distribution.

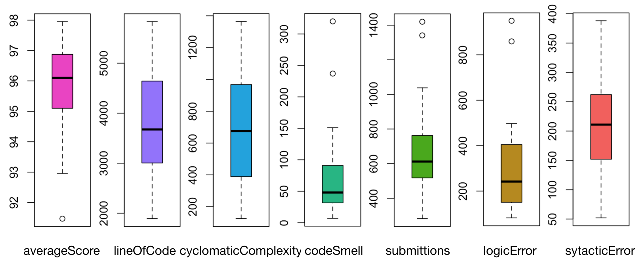


FIGURE 7. Outliers in the averageScore, codeSmell, submission, and logicError features.

According to the literature, if ties exist in the data, regardless of whether the proportion of the ties is large or small, the Spearman method is superior to the Kendall method [34]. Thus, we investigated and confirmed the existence of ties in the data. All the ties are highlighted in Table 1. This result implies that the Spearman method is more suitable for our data for measuring the correlation between features.

To determine whether and how the seven features are related, we remove the influence of other control variables and perform partial Spearman's rank correlation on the feature pairs. Fig. 8 presents the Spearman partial correlation of the seven features. Furthermore, we run statistical significance tests to confirm whether our data can represent a correlation in the population, with the result shows in Table 2. A $p-value < 0.05$ demonstrates significant correlation, and a $p-value < 0.01$ demonstrates very significant correlation.

As seen in Table 2, we can trust that the correlation of these feature pairs is statistically significant, while the correlation

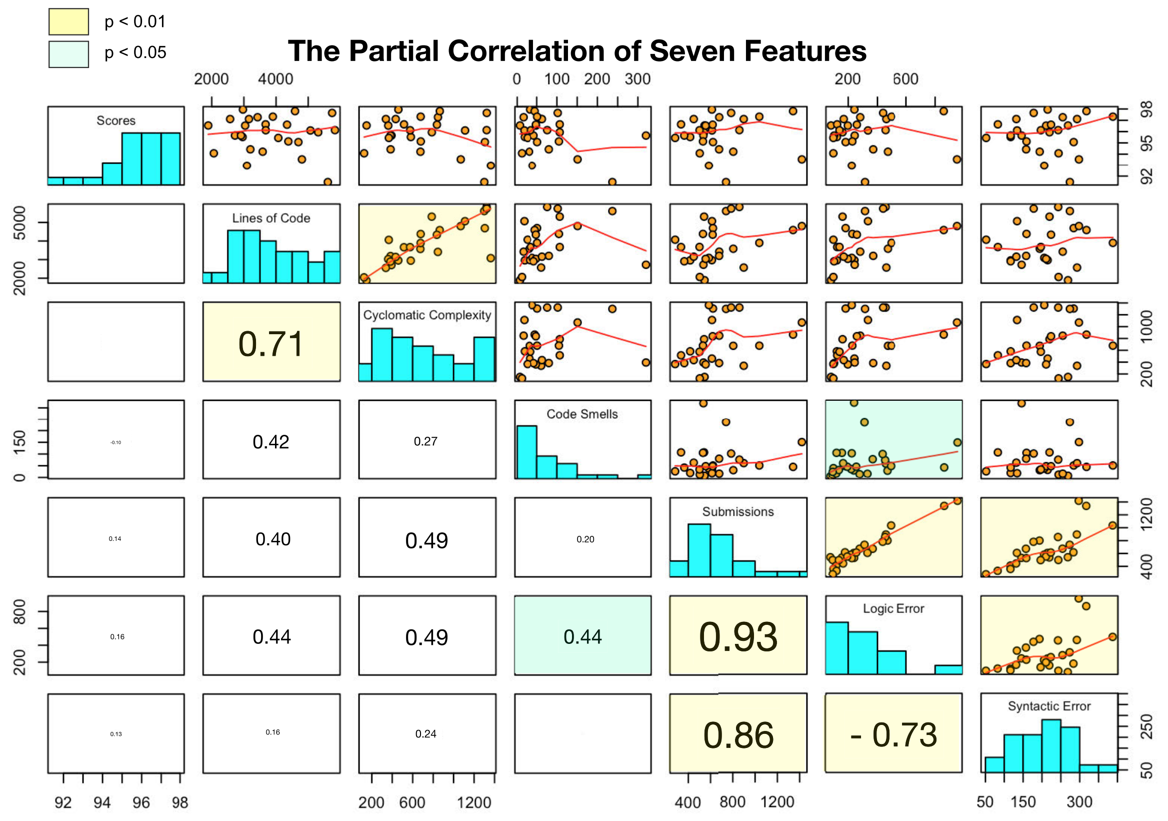


FIGURE 8. **The partial correlations of the seven features. The decimal number in the lower left triangle presents the Spearman partial correlation coefficients of the paired variables, and trend lines on the scatter plots were used to obtain an intuitive visual perception of the correlation. In addition, feature pairs with a $p - value < 0.05$ are marked in green, and feature pairs with a $p - value < 0.01$ are marked in yellow.**

coefficients of the remaining feature pairs are considered accidental, regardless of effect size.

Again, we use the same color codes in Table 2 to mark the feature pairs that demonstrate a significant correlation in Fig. 8. We observe the following:

(1) The value of the Spearman partial correlation coefficient is 0.93, proving that there is an extremely strong positive correlation between the feature pair (submissions, logicError). This result also confirms our previous analysis, that is, logical errors are the main type of error that determines whether a student passes the test.

(2) The Spearman partial correlation coefficient for the feature pair (lineOfCode, cyclomaticComplexity) is 0.71, demonstrating a strong positive correlation. Although earlier studies reported a strong correlation between the lines of code and cyclomatic complexity, Landman et al. observed different results when applying cyclomatic complexity at the method level instead of the project level. More specifically, there is no strong linear correlation between these two variables [35]. In our case, none of the 27 assignments reached the project level, while the source lines of code demonstrate a strong positive correlation with cyclomatic complexity. For beginners, solely aiming to pass the test without any effort to reduce cyclomatic complexity is not advisable. In addition, a Spearman partial correlation coefficient value of 0.86 suggests a strong positive correlation between the features "submission" and "syntacticError". The feature "syntacticError" also directly affects the frequency of submission by students, yet it is not the main reason preventing students from passing the test.

(3) Interestingly, the partial correlation coefficient between "logicError" and "syntacticError" is −0.73, which means they are strongly negatively correlated, yet the scatter plot is positively correlated. According to the scatter plot of this feature pair, the distribution of its points is almost equal to that of the feature pair (submissions, syntacticError), which indicates that the "submission" feature seriously affects the relationship of the feature pair (logicError, syntacticError). To clearly see the true correlation of the feature pair (logicError, syntacticError), we performed regression analysis to calculate the residuals of (submissions, logicError) and (submissions, syntacticError), as shown in Table 3. We denote the residual of (submissions, logicError) as RES_{sl}, and the residual of (submissions, syntacticError) as RES_{ss}. Each residual needs to be added to the corresponding average, expressed as AVGlogicError and AVGsyntacticError, to correct the data.

TABLE 3. The residuals of the feature pairs (submissions, logicError) and (submissions, syntacticError).

	submissions	logicError	syntacticError	RES_{sl}	RES_{ss}	RES'_{sl}	RES'_{ss}
1	546	82	269	-129.418620	85.797958	170.9518	293.5387
2	503	98	244	-80.222547	69.955602	220.1478	277.6963
3	542	161	220	-47.330613	37.649832	253.0398	245.3906
4	417	122	117	10.169598	-38.729111	310.5400	169.0116
5	622	182	284	-88.090748	84.612355	212.2796	292.3531
6	612	242	215	-20.370732	17.742040	279.9996	225.4828
7	536	244	146	40.301397	-35.072358	340.6718	172.6684
8	898	461	293	-22.163215	34.833062	278.2072	242.5738
9	585	224	206	-17.526686	14.492188	282.8437	222.2329
10	678	282	255	-31.322843	43.686122	269.0475	251.4269
11	532	228	159	27.389404	-21.220484	327.7598	186.5203
12	677	371	158	58.449158	-53.100910	358.8195	154.6398
13	556	197	211	-22.138637	25.668273	278.2317	233.4090
14	364	103	117	32.085688	-27.441783	332.4561	180.2990
15	1420	950	298	63.851902	-71.336472	364.2223	136.4043
16	739	315	274	-45.414946	49.695046	254.9554	257.4358
17	449	169	132	32.465544	-30.544102	332.8359	177.1966
18	335	124	83	75.473737	-55.265698	375.8441	152.4750
19	804	472	194	61.404944	-44.147904	361.7753	163.5928
20	1341	859	318	33.840035	-34.511964	334.2104	173.2288
21	859	456	243	2.944851	-6.861169	303.3152	200.8796
22	613	335	134	71.857267	-63.470929	372.2276	144.2698
23	785	439	178	43.072976	-56.101503	343.4433	151.6392
24	501	140	197	-36.678544	23.381538	263.6918	231.1223
25	1038	497	388	-94.243452	100.017478	206.1269	307.7582
26	619	259	224	-8.774743	25.251261	291.5956	232.9920
27	282	98	52	90.389827	-74.978370	390.7602	132.7624

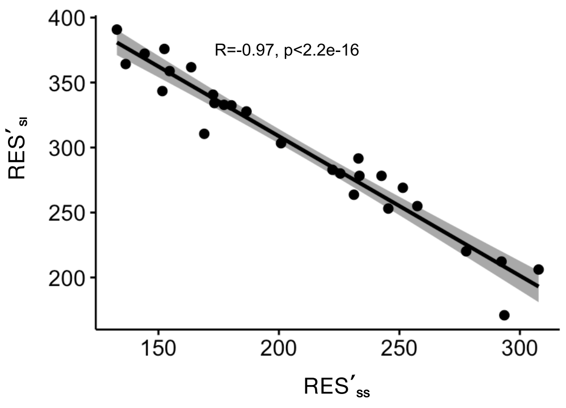


FIGURE 9. Partial correlation scatter plot of the residuals RES_{sl} and RES_{ss}. We denote the residual of (submissions, logicError) as RES_{sl} and the residual of (submissions, syntacticError) as RES_{ss}.

The corrected data are represented as: RES'_{sl} and RES'_{ss}, where:

$$RES'_{sl} = AVG_{logicError} + RES_{sl} \tag{1}$$

$$RES'_{ss} = AVG_{syntacticError} + RES_{ss} \tag{2}$$

Based on these two values, a partial correlation scatter plot of the feature pair (logicError, syntacticError) is plotted, as shown in Fig. 9. The feature pair (logicError, syntacticError) is strongly negatively correlated once the effects of other features are excluded. In other words, the more syntactic errors students have in these 27 assignments, the fewer logical errors there are; the fewer syntactic errors there are, the more logical errors there are. We suspect that the reason for this phenomenon is that the logic of the assignments is relatively simple at the beginning of the course, and thus there are mainly syntactic errors. After further study, students will struggle with increasingly complex programming logic, but they are now more familiar with syntax.

(4) The Spearman partial correlation coefficient value of 0.44 confirms that there is a moderate positive correlation between features codeSmell and logicError, which is not surprising. Because a smelly code segment is not easy to maintain, the likelihood of bug initiation in these segments is very high [36]. However, the frequency of codesmell per assignment in the experimental data is not large, which may be related to the size of files that were analyzed. However, introducing good programming patterns to students will improve the pass rate of the exam.

(5) As we expected, "averageScore" is independent of the other features, which may be an accidental phenomenon because all significance levels are greater than 0.05. However, we speculate that, even if its level of significance meets the requirements, "averageScore" is still unrelated to the other features. This is attributed to Trustie's scoring mechanism: students can modify and submit the code multiple times until they get a satisfactory score, thus making the average scores for each assignment very close to each other.

IV. LESSONS LEARNED

Several interesting observations were made from the analysis of the learning data of the CS1 programming course.

First, the frequency of issuing online assignments has a strong effect on the quality of students' submissions. We find that the frequencies of late submission and nonsubmission increase greatly when two assignments are submitted very closely together. However, the frequencies of late submission and nonsubmission do not increase greatly when the assignment is relatively more difficult. This indicates that the difficulty of assignments has little impact on students' enthusiasm for learning. Moreover, intensive assignments, in terms of submission, will make students extremely busy. To enhance the teaching impact, we recommend that teachers maintain a suitable publication frequency, preferably not publishing twice within five days. This is because students will not have much time to complete all assigned work at the end of the semester.

Second, programming logic training is more important than syntax training in programming courses. Beginner students always focus on grammar training rather than logic training. However, when difficulty levels increase, the frequency of syntactic errors does not increase very much, while logic errors increase greatly. Therefore, when students pay more attention to logic training, their learning of programming will be more successful in the long-term. Hence, instructors should concentrate more on logic training to improve the ability of students to think logically. This is also very helpful for students in building a solid foundation of programming, and even for their future career.

Third, a good programming style helps students to pass the test in a shorter period of time. A badly styled program is more at risk of error, and the experimental results also show a moderate positive correlation between code smell and logic errors. In other words, a good programming style helps reduce the frequency of logical errors. Therefore, in addition to strengthening logic training, teachers should also focus on fostering students' programming styles. It is wise to let students know how bad practice can pose unexpected risks to programs, and the sooner the better. Aside from discussions in class, reviewing the code among students and reading well-written code are also recommended.

Fourth, students need to improve their awareness of optimizing code as early as possible. The 27 assignments used in this study are far from reaching project-level, yet the source lines of code had a strong positive correlation with cyclomatic complexity. Although beginners can gain a sense of accomplishment from the amount of code, we still recommend that students develop an awareness for optimizing code as early as possible, which can diminish their risk of error in the future and greatly reduce debugging time.

V. CONCLUSION

In this paper, we tracked students' learning data through the Trustie platform and analyzed the submitted code from multiple aspects using CppCheck and SonarQube. We determined several factors that can help teachers develop more targeted teaching strategies. For example, the timing of an assignment release has an impact on the submission; logic errors are the main challenge for students; and code smell also contributes to logic errors. Such automated code quality analysis overcomes the limitations of the current literature on language types and can be used for up to 25 common programming languages, or even for multilanguage mixed projects. The limitation of this paper is that none of the assignments in the CS1 course have reached project level, so we do not have evidence to prove that code smell is strongly correlated with logical errors. At present, we can only prove that code smell and logical errors are moderately correlated.

Future work will focus on: (1) analyzing the relevance of code smell and logical errors in projects at the project level; (2) analyzing the optimal separation of assignment releases; and (3) analyzing whether the code review commonly used in software development plays a positive role in improving students' logic building and improving student programming style and how much can be gained. We believe that our work can assist teachers in developing more diverse and personalized teaching strategies.

ACKNOWLEDGMENT

The authors would like to thank our students for their active participation in the course, and the cooperation of TRUSTIE.

REFERENCES

[1] N. Parlante, "Nifty reflections," *ACM SIGCSE Bull.*, vol. 39, no. 2, pp. 25–26, Jun. 2007.

[2] K. Nguyen, "Automatic evaluation of Python and C Programs with codecheck," M.S. thesis, Dept. Comput. Sci., San José State Univ., San Jose, CA, USA, 2014. doi: 10.31979/etd.cukn-qkh2.

[3] D. Hovemeyer, M. Hertz, P. Denny, J. Spacco, A. Papancea, J. Stamper, and K. Rivers, "CloudCoder: Building a community for creating, assigning, evaluating and sharing programming exercises," in *Proc. 44th ACM Tech. Symp. Comput. Sci. Edu.*, Mar. 2013, p. 742.

[4] H. Wang, G. Yin, X. Li, and X. Li, "TRUSTIE: A software development platform for crowdsourcing," in *Crowdsourcing*. Berlin, Germany: Springer, 2015, pp. 165–190.

[5] Y. Lu, X. Mao, T. Wang, G. Yin, Z. Li, and H. Wang, "Continuous inspection in the classroom: Improving students' programming quality with social coding methods," in *Proc. 40th Int. Conf. Softw. Eng., Companion*, Jun. 2018, pp. 141–142.

[6] S. K. Kummerfeld and J. Kay, "The neglected battle fields of syntax errors," in *Proc. 5th Australas. Conf. Comput. Educ.*, vol. 20, 2003, pp. 105–111.

[7] T. Koulouri, S. Lauria, and R. D. Macredie, "Teaching introductory programming: A quantitative evaluation of different approaches," *ACM Trans. Comput. Edu. (TOCE)*, vol. 14, no. 4, 2015, Art. no. 26.

[8] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, Jun. 2014, pp. 252–261.

[9] M. M. T. Rodrigo, R. S. Baker, M. C. Jadud, A. C. M. Amarra, T. Dy, M. B. V. Espejo-Lahoz, S. A. L. Lim, S. A. M. S. Pascua, J. O. Sugay, and E. S. Tabanao, "Affective and behavioral predictors of novice programmer achievement," *ACM SIGCSE Bull.*, vol. 41, no. 3, pp. 156–160, 2009.

[10] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Identifying at-risk novice java programmers through the analysis of online protocols," in *Philippine Comput. Sci. Congr.*, 2008, pp. 1–8.

[11] G. Dyke, "Which aspects of novice programmers' usage of an IDE predict learning outcomes," in *Proc. 42nd ACM Tech. Symp. Comput. Sci. Edu.*, Mar. 2011, pp. 505–510.

[12] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers," *Comput. Sci. Educ.*, vol. 18, no. 2, pp. 93–116, 2008.

[13] S. H. Edwards and M. A. Perez-Quinones, "Web-CAT: Automatically grading programming assignments," *ACM SIGCSE Bull.*, vol. 40, p. 328, Sep. 2008.

[14] C. M. Hawkins, J. D. Pittman, F. B. Prats, W. E. Wheeler, D. Brown, J. Dalton, and B. Johnstone, "WebCAT: The design and implementation of the Web-based crime analysis toolkit," in *Proc. IEEE Syst. Inf. Eng. Design Symp.*, Apr. 2003, pp. 233–239.

[15] K. Van Haaster and D. Hagan, "Teaching and learning with BlueJ: An evaluation of a pedagogical tool," *Issues Informing Sci. Inf. Technol.*, vol. 1, pp. 0407–0455, Jan. 2004.

[16] D. Hagan and S. Markham, "Teaching java with the BlueJ environment," in *Proc. Australas. Soc. Comput. Learn. Tertiary Edu. Conf. (ASCILITE)*, 2000, pp. 1–12.

[17] A. Patterson, M. Kölling, and J. Rosenberg, "Introducing unit testing with BlueJ," *ACM SIGCSE Bull.*, vol. 35, no. 3, pp. 11–15, 2003.

[18] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "CodeWrite: Supporting student-driven practice of java," in *Proc. 42nd ACM Tech. Symp. Comput. Sci. Edu.*, Mar. 2011, pp. 471–476.

[19] R. Pettit, J. Homer, R. Gee, S. Mengel, and A. Starbuck, "An empirical study of iterative improvement in programming assignments," in *Proc. 46th ACM Tech. Symp. Comput. Sci. Edu.*, Mar. 2015, pp. 410–415.

[20] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proc. 45th ACM Tech. Symp. Comput. Sci. Edu.*, Mar. 2014, pp. 223–228.

[21] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ.*, Jul. 2017, pp. 110–115.

[22] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting at-risk novice java programmers through the analysis of online protocols," in *Proc. 7th Int. Workshop Comput. Edu. Res.*, Aug. 2011, pp. 85–92.

[23] C. Norris, F. Barry, J. B. Fenwick, Jr., K. Reid, and J. Rountree, "Clockit: Collecting quantitative data on how beginning software developers really work," *ACM SIGCSE Bull.*, vol. 40, no. 3, pp. 37–41, Sep. 2008.

[24] A. L.-R. Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, and C. Szabo, "Introductory programming: A systematic literature review," in *Proc. Companion 23rd Annu. ACM Conf. Innov. Technol. Comput. Sci. Educ.*, Jul. 2018, pp. 55–106.

[25] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall, "Analyzing student work patterns using programming exercise data," in *Proc. SIGCSE*, Mar. 2015, pp. 18–23.

[26] J. Sanchez and M. P. Canton, *Microcontrollers: High-Performance Systems and Programming*. Boca Raton, FL, USA: CRC Press, 2018.

[27] M. Kessentini, "Understanding the correlation between code smells and software bugs," Dept. Comput. Inf. Sci., Univ. Michigan-Dearborn, Dearborn, MI, USA, Tech. Rep., 2019. [Online]. Available: https://deepblue.lib.umich.edu/bitstream/handle/2027.42/147342/CodeSmellsBugs.pdf

[28] R. Nascimento and C. Sant'Anna, "Investigating the relationship between bad smells and bugs in software systems," in *Proc. 11th Brazilian Symp. Softw. Compon., Architectures, Reuse*, Sep. 2017, Art. no. 4.

[29] J. Hauke and T. Kossowski, "Comparison of values of Pearson's and Spearman's correlation coefficients on the same sets of data," *Quaestiones Geographicae*, vol. 30, no. 2, pp. 87–93, 2011.

[30] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: Appropriate use and interpretation," *Anesthesia Analgesia*, vol. 126, no. 5, pp. 1763–1768, May 2018.

[31] T. K. Burdenski, Jr., "Evaluating univariate, bivariate, and multivariate normality using graphical procedures," *Multiple Linear Regression Viewpoints*, vol. 26, no. 2, pp. 15–28, 2000.

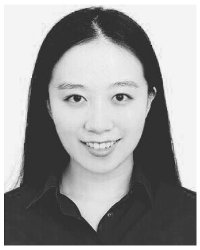
[32] J. P. Stevens, *Applied Multivariate Statistics for the Social Sciences*. Evanston, IL, USA: Routledge, 2012.

[33] R. E. Kass, U. Eden, and E. Brown, *Analysis of Neural Data*, vol. 491. New York, NY, USA: Springer, 2014.

[34] M.-T. Puth, M. Neuhäuser, and G. D. Ruxton, "Effective use of spearman's and Kendall's correlation coefficients for association between two measured traits," *Animal Behav.*, vol. 102, pp. 77–84, Apr. 2015.

[35] D. Landman, A. Serebrenik, and J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of java methods," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep./Oct. 2014, pp. 221–230.

[36] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of fault-proneness in methods by focusing on their comment lines," in *Proc. 21st Asia–Pacific Softw. Eng. Conf.*, vol. 2, Dec. 2014, pp. 51–56.



YU BAI received the B.S. and M.S. degrees in computer science from the National University of Defense Technology (NUDT), Changsha, China, in 2009 and 2014, respectively, where she is currently pursuing the Ph.D. degree in computer science. Her research interests include open source software engineering, data mining, machine learning, and adaptive learning in education.

TAO WANG received the B.S. and M.S. degrees in computer science from the National University of Defense Technology (NUDT), China, in 2007 and 2010, respectively, where he is currently pursuing the Ph.D. degree in computer science. His research interests include open source software engineering, machine learning, data mining, and knowledge discovering in open source software.

HUAIMIN WANG received the Ph.D. degree in computer science from the National University of Defense Technology (NUDT), China, in 1992. He is currently a Professor and the Chief Engineer of the Department of Educational Affairs, NUDT. He has published more than 100 research articles in peer-reviewed international conferences and journals. His current research interests include middleware, software agent, and trustworthy computing. He was a recipient of the Chang Jiang Scholars Program Professor and the Distinct Young Scholar.