International Journal of Engineering & Technology, 7 (3.28) (2018) 278-283

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Automated Ranking Assessment based on Completeness and Correctness of a Computer Program Solution

S. Suhailan[1]*, M.K. Yusof[1], A.F.A. Abidin[1], S.A. Fadzli[1], M.S. Mat Deris[1], S. Abdul Samad[2]

[1]*Faculty of Informatics and Computing, Universiti Sultan Zainal Abidin, 22200 Terengganu, Malaysia*
[2]*School of Information Technology, Faculty of Business and Information Science, UCSI University, Kuala Lumpur, Malaysia*
**Corresponding author E-mail: suhailan@unisza.edu.my*

Abstract

Many automated programming assessment methods requires program to be represented into certain calculated features. In order to assess the difficulty of a program in answering a computational programming question, two main factors need to be considered in extracting the features; program incompleteness and solution correctness. Common features were based on solution's template matching to assess a program correctness. However, incomplete program that usually occurs among novice learners may rise difficulty for the technique in parsing the program's structure. This research proposes program's scoring features based on instruction template's sequence and ratio to represent the programs into a solution ranking list in solving a programming question. The features were evaluated against manual rubric's assessment of 67 incomplete Java programs. The result shows that the proposed features were highly correlated with the manual rubric's assessment (rho = 0.9142086, S = 4299.5, p-value < 2.2e-16). Thus, the proposed features can be used to automatically rank computer programs based on expected instruction-based of solution templates. The ranking result can be used to identify most struggled user especially in assisting students in a programming lab exercise session.

Keywords: *program features; automated assessment; ranking features.*

1. Introduction

Extracting incomplete source code features are among important factor need to be considered in the context of assisting learning difficulties for computational programming exercises. It will be a tough role for a teacher to assist on each student's difficulties during programming exercises. However, in most cases, students' difficulties are commonly shared among others. Thus, it is possible to group them together using clustering technique. Learning difficulties can be measured using ordinal features of source codes that denote to certain ranking range.

Many computer program features were addressed especially in realizing the automated assessment process. The assessment is not only in the scope of solution correctness for marking purposes, but also to assess program's quality in term of maintainability, complexity, cost and plagiarism similarity. However, as features were meant on specific aim and objective, there were no universal features that can meet all application requirement. Thus, different practitioners may adopt different strategies to represent the meaning of respective data [1].

In respect to assessing specific computational program's correctness, two general strategies were commonly used covering test cases analysis and solution template matching [2]. Unfortunately, test cases analysis cannot be executed on an incomplete computer program that may contain errors. On the other hand, many solutions template matching were implemented to find mismatches between computer program's structure with the solution model. However, this kind of approach may be abused by students through trial and error strategy to get closer to the solution without really understand the reason why the changes need to be performed accordingly to match the template [3]. The matching process that was based on the discrete structure of the solution template using syntax tree has also limited the approach to be only applicable on a specific programming language syntax. It also requires more efforts for an expert user to provide the solution templates.

2. Source Code Features

In general, source code features are usually extracted using three kind of methods; parser-based, token-based and text-based method [4-5]. Parser-based method recognizes the structure of codes based on specific grammar using abstract syntax tree (AST) or dependency graph. In [6] use K-Means to group similar C source codes based on ordinal features such as number of loops, number of selection, number of modules/functions/classes, number of variables, number of jump statement instructions, and number of expression. In [7] extract Java source code features based on number of variables, number of objects instantiation, and number of return value in order to classify its class pattern (e.g. utility, DOA, builder, bean, adapter, etc.). These features are extracted from the codes based on their matched parse tree of specific programming language grammar. However, designing own syntax tree to extract occurrence of specific features of incomplete source code patterns requires a lot of works and vary among languages. The easiest way to extract incomplete source code features is by using syntax errors report generated by existing compiler tool [8]. However, some of syntax errors can be arguable when they are used in representing difficulties in programming. For example, consider missing semicolon or unbalance braces which may generate cascading errors, these occurrences are more related to mistakes rather than misunderstanding or difficulties. On the other hand, token-based and text-based method represent source code features based on token frequency or sequence. The only different between these methods is on their token identification technique. Token-based method identify tokens based on syntactical tokens generated by specific programming language parser. In [9] use this method to extract source code features based on n-gram token sequences such as header, keywords, identifier, operators, and numerals. Although, this approach is language-dependent, there are many independent tools available that can easily tokenize the source codes. Meanwhile, text-based method identifies tokens based on characters or words. In [10] use n-gram to represent the source code as set of string sequences. In [11] extract source code features based on frequency of characters and words in a line, frequency of whitespace (in the beginning, interior and ending) of non-whitespace lines and frequency of underscore. Although source code feature using token-based or text-based is based on cardinal numbers that cannot be used to rank source code [12], however, both method has its own strength as it can represent feature for incomplete codes.Ranking source codes can be done through three approaches; dynamic testing, software metrics and solutions template [13]. Dynamic testing involves with execution of source code with certain test cases. Good source codes can be represented by having high number of successful test cases of his source code. In [14] review strategies to automatically assess correctness of test cases by tracing the program output. Among them are simple output character matching using exact-match or filtered-match such as eliminate whitespaces and case insensitive [15-16]. Regular expression [17] and token pattern [18] are also among filtered-match of the program output. Unfortunately, most of learning difficulties in programming occurs in syntax errors which resulting dynamic testing failed to be executed. Meanwhile, software metrics are usually evaluated manually based on expert views by assessing source code's metrics at four different levels; system (e.g. maintainability), module (e.g. interoperability), class (e.g. reusability) and method (e.g. understandability) [19]. However, some of the attributes can be automatically extracted such as size of codes (smaller is better), comment percentage (higher is better), cyclomatic complexity (low is better), and number of methods (more is better) [20]. The metrics represent good or bad category of source codes based on higher or lower marks obtained. However, these kinds of features may not applicable for incomplete codes (i.e. codes with syntax and semantic errors). As an example, size of codes of an incomplete source codes do not really represent good or bad solution as it may consist of trial and error statements. On the other hand, solution template is used as syntax structure to find any mismatches or missing element from student's codes structure while answering a computational programming question [21-24]. This approach can possibly assign the ordinal features to the source codes by assessing the matching percentage of solution template. However, providing full solution codes as template for each question may increase workload of a teacher especially when new problem set are needed for different student groups. Furthermore, as features are extracted using parser-based using language-dependent syntax tree, this approach is not flexible for other languages and static for specific set of predefined problems.

On the other hand, extracting source code features using text-based method are more generic and flexible for incomplete codes. Many of researches are using *N*-gram method in extracting source code features for closeness measurement such as plagiarism [25], authorship [26], and malware [27-28]. *N* is usually set to four [29] to be meaningful in representing features of source code. Most of these source codes features are based on set of cardinal numbers that represent source code identification or fingerprint. To be used in ranking application, features need to be in ordinal representation. This paper proposes new ranking-based source codes feature extraction for incomplete codes based on solution template using text-based method.

3. Ranking-based Program Features

Computer program features have been widely proposed by other researchers especially in software engineering and plagiarism detection domain, but most of them are meant for a complete or working program and lack of ordinal representation [30]. Ordinal feature is important to enable ranking of computer programs from least to worst program difficulty. To access the correctness of a computer program, these features need to be related with related knowledge model such as by using solution templates. In this research, two ordinal features are proposed to process the incomplete program; *instruction-gram ratio (IGR)* and *instruction count ratio(ICR)*. These text-based features eliminate the rigidness in preparing complete program as solution templates on each specific computational programming question. They are more flexible and independent from rigidness of syntax rules as the template can just be provided using sequence of symbols or instructions.

3.1. Instruction-gram ratio (IGR)

Instruction-gram Ratio (IGR) feature is derived from *n-skip-gram* model. It counts the ratio of consecutive instructions or symbols sequence in a program as compared to the template's instructions or symbols sequence with certain skippable instruction(s). This feature is meant to represent student's level of difficulty to formulate solution logics or flows in solving specific computational programming question. Multiple templates can be provided to accept various solution. In these case, the highest IGR among the templates matching will be taken as the program's features. The template(s) consists of expected instructions sequence prepared by an expert user on each computational programming question. These instruction sequence will be processed as a list of words. The algorithm to calculate IGR is given as the following.

Algorithm 1: Instruction-Gram Ratio with skip sequence

```
1:  IGR = 0
2:  N = number of instructions (IS) in template
3:  loc = 0
4:  nskip = 0
5:  TOTALSKIP= number of allow skip sequence
6:  for i=0 to N do
7:      found_loc = find location of $IS_i$ in program start at loc
8:      if found_loc > 0 and found_loc > loc
9:          IGR++
10:     loc = found_loc + sizeof($IS_i$)
11:     else if nskip < TOTALSKIP
12:         nskip++
13:     else
14:         exit for
15:     end if
16: end for
17: return IGR / N
```

As an example, consider the following computational programming question.

"Write a program that can receive THREE integer input and determine whether the input is greater than or less than or same number with 100."

A solution template for the above question can be prepared using sequence of Java instructions as {"*nextInt*", "*nextInt*", "*nextInt*", "*if*", "*else if*", "*else if*", "*if*", "*else if*", "*else if*", "*if*", "*else if*", "*else if*"}. Then, let consider an example of student answer as the following.

```
Scanner k = new Scanner(System.in);
int num = k.nextInt();
if(num>100)
```

```
   System.out.println(num+" is greater than 100");
else if (num ==100)
   System.out.println(num+" is a same number");
else
   System.out.println(num+" is less than 100");
```

Based on the given solution template, total instruction (*N)* is 12. The IGR calculation starts by initializing *loc* to 0 and searches the first instruction in the sequence, IS_1 ("nextInt"). This instruction is found at location 40 of non-empty space characters and the location is stored to *found_loc*. As *found_loc* is greater than 0 and the *loc*'s value, IGR is then incremented to 1 and *loc* is updated to the new search location of *found_loc (40)* + size of $IS_1(7)$. Then, it continues to search next consecutive instruction such as IS_2 ("nextInt") started from the new search location of *loc*. The searching process is continued until all the instructions are searched. However, if one or number of skip-able instruction is not found, the searching process is terminated. In this research, four IGR features are proposed based on zero to three of skip-able instructions.
Alternatively, there are other correct solution template that may be also included such as {"*nextInt*", "*if*", "*else if*", "*else if*", "*nextInt*", "*if*", "*else if*", "*else if*", "*nextInt*", "*if*", "*else if*", "*else if*"}. Implementing loop for input and selection instruction is also another correct sequence that can be considered as one of the templates. In the case of multiple templates are provided, IGR feature for a program is selected based on the highest IGR value among the list of templates

3.2. Instruction count ratio (ICR)

Instruction Count Ratio (ICR) is an average ratio of all unique instructions count in a program that matches with all unique instructions count specified in a template. The template is a same template that used in extracting previous IGR feature. This feature is meant to represent student's level of difficulty in identifying specific computer statements (e.g. numbers of input, numbers of selection, or looping statement) that are required in solving a specific computational programming question. The algorithm to calculate ICR is given in Algorithm 2.

Algorithm 2: Instruction Count Ratio Average

```
1:   $N$ = number of unique instructions ($I'$) in the template
2:   $ICR = 0$
3:   for $i=1$ to $N$ do
4:        $Nt$ = number of $I'_i$ in template
5:        $Np$ = number of $I'_i$ in program
6:        if $Np / Nt <= 1$
7:                  $ICR = ICR + (Np / Nt)$
8:        else
9:                  $ICR = ICR + 1$
10:       end if
11:  end for
12:  $ICR = ICR / N$
13:  if $ICR>1$
14:       $ICR=1$
15:  return $ICR$
```

Considering previous example of instruction sequence template, they are three unique instructions covering "nextInt", "if" and "else if". Each of them is counted as three. By having instruction map table, ICR is calculated on each instruction as in Table 1.

Table 1: Instruction Count Ratio

Unique Instruction in Template	Template Count	Program Count	Ratio
nextInt	3	1	1/3
if	3	1	1/3
else if	3	1	1/3
ICR (average)	(1/3 + 1/3 + 1/3) / 3 = 1.33		

All the ratio values on each unique instruction specified in a template will be total up together and its average will represent the ICR feature for a program. If the value is above 1, then its ICR will be set to 1. In case there are more than one set of templates, ICR feature will be calculated based on the template that carry highest IGR value.

4. Results and Analysis

A query was performed on AOPC's MySQL database to extract all the latest Java program attempts on "Hangman Question" which dated on 12 December 2013 12:00pm as a dataset for simulating assisted feedback's experiments. The database can be downloaded at https://figshare.com/s/d68c4af2abef31811cb4. The attempts were made by 67 participants of first year students from one of public university. The same dataset is also used by researcher to manually give marks based on rubrics for each program's attempt. The information on the question detail, answer template, rubric's specification, programs' attempt, rubric's marks, ranking order and features result can be located in *https://dx.doi.org/10.6084/m9.figshare.3159967*.
A ranking list of computer programs in answering a question were generated based on weighted sum method using the dataset. Five proposed features; IGR with no-skip sequence (IGR0), IGR with one-skip sequence (IGR1), IGR with two-skip sequences (IGR2), IGR with three-skip sequences (IGR3) and an average of ICRs (ICR) were extracted based on answer templates that consist of 11 instructions with 7 of unique instructions. There are four templates specified for the question and the features' value were taken based on the highest score among of these templates. Based on these extracted features, ranking of computer programs were generated using simple weighted sum method. To compare these feature-based ranking result, a manual ranking list was also generated through researcher's assessment on the computer programs using a rubric. The rubric contains five criteria that represent the expected structure of correct answer. Table 2 shows the complete ranking result of the computer programs.

Table 2: Ranking result of computer programs

ID	IGR0	IGR1	IGR2	IGR3	ICR	Total Features(%)	Rubric's Mark (%)	Rubric's Rank	Features' Rank
75	1	1	1	1	1	100	100	62	60
113	1	1	1	1	1	100	100	62	60
103	1	1	1	1	1	100	100	62	60
82	1	1	1	1	1	100	84	59	60
117	1	1	1	1	1	100	100	62	60
95	1	1	1	1	1	100	68	47	60
51	1	1	1	1	1	100	100	62	60
96	1	1	1	1	1	100	92	61	60
89	0.90909	0.90909	0.90909	0.90909	0.85714	89.87	52	32	59
141	0.81818	0.90909	0.90909	0.90909	0.85714	88.0518	60	43	55
85	0.81818	0.90909	0.90909	0.90909	0.85714	88.0518	80	57	55
54	0.81818	0.90909	0.90909	0.90909	0.85714	88.0518	84	59	55
69	0.81818	0.90909	0.90909	0.90909	0.85714	88.0518	100	62	55
143	0.81818	0.81818	0.81818	0.81818	0.71429	79.7402	80	57	54

134	0.63636	0.72727	0.81818	0.81818	0.71429	74.2856	56	40	47
138	0.63636	0.72727	0.81818	0.81818	0.71429	74.2856	68	47	47
68	0.63636	0.72727	0.81818	0.81818	0.71429	74.2856	72	52	47
131	0.63636	0.72727	0.81818	0.81818	0.71429	74.2856	72	52	47
73	0.63636	0.72727	0.81818	0.81818	0.71429	74.2856	72	52	47
112	0.63636	0.72727	0.81818	0.81818	0.71429	74.2856	68	47	47
100	0.63636	0.72727	0.81818	0.81818	0.71429	74.2856	72	52	47
91	0.09091	0.90909	0.90909	0.90909	0.85714	73.5064	76	56	46
86	0.63636	0.72727	0.72727	0.72727	0.71429	70.6492	52	32	45
126	0.63636	0.72727	0.72727	0.72727	0.57143	67.792	68	47	44
99	0.54545	0.54545	0.63636	0.63636	0.71429	61.5582	68	47	43
124	0	0.72727	0.72727	0.72727	0.71429	57.922	40	12	42
88	0.45455	0.54545	0.54545	0.54545	0.57143	53.2466	64	46	41
93	0.09091	0.54545	0.63636	0.72727	0.57143	51.4284	56	40	39
109	0.09091	0.54545	0.63636	0.72727	0.57143	51.4284	56	40	39
110	0.09091	0.54545	0.63636	0.63636	0.57143	49.6102	52	32	37
128	0.09091	0.54545	0.63636	0.63636	0.57143	49.6102	44	18	37
90	0.18182	0.36364	0.36364	0.36364	0.57143	36.8834	60	43	36
108	0.18182	0.27273	0.27273	0.27273	0.71429	34.286	60	43	34
130	0.18182	0.27273	0.27273	0.27273	0.71429	34.286	52	32	34
129	0.18182	0.27273	0.27273	0.27273	0.57143	31.4288	52	32	33
125	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	52	32	25
59	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	44	18	25
78	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	48	21	25
70	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	48	21	25
87	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	48	21	25
83	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	32	5	25
80	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	48	21	25
98	0.18182	0.27273	0.27273	0.27273	0.42857	28.5716	52	32	25
145	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	52	32	12
122	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	48	21	12
107	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	40	12	12
139	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	48	21	12
136	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	48	21	12
79	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	48	21	12
60	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	40	12	12
114	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	44	18	12
101	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	48	21	12
142	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	48	21	12
62	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	40	12	12
84	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	48	21	12
77	0.18182	0.27273	0.27273	0.27273	0.28571	25.7144	40	12	12
56	0.09091	0.09091	0.27273	0.27273	0.42857	23.117	36	10	10
71	0	0	0.36364	0.36364	0.42857	23.117	40	12	10
116	0.09091	0.09091	0.18182	0.18182	0.28571	16.6234	28	3	8
115	0.09091	0.09091	0.18182	0.18182	0.28571	16.6234	36	10	8
146	0.09091	0.09091	0.18182	0.18182	0.14286	13.7664	32	5	3
92	0.09091	0.09091	0.18182	0.18182	0.14286	13.7664	28	3	3
67	0.09091	0.09091	0.18182	0.18182	0.14286	13.7664	32	5	3
94	0.09091	0.09091	0.18182	0.18182	0.14286	13.7664	32	5	3
144	0.09091	0.09091	0.18182	0.18182	0.14286	13.7664	32	5	3
97	0	0.09091	0.09091	0.09091	0.28571	11.1688	4	1	2
118	0	0	0	0.09091	0.14286	4.6754	8	2	1

This feature-based ranking result is effective if it generates rank of computer programs that is similar to the manual or normal ranking approach. A Spearman's rank correlation was used to assess the correlation between the computer program's rank using the proposed automated ordinal features and manual rubrics assessment. The correlation results were generated using R software version 3.2.2. The result in Table 3 shows that there was a very strong positive correlation between ranking result of the automated proposed features and manual marks especially when all the features were combined (rho = 0.9142086, S = 4299.5, p-value < 2.2e-16).

Table 3: Spearman's rank correlation using IGR and ICR features for program ranking

Type of Features	Rho Correlation
IGR0	0.8182933 (S = 9106.4, p-value < 2.2e-16)
IGR0 and IGR1	0.9010911 (S = 4956.9, p-value < 2.2e-16)
IGR0, IGR1 and IGR2	0.9058649 (S = 4717.7, p-value < 2.2e-16)
IGR0, IGR1, IGR2 and IGR3	0.9078048 (S = 4620.5, p-value < 2.2e-16)
IGR0, IGR1, IGR2, IGR3 and ICR	0.9142086 (S = 4299.5, p-value < 2.2e-16)

5. Discussion

The proposed features use *n-gram* based method called as *instruction-gram ratio* (IGR) to assess computer program based on text-based solution template. Rather than writing complete computer program as a solution template, this method only requires a sequence of main expected computer instructions (i.e. Keywords) to be provided. Instead of calculating several correct instruction sequences, its ratio was chosen to represent the computer feature. This is due to the diversity of end user's program that can be compared with different templates where each template has a number of different instructions. The proposed IGR has also applied the *n-skip-gram* based method that allowed the sequence to be skipped up to 3 sequences. This is to compensate computer program that may be missing in the earlier sequence, but matching well in the following instructions' sequence. It is important to note that those who has the lowest score in igr-0 but highest in igr-1 may be possibly ranked in the same group with who has scored higher in igr-0. For example, a computer program of a maybe missing the last sequence, while a computer program of b may be missing only the first sequence. These computer programs are having the same difficulty level of only one mistake. Thus, it is important to also include igr-1, igr-2 and igr-3 as the main features to be used in assessing the similarity of difficulty level.

Meanwhile, a feature of *instruction-count ratio* (ICR) was also used to represent similar difficulty level among computer programs without considering the correct sequence. This feature is important, especially in the differentiating computer program of same IGRs values. There may be a case of among similar lowest score of IGRs, there are computer programs that should be ranked in a different position due to its highest score in meeting all the instructions template without sequence constraint. In a computer program, although using same instructions, some of the sequences may be flexible. Consider the statements of "*if (a==1) print 1; else print 0;*" and expected sequence template is "*if =>print 1=>else=>print 0*". If a computer program was written as not in the expected sequence such as "*if (a!=1) print 0; else print 1;*", the score of IGR-1 will be 0.25 (1/4) although this is a correct solution. On the other hand, when evaluated using ICR, the program will gain full score. Thus, including the ICR feature is important especially in supporting the limitation of template insufficiency to cover the varieties of the correct solution.

It also noted that these features were purposely designed to represent ordinality rather than common cardinal features such as a line of codes, cyclometric complexity, number of methods, number of selection, number of loops, number of comments and etc. These ordinal features were used to represent a certain level of computer program's difficulty in answering computational programming exercise. Based on this representation, the computer program can be ranked and sorted for further analysis

6. Conclusion

Automated computer program ranking is important to sort students into certain difficulty level in answering a computational programming question. It can help educators to tackle most struggled students in practicing computer programming. However, current practices put burden on educators to prepare a complete solution template for automating student's answer assessment. This research has proposed features that can be used to automatically assess student's answer using text-based parser based on keyword-solution template of computer instruction sequence. The features have managed to sort among good and bad computer programs according to the specific items of solution structure. However, the features cannot provide detail assessment on the correctness level of an item. As in the rubric assessment, the goodness of an item can be represented with detail ordinal mark such as ranging from 0 to 5. On the other hand, the proposed features were only capable of accessing availability (0 or 1) of an item rather than its quality. Thus, a future research on this issue can enhance the result to be applicable as an automated marking assessment method.

Acknowledgement

This work has been supported and funded by ministry of higher education (MOHE) Malaysia under SLAI scholarship.

References

[1] M. Joy, 2010, “Automated Assessment,” University of Warwick. https://www2.warwick.ac.uk/fac/sci/dcs/research/edtech/automated assessment/.

[2] S. Safei, A. S. Shibghatullah, and B. Mohd Aboobaider, 2014, “A Perspective of Automated Programming Error Feedback Approaches,” Journal of Theoretical and Applied Information Technology, 70(1), 121–129.

[3] B. E. Vaessen, F. J. Prins, and J. Jeuring, 2014, “Computers and Education University Students’ Achievement Goals and Help-Seeking Strategies in An Intelligent Tutoring System,” Computers and Education, 72, 196–208.

[4] Y. Udagawa, “A Novel Technique for Retrieving Source Code Duplication,” Proceedings of the Ninth International Conference on Systems, 2014, pp. 172–177.

[5] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer, “Comparison of Similarity Metrics for Refactoring Detection,” Proceedings of the 8th Working Conference on Mining Software Repositories, 2011, pp. 53–62.

[6] E. Stankov, M. Jovanov, A. M. Bogdanova, and M. Gusev, 2013, “A New Model for Semiautomatic Student Source Code Assessment,” Journal of Computing and Information Technology, 21(3), 185–194.

[7] M. Mojzeš, M. Rost, J. Smolka, and M. Virius, “Feature Space for Statistical Classification of Java Source Code Patterns,” Proceedings of the 15th International Carpathian Control Conference, 2014, pp. 357–361.

[8] C. Fernandez-Medina, J. R. Pérez-Pérez, V. M. Álvarez-García, and M. D. P. Paule-Ruiz, “Assistance in Computer Programming Learning Using Educational Data Mining and Learning Analytics,” Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, 2013, pp. 237–242.

[9] S. Sharma, C. S. Sharma, and V. Tyagi, “Plagiarism Detection Tool ‘Parikshak,’” Proceedings of the International Conference on Communication, Information and Computing Technology, 2015, pp. 1–7.

[10] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local Algorithms for Document Fingerprinting,” Proceedings of the ACM SIGMOD International Conference on on Management of Data, 2003, pp. 76–85.

[11] U. Bandara and G. Wijayarathna, 2013, “Source Code Author Identification With Unsupervised Feature Learning,” Pattern Recognition Letters, 34(3), 330–334.

[12] R. Lange and S. Mancoridis, “Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics,” Proceedings of the Genetic and Evolutionary Computation Conference, 2007, pp. 2082–2089.

[13] T. Wang, X. Su, Y. Wang, and P. Ma, 2007, “Semantic Similarity-Based Grading of Student Programs,” Information and Software Technology, 49(2), 99–107.

[14] C. M. Tang, Y. T. Yu, and C. K. Poon, “A Review of the Strategies for Output Correctness Determination in Automated Assessment of Student Programs,” Proceedings of the 14th Global Chinese Conference on Computers in Education, 2010, pp. 584–591.

[15] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, “Understanding the Syntax Barrier for Novices,” Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, 2011, pp. 208-212.

[16] A. Papancea, J. Spacco, and D. Hovemeyer, “An Open Platform for Managing Short Programming Exercises,” Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research, 2013, pp. 47–51.

[17] D. S. Morris, “Automatic Grading of Student’s Programming Assignments: An Interactive Process and Suite of Programs,” Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference, 2003, pp. 1–6.

[18] C. M. Tang, Y. T. Yu, and C. K. Poon, “An Approach Towards Automatic Testing of Student Programs Using Token Patterns,” Proceedings of the 17th International Conference on Computers in Education, 2009, pp. 188–190.

[19] P. Garg, S. Sangwan, and R. K. Garg, 2014, “Design an Expert System for Ranking of Software Metrics,” International Journal for Research in Applied Science and Engineering Technology, 2(8), 109–117.

[20] H. M. Manoj and A. N. Nandakumar, 2014, “A Survey on Modelling of Software Metrics for Ranking Code Reusability in Object Oriented Design Stage,” International Jornal of Engineering Research and Technology, 3(12), 538–544.

[21] M. Suarez and R. Sison, 2008, “Automatic Construction of a Bug Library for Object-Oriented Novice Java Programmer Errors,” Intelligent Tutoring System, 5091, 184–193.

[22] E. R. Sykes, 2005, “Qualitative Evaluation of the Java Intelligent Tutoring System,” Journal of Systemics, Cybernetics and Informatics, 3(5), 49–60.

[23] R. Singh, S. Gulwani, and A. Solar-lezama, “Automated Feedback Generation for Introductory Programming Assignments,” Proceed-

ings of the ACM Programming Language Design and Implementation, 2013, pp. 15–26.
[24] A. Bagini, 2011, “Automatic Assessment of Java Programming Patterns for Novices,” University of Western Australia.
[25] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, 2014, “Using Web Corpus Statistics for Program Analysis,” ACM SIGPLAN Notices, 49(10), 49–65.
[26] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. E. Chaski, and B. S. Howald, 2007, “Identifying Authorship by Byte-Level N-Grams: The Source Code Author Profile (SCAP) Method,” International Journal of Digital Evidence, 6(1), 1–18.
[27] A. Pektaş, “Proposal of n-gram Based Algorithm for Malware Classification,” Proceedings of the Fitth International Conference on Emerging Security Information, Systems and Technologies, 2011, pp. 14–18.
[28] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, “Detection of New Malicious Code Using N-Grams Signatures,” Proceedings of the Second Annual Conference on Privacy, Security and Trust, 2004, pp. 193–196.
[29] A. Jadalla and A. Elnagar, 2008, “PDE4Java: Plagiarism Detection Engine for Java Source Code: A Clustering Approach,” International Journal of Business Intelligence and Data Mining, 3(2), 121–135.
[30] S. Safei, S. Abdul Samad, M. A. Burhanuddin, and A. H. Nazirah, 2017, "Program Statement Parser for Computational Programmng Feedback", Journal of Engineering and Applied Science, 12(5), 7057-7062.