12-2016

Semi-automated tool for providing effective feedback on programming assignments

Min Yan BEH
Singapore Management University, minyan.beh.2014@sis.smu.edu.sg

Swapna GOTTIPATI
Singapore Management University, SWAPNAG@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Venky SHANKARARAMAN
Singapore Management University, venks@smu.edu.sg

Chen, W. et al. (Eds.) (2016). Proceedings of the 24th International Conference on Computers in Education. India: Asia-Pacific Society for Computers in Education

Semi-automated Tool for Providing Effective Feedback on Programming Assignments

Min Yan BEH, Swapna GOTTIPATI*, David LO, Venky SHANKARARAMAN
School of Information Systems, Singapore Management University
minyan.beh.2014,*swapnag,davidlo,venks@smu.edu.sg

Abstract: Human grading of introductory programming assignments is tedious and error-prone, hence researchers have attempted to develop tools that support automatic assessment of programming code. However, most such efforts often focus only on scoring solutions, rather than assessing whether students correctly understand the problems. To aid the students improve programming skills, effective feedback on programming assignments plays an important role. Individual feedback generation is tedious and painstaking process. We present a tool that not only automatically generates the static and dynamic program analysis outcomes, but also clusters similar code submissions to provide scalable and effective feedback to the students. We studied our tool on data from introductory Java programming assignments of year 1 course in School of Information Systems. In this paper, we share the details of our tool and findings of our experiments on 261 code submissions.

Keywords: Automated grading, effective feedback, programming assignments, clustering

1. Introduction

Human grading of introductory programming assignments is a tedious and error-prone task, a problem compounded by the large student cohorts of programming courses. It is a challenge for instructors to review individual code submissions and provide specific feedback for each student. Therefore, the instructors tend to provide a general feedback on correctness of code and common errors. As a result, we observe that most of the students in introductory programming classes struggle to improve in their mastery of programming techniques.

One of the most common solutions to this problem is to automate the grading process such that students can electronically submit their programming assignments and receive instant feedback. Several tools are developed for assisting teachers in assessing student programs through assessing the program output and program code (Ala-Mutka, 2005). Most systems evaluate the function correctness of student programs by compiling and executing the programs with test inputs and comparing the output of student programs with that of the model program (Higgins, et al. 2003, Cheang, et al. 2003, Joy, et al. 2005). Auto-assessment often focuses only on scoring solutions, rather than assessing whether students correctly understand the problem and then providing individual feedback.

Many universities take on a test-case-based approach to evaluate submissions for student assignments and timed assessments that put the students' practical programming skills to test (Cheang, et al. 2003). As such, there is a lack of personalised and comprehensive feedback for most introductory programming classes, since the overwhelming number of student submissions precludes the option of manual evaluation (Irene, et al. 2015). Specific feedback on areas of improvement for code submissions is critical for novice programmers, who lack the experience to decipher their own mistakes for further improvement. Therefore, code feedback should not only focus on programming errors, but also on the logical approach and correctness improvements (Melina, et al. 2012).

In this paper, we propose a semi-automated tool based on static and dynamic program analysis, and clustering model (Maimon et al. 2009) that can facilitate the process of feedback generation. The tool takes programming assignments as input and generates clusters of similar codes together with the compilation errors and code structures. Through summarized visual outputs, instructor will be able to provide relevant feedback text to be propagated to all submissions within that cluster.

We studied the tool on 261 student code submissions to a "String Manipulation" programming assignment during a timed programming assessment for an introductory Java programming course, IS Software Foundations, at School of Information Systems, Singapore Management University. The tool generated eight clusters and visual outputs of code details for each cluster. Instead of evaluating 261

submissions separately, the instructor is now only required to look at representatives from just eight clusters. This saves significant time and energy on the instructors' part and is a useful tool for scaling up feedback frequency on student programming assignments. The rest of the paper is organized as follows. In section 2, we explore the current literature on related research in three areas namely auto-grading, program analysis and clustering techniques for auto-grading. Section 4 describes the details of our tool. Section 5 introduces the datasets followed by analysis of results from our experiments. Conclusions drawn from this research are presented in section 6.

2. Related Work

Grading programming assignments: The use of auto-grading systems in introductory programming courses has been studied by many researchers. Several advantages of automatic assessment in programming courses have been observed. Ala-Mutka (2005) described speed, availability, consistency and objectivity of assessment. For code correctness, Higgins, et al. (2003) constructed test by specifying the content of output file for a given input file. Lane (2004) used Junit test cases for code correctness. Static analysis is usually more efficient but less precise than dynamic analysis and testing, and their complementarity is well defined by Ernst et al. (2003). In our tool, we used test-case based approach for dynamic program analysis and employed the similar idea of performing both static and dynamic program analysis to extract logical structure and code correctness.

Effective Feedback: Though automated assessment saves time for instructors, immediate feedback is more important for supporting the students in their learning process. Milena et al. (2013) highlighted the advantage of meaningful and comprehensible feedback for students, especially for novices who can benefit from early disambiguation of misconceptions in introductory programming courses. Glassman et al. (2015) discussed the importance of effective feedback and need for an automated tool.

Program Analysis: Java compilers flag some of the programming errors, often the Java error messages are usually cryptic especially to novice students and thus they have difficulty in identifying the errors and making corrections. Program analysis techniques are popularly used by programmers or developers to enable discovery of comprehensive characteristics of code **(**Ernst et al. 2003). This can aid the students comprehend the error messages. Two types of program analysis techniques are widely used namely static and dynamic. On the one hand, static program analysis is conducted in a non-runtime environment, and involves a thorough inspection of source code to identify any flaws in the logical flow of the program. However, this technique does not help to verify the correctness of the program code in terms of the results it is supposed to produce. On the other hand, dynamic program analysis is carried out in the runtime environment, where the functional behavior of the code is monitored. Using this technique, one can determine the correctness of code submissions with reference to the expected results (Lane 2004) and the errors in the code.

Clustering models: Applying data mining and analytics techniques for curriculum enables analyzing the content and assessments of the course (Gottipati et al. 2014a). Gottipati et al. (2014b) used key phrase extraction techniques for analyzing the assessments against learning outcomes. Therefore, unsupervised data mining techniques are useful for analyzing data which is unlabeled. The goal of an unsupervised clustering algorithm is to create clusters that are similar internally, but are clearly different from each other. Hierarchical clustering outputs a hierarchical structure through a clustering process that starts from bottom up, with every student code in its own cluster. It starts by finding the closest pairs of clusters, based on the Ward's minimum variance method, and merging them together so that there is one less cluster after each merge (Glassman et al. 2014). This process repeats itself until we are left with one cluster, which contains all code submissions in the data set. Clustering technique is very befitting in our context of finding patterns in student code submissions in an automated manner. In our solution, we used clustering technique to cluster similar codes with the help of decision trees. Clusters can reduce painstaking task of evaluating and writing feedback for each and every student individually.

Clustering techniques for auto-grading: Clustering techniques for auto-grading is an active research in education community (Glassman, et al. 2014). Glassman, et al. (2014) used a clustering technique and feature engineering for the grading of codes. They performed a hierarchical clustering (Maimon et al. 2009) of student codes. They found that in order to cluster submissions effectively, both abstract and concrete features needed to be extracted and clustered. OverCode uses clustering

techniques to aid teachers write general feedback for the entire class (Glassman et al. 2015). In our tool, we adopt a similar approach where we use clustering algorithm to cluster similar codes. However, our tool also aids in generating more specific feedback for each cluster instead of the entire class. Our solution approach combines both static and dynamic program analysis to aid the instructors provide effective feedback along three dimensions namely analysis of the logical approach adopted by the student in structuring the code, evaluation of the code for correctness in terms of the results it is supposed to produce, and an analysis of the errors in the code.

3. Solution

Our solution approach takes programming assignments, test cases and student submissions as inputs and generates visual outputs of clusters of codes, static analysis results and dynamic analysis results. Figure 1 depicts the overall solution framework of our programming assignment feedback tool.

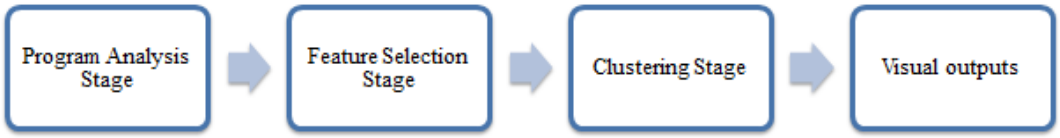


Figure 1: Framework of semi-automated tool for feedback on programming assignments

Program Analysis Stage: Static program analysis is performed by extrating the structures of the code using lexical parsers written in Java. A snippet of parser is shown below.

```
static void extractMethods(String student, String content){
        Pattern pMtd = Pattern.compile("([0-9_a-zA-Z]*[.][0-9_a-zA-Z, -<>(]+[)])");
   Pattern pNew = Pattern.compile("(new[ ][0-9_a-zA-Z(,]+[)])");
   Scanner sc = null;
        try{
                writer.write(student + "," + extractConstructs(content) + "," );
```

Dynamic program analysis is performed using test-case based approach. The tool embeds a script that extracts the output from java compiler, summarizes the compile time and runtime outcomes. A snippet of the script is shown below.

```
javac -classpath ${i%%}/Q2  ${i}/Q2/Q${QN}Tester.java 2> ${i}/Q2/op-CPerror${QN}.txt
        if [[ -s ${i}/Q2/op-CPerror${QN}.txt ]] ; then
                # error=`tr '\n' ' ' < ${i}/Q2/op-CPerror${QN}.txt`
                errtype=`./error-extractor.sh 1 /${root}/${i}/Q2/op-CPerror${QN}.txt`
                errline=`./error-extractor.sh 2 /${root}/${i}/Q2/op-CPerror${QN}.txt`
                summary=${summary},0,${errtype},${errline},-,-,-
```

Feature Selection Stage: Features for each student data point includes; methods such as *length*, *charAt* etc. and controls such as *if*, *for*, *while* etc. All the features are then represented as a matrix which is suitable for classification models. Decision trees generates a list of releavant features for programming assignment and their corresponding importance. The features with low importance are removed from the data points to improve the performance of the Clustering algorithm.

Clustering Stage: We use Jaccard Coefficient (Maimon et al. 2009) to measure the similarity between two codes, a popular technique for text mining tasks. Finally, hierarchical clustering algorithm identifies common features on the codes and clusters the codes into clusters.

Visual Outputs: The outputs of all the three stages are shown in a comprehensive view using html pages. Students' information, clusters, code structures and errors are represented in a table. The instructors will be able to use the comprehensive information and provide more meaningful feedback on the codes to the students.

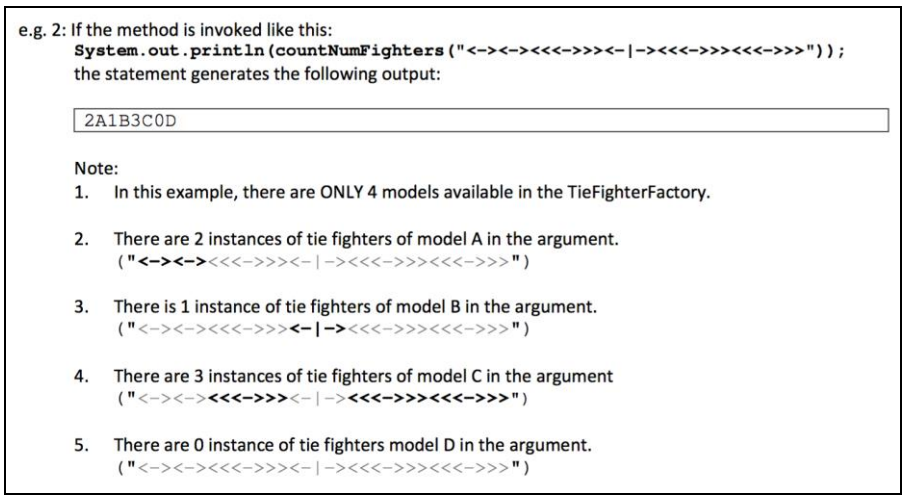
4. Experiments

4.1 Data Sets

The tool is tested on foundation course, IS Software Foundations, from School of Information Systems in Singapore Management University. Seven sections, G1 to G7 are run concurrently for year 1 students.

The course instructor provided 261 code submissions from the students, as well as a list of test cases for a particular programming problem as inputs to our tool. These test cases are reasonably sufficient to verify the correctness of the student's code submission, as these are the same test cases used to grade the students' code submissions.

Programming assignment: Given a combined string representation that aggregates several string patterns, students are required to write a static method called "countNumFighters", which counts the number of occurrences of each *pattern within the string*. "TieFighterFactory" is the testcase with four models, A-B, with different string patterns. Figure 2 explains an example test case of the problem, and how the method output was derived. Table 1 shows the statistics of the codes in which out of 261 submissions, 224 codes failed the test cases.


e.g. 2: If the method is invoked like this:
`System.out.println(countNumFighters("<->-><->-><<<->>><-|-><<<->>><<<->>>"));`
the statement generates the following output:

```
2A1B3C0D
```

Note:
1. In this example, there are ONLY 4 models available in the TieFighterFactory.
2. There are 2 instances of tie fighters of model A in the argument.
   ("**<–><–>**<<<–>>><–|–><<<–>>><<<–>>>")
3. There is 1 instance of tie fighters of model B in the argument.
   ("<–><–><<<–>>>**<–|–>**<<<–>>><<<–>>>")
4. There are 3 instances of tie fighters of model C in the argument
   ("<–><–>**<<<–>>>**<–|–>**<<<–>>><<<–>>>**")
5. There are 0 instance of tie fighters model D in the argument.
   ("<–><–><<<–>>>><–|–><<<–>>><<<–>>>")

Figure 2: Sample method call and the corresponding output.

Table 1: Statistics of Code Submission.

	Total	Verified correct	Verified incorrect
No of Code Submissions	261	37	224

4.2 Dynamic and Static Program Analysis

Dynamic Analysis: The tool evaluated the correctness of each student submission based on a list of test cases in the data set. During the evaluation, outcome of java compiler and java executer are recorded. In cases where the code execution reaches a runtime of more than 30 seconds, the tool will terminate the program and record the outcome with a runtime error of "timeout". The tool is accurate in determining the correctness of the code which will be discussed in next sub section.

Static Analysis: After recording the dynamic program analysis results, the tool executes the static analysis on the code. The tool extracts method calls, object instantiation and control structures such as for-loops, enhanced for loops, while loops, do-while loops, if-else statements and try-catch statements. For each student submission, these static analysis outcomes are also recorded. Figure 3 shows the features extracted from the program analysis stage. We observe that some codes fail to compile while others fail in correctness. For codes which were successfully compiled, the code structures such as "if controls", "while controls" etc., and methods such as "length", "equals" etc., are extracted for each code submission. These features are not only useful for clustering, but also helps in providing relevant feedback by the instructor.

	ID	COMPILE	CORRECT	CTRLS	METHODS
0	G1/[redacted].2014	0	0		
1	G1/[redacted].2015	1	1	for{}if{if{}for{for{if{if{...	new TieFighterFactory()//g...
2	G1/[redacted].2015	0	0		
3	G1/[redacted].2015	0	0		length//
4	G1/[redacted].2015	1	1	if{}for{dowhile{if{}}}	new TieFighterFactory()//g...
5	G1/[redacted].2012	1	0	for{if{}else{while{while{}...	new TieFighterFactory()//g...
6	G1/[redacted].2015	1	0	if{}while{if{if{}if{}if{}i...	length//length//equals//eq...

Figure 3: Sample features for each student on correctness (dynamic) and structure (static) of code.

Further at runtime, Java compiler outputs the error messages which are useful for clustering and feedback. Figure 4 shows some sample dynamic analysis outcomes.

cannot find symbol	non-static method retrieve(char) cannot be ref...	bad operand types for binary operator '<'
missing method body or declare abstract	';' expected	missing return statement

Figure 4: Sample errors generated from program analysis

The outcomes from dynamic and static program analysis serve as features to the clustering stage of the tool. We first convert the features into a matrix format and apply classification approach for feature selection. Using decision trees, we shortlisted 35 features from the original 746 features in order to control the number of dimensions.

4.3 Clustering Results

Hierarchical clustering algorithm generated clusters of similar codes and we used elbow method to determine the number of clusters. We observed in our preliminary experiments that the cut off distance of 5 maximizes the average Jaccard similarity within each cluster and derives eight clusters in total.

To accurately evaluate the performance of our methodology, we manually review each student's code submission to identify areas of feedback. After that, we simulate the process of feedback generation based on the cluster labels from clustering output, in these 2 steps:

1. Identify a common feedback for each cluster in terms of static and dynamic program analysis.
2. Evaluate if feedback given is appropriate for each code submission (Yes / No).

Table 2 shows the evaluation results of the clusters generated by the tool. The tool identified 90% of codes for common feedback and 84.4% are classified into correct clusters for similar feedback.

Table 2: Results of feedback generation from clustering stage.

No. of Submissions clustered	Identified common feedback	% identified	Gave appropriate feedback	% Total
224	203	90.63%	189	84.38%

Figure 5 shows the sample output of cluster 2 generated by the tool for the instructor. The controls shows that these students commonly used "for loops" and "if loops" in the codes. They failed to use the methods such as "length" and "equals" which are critical for this programming assignment. The instructor can now draft a feedback based on these two observations along with the test-case outputs. Therefore, the common features on code structure aids the instructor to draft the feedback to the students for corrections and code structure improvements. We observe that student, "118", is wrongly clustered. We discuss the improvements to the tool in the next section.

```
ares['CLUSTER'] = clusters
ares[['CLUSTER','ID','CTRLS','METHODS']][ares.CLUSTER == 2]
```

	CLUSTER	ID	CTRLS	METHODS
11	2	G1/[redacted].2015	for{if{}}	new TieFighterFactory()//getAllAvailableModels...
18	2	G1/[redacted].2015	for{}	new TieFighterFactory()//getAllAvailableModels...
35	2	G2/[redacted].2015	for{}	new TieFighterFactory()//getAllAvailableModels...
61	2	G3/[redacted].2015	for{if{}}	new TieFighterFactory()//getAllAvailableModels...
82	2	G3/[redacted].2015	for{while{}}	new TieFighterFactory()//getAllAvailableModels...
104	2	G4/[redacted].2015	for{}	new TieFighterFactory()//getAllAvailableModels...
118	2	G4/[redacted].2015	if{}if{}for{while{}}	new TieFighterFactory()//getAllAvailableModels...
131	2	G5/[redacted].2015	for{if{}}	new TieFighterFactory()//getAllAvailableModels...

Figure 5: Sample visual output for cluster 2.

4.4 Discussions

We observe that there is an area for improvement for the clustering algorithm to better assist the instructor in feedback generation. We also observed that some codes are incorrectly clustered and one of eight clusters cannot be labelled coherently due to lack of common features. A proposed method would be to generate more clusters to improve homogeneity in each cluster. Another approach is to study other clustering techniques such as agglomerative and k-means and feature selection techniques.

One limitation of our clustering methodology is that it produces hard clustering outputs, as opposed to soft clustering. In hard clustering, we see that each code submission is a member of exactly one cluster. In contrast, in soft clustering technique, a submission may have fractional membership in several clusters, which could be more helpful in generating more than one feedback point for the same code submission. It is suggested that the hidden Markov model and fuzzy c-means algorithms would be useful areas to explore should we venture into this scope in the future. We also study on extending our solution to a fully automated feedback tool where the model is trained on instructors' feedback. Further, applying analytics on student codes can aid the instructors in discovering common challenges in programming learning process.

5. Conclusion

In this paper, we presented a feedback tool for introductory programming course code assignments. The tool aids instructors provide specific feedback in an effective manner by discovering the similar codes and clustering them into groups. It can be seen from this study that the use of automated program analysis, feature selection techniques and clustering algorithms can facilitate the process of effective feedback generation. The instructor uses visual outputs for each generated cluster to provide relevant feedback text to be propagated to all submissions within that cluster.

References

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments, *Computer Science Education*, 15 (2), pp. 83-102.

Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003). On automated grading of Programming Assignments in an academic institution. *Computers & Education*, 41, 121-131.

Glassman, E. L., Scott, J., Singh, R., and Miller, R. C. (2015) Overcode: visualizing variation in student solutions to programming problems at scale. *In Proceedings of the adjunct publication of the 27th annual ACM symposium on User interface software and technology*, 129–130, 2015

Glassman, E. L., Singh, R., and Miller, R. C. (2014). Feature engineering for clustering student solutions. *In Proceedings of the First ACM Conference on Learning @ Scale Conference*, ACM (New York, NY, USA).

Gottipati, Swapna and Shankararaman, Venky. Learning Analytics Applied to Curriculum Analysis. (2014). *In proceedings of the International Conference on Informatics Education and Research*. (SIGED IAIM). 2014.

Higgins, C., Hergazy, T., Symeonidis, P., & Tsinsifas, A. (2003). The CourseMarker CBA system: Improvements over Ceilidh. *Education and Information Technologies*, 8, 287 – 304.

Irena Koprinska, Joshua Stretton, and Kalina Yacef. (2015). Students at risk: Detection and remediation. *The 8th International Conference on Educational Data Mining*, 2015.

Joy, M., Griffiths, N., & Boyatt, R. (2005). The BOSS Online Submission and Assessment System, *ACM Journal on Educational Resources in Computing*, Vol. 5, No. 3, Article No. 2

Lane D. (2004). Junit: The Definitive Guide. *O'Reilly*, Sebastopol , CA.

M. D. Ernst. (2003). Static and dynamic analysis: synergy and duality. *In Proceedings of WODA'2003 (ICSE Workshop on Dynamic Analysis)*, Portland, May 2003

Maimon, Oded, and Lior Rokach. (2009). Introduction to knowledge discovery and data mining. *Data mining and knowledge discovery handbook*. Springer US, 2009. 1-15.

Milena Vujošević-Janicić and Viktor Kuncak. (2012). Development and evaluation of LAV: an SMT-based error finding platform. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, LNCS, 2012

Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. (2013). Software verification and graph similarity for automated evaluation of students' assignments. *Inf. Softw. Technol.* 55, 6 (June 2013).

Shankararaman, Venky and Gottipati, Swapna. Aligning Assessments with Competencies using Keyphrase Extraction. (2014). *In proceedings of Teaching, Assessment, and Learning for Engineering (TALE)*. 2014.