Computer Science Education

ISSN: 0899-3408 (Print) 1744-5175 (Online) Journal homepage: https://www.tandfonline.com/loi/ncse20

Using automatic machine assessment to teach computer programming

Phil Maguire, Rebecca Maguire & Robert Kelly

To cite this article: Phil Maguire, Rebecca Maguire & Robert Kelly (2017) Using automatic machine assessment to teach computer programming, Computer Science Education, 27:3-4, 197-214, DOI: 10.1080/08993408.2018.1435113

To link to this article: https://doi.org/10.1080/08993408.2018.1435113

View supplementary material

Published online: 07 Feb 2018.

Submit your article to this journal

Article views: 311

View related articles

View Crossmark data

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=ncse20

COMPUTER SCIENCE EDUCATION, 2017
VOL. 27, NOS. 3–4, 197–214
https://doi.org/10.1080/08993408.2018.1435113

Routledge
Taylor & Francis Group

Check for updates

Using automatic machine assessment to teach computer programming

Phil Maguire[a], Rebecca Maguire[b] and Robert Kelly[a]

[a]Department of Computer Science, National University of Ireland, Maynooth, Ireland; [b]Department of Psychology, National University of Ireland, Maynooth, Ireland

ABSTRACT

We report on an intervention in which informal programming labs were switched to a weekly machine-evaluated test for a second year Data Structures and Algorithms module. Using the online HackerRank system, we investigated whether greater constructive alignment between course content and the exam would result in lower failure rates. After controlling for known associates, a hierarchical regression model revealed that HackerRank performance was the best predictor of exam performance, accounting for 18% of the variance in scores. Extent of practice and confidence in programming ability emerged as additional significant predictors. Although students expressed negativity towards the automated system, the overall failure rate was halved, and the number of students gaining first class honours tripled. We infer that automatic machine assessment better prepares students for situations where they have to write code by themselves by eliminating reliance on external sources of help and motivating the development of self-sufficiency.

ARTICLE HISTORY
Received 13 October 2017
Accepted 29 January 2018

KEYWORDS
Programming instruction; programming confidence; automatic correction; skill development; constructive alignment; automatic feedback

Introduction

Many students taking computer science (CS) find programming very challenging, with up to a quarter dropping out and many others performing poorly (Fowler & Yamada-F, 2009; Peters & Pears, 2012; Williams & Upchurch, 2001). The high variability of students' backgrounds typically found in introductory programming courses can undermine some students' motivation, and make it more difficult to ensure the desired competency and retention rates (Barros, Estevens, Dias, Pais, & Soeiro, 2003). As well as leading to significant attrition at university level, the perception of CS as a "difficult" subject may also discourage students from choosing to study it in the first place (Bennedsen & Caspersen, 2007).

CONTACT Phil Maguire pmaguire@cs.nuim.ie
The supplemental data for this article is available online at https://doi.org/10.1080/08993408.2018.1435113.

© 2018 Informa UK Limited, trading as Taylor & Francis Group

A number of studies have attempted to explain the cause of such high failure rates (e.g. Bergin, Mooney, Ghent, & Quille, 2015). One pertinent factor may be the way in which programming is taught. Like other disciplines that require procedural knowledge, programming is best learned through practice and experience (Traynor & Gibson, 2004). Students' lack of fundamental problem-solving skills have been identified as one of the main reasons for attrition and weak programming competency (Beaubouef, Lucas, & Howatt, 2001; Thweatt, 1994). Unfortunately, textbooks and lecture material in CS are often heavy on declarative knowledge, with particular emphasis on the features of programming languages and how to use them (Robins, Rountree, & Rountree, 2003). Changes to teaching methods, such as the use of clearer textbooks and the introduction of online resources, have done little to improve programming competence (Miliszewska & Tan, 2007).

Although programming is a practical skill, the opportunities provided for practice are often insufficient (Hawi, 2010). The best means of implementing the practical component of such modules remains a contentious issue (Linn & Dalbey, 1989; Maguire & Maguire, 2013). Research has shown that students must be active participants in the learning process in order for deep learning to occur (Mayer et al., 2009). Knowledge must be put into practice in order for misunderstandings to rise to the surface where they can be challenged and corrected (McKeachie, 1999). According to Trees and Jackson (2007), the ideal learning environment should involve mastery-oriented feedback, choice-making opportunities, and the chance for students to evaluate their own learning. In the case of developing skills in programming, this implies tackling open-ended questions which require creative thinking, and getting prompt objective feedback on what works and what doesn't work.

A particular problem with CS coursework, given the group-based setting in which labs are typically conducted, is that work may be shared and copied with very little effort (Fraser, 2014). Rather than have to solve a difficult problem independently, it can sometimes be easier to paste in a solution that somebody else has developed. For instance, Roberts (2002) reviewed incidents of dishonesty at Stanford University over a decade, and found that 37% of all incidents were attributed to CS courses, despite the fact that these students represented less than 7% of the student population. In a programming lab environment, some students may realize that directing energy towards obtaining code from others leads to greater payoffs than actually attempting the problem themselves; they confess to cheating simply because they are "lazy" (Dick et al., 2003; Sheard, Carbone, & Dick, 2003; Wilkinson, 2009). According to Fraser (2014), unlimited collaboration increases the amount of copying that takes place, and thus damages the average student's learning experience.

Programming anxiety, which represents an important predictor of achievement in CS modules, may also play a role in discouraging students from attempting to program independently. Students can find learning programming intimidating, giving rise to lack of confidence and loss of self-esteem. Connolly, Murphy, and

Moore (2009) report that the best approach to breaking the cycle of anxiety is to change the way students think, focusing on the development of rational skills, which can be used to deal with all computer programming. Barros et al. (2003) identify two obstacles to the development of such skills, namely, an excessive dependency on group work, and insufficient assessment opportunities, leading to fraudulent behaviour. In light of these obstacles, Barros et al. radically modified the assessment and grading system of a CS module, with the objectives of increasing programming practice, decreasing fraud and dependency on others, and decreasing student drop out. Barros et al. found that switching to a regular lab-exam paradigm greatly enhanced competency, programming confidence, and lowered the drop-out rate. Students preferred the new system over the previous open lab group assignments, perceiving it as fairer and more relevant to the exam. Knorr and Thompson (2017) also investigated the use of regular lab programming tests, finding that it enhanced confidence in programming, albeit without impact on the final exam grade.

Teaching programming in Maynooth University

The current system employed in Maynooth University for teaching Data Structures and Algorithms (an intermediate level programming module) is that students attend two hours of lectures per week, followed by two hours of labs where they put into practice what they have learned. Thirty per cent of the module mark is awarded for work carried out in the labs, with the remaining 70% awarded for the end of semester paper-based written examination. In previous years students were given programming questions during the week, which they would complete in the lab in an open setting. Demonstrators would quiz students on their work at the end of the lab, and award marks appropriately.

Over the years a number of potential drawbacks of this system became apparent from demonstrators' reports on student behaviour. For example, not all of the weekly lab exercises were completed independently by students. Informal feedback from demonstrators suggested that, when asked to explain their code, students appeared to have learned off a script from which they were unable to deviate. Another problem was that students relied heavily on demonstrators' input to complete their exercises. Despite only a small fraction of the class being capable of writing computer programs independently, most successfully completed the labs and earned the marks. As such the correlation between CA mark and programming exam mark was low.

Arguably, there is little point learning off facts about data structures and algorithms if one does not have the ability to put that knowledge into practice through programming. Maynooth students coming out of first year CS have basic programming experience of writing simple programs. However, because an essential aspect of programming involves the deployment of specialized data structures and algorithms, their abilities are still developing. Most of the learning outcomes

for the Data Structures and Algorithms module make direct reference to applied knowledge, such as (1) identifying data structuring strategies appropriate to a given context, (2) designing, developing and testing programs, and (3) applying data structuring techniques to the design of computer programs. In order to address the misalignment between learning outcomes and ultimate evaluation, the exam format in 2015 was changed so that the entire paper involved reading and writing computer code and nothing else. The new-style exam featured five fully programming-based open-ended questions (see supplementary material – exam questions). The marks were very poor, suggesting that, after two years studying computer science, many students were not able to program solutions to even relatively simple problems. These results motivated us to seek better alignment between the learning outcomes, lab activities and the final exam for future years.

Porter, Guzdial, McDowell, and Simon (2013) discuss several different enhancements that can improve programming skills, such as peer instruction and pair programming. For example, peer instruction modifies a standard lecture to revolve around several questions, in which students can discuss the problem in groups and then answer using clickers. Porter et al. (2013) report that students taking computer science in peer instruction classes experience a 61% reduction in failure rates, and outperform students given traditional lectures by 5% on identical final exams.

We previously found that the use of clicker questions in lectures within this module greatly raised attendance, with students reporting increased attention during lectures. However, this did not lead to increased performance in final exam grades (Maguire & Maguire, 2013). This may have been because the questions asked in lectures were verbal and conceptual, and did not involve directly programming a machine. Again, the questions students were being asked in lectures were not well aligned with the learning outcomes.

Another form of intervention which may be helpful for developing programming skills is "paired programming". This involves two people working together at a workstation, one of them as the "driver" and the other as the "observer". The assumption is that students should learn from the collaboration, with promising results reported in the literature (Porter et al., 2013). For example, McDowell, Werner, Bullock and Fernald (2002) found that more students passed in the pairing sections (72%) vs. the solo sections (63%), were more likely to continue on into the next programming course (85% vs. 67%), and were more likely to have declared a CS major one year later (57% vs. 34%).

Nevertheless, when we introduced paired programming to our module, the initiative failed to enhance programming performance (Maguire, Maguire, Hyland, & Marshall, 2014). Upon investigation, weaker students appeared to rely on the knowledge of stronger classmates, and did not get the chance to experience the process of resolving difficulties themselves. Pairing students with others of similar ability may have mitigated this effect. For example, Braught, MacCormick, and Wahls (2010) compared random pairings vs. pairing by ability and found that the

lowest-quartile students who were paired by ability performed better than those who were paired randomly and those who worked alone.

Based on our previous experiences, we identified the following working assumptions regarding our teaching of computer programming, echoing closely those of Barros et al. (2003):

(1) Raising lecture attendance or lab attendance *per se* is not sufficient to improve programming performance when activities are not aligned with learning outcomes
(2) Allowing weaker students to rely on the work of stronger peers can be damaging for developing practical programming skills, because it prevents the them from confronting their own lack of knowledge
(3) Putting text-based questions on the exam can be damaging for developing practical programming skills, because it allows students to earn marks without programming, thus reducing their motivation to develop the skill during the module

In essence, the problem was that the content being examined in the final exam was not aligned with the way in which the lab practicals were being run. Constructive alignment (Biggs, 1996) is the idea of systematically aligning teaching methods and assessment through the medium of constructivist teaching, where learners are actively involved in the process of meaning and knowledge construction, as opposed to passively receiving information. According to this paradigm, students learn best when they are allowed to construct a personal understanding of the assessment targets based on experiencing things and reflecting on those experiences. Shaffer and Resnick (1999) hypothesize that different forms of "authentic" learning can be blended together in learning environments that provide "thick authenticity", enabling simultaneous creation of personal (aligned with what learners want to know), real-world (aligned with the expectations of industry), disciplinary (aligned with the academic discipline) and assessment authenticity. Inspired by this approach, we identified the following action points to actively encourage students to develop "thickly authentic" programming skills:

(1) All assessment in the module should be programming-based so that the only way to pass is by successfully programming, thereby directly addressing the module learning outcomes, and the expectations of industry and academia
(2) Students must take full personal responsibility for their own ability to program, coding independently using their own ideas, and obtaining immediate, objective feedback throughout the semester

Identifying similar action points, both Barros et al. (2003) and Knorr and Thompson (2017) switched to regular lab exams, which were corrected manually. For example, Barros et al. used a grading system for labs of 0 to 3, with partial marks awarded for following style rules and being "near to the correct solution". This

grading scale, however, is subjective and therefore requires human intervention, drawing down significant labour from teachers, and also raising the spectre of students haggling for marks. Daly and Waldron (2004) argue that students should not be given any marks for programs that do not work, because it disincentivizes students from seeking successful problem-solving skills. Embracing the concept of constructive alignment, we sought to develop a paradigm whereby students could discover the importance of issues such as style rules by themselves, rather than having subjective rules imposed by fiat. We also sought to leave the approach to the problem completely open, awarding marks only for the correct output and nothing else, thus allowing students to engage their creativity, make mistakes and discover efficient solutions for themselves, rather than handing them the answer.

For example, presenting programming problems as game-type challenges encourages students to develop skills by engaging them more deeply in the material (Jiau, Chen, & Ssu, 2009). Lawrence (2004) reports that allowing students to evaluate and improve their programs to achieve competitive success is also a significant motivator for increased performance. Students work harder and are more interested in programming when they have creative control over solving a well-defined, though interesting problem.

Most of all, we wished to have the whole process automated. Systems which assess code automatically are gaining popularity within computer programming courses (Rubio-Sánchez, Kinnunen, Pareja-Flores, & Velázquez-Iturbide, 2014). A variety of different systems have been developed and deployed for teaching in the past (see Douce, Livingstone, & Orwell, 2005; Ihantola, Ahoniemi, Karavirta, & Seppälä, 2010; for a comprehensive review). For example, CodeWrite is a web-based tool that gives students responsibility for developing exercises which are then shared with their classmates, a pedagogical approached known as "constructive evaluation" (Luxton-Reilly & Denny, 2010). This means that students are exposed to a variety of solutions to a problem that they have also solved, showing them that there are multiple correct solutions, some of which may be more succinct and easier to understand than their own (Denny, Luxton-Reilly, Tempero, & Hendrickx, 2011). Web-CAT, the Web-based Center for Automated Testing, is another widely used open-source automated grading system that provides rapid, directed comments on students' work. Supporting a wide variety of programming languages and assessment strategies, it encourages students to write software tests for their own work, giving them the responsibility of demonstrating the correctness and validity of their own programs (Edwards & Perez-Quinones, 2008).

In this study, we describe the use of a recently developed online system which met all of our desiderata with minimum overhead, namely HackerRank. HackerRank's principal advantage over other automatic grading systems is that it is used in practice by industry to recruit programmers and run programming competitions, thus potentially providing the closest alignment with the learning outcomes of a programming course and the expectations of industry.

Overview of HackerRank

HackerRank is a company that focuses on competitive programming challenges for consumers, educators and businesses, boasting an online community of over one million computer programmers (Kosner, 2014). Its programming challenges can be solved in a variety of programming languages (including Java, C++, PHP, Python, SQL), and span many areas of CS. When a programmer submits a solution to a programming challenge, the submission is scored on the accuracy of the output and the execution time of the solution.

HackerRank for Work is a subscription service offered by HackerRank that aims to help companies source, screen, and hire engineers and other technical roles. HackerRank also provide the same service to educators for free. The platform gives users the option to avail of a built-in library of programming challenges, or to write their own. Candidate's solutions, once submitted, are automatically scored and the results are then provided to the technical recruiter or educator for review.

HackerRank provided Maynooth University with an unlimited allocation of free invitations, allowing instructors to email tests out to students in the class on a weekly basis. Given that the platform supports a wide range of programming languages, it immediately cut the tie of the module to a particular language (which had previously been Java based). The platform also has the advantage of being run through a webpage, meaning there was no need to install any software to allow students to avail of the different languages. Furthermore, the HackerRank website could easily be displayed in lecture theatres, with the solution programmed by the lecturer live. Given that there was no need to install specialized software on lecture theatre machines, this saved the lecturer from having to bring a laptop.

The way the programming challenges work is that the test designer specifies an input–output function (see supplementary material – topics covered). The designer provides a textual description of what the function should do, with sample cases given of input mapping to some output. The designer also inputs hidden test cases, which students do not see. In labs, students receive an email which they click on to bring them to the HackerRank website. A timer then starts ticking down, informing them how much time they have left. When students have written their program they hit RUN, and find out immediately how many of the hidden outputs their program has managed to produce. If they have identified all of them correctly, they can submit their code and get full marks. If not, they are free to go back and edit the program and try to produce more of the outputs. A "code stub" can also be provided in the question, which means that students need only add a few methods into an existing data structure to get it working.

In sum, HackerRank provides choice-making opportunities, because it allows students to approach a problem from any angle, the only requirement being to produce the appropriate output. It also provides students with the chance to evaluate their own learning by providing immediate objective information on

programming ability each week, which is constructively aligned with the final exam assessment.

Objectives of intervention

The goal of our intervention was to ascertain, first, whether HackerRank could be successfully used to run all the labs covering the various topics in the module, and more importantly, to find out whether this form of regular automated testing of students would lead to improved outcomes, as measured by performance on the final exam. To this end, we aimed to evaluate the validity of HackerRank performance for predicting exam performance after controlling for a number of known associates of programming performance.

Method

Participants

This study was carried out with a second year undergraduate class in the Maynooth University taking the module in Data Structures and Algorithms. There were a total of 230 students enrolled in the module, studying for a range of degrees, such as the degree in Computer Science and Software Engineering, the degree in Computational Thinking, the HDip in Information Technology, as well as degrees in Science, Arts and Multimedia.

The Computational Thinking and HDip students had taken an intensive pre-semester course in Java programming, while everybody else had completed two first year modules in Java programming. It was therefore assumed that on entry the students had introductory knowledge of programming, though without the experience of applying it to the development of data structures and algorithms.

Initial questionnaire

In a meta-review, Watson and Li (2014) found a global pass rate of 68% for introductory programming classes, without any substantial variation over time or according to country, grade level, class size or programming language used. Building on this, Bergin et al. (2015) carried out investigations as to the factors that predict programming performance at the level of the individual (see also Watson & Li, 2014, for a review of global introductory programming pass rates). After developing numerous models, they found that three significant factors emerged, namely final mathematics examination result prior to university entry, the number of hours spent playing computer games and programming self-esteem. In light of these findings, we invited students to complete a background questionnaire which provided sociodemographic and prior academic information including gender, total CAO points achieved in their Leaving Certificate (which is taken around the age of 18 and is typically used to determine entry to university courses), and Math points

achieved in the Leaving Certificate. Both of these variables were taken as measures of prior academic ability. Total CAO points can range from 0 to 625, while Maths points can range from 0 to 125, with higher scores indicating higher achievement prior to university entry. In addition, students indicated whether they had any prior programming experience. Since most of these students had already undertaken programming modules, it was expected that the majority would have at least one year's experience. Therefore, for the purpose of analysis, this variable was recoded into those who had low (0–1 year) and high (2 years and above) experience with programming.

Students were also asked to rate their level of confidence in their programming ability. This was assessed using a single item, where students rated their agreement on a scale of 0–10, with higher scores indicating higher levels of confidence or self-efficacy.

Finally, given Bergin et al.'s (2015) findings, students indicated how many hours per week they engaged in social media usage and gaming per week. Due to non-normal distribution of these variables, the data was classified into tertiles. In the case of gaming, participants were reclassified as either spending no time gaming per week (i.e. 0 h) and then a median split was employed to categorize the remaining participants into those engaging in low and high levels of gaming. Since only a handful of participants indicated that they did not engage in social media usage, this variable was split into equal thirds whereby participants were classified as having either a low, mid or high level of usage.

Module engagement

Two measures provided information on students' level of engagement with the module. First, levels of attendance at weekly labs were recorded (totaling 11 weeks). Second, students were sent out a set of practice questions before the exam and the number of questions they attempted was recorded. Based on these data, students were classified as to whether they had engaged in no practice or some practice.

Lab procedure

The module entailed weekly practical lab-based sessions. Students were informed that each week there would be a question relevant to the material covered in lectures, but they were not told what the question would be. Each lab lasted for two hours. In the first hour, students were given time to practice and to ask the demonstrators questions about the code they had written. Sometimes the demonstrators would give brief tutorials at the white-boards, giving the students information they might need in the lab. In the second hour, the students did the HackerRank test in silence. Each student was sent out an email, with a link which they clicked to bring them to the test on the HackerRank website. The HackerRank test was open book, meaning that students were allowed to bring in any material they wanted.

They were also free to access programming language websites, (e.g. https://docs.oracle.com/ for Java; https://www.haskell.org/hoogle/ for Haskell), and all of the lecture notes and past solutions on the module website. Five minutes into the test, web access to any other sources than these was taken down, thus minimizing the potential for students to email solutions to each other.

The students had one hour to complete the test. If they did not hit the submit button before one hour, then the code on their screen was automatically submitted after the hour had elapsed. Each question involved ten input-output pairs, for which students had to code up an appropriate function. Each test case matched earned 10% of the marks for that lab. Usually, only 8 of the test case outputs were hidden, meaning that two could be hard-coded to earn 20% of the marks for the lab (e.g. if input = "15 18 3", then output "3"; see supplementary materials – topics covered). For this reason, everybody attending the lab would, in theory, get at least 20%. Although students could resubmit their work as many times as possible, they were not able to run the 8 hidden test cases until passing both of the two visible ones, thus reducing the motivation to "hack" their way to a correct solution through trial and error (see Ihantola et al., 2010). Buffardi and Edwards (2015) report that, while automated graded systems provide students with prompt feedback, they may inadvertently discourage students from thinking and testing thoroughly, instead encouraging dependence on the instructor's tests. In light of this, we sought to deliberately include a selection of "curveball" test cases that would thoroughly test the robustness of any submitted program (e.g. zero inputs, defective inputs, negative inputs, large inputs).

Following completion of the lab, all the marks could be easily downloaded into an excel spreadsheet from the HackerRank website, saving great effort. This contrasted with previous years, when all the data had to be entered manually. The marks were posted up within minutes, so students could see what mark they got. Students were uniquely identified by their email addresses, and by the student numbers they were asked to enter into the HackerRank system when accepting an invitation. Demonstrators verified who was in the lab in the first hour, so marks were only given to those who attended, and not those who might have completed the test at home. All labs were weighted equally in determining final CA (see supplementary materials – topics covered). In the spirit of constructive evaluation (see Luxton-Reilly & Denny, 2010), a range of the top solutions submitted by students were uploaded online and discussed in lectures.

Statistical analysis

A hierarchical regression analysis with four blocks of factors was conducted to establish the relationship between students' HackerRank score and their final exam mark, after controlling for nine additional predictor variables. The first block of factors measured sociodemographic and academic characteristics prior to course commencement (gender, CAO points, maths score, programming experience).

The second block of factors measured self-efficacy and engagement in gaming and social media (confidence in programming, hours gaming per week and social media usage per week), while the third block of factors identified behavioural engagement with the module (attendance and practice) and, finally, the fourth block examined the role of the HackerRank score. Prior to analysis, descriptive statistics were calculated and preliminary analysis was conducted in order to ensure that no violations in the assumptions of normality, linearity and homoscedasticity were observed. Correlations between the predictor variables were also examined to ensure no problems with multicollinearity.

Results

Descriptive statistics

Descriptive statistics for both the categorical and continuous variables in our study are presented in Table 1. As can be seen here, the vast majority (84%) of students were male. Most students reported between 6 and 10 h of social media usage per week and between 1 and 9 h of gaming per week. Only 19% of students had two years or more programming experience, with most only having experience based

Table 1. Descriptive statistics for continuous variables in study.

Categorical variables	No.	%	Valid %
Gender			
Male	177	83.5	83.5
Female	35	16.5	16.5
Missing	0	0	
Programming experience			
One year or less	172	81.1	81.1
Two years or more	40	18.9	18.9
Missing	0	0	
Amount of gaming per week			
0 h	47	22.2	23.2
1–9 h	80	37.7	39.6
10 h+	75	35.4	37.1
Missing	10	4.7	
Social media usage per week			
0–5 h	68	32.1	33.0
6–10 h	70	33.0	34.0
11 h+	68	32.1	33.0
Missing	10	2.8	
Engaged in any practice			
No practice	181	85.4	85.4
At least some practice	31	14.6	14.6
Missing	0	0	
Continuous variables	Mean	SD	Range
Total points	424.67	67.13	95–595
Maths points	58.27	19.24	0–100
Confidence in programming ability	5.88	1.91	0–10
Attendance	9.30	2.21	1–11
Hackerrank score (CA)	19.98	6.50	1.5–30
Exam score	58.42	28.41	2.50–100

on their first year in CS. Only 15% of students engaged in HackerRank practice prior to the exam. Examination of the continuous variables revealed that, while generally reporting a high attendance, students exhibited variability in their levels of confidence, their HackerRank score and their exam grades. The mean exam score of 58% was an improvement on previous years (see Table 2).

Regression analysis

Table 3 displays the results of the regression analysis. All blocks of factors contributed significantly to the model, with block 1 (sociodemographic and academic factors) contributing 17% of the variance in exam scores ($p < .01$), block 2 (confidence, social media and gaming) contributing a further 13% of the variance ($p < .01$), and block 3 (behavioural engagement) contributing 14% of the variance ($p < .01$). Block 4 (HackerRank performance) contributed the greatest amount of variance at 18% ($p < .01$). As a whole, the model was significant ($F(10, 168) = 28.05$; $p < .001$) and successfully explained 59% of variance in final exam scores. Three

Table 2. Comparison of performance between 2015 (demonstrator correction) and 2016 (machine correction) cohorts.

	2015	2016
N Sitting Exam	221	230
Average Mark	36.6%	56.9%
Failure Rate	60.6%	31.7%
First Class Honours Rate	13.1%	39.1%
Correlation between exam and CA (*r*)	0.35	0.64

Table 3. Hierarchical Regression Analyses for Variables Predicting Exam Score.

Variables	*β*	*p*	*t*	*B*	SE	CI95%	
Block 1: Sociodemographic and academic factors							
Gender [0 = female; 1 = male]	0.04	0.51	0.66	2.84	4.27	−5.59	11.26
CAO points	0.08	0.26	1.13	0.03	0.03	−0.03	0.10
Maths points	0.06	0.44	0.77	0.09	0.11	−0.14	0.31
Programming experience [0 = 0–1 years; 1 = 2+ years]	−0.05	0.28	−1.09	−3.98	3.64	−11.16	3.20
R^2 Change = 0.17							
Block 2: Confidence, social media and gaming usage							
Gaming usage per week	−0.03	0.63	−0.48	−0.99	2.08	−5.11	3.12
Social media usage per week	−0.07	0.16	−1.40	−2.49	1.78	−6.01	1.03
Confidence	0.13*	0.03	2.26	2.00	0.89	0.25	3.75
R^2 Change = 0.12							
Block 3: Behavioural engagement							
Attendance	−0.12	0.10	−1.64	−1.49	0.91	−3.29	0.30
Practice	0.11*	0.03	2.21	8.75	3.97	0.93	16.58
R^2 Change = 0.14							
Block 4: Hackerrank							
Hackerrank score (CA)	0.72***	0.00	8.85	3.15	0.36	2.45	3.86
R^2 Change = 0.18							
R^2 = 0.59							

Statistical significance:*$p < .05$;**$p < .01$;***$p < .001$.

predictors independently contributed to the model which, in order of strength, were HackerRank score ($\beta = .72$, $p < .001$), confidence in programming ability ($\beta = 0.13$; $p < .05$), and HackerRank practice ($\beta = .11$; $p < .05$). These results suggest that those scoring higher in HackerRank, who had a greater confidence in their programming ability, and who opted for a greater amount of practice, did better in their exams scores.

In sum, HackerRank lab performance was demonstrated to be a highly accurate predictor of performance in a programming-only exam, thus supporting the hypothesis that automatic machine evaluation is an appropriate, constructively aligned, tool to use for teaching programming.

Comparison of machine corrected labs vs. demonstrator corrected labs

At the conclusion of the module, before the exam, all students were invited to provide feedback on how the module might be improved for future years. Students did not report high levels of satisfaction with the HackerRank intervention. Most recommended that the system be abolished or scaled back (see supplementary material – student feedback – for examples).

Students viewed the introduction of automatic assessment negatively, fearing it would lead to a drop in grades. Early in the semester, the class requested a means of earning marks outside of labs. In response, it was decided to send out a "practice" exercise at the beginning of the week, which was to be completed by the end of the week, and which would introduce them to the topic being tested on the Friday. The practice question was worth 30% of the weekly CA mark, with performance in the lab session itself earning the remaining 70%. This process of sending out a practice question was implemented from week 3 onwards. HackerRank's plagiarism detector suggested that the majority of answers to the practice questions were copied or shared. Nevertheless, the boundary between collaboration and collusion is ill defined and highly variable, and students have a poor understanding of it (Joy, Cosma, Yau, & Sinclair, 2011). Any students challenged about the issue said that, although they had worked out the general idea in a group, they had implemented their own solution independently. Because of the difficulty of establishing plagiarism for short problems where standard solutions are often discovered independently, no actions were taken. Nevertheless, the possibility of widespread copying suggests that automatic assessment can only be effective in a controlled lab environment where students are unable to pass code between themselves.

Based on student feedback, it was agreed to pick the best 8 out of the 11 labs, thereby boosting CA marks by an average of 24.2% per student. This meant that by the end of the semester, the best students already had maximum CA, so did not need to attend the final labs.

Discussion

In line with the findings of Barros et al. (2003), the use of HackerRank was positively associated with improved exam results, with the failure rate dropping by two thirds in comparison to the previous year. There were no changes as to the content of the module over the two years, other than the manner in which the labs were structured. These results provide convincing evidence that automatic machine assessment supports an enhanced learning experience for developing programming skills. Furthermore, HackerRank proved easy to use, saving considerable amounts of labour on the part of the demonstrators and the lecturer. Students received consistent, objective feedback on their coding ability during the semester, and this valuable source of information led to a significant boost in programming ability and higher progression rates. The stress of having to perform each week, and of being caught out dramatically by their own coding errors, challenged students to improve their skills in programming.

In line with previous studies examining predictors of programming success (e.g. Ahadi, Lister, Haapala, & Vihavainen, 2015), lab scores turned out to be the strongest predictor of exam results, as opposed to age, gender or prior programming experience. This means that students obtaining low scores in initial HackerRank labs could in future be targeted with supplementary interventions to help improve their confidence and conceptual understanding, thus helping to close the gap in achievement.

There were, however, drawbacks to the use of the HackerRank system, most notably the low levels of student satisfaction reported. In spite of the improvement in performance in comparison to previous years, most students reported that HackerRank had a *negative* impact on their learning, and recommended that it should be dropped, or at least scaled back. Similar responses have been noted in computer programming classes following the introduction of automatic assessment systems. Rubio-Sánchez et al. (2014), for example, suggested that feedback needs to be richer in order to lead to acceptance among the student cohort. Falkner, Vivian, Piper, and Falkner (2014) found that increasing the amount and granularity of automated feedback led to improved results. HackerRank, however, can only provide the most granular feedback of passing and failing test cases.

It also became apparent in our intervention that not every kind of topic was as well suited to automatic assessment. For example, the labs on linked lists, on mergesort, on recursion and on bit-shifting were somewhat contrived, involving code stubs to force students to program in a particular way. Accordingly, in future we might be more selective, combining some element of automatic assessment with more informal labs in weeks where the topic is not as amenable to such evaluation.

There are a number of changes that could be made to improve student experience. For example, the feedback from students revealed that they wanted to see the test cases that were causing their programs to fail. In future we will email students another HackerRank invitation after the lab, with the test cases visible,

so they can see where they went wrong and continue to develop their programs. Based on our experiences, we will also restructure the two hour lab as follows: the HackerRank test will be extended to two hours, and will begin immediately. For the first hour, nobody will be allowed to communicate, and demonstrators will not be allowed to help. Any student completing in the first hour will receive 100% of the marks they are awarded by HackerRank. In the second hour, the marks awarded will fall linearly over time, from 100% at the hour mark, to 40% at the two hour mark. After the hour mark, demonstrators can advise students on bugs and provide a general approach to solving the problem on the whiteboard. As more time passes they can provide more extensive help, with the goal that by the end of the two-hour lab everybody should have a working program.

We hope that this system of increasing the availability of help over the two hours will relieve student anxiety. According to Falkner et al. (2014), even when initial marks are low, the prospect of making progress gives students more resilience and leads them to strive for longer. Providing feedback should also take away the pressure to cheat: rather than take the risk of plagiarising (which HackerRank detects), students who are really struggling will wait until the hour mark and then get help. Over time, this system should have the effect of improving programming confidence, which has been shown to have significant influence on students' achievements in computer science (Connolly et al., 2009). Programming confidence also emerged as an independent predictor of exam performance in our study, further highlighting the important role that psychological variables can have in predicting later success (Maguire, Egan, Hyland, & Maguire, 2017).

Palazzo, Lee, Warnakulasooriya, and Pritchard (2010) suggested that a course design which involves greater teacher interaction encourages students to believe that the instructor is concerned with their learning (rather than simply assigning a grade based on their performance). They found that modifying physics courses at MIT to enhance teacher interaction led to a 75% reduction in cheating (Fraser, 2014). Accordingly, we believe that allowing greater interactions between students and demonstrators in the second hour may improve morale. Students will be able to learn about the bugs in their code, and hopefully get coached all the way through to developing a fully functioning program, allowing them to study their own solutions for the exam.

In conclusion, HackerRank is easy to use, reduces much administrative burden, corrects everything instantly without requiring demonstrator input, gives instant feedback, and is more objective than the previous system employed in this module. Our study has shown that the system is effective in helping students develop programming skills, providing them with high quality objective feedback as to their programming ability. Nevertheless, it should also be noted that students did not report a positive experience. Accordingly, we suggest that the blending of automatic assessment with some element of teacher intervention and peer interaction could lead to a more satisfactory learning experience for struggling students.

Disclosure statement

No potential conflict of interest was reported by the authors.

Notes on contributors

Phil Maguire is a lecturer in the Department of Computer Science at Maynooth University and was the module coordinator for the Data Structure and Algorithms module described in the paper. His research interests are computer science education, philosophy of measurement, and psychology of power and fintech.

Rebecca Maguire is a lecturer in the Department of Psychology at Maynooth University. She has a particular interest in student engagement and learning in Higher Education and the role that cognitive representations play in health and well-being.

Robert Kelly is a PhD student in the Department of Computer Science at Maynooth University and was a Data Structure and Algorithms module demonstrator. His research interests include concurrent data-structure design and implementation, with particular focus on non-blocking data-structures, garbage collection, memory reclamation, and concurrent language design.

References

Ahadi, A., Lister, R., Haapala, H., & Vihavainen, A. (2015, July). Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 121–130). ACM.

Barros, J. P., Estevens, L., Dias, R., Pais, R., & Soeiro, E. (2003). Using lab exams to ensure programming practice in an introductory programming course. *ACM SIGCSE Bulletin, 35*(3), 16–20.

Beaubouef, T., Lucas, R., & Howatt, J. (2001). The UNLOCK system: Enhancing problem solving skills in CS-1 students. *ACM SIGCSE Bulletin, 33*(2), 43–46.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin, 39*(2), 32–36.

Bergin, S., Mooney, A., Ghent, J., & Quille, K. (2015). Using machine learning techniques to predict introductory programming performance. *International Journal of Computer Science and Software Engineering, 4*(12), 323–328.

Biggs, J. (1996). Enhancing teaching through constructive alignment. *Higher Education, 32*(3), 347–364.

Braught, G., MacCormick, J., & Wahls, T. (2010). The benefits of pairing by ability. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 249–253). ACM.

Buffardi, K., & Edwards, S. H. (2015). Reconsidering automated feedback: A test-driven approach. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 416–420). ACM.

Connolly, C., Murphy, E., & Moore, S. (2009). Programming anxiety amongst computing students – A key in the retention debate? *IEEE Transactions on Education, 52*(1), 52–56.

Daly, C., & Waldron, J. (2004). Assessing the assessment of programming ability. *ACM SIGCSE Bulletin, 36*(1), 210–213.

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). CodeWrite: Supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 471–476). ACM.

Dick, M., Sheard, J., Bareiss, C., Carter, J., Joyce, D., Harding, T., & Laxer, C. (2003). Addressing student cheating: Definitions and solutions. *ACM SIGCSE Bulletin, 35*(2), 172–184.

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC), 5*(3), article 4.

Falkner, N., Vivian, R., Piper, D., & Falkner, K. (2014). Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 9–14). ACM.

Edwards, S. H., & Perez-Quinones, M. A. (2008). Web-CAT: Automatically grading programming assignments. In *ACM SIGCSE Bulletin* (Vol. 40, No. 3, pp. 328–328). ACM.

Fowler, S., & Yamada-F, N. (2009). A brief survey on the computer science programs in the UK higher education systems. *Journal of Scientific and Practical Computing, 3*(1), 11–17.

Fraser, R. (2014). Collaboration, collusion and plagiarism in computer science coursework. *Informatics in Education, 13*(2), 179–195.

Hawi, N. (2010). Causal attributions of success and failure made by undergraduate students in an introductory-level computer programming course. *Computers & Education, 54*(4), 1127–1136.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 86–93). ACM.

Jiau, H. C., Chen, J. C., & Ssu, K. F. (2009). Enhancing self-motivation in learning programming using game-based simulation and metrics. *IEEE Transactions on Education, 52*(4), 555–562.

Joy, M., Cosma, G., Yau, J., & Sinclair, J. (2011). Source code plagiarism – A student perspective. *IEEE Transactions on Education, 54*(1), 125–132.

Knorr, E. M., & Thompson, C. (2017). In-Lab programming tests in a data structures course in C for non-specialists. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 339–344). ACM.

Kosner, A.W. (2014, June 12). Hackerrank solves tech hiring crisis by finding programmers where they live. *Forbes*.

Lawrence, R. (2004). Teaching data structures using competitive games. *IEEE Transactions on Education, 47*(4), 459–466.

Linn, M. C., & Dalbey, J. (1989). Cognitive consequences of programming instruction. In E. Soloway & J. C. Sphorer (Eds.), *Studying the novice programmer* (pp. 57–81). Hillsdale, NJ: Lawrence Erlbaum.

Luxton-Reilly, A., & Denny, P. (2010). Constructive evaluation: A pedagogy of student-contributed assessment. *Computer Science Education, 20*(2), 145–167.

Maguire, R., Egan, A., Hyland, P., & Maguire, P. (2017). Engaging students emotionally: The role of emotional intelligence in predicting cognitive and affective engagement in higher education. *Higher Education Research & Development, 36*(2), 343–357.

Maguire, P., & Maguire, R. (2013). Can clickers enhance team based learning? Findings from a computer science module. *AISHE-J: The All Ireland Journal of Teaching & Learning Higher Education, 5*(3), 1421–14217.

Maguire, P., Maguire, R., Hyland, P., & Marshall, P. (2014). Enhancing collaborative learning using paired-programming: Who benefits?. *AISHE-J: The All Ireland Journal of Teaching and Learning Higher Education, 6*(2), 1411–14125.

Mayer, R., Stull, A., DeLeeuw, K., Almeroth, K., Bimber, B., Chun, D., ... & Zhang, H. (2009). Clickers in college classrooms: Fostering learning with questioning methods in large lecture classes. *Contemporary Educational Psychology, 34*, 51–57.

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bulletin, 34*(1), 38–42.

McKeachie, W. J. (1999). *Teaching tips: Strategies, research, and theory for college and university teachers* (10th ed.). New York, NY: Houghton Mifflin.

Miliszewska, I., & Tan, G. (2007). Befriending computer programming: A proposed approach to teaching introductory programming. *Informing Science: International Journal of an Emerging Transdiscipline, 4*, 277–289.

Palazzo, D. J., Lee, Y.-J., Warnakulasooriya, R., & Pritchard, D. E. (2010). Patterns, correlates, and reduction of homework copying. *Physical Review Special Topics: Physical Education Research, 6*(1), 010104-1–010104-11.

Peters, A. K., & Pears, A. (2012). Students' experiences and attitudes towards learning Computer Science. In *Proceedings of the 42nd ASEE/IEEE Frontiers in Education Conference* (pp. 88–93). IEEE.

Porter, L., Guzdial, M., McDowell, C., & Simon, B. (2013). Success in introductory programming: What works? *Communications of the ACM, 56*(8), 34–36.

Roberts, E. (2002). Strategies for promoting academic integrity in CS courses. *Frontiers in Education, 3*, F3G14–F3G19.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137–172.

Rubio-Sánchez, M., Kinnunen, P., Pareja-Flores, C., & Velázquez-Iturbide, Á. (2014). Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior, 31*, 453–460.

Sheard, J., Carbone, A., & Dick, M. (2003). Determination of factors which impact on IT students' propensity to cheat. In *ACE '03: Proceedings of the fifth Australasian Conference on Computing Education* (pp. 119–126). Australian Computer Society, Inc.

Shaffer, D. W., & Resnick, M. (1999). "Thick" Authenticity: New Media and Authentic Learning. *Journal of Interactive Learning Research, 10*(2), 195–215.

Thweatt, M. (1994). CSI closed lab vs. open lab experiment. *ACM SIGCSE Bulletin, 26*(1), 80–82.

Traynor, D., & Gibson, P. (2004). Towards the development of a cognitive model of programming: A software engineering approach. In *Proceedings of the 16th Workshop of Psychology of Programming Interest Group*.

Trees, A. R., & Jackson, M. H. (2007). The learning environment in clicker classrooms: Student processes of learning and involvement in large university course using student response systems. *Learning, Media, and Technology, 32*(1), 21–40.

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 39–44). ACM.

Wilkinson, J. (2009). Staff and student perceptions of plagiarism and cheating. *International Journal of Teaching and Learning in Higher Education, 20*(2), 98–105.

Williams, L., & Upchurch, R. (2001). In support of student pair programming. *SIGCSE Conference on Computer Science Education,* 327–331.