Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance

Alireza Ahadi and Raymond Lister
University of Technology, Sydney
Australia
alireza.ahadi@uts.edu.au
raymond.lister@uts.edu.au

Heikki Haapala and Arto Vihavainen
Department of Computer Science
University of Helsinki
Finland
heikki.haapala@cs.helsinki.fi
arto.vihavainen@cs.helsinki.fi

ABSTRACT

Methods for automatically identifying students in need of assistance have been studied for decades. Initially, the work was based on somewhat static factors such as students' educational background and results from various questionnaires, while more recently, constantly accumulating data such as progress with course assignments and behavior in lectures has gained attention. We contribute to this work with results on early detection of students in need of assistance, and provide a starting point for using machine learning techniques on naturally accumulating programming process data.

When combining source code snapshot data that is recorded from students' programming process with machine learning methods, we are able to detect high- and low-performing students with high accuracy already after the very first week of an introductory programming course. Comparison of our results to the prominent methods for predicting students' performance using source code snapshot data is also provided.

This early information on students' performance is beneficial from multiple viewpoints. Instructors can target their guidance to struggling students early on, and provide more challenging assignments for high-performing students. Moreover, students that perform poorly in the introductory programming course, but who nevertheless pass, can be monitored more closely in their future studies.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; H.2.8 [**Database Applications**]: Data mining

Keywords

introductory programming; source code snapshot analysis; programming behavior; educational data mining; learning analytics; novice programmers; detecting students in need of assistance

.

1. INTRODUCTION

Every year, tens of thousands of students fail introductory programming courses world-wide, and numerous students pass their courses with substandard knowledge. As a consequence, studies are retaken and postponed, careers are reconsidered, and substantial capital is invested into student counseling and support. World-wide, on average one third of students fail their introductory programming course [4, 40]. Even when looking at statistics describing pass rates after teaching interventions, as many as one quarter of the students still fail the courses [38].

One of the challenges in organizing teaching interventions is that any change is likely to also affect students for whom the prevalent situation is more suitable. For example, if a student is already at a stage where she could work on more challenging projects on her own, mandatory excessively structured learning activities that everyone needs to follow may even be counterproductive for her [16, 31]. To provide another example, while collaborative learning practices such as pair programming [45] have been highlighted as efficient teaching approaches for introductory programming [23, 38], there are contexts in which students mostly work from a distance and rarely attend an institution.

This diversity of institutions, students, and teaching approaches is the setting upon which our work builds. We believe that the appropriate next step in teaching interventions is the transition towards interventions that address only those students that are in need of guidance, and work towards that goal by analyzing methods for detecting such students as early as possible. More specifically, in this work, we explore methods for detecting high- and low-performing students in an introductory programming course already based on the performance during the very first week of the course. Variants of the topic have been investigated previously, for example, by Jadud, who proposed an approach to quantify students' ability to solve errors using source code snapshots [15], Ahadi et al., who measured students' knowledge using tests [1, 2], and Porter et al., who used in-class clicker data as a lens into students' performance [24, 25].

This work is organized as follows. First, in Section 2, we provide an overview of the evolution of the field of understanding factors that contribute to students' performance in introductory programming. Then, in Sections 3 and 4 we outline our research questions and data in more detail, as well as explain the methodology and outline the results. The results are discussed in Section 5, and finally, Section 6 concludes the work and outlines future research questions.

2. BACKGROUND

In the article "*What best predicts computer proficiency*" [9], Evans and Simkin describe early advances into understanding attributes that contribute to the ability of learning to program. This ability, *programming aptitude*, is often defined as the student's ability to succeed in an introductory programming course, and is measured through e.g. the course grade or a finer-grained measure such as within-course point accumulation. Before 1975, the research focused mainly on demographic factors such as educational background and scores from previous courses, while by the end of the 1970s, the focus moved slowly to evaluating static tests that measure programming aptitude. This was followed by research that started to investigate the effect of cognitive factors such as abstraction ability and the ability to follow more complex processes and algorithms [9]. Such research has continued to this day by introducing factors related to study behavior, learning styles and cognitive factors [42]. However, recently, dynamically accumulating data from students' learning process has gained researchers' attention [15, 25, 37, 41].

Overall, this stream of research has been motivated by multiple viewpoints, which include identification of students that have an aptitude for CS-related studies (e.g. [35]); studying and identifying measures of programming aptitude as well as combining them (e.g. [3, 5, 30, 43]); improvement of education and the comparison of teaching methodologies (e.g. [34, 36]); and identifying at-risk students and predicting course outcomes (e.g. [15, 41]).

Next, we outline some of this work in more depth. We begin by focusing on factors that do not change at all or change very slowly, and continue towards dynamic factors that change more rapidly and where new information may be constantly accumulated.

2.1 Gender

Studies in past often investigated gender as one of the factors that may explain programming aptitude – one of the reasons may be that the field of computing is at times seen as being dominated by males, and thus exhibits a male-oriented culture. However, the results show no clear trend. For example, in an analysis of introductory programming course grade and gender, Werth found no significant correlation ($r = 0.080$) [43]. In a similar study, Byrne and Lyons found that female participants in introductory programming course had a marginally higher point average than their male counterparts, but the difference was not statistically significant [6]. The role of gender was also investigated by Ventura, who studied the effect of gender by comparing students' programming assignment, exam, and overall course points, and found no effect that could be explained by gender [36].

Studies exist that suggest a referential connection between programming aptitude and gender. For example, in a small study ($n = 11$), Bergin and Reilly observed that female students had statistically significant and strong correlations ($r = 0.72 - 0.93$) between an Irish high-school leaving certificate test and programming course scores [5] – an effect that was not visible among male counterparts.

2.2 Academic Performance

The connection between students' academic performance and programming aptitude has been investigated in several studies. For example, Werth analyzed the connection between the amount of tertiary education mathematics courses and programming aptitude, but found no significant correlation ($r = -0.019; p > 0.1$). She suggested that a large amount of mathematics courses in tertiary education may actually be an indicator of improving a weak mathematics background [43]. Other studies have found connections between mathematics and introductory programming. For example, Stein studied the connection between Calculus and Discrete Mathematics and the grade from an introductory programming course. The correlations, overall, were weak (Calculus: $r = 0.244$; Discrete Math: $r = 0.162$) [34]. Watson, Li and Goldwin did a similar study, and, similarly, found no significant correlation between the Discrete Math and the introductory programming grade ($r = 0.06; p > 0.05$). However, there was a mediocre albeit not statistically significant effect between the Calculus course grade and programming course points ($r = 0.37; p = 0.06$) [42].

In addition to mathematics, factors such as language performance and overall grade averages have also been studied. For example, Leeper and Silver studied students' English language scores and the score of the verbal part of the SAT test. In their study, only the verbal SAT score had a mediocre correlation with the introductory programming course grade ($r = 0.3777$) [19]. Werth found no significant correlation between secondary education grade average and the grade achieved in an introductory programming course ($r = 0.074; p > 0.1$), but she did find a weak correlation between university-level grade average and the introductory programming course grade ($r = 0.252; p < 0.01$) [43]. Similarly, Watson et al. studied correlations between various secondary education courses and course averages, but found no statistically significant correlations [42].

2.3 Past Programming Experience

It is natural to assume that past programming experience influences programming course scores, and thus, the connection has been studied in a number of contexts, albeit with contradictory results. Hagan and Markham found that students with previous programming experience received considerably higher course marks than the students with no programming experience [10]. Wilson and Shrock utilized five variables related to programming and computer use, such as formal programming education, the use of internet, and the amount of time spent on gaming. The combination of these variables had a significant correlation with the midterm score in an introductory programming course ($r = 0.387; p < 0.01$) [7]. Similarly, in 2004, Wiedenbeck et al. reported on a study in which the number of ICT courses taken by students, the number of programming courses taken, the number of programming languages students had used, the number of programs students had written, and the length of those programs were combined into a single factor. The combination had a weak but significant correlation with the introductory programming score ($r = 0.25; p < 0.05$) [44].

While multiple studies indicate a positive correlation between past programming experience and introductory programming course outcomes, somewhat contradictory results also exist. For example, Bergin and Reilly found that students with no previous programming experience had a marginally higher mean overall score in an introductory programming course, and found no statistically significant difference between students with and without previous programming experience [5]. In another study, Watson et al. found that while students with past programming experience had significantly higher overall course points than those with no previous programming experience [42], programming experience in years had a weak but statistically insignificant negative correlation with the course points ($r = -0.20$) [42].

2.4 Behavior in Lectures and Labs

Rodrigo et al. studied students' observed behavior in programming labs [26]. They studied students' gestures, outbursts, and other factors including collaboration with other students, and sought to identify factors that are potentially related to students' success. In addition, they collected source code snapshots from students' programming process. Six statistically significant factors ($p < 0.05$) that had a mediocre correlation with an introductory programming course midterm score were identified. Four of them were related to students' behaviors; confusion ($r = -0.432$), boredom ($r = -0.389$), focus ($r = 0.346$), and discussion about the programming environment (-0.316), while two were related to snapshots. The number of consecutive snapshots with errors ($r = -0.326$) and compilation events in which the student had worked on the same area in the source code ($r = -0.336$) were both negatively correlated with the midterm score [26].

Another angle at studying students behavior was recently proposed by Porter et al. [25], who studied students' responses to clicker questions in a peer instruction setting. In their study, they identified that the percentage of correct clicker answers from the first three weeks of a course was strongly correlated with overall course performance ($r = 0.61; p < 0.05$).

2.5 Source Code Snapshots

In "*Methods and Tools for Exploring Novice Compiling Behaviour*" [15], Jadud presents a method to quantify a student's tendency to create and fix errors, which he called the *error quotient*. In his study, the correlation between the error quotient and the average score from programming assignments was mediocre and statistically significant ($r = 0.36; p = 0.012$), while the correlation between the error quotient and the grade from a course exam was high ($r = 0.52; p = 0.0002$) [15]. Rodrigo et al. used an alternative version of Jadud's error quotient, and found that in their context the correlation between the error quotient and the midterm score of an introductory programming course was strong and statistically significant ($r = -0.54; p < 0.001$) [27]. In essence, this suggests that the less programming errors a student makes, and the better she solves them, the higher her midterm grade will tend to be [27].

Watson et al. also conducted a study using Jadud's error quotient, and found a significant correlation between the error quotient and their programming course scores ($r = 0.44$) [41]. They proposed that the amount of time that students spend on programming assignments should be taken into account, and that one should consider the files that a student is editing as a part of the error quotient calculation [41]. They proposed an improvement to the error quotient called *Watwin*, and found that with this improvement the correlation increased from ($r = 0.44$) to ($r = 0.51$) [41]. They also noted that a simple measure, the average amount of time that a student spends on a programming error, is strongly correlated with programming course scores ($r = -0.53; p < 0.01$).

Source code snapshots have been used to elicit information in finer detail as well. For example, Piech et al. [22] studied students' approaches to solving two programming tasks, and found that students' solution patterns are indicative of course midterm scores. Programming patterns were also studied by Hosseini et al., who identified students' behaviors within a programming course – some students were more inclined to build their code step by step, while others started from larger quantities of code, and reduced their code in order to reach a solution [14]. Another approach recently proposed by Yudelson et al. was to use fine-grained concepts extracted from source code snapshots, and to model students' understanding of these concepts as they proceed [46].

Next, we explore some of these methods for source code snapshot analysis, as well as provide researchers with an outline for performing such studies.

3. RESEARCH DESIGN

This study is driven by the question of identifying high- and low-performing students as early as possible in a programming course to provide better support for them. By high- and low-performing students, we mean students in the upper- and lower-half of course scores, and by early, we mean after the very first week of the programming course. This means that instructors could plan and provide additional guidance to specifically selected students already during the second week of the course.

For the task, we explore previously proposed methods for predicting students' performance from source code snapshots, and evaluate a number of machine learning techniques that have previously received little attention for the task at hand.

3.1 Research Questions

Our research questions for this study are as follows.

RQ1 Given our dataset, how do the methods proposed by Jadud and Watson et al. perform for detecting high- and low-performing students?

RQ2 Given our dataset, how do standard machine learning techniques perform for detecting high- and low-performing students?

To answer the first question, we have implemented the algorithms described in [15, 41], and evaluate their performance on our data. For the second question, we first identify relevant features from a single semester, then evaluate different machine learning techniques to build a predictive model using the extracted features to determine a top-performing approach. Finally, the top-performing predictive model is evaluated on a dataset from a separate semester to determine cross-semester performance of the selected model.

3.2 Data

The data for the study comes from two semesters of an introductory programming course organized at the University of Helsinki. The course lasts six weeks, is taught in Java, and uses a blended online textbook that covers variables, basic I/O, methods, conditionals, loops, lists, arrays, elementary search algorithms and elementary objects. In the programming course, the main focus is on working on practical programming assignments, accompanied by a weekly two-hour lecture that covers the basics needed to get started with the work. Support is available in open computer labs, where teaching assistants and course instructors are available some 20-30 hours each week (see [18] for details).

Although no socio-economic factors were available for this study, the studied population is relatively homogenic, and the educational system in the context is socially inclusive, meaning that there is both a minimal underrepresentation of students from low education background and a minimal overrepresentation of students from high education background [21]. There are also no tuition fees, and students receive student benefits such as direct funding from the state, assuming that they progress in their degree work.

For the purposes of this study, students' programming process was recorded using Test My Code [39] that is used for automatically assessing students' work in the course. For each student that consented to having their programming process recorded, every key-press and related information such as time and assignment details was stored. The students used the same programming environment both from home and at the university. Students were asked to provide information on whether they had prior programming experience, and access to information on students' age, gender, grade average, and major was given for the researchers for the purposes of this study. In the studied context, major is selected before enrollment, and in both semesters, over 50% of the students had other subjects than computer science as their major – for students with CS as a major, the studied course is the first course that they take.

In the first semester (spring), a total of 86 students participated in the study, and in the second semester (fall), a total of 210 students participated in the study. Full fine-grained key-log data is available only for the first semester, while for the second semester, only higher level actions such as saves, compilation events, run events and test events are available.

While attendance in the course activities is not mandatory, 50% of total course points comes from completing programming assignments. The rest of the course points comes from a written exam, where students answer both essay-type questions as well as programming questions. To pass the course, the students have to receive at least half of the points from the exam as well as half of the points from the programming assignments, while the highest grade in the course can be received by gathering over 90% of the course points.

4. METHODOLOGY AND RESULTS

The students were divided into groups based on their performance in (1) an algorithmic programming question given in the exam, (2) the overall course, and (3) a combination of the two. The first division into groups is motivated by students' struggling with writing programs even at a later phase of their studies [20], and has also been the focus in related studies, such as the work by Porter et al. [25]. The algorithmic programming question is a variant of the Rainfall Problem [32], where students have to create a program that reads numbers, possibly filters them, and prints attributes such as the average of the accepted numbers. The second division into groups outlines the students' overall performance, and the third division combines the previous. Table 1 shows student counts in these groups for the dataset that is used to evaluate the algorithms in RQ1, and to train the predictive model for RQ2.

4.1 Research Question 1

To answer the first research question, "*Given our dataset, how do the methods proposed by Jadud and Watson et al. perform for detecting high- and low-performing students?*", we implemented these algorithm's as they were described [15, 41]. Both algorithms use a set of successive compilation event pairings to quantify the students' ability to fix syntactic errors in the programs that they are writing. The main difference between Jadud's error quotient and the Watwin-algorithm is that the Watwin-algorithm also considers the possibility that students may be working on multiple files, where one file has errors, and the other does not. Thus, changing from one file to the other is not seen as if the user fixed the errors. Moreover, the Watwin algorithm also takes into account the amount of time that students spend on fixing errors.

Table 1: Student counts for the studied population, binned based on the predicted variable.

Target class	Median or Above	Below Median
Exam Question	47	39
Final Grade	48	38
Combined	43	43

Unlike the data used by Jadud and Watson et al., the data recorded from standard programming environments do not have explicit compilation events as the environments continuously compile the code and highlight errors to developers. To approximate these explicit compilation events for Jadud EQ and Watwin algorithm, two options were evaluated: (1) use only snapshots where students perform an action that does not involve changing the code, i.e. run their code, test their code, or submit the code to the assessment server (i.e. *action* in Table 2), and (2) use only snapshot pairs between which the students have taken at least a ten second pause from programming (*pause* in Table 2). The value for the pause was determined by evaluating the algorithms with 60, 30, 10 and 5 second pauses, after which the value which resulted in the best average performance was selected. Our rationale for the use of actions is that in such cases, the students want explicit feedback from the system, while the rationale for pauses is that the students have stopped to, for example, debug their program. Option (2) is only available for the first semester, as fine-grained key-log data is not available for the second studied semester.

Pearson correlation coefficients between the predicted variables (Table 1) and Jadud's error quotient and Watwin-score are given in Table 2. The correlations are given as absolute values, and are all low ($r < 0.3$).

Table 2: Pearson Correlation coefficients for between the Jadud's error quotient, the Watwin-score, and the predicted variables.

Variable	Semester	Jadud		Watwin	
		action	pause	action	pause
Exam Quest.	First	.15	.21	.25	.18
	Second	.20	-	.09	-
Final Grade	First	.03	.08	.01	.13
	Second	.10	-	.005	-
Combined	First	.02	.08	.01	.13
	Second	.12	-	.01	-

4.2 Research Question 2

To answer the second research question, "*Given our dataset, how do standard machine learning techniques perform for detecting high- and low-performing students?*", the problem was approached as a supervised learning task, where existing data is used to infer a function that can be used to categorize incoming data into groups [13].

First, features were extracted from the dataset. Then, to avoid the use of irrelevant or redundant features, feature selection was used to identify relevant features. Once a relevant subset of features had been selected, we evaluated a number of classifiers. Finally, when a classifier had been selected from the evaluated classifiers, we tested the model against a data set from a separate semester. Feature selection and classifier evaluation was performed using the WEKA Data Mining toolkit [11].

Feature Extraction

For the study, we extracted two types of attributes: (a) Attributes based on previously studied success factors, such as previous academic performance (tertiary education) and past programming experience; (b) programming assignment specific Source-code snapshot attributes that potentially reflect students' persistence and success with the course assignments. For each assignment, the number of steps that a student took, measured in key-presses and other actions, as well as the maximum achieved correctness when measured by automated tests was extracted. The Source-code snapshot attributes were programmatically extracted from the programming process data, which is recorded by Test My Code as students are working on the assignments. An overview of the used attributes is given in the Table 3. The datasets were also normalized.

Table 3: Features extracted for the study

Features	Type
Gender	Categorical
Major	Categorical
Grade Average	Numerical
Age	Numerical
Programming experience	Binary
Maximum obtained correctness for each programming assignment	Numerical $[0-1]$
Amount of steps taken in each of the programming assignment	Numerical $[0-\infty]$

Feature Selection

After the feature extraction phase, there was a total of 53 features. To reduce the amount of overlapping features, possible over fitting, and to potentially improve predictive accuracy of the feature set, feature selection was performed. We used correlation-based feature subset selection [12], where individual predictive ability of each feature along with the degree of redundancy between them was evaluated using three methods; (1) genetic search, (2) best first method and (3) greedy stepwise method. Results of the feature selection phase are given in Table 4.

After this phase, the information gain of each feature was measured to reveal features that had little or no predictive value. Information gain, or Kullback-Leibler divergence [17], is used to measure the amount of information that the feature brings about a predicted value, assuming that they are the only two existing variables, and is measured by the difference of two probability distributions (in our case, e.g., the difference of the probability distributions of the exam question results and grade average). After measuring information gain for each of the features and predicted value, the low-contributing features were removed. The features above the line in Table 5 were retained in the training set.

Table 5: Information gain of the features. Features below the line were excluded from further use.

Feature	Exam question	Grade	Both
Grade Average	0.34	0.36	0.44
Correctness of a20	0.40	0.40	0.38
Steps for a23	0.44	0.32	0.29
Steps for a21	0.23	0.20	0.20
Steps for a22	0.22	0.16	0.19
Major	0.17	0.11	0.13
Steps for a17	0.27	0.15	0.12
Steps for a20	0.26	0.15	0.12
Steps for a18	0.14	0.15	0.12
Steps for a19	0.23	0.13	0.11
Age	0.11	-	0.11
Prog. Exp	-	0.05	0.07
Gender	0.01	0.008	0.003

Classifier Evaluation

As is typical for studies that explore machine learning methodologies, a number of classifiers were evaluated. In our case, we evaluated three families of classifiers; Bayesian classifiers, Rule-learners, and Decision tree -based classifiers, and chose a total of nine classifiers from these three families. All of these approaches are commonly used for classifying students [28, 29]. The evaluation was performed using two separate validation options: k-fold cross validation (with k=10), and percentage split (2/3 of the dataset used for training and 1/3 for testing). This means that during the classifier training and evaluation phase, parts of the data was hidden during the training, and was then used for the evaluation. Table 6 presents the results for the classification algorithms that were investigated in this study.

As can be seen in Table 6, the overall accuracy of decision trees is higher than that of the other two classifier families. Among decision trees, Random Forest has on average the highest accuracy for all predictive variables with 86%, 90% and 90% accuracy for predicting Exam question, Final Grade and the combination of both. To show the predictive accuracy in more detail, Table 7 shows the confusion matrix of the Random Forest classifier when predicting the combination of the Exam Question and Final grade, when using 10-fold cross-validation on the training data set.

Table 7: Confusion matrix of Random Forest on predicting whether students are equal-to-or-above or below the median score on the combination of exam question and final grade

	Predicted above	Predicted below
Actual above	38	5
Actual below	3	40

Thus, we selected the Random Forest as the classifier that is used to evaluate students' performance. More detailed evaluation of the performance of the Random Forest classifier is given in Table 8. The F1-Measure, which represents the balanced precision-recall, shows that Random Forest provides a strong result in this prediction task. Moreover, the Receiver operating characteristic value (ROC) suggests that the classifier still performs well when the classification threshold is changed from the median, i.e. if we would rather

Table 4: Features selected during feature selection. The left-hand side describes the feature selection method, and the columns describe the features selected for the different predictive variables. *Steps* **denotes the number of recorded events for a student on a specific programming assignment.** *Correctness* **denotes the percentage of tests passed by a student on a specific programming assignment.**

Method	Exam question	Final Grade	Both
Best First	Age; Grade Average; Steps for a17, a21, and a23	Grade Average; Steps for a21 and a23; Correctness for a23	Grade Average; Steps for a21 and a23; Correctness for a20
Genetic Search	Age; Grade Average; Steps for e17, e19, e20, e21, and e23; Correctness for e2, e6, e11, and e12	Grade Average; Steps for e20, e21, and e23; Correctness for e23	Grade Average; Steps for e21, and e23; Correctness for e20
Greedy Stepwise	Age; Grade Average; Steps for e17, e21, and e23	Grade Average; Steps for e21 and e23; Correctness for e23	Grade Average; Steps for e21 and e23; Correctness for e20

Table 6: Classifier accuracy when performing evaluation of the classifiers on the training set from a single semester. The highest accuracies are marked with bold. Exam question is shown as Q in the Table.

Classifier	Family	10-fold cross-validation accuracy			percentage split accuracy		
		Q	Final grade	Q + Final grade	Q	Final grade	Q + Final grade
Naive Bayes	Bayesian	80%	80%	77%	86%	86%	86%
Bayesian Network	Bayesian	81%	77%	76%	82%	76%	72%
Decision Table	Rule Learner	78%	73%	84%	86%	76%	90%
Conjuctive Rule	Rule Learner	73%	80%	83%	72%	86%	90%
PART	Rule Learner	85%	79%	**93%**	90%	76%	82%
ADTree	Decision Tree	80%	85%	86%	90%	83%	83%
J48	Decision Tree	83%	82%	**93%**	**93%**	89%	83%
Random Forest	Decision Tree	**86%**	**90%**	90%	90%	**90%**	**93%**
Decision Stump	Decision Tree	73%	76%	84%	83%	**90%**	90%

seek to identify the lowest performing quartile of students, and the Matthews correlation coefficient (*MCC*) shows a high correlation ($r = 0.71 - 0.81$) between the classifier and the predicted values.

Evaluation on a Separate Semester

As the data that is produced within educational settings varies between semesters, due to variations in student cohorts and course changes, the generalizability of the model needs to be evaluated using data from a separate semester. Accordingly, we evaluated the Random Forest -classifier (i.e. our best performing classifier from above) on data from a separate semester with $n = 210$ students. We found that the Random Forest -classifier was able to categorize students on the Exam Question, the Final Grade, and the combination of both with the accuracy of 80%, 73%, and 71% respectively, when the training of the model was performed on the data from the first semester with $n = 86$ students.

5. DISCUSSION

5.1 Research Question 1

To answer research question one, "Given our dataset, how do the methods proposed by Jadud and Watson et al. perform for detecting high- and low-performing students?", the performance of the approaches differs from the studies in which the algorithms have traditionally been evaluated. Next, we discuss factors which may explain this result.

First, we use data from a considerably shorter period than Jadud and Watson et al. use in their studies. The first results in the article by Watson et al. [41] are given after three weeks into the course, and at that time, the correlation coefficients are near 0.3 for both Watwin-score and Jadud's error quotient – marginally better than our results. Moreover, in Watson et al.'s work, the analysis is performed against overall coursework mark, that is, the overall score from programming assignments [41], and not against the performance in a written exam.

Second, the programming environment used in the studies by Jadud and Watson et al. expects the student to take an extra step for her to receive information on whether her code compiles or not, while such a step is not necessary in current programming environments. It is possible that such a feature stimulates specific working behavior, which in turn may have contributed to previously observed outcomes.

A third factor is related to the quantity and type of the programming assignments. In the context of our study, the students work on a relatively large number of programming assignments during the very first week. Many of the assignments are relatively straightforward, and have been designed to help students gain confidence. This means that it is possible that the predictive approaches that are based on students' programming errors may also be dependent on the programming assignments being non-trivial for the students, which is not always the case in the studied context. These details from the contexts of Jadud and Watwin are not at our disposal.

Finally, the fourth factor is the guidance that students receive during the course. For example, in the context of Watson et al. [41], the students have specific and limited lab hours during which they can receive support on the programming assignments, while in the context that we studied, the labs are open most of the time, and anyone can attend. It is also possible that the type of guidance provided in labs differs.

Table 8: Statistical measures for the Random Forest -classifier when predicting the considered target variables. TPR stands for True Positive Rate, FPR stands for False Positive Rate, ROC stands for Receiver Operating Characteristic, and MCC stands for Matthews Correlation Coefficient.

Class	TPR	FPR	Precision	Recall	F1-Measure	ROC	MCC
Exam question	0.86	0.14	0.86	0.86	0.86	0.92	0.71
Final grade	0.89	0.10	0.89	0.89	0.89	0.92	0.78
Exam question & Final grade	0.90	0.09	0.90	0.90	0.90	0.95	0.81

5.2 Research Question 2

To answer research question two, ”Given our dataset, how do standard machine learning techniques perform for detecting high- and low-performing students?”, we both described the workflow of creating and evaluating machine learning algorithms as well as outlined the results. The process starts with feature extraction, continues with feature selection that is followed by classifier evaluation, and finally concludes with evaluation with a separate data set – in our study, from a separate semester. While the performance of the classifier was high when evaluating the approach within a single semester, ranging from 86% to 90% accuracy with 10-fold cross-validation, the performance was lower (ranging from 71% to 80%) when the predictive model was evaluated on data from a separate semester.

When extracting and selecting the most important features, it was observed that most *a priori* features such as past programming experience, age, and gender made relatively little contribution to the predicted values. This is in line with previous research, which was discussed in Section 2. The information provided by *a priori* features was lower than that of the performance in the actual programming assignments. The most important features were students' grade average, the maximum percentage of automated tests that a student's solution to a specific programming assignment reached, number of steps that students took in a number of programming assignments, and the students' major. Note that for the students who have CS as their major, no grade average was available as the programming course was the very first course that they took – tree-based models handle this well.

5.3 Analysis of Programming Assignments

The feature selection process selected a number of programming assignments as important for the predictive process. All of the programming assignments were from the later part of the week – assignments 17 to 23 were selected, out of a total of 24 assignments in the first week. In all of these programming assignments from the first week, students were given a class that had an empty main-method. In the assignments leading to assignment 17, students had practiced producing different kinds of outputs, the use of variables such as `int` and `String`, reading input from the keyboard, simple comparisons with `if` and `if-else` structures, and combinations of these. Instructions for assignments 17 to 23 are given in Table 9. In addition to what is shown in the table, students had one or two examples of the program output. Also, assignment 23 had an API description of the visualization library.

As with assignments 1-16, assignments 17 and onwards introduce new concepts step-by-step. For example, in assignment 17, the students practice the use of an `else if` structure for the first time, and in assignment 18, the students are expected to use multiple `else if` statements. In assignment 19, the students are practicing the same concepts as in assignment 18, but with a different task and a smaller number of cases that need to be taken into account. Assignment 20 is the first assignment in which the students compare String variables. Assignments 21 is more algorithmic in nature than earlier assignments. Finally, assignments 22 and 23 are programs that require the student to use a loop for the first time. These are concepts that are known not to be easy in other contexts as well (see e.g. [8]).

Table 9: Programming assignments that were highlighted during the feature selection process. Examples of input/output were also given to students.

#	assignment instructions
17	Write a program that reads in two numbers from the user, and prints the larger of them. If the numbers are equal, the program should output ”they are equal.”
18	Write a program that reads in a number between 0 and 60, and transforms it to a grade using the following rules: 0-35 should be F, 36-40 D, 41-45 C, 46-50 B, and 51-60 A.
19	Write a program that reads in a number and checks that it is a valid age [0-120]. If the number is within the range, the program should output ”OK!”, otherwise the program should output ”Impossible!”.
20	Write a program that reads an username and a password, and compares them to user credentials that are given with the assignment. The program should print ”correct”, if the credentials are correct, otherwise, ”false”.
21	Write a program that reads in a number, and determines whether it is a leap year or not.
22	Write a program that continuously asks for a password until the user types in the right password.
23	Write a program that continuously reads in numbers, if the numbers are between [-30, 40], they are to be added to a plot (a ready library given). The program execution should never end.

It is somewhat surprising that assignment 20 was the only assignment for which the student's maximum achieved correctness, i.e. the percentage of tests passed, was highlighted as an important feature. Upon further analysis, as the students were accustomed to comparing numbers, many had initially challenges with comparing strings and the use of the `equals` method, which was needed in the assignment. Most of the students eventually did tackle this, and some of those that did not seemed to be confused with comparing multiple strings at the same time; even if not completing the assignment, students eventually moved forward. At the same time, a persistent student could work through the assignments with the support from the programming environment and course staff, given that she would not start too late, which likely also explains parts of the correctness not being important. From the viewpoint of a material designer, the first finding could imply that it might be beneficial to consider an assignment with simpler string comparisons at first, e.g. by comparing just a single string, instead of the first assignment being one where two strings are compared at the same time. However, this was no longer an issue in assignment 22, where the students combined the same behavior with a loop construct.

Overall, when considering the number of steps that the students took to reach a solution, the students in the high-performing group took more steps on average than the students in the low-performing group. While initially one would assume that this would be explained simply by the low-performing students not attempting the assignments, this was not the case. It simply seems that the students in the high-performing group, when generalized, tried out more than a single approach and were not always content with simply reaching a working solution. Such behavior was also encouraged by the course staff.

5.4 Misclassified Students

We also performed an analysis of the students who were misclassified, i.e. students who were classified into another category than that to which they belonged. When considering the students who were classified as high-performing but belonged to the low-performing group, a number of them had adopted a work behavior where they diligently worked through the assignments by battling their way through the automatic tests. This is likely due to a result that has been previously pointed out by Spacco, i.e., if students are given full test results, they may adopt the habit of ”*programming by 'Brownian motion', where students make a series of small, seemingly random changes to the code in the hopes of making their program pass the next test case”* [33] – currently, the programming environment used does not provide ways to battle this behavior.

Similarly, when considering the students who were classified as low-performing, but were high-performing, some of them used copy-paste in a quantity that had the classifier consider them as students who did not explore the solutions at length. Note that this does not mean that these students were plagiarizing their solutions from others, but seemed to extensively utilize their solutions from previous assignments.

5.5 Practical Implications

Our work implies that one can differentiate between the high- and low-performing students in a programming class already based on the performance of a single week with a relatively high accuracy. This means that instructors may, potentially, provide targeted interventions already during the second week. Practices such as additional rehearsals could be introduced for low-performing students, while high-performing students may benefit from additional challenges.

The results also indicate that students' programming behavior during the class is more important than background variables such as age, gender, or past programming experience, which is in line with previous studies. Moreover, in the studied context, the correctness of the students' solutions was not as important as the effort. That is, students who simply pushed towards a solution did not benefit from the programming tasks as much as the students who did additional experiments. It is plausible that such information on students' behavior can also be used to guide students towards more productive learning strategies.

5.6 Limitations of work

Predictive models are generalizations over a dataset gathered during a single or a number of semesters, and should always be validated using an additional dataset. As is evident in our case, the new dataset, when gathered within the same context but during a different semester, had different results than those from the initial evaluation. At the same time, the comparison was strict as we compared the performance of a model built on data from a spring semester against data from a fall semester. This effectively demonstrates that if the teaching approach, materials, or other related variables change, the performance of the predictive model may also change. That is, the predictive model is tuned to a specific context and dataset, and thus, it should be adjusted if the context changes.

Naturally, while the machine learning approach described in this article generalizes to other contexts, one should not assume that the same features would be the best features in other contexts as well. That is, the process should be started from the first step, i.e. extracting features, and followed as described in this article. That is, the predictive model that works on our data set would likely be different from a predictive model from other data sets – how different is a question that is left for future work. This is likely similar for all related studies.

6. CONCLUSIONS AND FUTURE WORK

In this work, we explored methods for early identification of students to guide from naturally accumulating programming process data. Such information can be useful for instructors and course designers, and can be used to create targeted interventions and to adjust materials accordingly. For example, the students who are performing well in the course may benefit from additional, more challenging tasks, while the students who are performing poorly are likely to benefit from rehearsal tasks as well as other activities that are typically used to help at-risk students.

The three main contributions of this article are as follows: (1) Analysis of the performance of existing source code snapshot-based methods for identifying high- and low-performing students in a new context; (2) Exploration of machine learning techniques for identifying high- and low-performing students; and (3) Analysis of cross-semester performance of the predictive models.

When analyzing the performance of the methods proposed by Jadud and Watson et al., we observed that the approaches had relatively poor performance on the data at our disposal. When exploring the performance of the machine learning techniques, the within-dataset performance was higher than that of the cross-semester performance, which was measured based on the predictive performance during a separate semester. This is explainable by the natural variance between semesters and student populations. Even so, with the cross-semester accuracy that ranges between 70% and 80%, reaching many of the right students is possible.

As a part of our future work, we are tuning the predictive models using additional data, seeking to further understand the students' behavior by delving deeper into their programming process, and conducting interviews that hopefully will shed further light on students' working practices as well as to those students who were misclassified. We are also performing targeted interventions within the studied context.

7. REFERENCES

[1] A. Ahadi and R. Lister. Geek genes, prior knowledge, stumbling points and learning edge momentum: Parts of the one elephant? In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 123–128, New York, NY, USA, 2013. ACM.

[2] A. Ahadi, R. Lister, and D. Teague. Falling behind early and staying behind when learning to program. In *Proceedings of the 25th Psychology of Programming Conference*, PPIG '14, 2014.

[3] J. Bennedsen and M. E. Caspersen. Abstraction ability as an indicator of success for learning object-oriented programming? *ACM SIGCSE Bulletin*, 38(2):39–43, 2006.

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

[5] S. Bergin and R. Reilly. Programming: factors that influence success. *ACM SIGCSE Bulletin*, 37(1):411–415, 2005.

[6] P. Byrne and G. Lyons. The effect of student attributes on success in programming. In *ACM SIGCSE Bulletin*, volume 33, pages 49–52. ACM, 2001.

[7] B. Cantwell Wilson and S. Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. In *ACM SIGCSE Bulletin*, volume 33, pages 184–188. ACM, 2001.

[8] Y. Cherenkova, D. Zingaro, and A. Petersen. Identifying challenging CS1 concepts in a large problem dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 695–700, New York, NY, USA, 2014. ACM.

[9] G. E. Evans and M. G. Simkin. What best predicts computer proficiency? *Communications of the ACM*, 32(11):1322–1327, 1989.

[10] D. Hagan and S. Markham. Does it help to have some programming experience before beginning a computing degree program? *ACM SIGCSE Bulletin*, 32(3):25–28, 2000.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[12] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[13] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[14] R. Hosseini, A. Vihavainen, and P. Brusilovsky. Exploring problem solving paths in a Java programming course. In *Proceedings of the 25th Workshop of the Psychology of Programming Interest Group*, 2014.

[15] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[16] H. Jang, J. Reeve, and E. L. Deci. Engaging students in learning activities: It is not autonomy support or structure but autonomy support and structure. *Journal of Educational Psychology*, 102(3):588, 2010.

[17] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

[18] J. Kurhila and A. Vihavainen. Management, structures and tools to scale up personal advising in large programming courses. In *Proceedings of the 2011 Conference on Information Technology Education*, SIGITE '11, pages 3–8, New York, NY, USA, 2011. ACM.

[19] R. Leeper and J. Silver. Predicting success in a first programming course. In *ACM SIGCSE Bulletin*, volume 14, pages 147–150. ACM, 1982.

[20] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull.*, 33(4):125–180, Dec. 2001.

[21] D. Orr, C. Gwosć, and N. Netz. *Social and economic conditions of student life in Europe: synopsis of indicators; final report; Eurostudent IV 2008-2011*. W. Bertelsmann Verlag, 2011.

[22] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

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

[24] L. Porter and D. Zingaro. Importance of early performance in CS1: Two conflicting assessment stories. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 295–300, New York, NY, USA, 2014. ACM.

[25] L. Porter, D. Zingaro, and R. Lister. Predicting student success using fine grain clicker data. In *Proceedings of the tenth annual conference on International computing education research*, pages 51–58. ACM, 2014.

[26] M. M. T. Rodrigo, R. S. Baker, M. C. Jadud, A. C. M. Amarra, T. Dy, M. B. V. Espejo-Lahoz, S. A. L. Lim, S. A. Pascua, J. O. Sugay, and E. S. Tabanao. Affective and behavioral predictors of novice programmer achievement. *ACM SIGCSE Bulletin*, 41(3):156–160, 2009.

[27] M. M. T. Rodrigo, E. Tabanao, M. B. E. Lahoz, and M. C. Jadud. Analyzing online protocols to characterize novice Java programmers. *Philippine Journal of Science*, 138(2):177–190, 2009.

[28] C. Romero and S. Ventura. Educational data mining: a review of the state of the art. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 40(6):601–618, 2010.

[29] C. Romero, S. Ventura, P. G. Espejo, and C. Hervás. Data mining algorithms to classify students. *Educational Data Mining 2008*.

[30] N. Rountree, J. Rountree, A. Robins, and R. Hannah. Interacting factors that predict success and failure in a CS1 course. In *ACM SIGCSE Bulletin*, volume 36, pages 101–104. ACM, 2004.

[31] E. Sierens, M. Vansteenkiste, L. Goossens, B. Soenens, and F. Dochy. The synergistic relationship of perceived

autonomy support and structure in the prediction of self-regulated learning. *British Journal of Educational Psychology*, 79(1):57–68, 2009.

[32] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, 29(9):850–858, Sept. 1986.

[33] J. Spacco. *Marmoset: a programming project assignment framework to improve the feedback cycle for students, faculty and researchers*. PhD thesis, 2006.

[34] M. V. Stein. Mathematical preparation as a basis for success in CS-II. *Journal of Computing Sciences in Colleges*, 17(4):28–38, 2002.

[35] M. Tukiainen and E. Mönkkönen. Programming aptitude testing as a prediction of learning to program. In *Proc. 14th Workshop of the Psychology of Programming Interest Group*, pages 45–57, 2002.

[36] P. R. Ventura Jr. Identifying predictors of success for an objects-first CS1. 2005.

[37] A. Vihavainen. Predicting students' performance in an introductory programming course using data from students' own programming process. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on.* IEEE, 2013.

[38] A. Vihavainen, J. Airaksinen, and C. Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 19–26, New York, NY, USA, 2014. ACM.

[39] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students' learning using Test My Code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 117–122. ACM, 2013.

[40] C. Watson and F. W. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44. ACM, 2014.

[41] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*, pages 319–323. IEEE, 2013.

[42] C. Watson, F. W. Li, and J. L. Godwin. No tests required: comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 469–474. ACM, 2014.

[43] L. H. Werth. *Predicting student performance in a beginning computer science class*, volume 18. ACM, 1986.

[44] S. Wiedenbeck, D. Labelle, and V. N. Kain. Factors affecting course outcomes in introductory programming. In *16th Annual Workshop of the Psychology of Programming Interest Group*, pages 97–109, 2004.

[45] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner. Building pair programming knowledge through a family of experiments. In *Proc. Empirical Software Engineering*, pages 143–152. IEEE.

[46] M. Yudelson, R. Hosseini, A. Vihavainen, and P. Brusilovsky. Investigating automated student modeling in a Java MOOC. In *Proceedings of The Seventh International Conference on Educational Data Mining 2014*, 2014.