Using data to improve programming instruction

Educational Data Mining in CS1 classes

Alisan Öztürk[1], Petra Bonfert-Taylor[2] and Armin Fügenschuh[3]

Abstract: Programming classes are difficult by nature and educators are eager to find ways to deal with high dropout rates. Todayĝs technologies allow us to capture programming-related student data, which can be used to identify students in need of assistance and in getting insights in student learning. In order to assist novice programming students in learning how to program, we developed a web-based programming environment, which is used by students throughout the whole course. While it also provides students with enhanced error messages, all data of studentsô interactions are captured. Through this data, we identified two metrics, related to small programming assignments, which highly correlate with student performance. These metrics along other features further enabled us to implement machine learning algorithms that could accurately predict dropout-prone students, early on in the course. Overall, methods of educational data mining can be utilized to assist both, students and educators in introductory programming courses.

Keywords: E-Learning, Machine Learning, Introductory Programming, Prediction, Enhanced Error Messages

1 Introduction

Programming skills are in increasing demand in todayĝs workforce, leading to a growth of introductory programming classes with a diverse student body. Many students struggle in these classes, resulting in high student dropout rates in universities. There are different approaches to tackle these issues and to understand how students learn to code. Because of todayĝs abilities to inexpensively store and analyze data, the area of educational data mining evolved, which allows for the identification of relevant data that comes from an educational setting. One area of educational data mining focuses on predicting student success in introductory programming classes. While past studies mainly collected and analyzed demographic data, more recent studies analyzed data that was directly collected from the platform within the coding platform. The ultimate goal is to eventually link student success in programming classes to such coding data. The Thayer School of Engineering at Dartmouth teaches a 10-week introductory programming course. The

[1] Helmut Schmidt Universität, Universität der Bundeswehr Hamburg, Holstenhofweg 85, 22043 Hamburg, Alisan.Oeztuerk@outlook.de

[2] Thayer School of Engineering, Dartmouth College, 14 Engineering Drive, 03755 Hanover, New Hampshire, USA, Petra.B.Taylor@dartmouth.edu

[3] Helmut Schmidt Universität, Universität der Bundeswehr Hamburg, Holstenhofweg 85, 22043, currently at Brandenburgische Technische Universität Cottbus, Platz der Deutschen Einheit 1, 03046 Cottbus, Fuegenschuh@b-tu.de

course is taught via an online coding environment, in which students take their first programming steps without hurdles to overcome, such as installation and compiler commands. At the same time, the data gathered from the coding environment enables us to apply educational data mining techniques to gather insights on student learning, as well as allows us to predict student success. These predictions can then be used as a decision support tool for instructors to provide targeted assistance for struggling students early on in the course. Whereas other studies analyzed data that was solely collected in a laboratory environment, the data from our study was collected during the entire course. We propose a number of methods to analyze and improve an introductory programming course, based on our course setting and the data collected therein. First, we show how compiler data can be captured and utilized to provide enhanced error messages to students. Then, we present a decision support tool for instructors that flags students based on a prediction of risk of them dropping the course.

2 Similar Work

Educators have been interested in understanding how students learn to program for more than fifty years. Research in the field of educational data mining focuses on many different aspects. [Ih15] categorizes work on student-focused research, such as predicting studentsô performance or programming related confusion and boredom, programming-focused research, aiming to identify programming behavior and lastly work on learning environments concerned with finding tools for instructors and various mechanisms for automated testing and grading of programming assignments. [FCJ04] introduces a system that responds with an easy to understand message on the most frequent java compiler error messages, as well as motivating feedback when a student compiles error-free code. [Ha10] presents another approach with an advanced system that stores compiler errors and student solutions in a database, which are suggested to help-seeking students. This has the advantage to provide assistance in situations, when it is impossible to find the root cause of the error message, which is especially true for the C programming language. [Fe12] shows an intelligent agent that detects student affective states based on key log data. [Be16a] presents a system that provides enhanced error messages which are further customized using parts of the studentôs source code. Recent work has focused on predicting student performance, which started with data sets that contained demographic data, see [Ko03] and [Ly09]. [MD10] and [Hu14] analyze data from a studentôs online learning management systems activity. Most recently, studies focus on predicting student performance in introductory programming courses by using data that is directly extracted from programming work. [Ta11], [Wa13] and [Be16b] propose various error compilation metrics and show a significant correlation to student performance. [Ah15] and [Ca17] present machine learning models to predict student performance with data sets containing features on the assignment level.

3 Methodology

In order to evaluate possible student performance metrics, we use linear regression models in combination with the coefficient of determination R^2, which describes the amount of variance explained by a regression model. Since it is important for these metrics to be as accurate as possible early on in a course, R^2 is calculated at different points of time for comparison purposes.

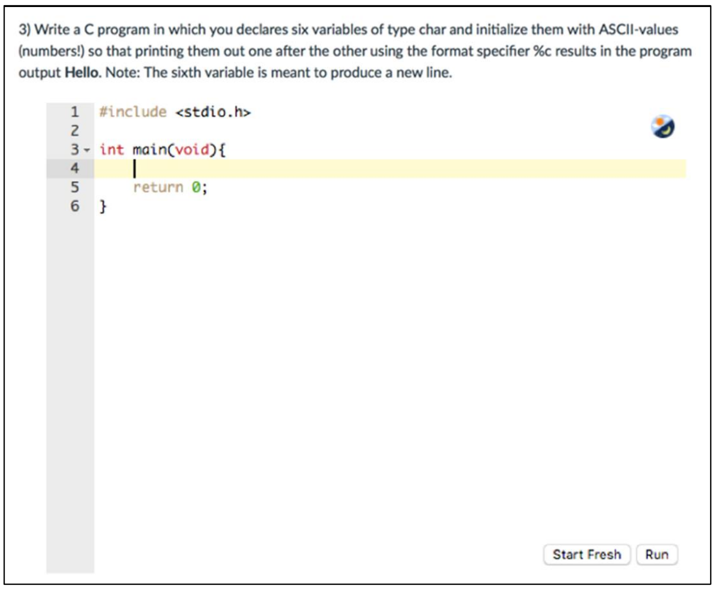
For the prediction models, we built machine learning classifiers which were, similar to the regression models, evaluated for every day of the week leading to the first midterm exam. A machine learning classifier is trained on a training data set, based on a set of pre-selected features, which optimally show a relationship with the target variable, in our case student overall performance. We selected our features by using a recursive feature elimination algorithm, based on [Gu02]. To train our models, we limited the number of selected features to five. This is one method to prevent overfitting, in order to not train a model that fits the data too well. There are various algorithms to train a classification model and we compared their performance. A logistic regression model performed best with our data set and was therefore selected to build our classifiers. Since we desire the ability of a classifier to accurately predict student success as early as possible in the course, we treat every day as a separate instance for our model, by training one classifier per day in the week before the first midterm exam. The classifiers predict a studentȏs performance on the first midterm exam, which in turn is predictive via a high correlation with a studentȏs overall course performance. The models were fine tuned in order to have a high specificity, which means the classifier correctly predicts all negative samples, which are in our case students with a low midterm performance. In addition, we used several other metrics, such as receiver operating characteristic (ROC) curve and the kappa-score ə. The ROC curve displays the trade-off between a classifiers false positive rate and true positive rate and allows to compare different classifiers. The area under the curve (AUC) is used to quantify this metric, see [PFK97] for more information. ə compares the observed accuracy with the expected accuracy, which measures how well the predictions agree with the ground truth, see [Co60]. All metrics were calculated by using a 10-fold cross validation, a technique to evaluate a model, by partitioning the data into ten equal slices and repeatedly training the model on nine slices and testing it on the remaining slice.

4 Course Setting

The course is taught as a traditional on-campus introductory C programming class with around 100 students. Throughout the term students have to solve various programming assignments in different contexts. This includes assignments completed in preparation for class, work performed during class as well as homework assignments. In addition, students take two midterm exams and one final exam. The first midterm exam is administered by the end of week three. The course grade is a composition of these efforts.

4.1 The Coding Environment

Development of an online coding environment started in 2015, with infrequent, experimental use prior to this study. We worked on a full integration of this environment into the course structure, which demanded a rework of the auto-grader system and an adaptation of several coding assignments. The online coding environment is embedded into the learning management system which provides students with all assignments, important deadlines and a course overview. It is a simple editor with syntax highlighting that allows to compile code on a server, reset a coding assignment to a starting state provided by the instructor and allow to provide user-input. Time-stamped data is collected on the level of individual keystrokes. Each data point is linked to a unique student and an assignment. In addition, every compile-attempt is saved with a snapshot of the source code and the compiler output. Fig. 1 shows an example of a coding assignment that is embedded in the learning management system.


3) Write a C program in which you declares six variables of type char and initialize them with ASCII-values (numbers!) so that printing them out one after the other using the format specifier %c results in the program output **Hello**. Note: The sixth variable is meant to produce a new line.

```
#include <stdio.h>

int main(void){

    return 0;
}
```

Fig. 1: Interface of the embedded coding editor

For the small programming assignment shown in Fig. 1, students were provided a pre-populated source code. In addition, such small programming assignments, which are part of the pre-class preparation, are automatically checked via our auto-grader. While in the assignment displayed in Fig. 1 we use input/output checking, since the assignment involves print statements, most assignments are checked via a function embedded but hidden to students within the pre-populated source code. This requires students to include a call to our ócheck functionôto pass their variable to this ócheck_functionô. It allows us to identify whether a student solved a programming problem without having to rely on output checking.

In addition, the instructor has access to an admin dashboard, which not only serves as an overview of a studentôs progress on a specific assignment, but also allows the instructor to join the studentôs programming session in order to provide help.

From the collected data we extracted up to 300 features of different types. Some are not engineered and are taken straight from the database, whereas some are inferred from the data and so we will refer to them as ñengineeredò features. Engineering new features from the data has no limits, which means that there is still unexplored potential in our data set that requires further analysis. Beyond that, features can be either categorized as assignment-level features or course-level features. Because every log entry is linked to a unique student and a specific assignment, we can calculate different metrics by only considering data pertaining to a specific assignment. For example, we can not only extract the total number of compiler errors a student encountered during the course, but also get the number of compiler errors a student encountered on a single assignment.

5 Results

This section is split into two parts. We start by presenting our implementation of enhanced error messages. Next, we briefly discuss our findings on student performance metrics, as well as our classification models. For a more detailed analysis see [Oe17].

5.1 Enhanced Error Messages

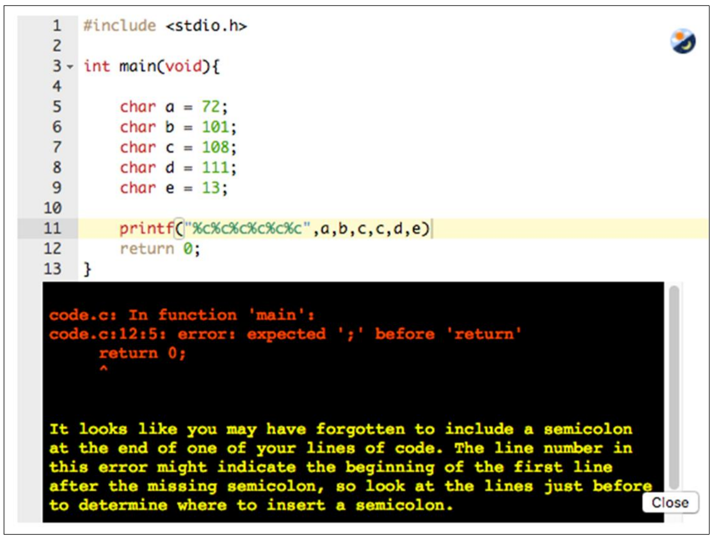
In the process of finding appropriate simplified explanations for compiler error messages it is important to not be too specific, because the same compiler error message can be the result of many different scenarios. At the same time the explanation must not be too general, in order to still be helpful for students.

We aggregated results from past student data and created explanations for the 50 most frequent compiler error messages, which cover 95 % of the error messages that students encountered. After the course ended, we compared this data to data from the current course and we were able to show that 15 out of 20 errors from the historic data set were also in the top 20 of the current data set. Out of over 150 unique error messages, the top 50 errors also covering 95 % of all errors encountered by the students. This shows that a data-driven approach to extract the most frequent errors proved effective, even when only a fraction of the data was available. The most frequent errors that were extracted from 68,248 submissions of 95 students are displayed in Table 1.

	Error	**Occurrence**
1	error: ôXôundeclared (first use in this function)	2903
2	error: expected expression before ôXôtoken	1189
3	error: expected ';' before 'X'	1108
4	error: expected identifier or '(' before 'X'	860
5	error: expected ';' before 'X' token	768
6	error: conflicting types for 'X'	761
7	error: expected '=', ',', ';', 'asm' or '__attribute__' before 'X'	644
8	error: parameter name omitted	643
9	error: ld returned 1 exit status	618
10	error: expected '=', ',', ';', 'asm' or '__attribute__' before 'X' token	571

Tab. 1: Most common C errors from our CS1 course

If the compile process fails, the enhanced error message is displayed below the original error message, see Fig. 2.



```
1   #include <stdio.h>
2
3   int main(void){
4
5       char a = 72;
6       char b = 101;
7       char c = 108;
8       char d = 111;
9       char e = 13;
10
11      printf("%c%c%c%c%c%c",a,b,c,c,d,e)
12      return 0;
13  }
```

```
code.c: In function 'main':
code.c:12:5: error: expected ';' before 'return'
     return 0;
     ^


It looks like you may have forgotten to include a semicolon
at the end of one of your lines of code. The line number in
this error might indicate the beginning of the first line
after the missing semicolon, so look at the lines just before
to determine where to insert a semicolon.
```

Fig. 2: Example of an enhanced error message

By the end of the course students should be able to use the original compiler error messages to help them debug their source code, without the need of enhanced error messages. Nonetheless, the benefits are intensely discussed. While [Be16] showed positive effects on students learning based on their own developed metric, [De14] claims that enhanced error messages were ineffective. We believe this is highly dependent on the course setting and the quality of the enhanced error messages.

Within our course we conducted a post course survey, where students did indeed find the enhanced error messages helpful, but this needs to be verified in future courses. In the future, we plan to integrate a rating system that allows us to identify unhelpful explanations that require a rework. The error compilation metrics were developed and tested with data from a different language and course setting, thus findings of our study might not be directly transferable to other studies, although our results strongly indicate an invariance of these metrics across programming languages.

5.2 Predicting student performance

In our analysis we evaluated several error compilation metrics. With our data set, we found that the error quotient performed best, and the regression model could explain up to 41 % of the variance. In addition, we found two metrics, linked to small programming assignments, that also showed a significant correlation with student performance. The first metric is the amount of time a student spent on small programming assignments. For this metric, we had to exclude many students that only used the coding environment when they had to. These students chose to code in other environments. Nonetheless, a regression model based on the time spent on the small programming assignments could explain up to 34 % of the variance. Lastly, we engineered a metric called solved ratio, which is calculated by dividing the sum of the solved small programming assignments, by the maximum of assignments solved by a student in the same course. The solved ratio could explain up to 50 %.

To assess the ability to predict student performance early on in the course, we built a classifier for each day of the third week into the course, leading to the first midterm exam. There is a high correlation between a student's first midterm performance and overall course performance, which indicates that a student who struggled on the first midterm exam is more likely to have difficulties in the course and might drop out of it. Due to this, it is sufficient enough to build classifiers that predict a student's midterm performance. We considered two scenarios, for the first scenario we only provided general features for the feature selection process, whereas for the second scenario all 300 features were available. Tab. 2 shows the performance of each classifier in the first scenario and the metrics we chose to evaluate the prediction performance.

Δ days	**Acc**	**AUC**	**κ**	**Specificity**	**At-risk Precision**
-7	0.76	0.82	0.39	0.71	0.45
-6	0.77	0.83	0.52	1.0	0.50
-5	0.81	0.90	0.58	1.0	0.54
-4	0.77	0.90	0.60	0.86	0.60
-3	0.82	0.90	0.56	0.71	0.62
-2	0.8	0.89	0.56	0.71	0.62
-1	0.82	0.90	0.66	0.86	0.67

Tab. 2: Classifier performance on the midterm exam using general features

Classifiers that predict student success are usually trained on highly imbalanced data sets. As such it would be possible for a classifier to score high accuracy (Acc) solely by classifying all students as high performing, which would be of no use. We instead use metrics such as AUC that compares the performance of a classifier to a random prediction (AUC = 0.5) and the kappa score ə both of which take into account imbalanced data sets. Recall that our goal is to have a high specificity. The at-risk precision of a classifier reflects its ability to only classify at-risk students as those, which definitely is a good goal, but in the end, the output of the classifier is a ranking based on the prediction probabilities assigned by the classifier. We found that even seven days before the midterm exam our model correctly predicts 71 % of the dropout-prone students. In the second scenario, the classifiers performed even better, which is as expected, because features unrelated to programming performance are omitted.

Building classifiers with assignment-level features revealed three distinct assignments, whose features were repeatedly favored by the feature selection algorithm, see Tab. 3.

Feature
Number of compiler errors encountered in assignment A
Amount of time spent on assignment A
Assignment B solved (binary variable)
Assignment C solved (binary variable)

Tab. 3: Important assignment-level features

All three assignments cover the basic concepts of programming and it was the first time students had to apply these concepts, which are branch statements (A), functions (B) and loops (C). By knowing this, instructors can make sure that all students understood these concepts by, for example, provide additional assignments about these concepts or spend additional lecture time reviewing these assignments.

Nonetheless, the second scenario has few uses beyond the current offering of the course. This is so since a future course would need to have an identical structure in order for the predictions to remain accurate. But this is impossible because instructors seek to continuously improve and adapt their course to student needs. For this reason, it is important to find general features, such as student performance metrics that can be extracted from heterogonous data sets and be therefore of use across institutions. Additionally, this would allow us to validate the classifiers with external data sets, which is a necessary step for further development of a reliable decision support tool for instructors.

6 Conclusion

This study shows the integration of an online coding environment within an on-campus course. In addition, we provide suggestions on how to implement enhanced error messages for the C programming language as a simple but effective feature for novice programming students. At the same time, enhanced error messages can reduce the amount of time students spend on fixing simple syntax errors and instructor time, which instead could be used to help students with more serious problems. Moreover, we demonstrate how features and student performance metrics can be engineered from programming log data, which then can be utilized to train classifiers that are able to predict student performance early on in the course. A decision support tool that ranks students based on prediction probabilities can then help to meaningfully allocate instructor time to students who might be in need of assistance. While we need to measure the effects of the enhanced error messages on student learning within our own course, we are required to validate the machine learning models on external data sets, which were similarly obtained from a full on-campus course. If predictions on student performance should be reliable early on in the course, then students need to start coding as soon as possible to increase the amount of available data. Ultimately, more findings extracted from these kinds of data sets have the potential to reveal more information on how students learn to code, which can then be applied to improve programming education.

References

[Ah15] Ahadi, A. et al.: 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. ACM, New York, pp. 121ï 130, 2014.

[Be16a] Becker, B.: An Effective Approach to Enhancing Compiler Error Messages. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education. ACM, New York, pp. 126ï 131, 2016.

[Be16b] Becker, B.: A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. In: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education. ACM, New York, pp. 296ï 301, 2016.

[Ca17] Castro-Wunsch, K. et al.: Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance. In: The 2017 ACM SIGCSE Technical Symposium. ACM, New York, pp. 111ï 116, 2017.

[Co60] Cohen, J.: A coefficient of agreement for nominal scales. In: Educational and Psychological Measurement vol. 1, pp. 37ï 46, 1960.

[De14] Denny, P. et al.: Enhancing Syntax Error Messages Appears Ineffectual. In: Proceedings of the 19th ACM Conference on Innovation & Technology in Computer Science Education. ACM, New York, pp. 273ï 278, 2014.

[FCJ04] Flowers, T.; Carver, C.; Jackson, J.: Empowering students and building confidence in novice programmers through Gauntlet. In: 34th Annual Frontiers in Education, IEEE, Savannah, pp. 433ï 436, 2004.

[Fe12] Felipe, D. et al.: Towards the development of intelligent agent for novice C/C++ programmers through affective analysis of event logs. In: Lecture Notes in Engineering and Computer Science vol. 1, pp. 511ï 518, 2012.

[Gu02] Guyon, I. et al.: Gene Selection for Cancer Classification using Support Vector Machines. In: Machine Learning vol. 46, no. 1, pp. 389ï 422, 2002.

[Ha10] Hartmann, B. et al.: What Would Other Programmers Do: Suggesting Solutions to Error Messages. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, pp. 1019ï 1028, 2010.

[Hu14] Hu, Y. et al.: Developing early warning systems to predict studentsô online learning performance. In: Computers in Human Behavior vol. 36, no. C, pp. 469ï 478, 2014.

[Ih15] Ihantola, P. et al.: Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In: Proceedings of the 2015 ITiCSE on Working Group Reports. ACM, New York, pp. 41ï 63, 2015.

[Ko03] Kotsiantis, S. et al.: Preventing Student Dropout in Distance Learning Using Machine Learning Techniques. In: Knowledge-Based Intelligent Information and Engineering Systems. Berlin, Heidelberg, pp. 267ï 274, 2003.

[Ly09] Lykourentzou, I. et al.: Dropout prediction in e-learning courses through the combination of machine learning techniques. In: Computers & Education vol. 53, Nr. 3, pp. 950ï 965, 2009.

[MD10] Macfadyen, L.; Dawson, S.: Mining LMS data to develop an ñearly warning systemò for educators: A proof of concept. In: Computers & Education vol. 54, no. 2, pp. 588ï 599, 2010.

[Oe17] Oeztuerk, A.: A Data-Driven Approach to Improve the Teaching of Programming, Masterôs Thesis, Applied Mathematics and Optimization Series AMOS#57, 2017.

[PFK97] Provost, F.; Fawcett, T.; Kohavi, R.: The Case Against Accuracy Estimation for Comparing Induction Algorithms. In: Proceedings of the 15th International Conference on Machine Learning. Morgan Kaufmann, San Francisco, pp. 445ï 453, 1997.

[Ta11] Tabanao, E. et al.: Predicting at-risk novice Java programmers through the analysis of online protocols. In: Proceedings of the Seventh International Workshop on Computing Education Research. ACM, New York, p. 85, 2011.

[Wa13] Watson, C. et al.: Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In: Proceedings of the 13th International Conference on Advanced Learning Technologies, IEEE, pp. 319ï 323, 2013.