Data-driven Curricular Decisions in Introductory Computing Classes

Prof. Petra Bonfert-Taylor, Dartmouth College

Petra Bonfert-Taylor is a Professor and an Instructional Designer at the Thayer School of Engineering at Dartmouth College. She received her Ph.D. in Mathematics from Technical University of Berlin (Germany) in 1996 and subsequently spent three years as a postdoctoral fellow at the University of Michigan before accepting a tenure-track position in the Mathematics Department at Wesleyan University. She left Wesleyan as a tenured full professor in 2015 for her current position at Dartmouth College. Petra has published extensively and lectured widely to national and international audiences. Her work has been recognized by the National Science Foundation with numerous research grants. She is equally passionate about her teaching and has recently designed and created a Massively Open Online Course “Analysis of a Complex Kind” on Coursera and is in the process of creating a seven-MOOC Professional Certificate on C-programming for edX. The recipient of the Binswanger Prize for Excellence in Teaching at Wesleyan University and the Excellence in Teaching Award at the Thayer School of Engineering, Petra has a strong interest in broadening access to high-quality higher education and pedagogical innovations that aid in providing equal opportunities to students from all backgrounds.

Mr. Alisan Oeztuerk, German Army

Mr. Ben Servoz, Dartmouth College
Data-Driven Curricular Decisions in Introductory Computing Classes

1. Introduction

Computer programming has become an essential skill in young people’s trajectories for academic success in STEM, entry into STEM professions, and increasingly across a broader spectrum of career choices. Yet, drop-out rates remain high in overcrowded introductory programming courses. At the same time, recruiting and retention of a diverse student body, particularly women and students from underrepresented populations, into computing and STEM careers remains a complex challenge involving broad differences in student preparation, cultures and needs, see [5], [12]. For a study on various reasons why students drop out of introductory computer science courses, see [8].

Learning how to code typically involves having to overcome profound initial barriers. For example, navigating unfamiliar and confusing programming environments can deter students from experimentation and playful practicing, and lacking feedback can leave students lost during homework. In order to eliminate any such barriers for programming novices, we have developed a student-centered online coding environment that provides instant feedback to students in introductory programming classes, that allows for early interventions and that increases and encourages student experimentation, exploration, and time spent coding. At the same time, our coding environment collects fine-grained data and compilation snapshots in order to help the instructor identify early on students who might benefit from extra support.

In this paper, we describe educational data mining performed on the collected data during a spring 2017 offering of an introductory programming course, with the aim to build an early intervention system. We describe insights gained from the data as well as changes to future iterations of the course based upon these insights.

2. Setting

The course discussed in this paper is a 10-week introductory programming course, which is a prerequisite to the engineering major at Dartmouth’s Thayer School of Engineering. Students, the majority of whom are freshmen (65%) or sophomores (25%) are introduced to the programming languages C and MATLAB. About 70% of participants in this course have no prior programming background. The course typically has around 100 students and is taught in the flipped classroom style; students watch short videos and engage in quizzes in preparation for class and work on programming assignments during class. Many of the pre-class quizzes contain simple coding questions, relevant to the videos students just watched, and often students are provided a “try-it-out” coding window, embedded in the quiz. While most of the in-class programming assignments are solved in pairs, after-class homework assignments are of individual nature. “Real” coding assignments start in week two; the first week of classes is spent going over computer architecture, binary number systems, software installations, the “hello world” program and more.
3. Review of Existing Similar Platforms to Improve Student Coding Skills

In this section, we review some of the existing programming solutions that allow for automatic grading of student solutions.

*CloudCoder*: CloudCoder [10] is an open source web-based environment designed for managing short programming exercises. The tool evaluates the correctness of a student submission via input-output checking of multiple pre-designed test cases and gives immediate feedback to students. As is the case for CeMeCode, CloudCoder uses the Ace editor. CloudCoder is inspired by CodingBat and supports C/C++, Java, Python, and Ruby.

*Python Classroom Response System*: Another example of utilizing a web-based coding environment is the Python Classroom Response System (PCRS) described in [13]. The idea and motivation behind the tool is to facilitate the method of Peer Instruction in a programming class. The instructor can create programming assignments and test cases built around expected misconceptions in order to observe student problems during class in real time so as to directly be able to address these. PCRS is written in Python and is designed for the Python programming language. A support for C has been established and a support for Java and SQL are in development.

*BlueJ*: The BlueJ system [6] is a free, integrated Java development environment specifically designed for teaching and learning object-oriented programming. It consists of a simple interface that is interactive and designed for teaching. The majority of educational data mining research for introductory programming classes in Java utilizes the BlueJ system.

Other free and commercial auto grading tools and platforms are emerging such as AutoGradr, HackerRank, Web-CAT, Vocareum, Taskgrader and more.

4. CeMeCode

Our coding window, CeMeCode, is a lightweight web based coding environment, developed by our team. It integrates with Dartmouth’s single sign-on authentication systems and can be embedded into other tools such as our Learning Management System (see Figure 1) and our in-class response system.

CeMeCode is embedded in pre-class quizzes, used during in-class coding exercises and at times deployed in after-class homework assignments. The coding environment is designed to increase and encourage student experimentation, exploration, and time spent coding, while providing instant feedback to students via automatically graded assignments. CeMeCode furthermore collects a robust amount of mineable data concerning student performance and progress, including fine-grained (keystroke-by-keystroke) data and compilation snapshots.
CeMeCode was initially developed with the goal to reduce the hump (installing and familiarizing oneself with an IDE, having to learn Linux, SSHing to a server, mastering compiler commands, etc.) students have to overcome when first starting to program in C. As mentioned earlier, the programming class described in this paper is taught in the flipped classroom style, with students watching short videos and completing knowledge check quizzes before class and working on programming assignments during class. In order to encourage students to playfully experiment with code, a coding window is embedded alongside each preparatory video, with only one button that compiles and upon successful compilation, runs the code. In case of a compilation error, only the first error message is displayed, along with a likely “friendly” explanation, derived from errors students have made previously at a similar stage of learning.
CeMeCode is now furthermore used during in-class coding exercises, homework and pre-class quizzes. Through one link (or via an iframe command), distributed to all (via our LMS or our in-class response system), each student gains access to their own coding environment in which their personal code is saved. The coding window can be pre-populated by the instructor with starter code upon which students have to build. An automatic grading feature can furthermore be invoked that provides instant feedback to students. Through a dashboard (see Figure 2) the instructor can monitor the class' progress (check whose code compiles, whose code runs correctly, etc.) and view (as well as edit) any one student's code in order to help them troubleshoot.

![Figure 2: An illustrative example of the dashboard available to the instructor.](image)

At the same time, keystroke-by-keystroke data along with compilation snapshots is collected in order to aid our team in designing an early intervention system.

One of the outcomes of our educational data analysis (see Section “Results”) is to modify the curriculum in future iterations of the course and start coding on the first day of classes with the aid of CeMeCode. Since no installation, account creation or other hurdle is necessary, all students can be walked through their first programming steps in class.

5. Data Collection in CeMeCode

When a student is presented a CeMeCode window, a permanent link with a Linux server is established “behind the scenes”. This link is used to provide interactivity between the instructor and the students, and also to monitor and to log, in real time, the student's interactions with the piece of code on which they are working. At its core, this is simple data mining with an educational twist. The server is also used to compile and, in case of successful compilation, run student code submissions which means that students are coding in a native environment. Lastly, the server is responsible for checking the integrity of submissions as well as checking solutions
for correctness. Because these operations happen on a trusted server, the results are equally trusted and can be used for grading purposes. The components of CeMeCode are outlined in Figure 3.

This largely contrasts with what is possible in a browser emulated coding environment. For example, in a browser emulated environment it would be impossible to trust the integrity of the student submissions as code running on their own machine (thus tamperable) would be responsible for the verification phase. Most of today’s emulated environments also tend to not be 100% faithful to a native environment and so student code might behave slightly differently when switching from the in-browser experience to the native Linux environment in which students are bound to develop eventually.

All the events describing student interaction with a piece of code are time-stamped & stored in a MySQL database. This includes every keystroke (insertions as well as deletions), regardless of whether the code window is focused in the browser or not. In addition, all code submitted to the server for compilation as well as any other user interface interactions are time-stamped and stored. This results in millions of events being gathered which can later be used to construct any number of models. An excerpt of the database is shown in in Figure 4.
6. Data Analysis Features

The data collection in CeMeCode allows us to extract a number of features which vary in type. 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 “engineered” features.

a. Non-engineered features:
These include features such as successful or failed compilation events which can be directly read from the code submission data in the database. We also monitor various features related to syntax errors, such as type and location of the error. Additionally, more general features such as the number of assignments worked on or the total number of submissions on a particular assignment, give us insight into a student's activity patterns.

- Time-related features:
These are features which can be inferred from the database as well and include for example the amount of time a student worked on a specific assignment as compared to other students in the class. Because a student can freely start and stop working on an assignment (as opposed to a lab setting), we need to differentiate between time spent actively coding and other activities such as having the coding exercise open yet not being actively focused on it.
• Course-level and assignment-level features:
  As seen in the previous section, every log entry is linked to a unique student and a specific assignment. This enables us to not only extract most of the above-mentioned features on the course level but also on an assignment level. For example, we can extract, from the database, the number of compiler errors a student encountered on a specific assignment just as we can compute this number over all assignments for a particular student.

b. Engineered features:
  Lastly, we construct engineered features, such as the keystroke latency [7], the Error Quotient [11], the Repeated Error Density [1], and others that might help to partially explain a student's success (or lack thereof).

Overall, we use more than 300 features from which we extract the most predictive ones in order to build machine learning classifiers (see Section Results).

7. Results

In our effort to predict students at-risk of dropping out of the course, we chose to build machine learning classifiers which predict a student's midterm performance based on features we identified through recursive feature elimination (RFE) [2]. See Figure 5.

Our 10-week course tests students in two midterm exams (one in week four, one in week seven) and a final exam. Data from previous years has shown that students who perform poorly on the first midterm exam have typically already fallen far enough behind so as to not be able to compensate for the deficit during the remainder of the course, see Figure 6. Our goal is therefore to predict such struggling students as early as possible in order to provide targeted interventions as early as in week two or three of the course.
To determine which students are at-risk of dropping out of the course, we defined a cut-off based on performance on the first midterm exam. Students who scored below 75% are defined as at-risk of either dropping out of or failing the course.

We therefore built, for each day of week three (second week of active student coding) of the course, one classifier using the data available by the end of that day. For each classifier, the best features were selected by RFE and the classifiers were evaluated based on several metrics. Our priority was to build classifiers with a high specificity, meaning correctly classifying the negative class (students scoring below 75% on the first midterm), while keeping the false negatives reasonably low. Based on the classification probabilities students can then be ranked to provide decision support for the instructor, such as which students to review first. While of course it would be ideal if the instructor could review all programming practices of each student individually, in reality this is impossible given the size of the class. When provided with a short list of students who could potentially be experiencing difficulties it becomes more feasible for the instructor to more carefully review these students’ progress in order to identify who might benefit from extra support.

For this identification task we compared several classifiers, and in the end a logistic regression classifier performed best with our data. Out of more than 300 features, we limited the number of features used to build each model to five, in order to prevent overfitting [3]. Furthermore, we performed a 10-fold cross-validation within our machine learning pipeline to assess the performance of our model.

Data was continuously generated via students working on a variety of tasks (homework, pre-class work, etc.). In total, seven classifiers were built (one for each day of week three) to evaluate the ability to predict at-risk students. Interestingly, some features were repeatedly
selected by RFE over all of the other available features. Note that not all features are available at all times, since some features are tied to assignments which are released over time as the course progresses.

Table 1 lists the five most frequently selected features in the creation of classifiers (recall that these selections occur automatically using RFE). The last column represents the number of days during which the feature was available for the feature selection process, aside hundreds of others.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Number of times selected (out of 7)</th>
<th>Availability (out of 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Number of compiler errors encountered in assignment X</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>(b) Error Quotient of the student</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>(c) Binary variable - 1 if the student solved assignment Y, else 0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>(d) Amount of time spent on assignment X</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>(e) Binary variable - 1 if the student solved assignment Z, else 0</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

*Table 1: Five most frequently selected features using RFE.*

Out of the top five features that were selected, two features derived from assignment X, one from assignment Y and one from assignment Z (see below for details on these assignments). Furthermore, the error quotient, a metric that quantifies the error compilation behavior of a student, was selected five out of seven times. The only other study that discusses the error quotient in a C-programming context is [9]. Given the importance of this feature in our course setting it definitely merits further investigation.

Assignment X is about branch statements: Students are asked to first read an integer from user input, determine whether the user input lies within a predetermined range and then assign a score based on the different possibilities by using if/else statements. The two features built from assignment X that stood out in the RFE process are the number of compilation errors encountered by a student while working on assignment X (see feature (a) in the table), and the amount of time a student spent on this assignment (see feature (d) in the table). The relevance of these two features is further emphasized by the observation that students who did not pass the first midterm exam encountered more compilation errors on average on assignment X than students who did pass the exam. In addition, they also spent more time on average, trying to solve this assignment. Note that assignment X is the first assignment in the course on which students use branch statements.
In assignment Y students are instructed to write a function which calculates the sum of two variables. This is the first assignment in which students use functions with parameters. Overall, of all code submissions for assignment Y (recall that each student submits and compiles code repeatedly), 75% contained syntax errors and resulted in a compilation error, indicating that this assignment was difficult for all students. This was the highest error rate amongst all assignments in the same category. Note that in the end, 86% of students were able to solve this assignment correctly. Feature (c) in the above table encodes this success or lack thereof.

Lastly, assignment Z instructs students to write a loop that adds up the numbers from \(n\) to \(2n\), where \(n\) is set by user input. This is the first assignment in which loops are practiced. Again, being able to solve this assignment correctly (see feature (e) in the table) crystallized as important for the prediction task and was therefore selected by RFE in the feature selection process each of the four times it was available.

One might think of the repeated selection of these five features, predictive of student success, as a random occurrence, but there may be more to this if one approaches the question from an educational point of view: All three concepts, branch statements, loops and functions are essential concepts of programming that involve non-linear program execution. In each case, the given assignments presented the first time where students had to apply these concepts to solve a programming task. Note that these early concepts are essential for student success in a tightly knit course since students’ ability to move on to deeper concepts is significantly hindered by a lack of understanding of the basics. Figure 7, which was generated via a principal component analysis (PCA) [4], helps to visualize the possible information gain from the selected features.

![Figure 6: Principal Component Analysis of the selected features.](image)

Students who did not pass the course are represented by triangles, the others by circles (recall that “pass” here refers to receiving a course grade above 75%). The two outliers represent students for whom no data was available for assignment X. The vectors represent the influence of the various features on students’ success and can partly explain why these features have been selected by RFE. For example, we can see that successful students are generally concentrated in
the lower left corner, which is achieved by solving assignments Y and Z and having a low error quotient at the same time. Furthermore, the PCA also emphasizes that students who are at-risk of failing the course do so for a variety of different reasons; these students are therefore not found concentrated in one area.

This knowledge can not only be utilized to identify students struggling to understand these concepts, but it also helps in the effort to enhance and redesign the course itself in order to better address these essential concepts and improve the learning experience for students in future courses.

8. Limitations

The quality of machine learning models as well as our findings from the data-driven analysis rely heavily on the data that was used. This issue is intensified when features on the assignment-level are used to build the prediction models. Even within the same institution, it is impossible to replicate a course offering precisely if the instructors continuously improve and adapt their courses to student needs. For this reason, further analysis and research are necessary in order to identify general features that are independent of course-specific implementations. At the same time, machine learning models trained on more general features can be implemented with data from different institutions to help with validation of the results. In addition, insights that lead to a change of the course structure have to be carefully studied in order to ascertain that they indeed have the desired impact on student learning.

9. Conclusion and Future Work

In response to the results of our recent data analysis, our introductory programming course is currently undergoing a curricular redesign. This is an ongoing process, informed by repeated analysis of newly collected data for each course offering. We do not expect to get everything “right” in this next iteration, but we hope to start converging towards a new instructional model that better serves and retains a more diverse group of students, especially students from underprepared backgrounds.

Traditionally, the first day of classes was spent attending to typical first-day-of-class housekeeping business as well as speaking about hardware and software and introducing the binary number system. Next came an introduction to Linux (since in order to minimize installation issues and related troubleshooting, we have students program on our remote servers), more on binary numbers, including the 2’s complement representation, the hexadecimal number system, etc. By the end of the first week of classes, the first program involving simple user input, a computation and simple output, would finally be discussed.

In our redesigned course, all of these topics will still be addressed, but not right away. Instead, learning by doing (programming) takes the center stage. On the first day of class, students will write programs to print statements to the screen as well as learn how to repeat statements via loops. Since non-linear coding structures form an essential part of programming and also present an essential difficulty for student understanding, ample time will be spent to allow each student to code a loop on their own. Only one concept will be introduced at a time, with all students actively engaged and student helpers circulating in the classroom to ensure every student is
following along. Other crucial coding concepts (such as branch statements and functions) will be introduced in similarly slow and step-by-step fashion, and early on during the curriculum. Students will still learn about binary representations of numbers and characters; but rather than front-loading the curriculum with these topics, they will be introduced in places where they can present an “aha” moment. For example, once students have experienced the results of integer division, the binary representation of integers will be introduced in order to explain the unexpected observed behavior. Once students have seen that counting (integers) up by 1’s will eventually yield a negative number, the binary 2’s complement will be introduced. And once they have experienced round-off errors with decimal numbers, they will learn how floating-point numbers are stored in memory. Overall, the curriculum will be driven by the goal to increase students’ time spent coding, with topics such as computer storage and memory organization added in using a just-in-time teaching approach.

We will continue to collect fine-grained data and analyze the newly collected data in order to update our early prediction classifiers. We recognize that it is not sufficient to simply be able to predict at-risk students early on. At the same time, we need to develop intervention strategies. Along with the overall course redesign we will furthermore develop individual student intervention strategies, guided by our data analysis.

By continuously re-analyzing and re-evaluating newly collected data we hope to eventually be able to design a student-centered curriculum. We cannot expect to design one curriculum that serves all perfectly, but we hope to design a curriculum that leaves no-one behind. Our ultimate goal is to identify differences in student learning styles and design an adaptive curriculum that adjusts to a particular student’s needs.
References


