Using Students' Programming Behavior to Predict Success in an Introductory Mathematics Course

Arto Vihavainen, Matti Luukkainen, Jaakko Kurhila
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
Fi-00014 University of Helsinki
{ avihavai, mluukkai, kurhila }@cs.helsinki.fi

ABSTRACT

Computer science students starting their studies at our university often fail their first mandatory mathematics course, as they are not required to have a strong background in mathematics. Failing can also be partly explained by the need to adjust to a new environment and new working practices. Here, we are looking for indicators in students' working practices that could be used to point out students that are at risk of failing some of their courses, and could benefit from an intervention. We present initial results on how freshman students' programming behavior in an introductory programming course can be used to predict their success in a concurrently organized introductory mathematics course. A plugin in students' programming environment gathers snapshots (time, code changes) from students actual programming process. Gathered snapshots are transformed to data items that contain features indicating e.g. deadline-driven mentality or eagerness. Our results using Bayesian networks indicate that we can identify students with a high likelihood of failing their mathematics course already at a very early phase of their studies using only data that represents their programming behavior.

Keywords

code snapshots, programming behavior, first-year challenges

1. INTRODUCTION

Students face a number of challenges during the first term of their studies at a higher education institution [4]. They need to find a place within a new community, and adjust their learning strategies and styles to fit the requirements of their chosen study. The multitude of challenges is often unforeseen and surprising for the students, and many end up failing some of their first courses. However, if a student has good study habits, she is more likely to succeed in her studies [12].

Computer science (CS) students typically start their studies with introductory programming courses and mathematics. Although mathematics is typically a minor subject, it is frequently considered as the basis of CS. Especially algorithms-related courses require a fair amount of mathematical maturity. As such, it is important that students fare well in their mathematics studies.

There are several systems that gather snapshots from students' programming process, e.g. [13, 15]. Currently, the research that utilizes snapshots has focused on e.g. modeling how students solve a specific exercise (see e.g. [11, 8]), and how e.g. compilation errors and successes can be used to predict success in a programming course (see e.g. [9, 14]). However, to our knowledge, there has not been much attention on how the students' *study practices* and *behavior*, especially their use of time extracted from snapshots, reflect on their results on current and parallel courses.

In our work, we are investigating students' programming process and seeking indicators of bad study habits that could be used to highlight students at-risk of failing some of their first term studies. In this article, we describe how we can identify students that are at risk of failing a 14-week introductory mathematics course using only their programming behavior from a parallel introductory programming course. Our initial results are promising, and with a population of 52 students, we are able to accurately predict the mathematics course success after only four weeks of programming.

This paper is organized as follows. In section 2, we briefly describe our educational arrangements as well as the tool that enables recording snapshots from students' programming process. Section 3 describes the data and the feature generation process. Section 4 describes the methodology used for analysing the data and presents our results, and Section 5 outlines future work.

2. CONTEXT, PEDAGOGY AND TOOLS

The academic year at the University of Helsinki is split into four seven-week teaching periods. Each period starts and ends simultaneously throughout the University, and each period is followed by a one week intermission before the start of the next period. The last week of each period is usually devoted to course exams.

The first two periods for CS majors are packed with mandatory courses: Introduction to Programming Part I (*7 weeks, period 1*), Introduction to Programming Part II (*7 weeks, period 2*), Software Modeling (*7 weeks, period 2*), that are offered by the Department of CS. In addition, students are expected to enroll into Introduction to University Mathematics (*14 weeks, periods 1 and 2*) that is organized by the Department of Mathematics and Statistics.

Both courses Introduction to University Mathematics and the Introduction to Programming Part I are organized using the Extreme Apprenticeship method (XA) [16], which is a modern interpretation of apprenticeship-based learning [5, 6]. XA values students' personal effort and intensive interaction between the learner and the advisor, and emphasizes deliberate practice [7] that aims towards mastering the craft.

As a craft can only be mastered by practising it, XA-based courses contain lots of exercises. For example, during the first week of their programming course, CS freshmen work already on tens of programming tasks.

As XA stresses activity to be as genuine as possible, the students start working with industry-strength tools from day one. We use NetBeans, which is an open source IDE (integrated development environment), bundled with an automated assessment service called Test My Code (TMC) [15], which is used to download and submit exercises; moreover, TMC is used to run tests on the students' code in order to verify the correctness of an exercise.

In addition to the assessment capabilities, on student's permission, TMC gathers data from students' programming process. Currently, a snapshot is taken whenever a student saves her code, compiles the code, or pastes code into the IDE. Each snapshot contains student id, timestamp, source code changes and possible configuration modifications.

3. DATA AND FEATURES

During the Fall 2012, we gathered over 48 000 snapshots from 52 students that participated to both Introduction to Programming Part I and Introduction to University Mathematics. The 52 students include only those that had more than 100 snapshots, i.e. they had not disabled the snapshot gathering mechanism at an early part of the course and had put at least some effort to solving the exercises. Out of the 52 students, 28 passed the mathematics course, and 24 failed it, while 43 passed the programming course. Although some students passed the courses later in a separate exam, we currently only consider their success in the actual course.

Students' snapshots are aggregated to describe weekly median, average, minimum, maximum and standard deviation of their working in the following dimensions:

1. hour of working
2. minutes to deadline
3. minutes between sequential snapshots
4. edit distance between sequential snapshots.

The first two aggregate statistics are generated by analysing the snapshot time and its distance from the deadline of the exercise that the student is currently working on. Aggregate statistics three and four are generated by sorting the snapshots based on their time and comparing the code changes and snapshot timestamps between sequential snapshots. Due to the large number of snapshots and the relatively small amount of code changes between each snapshot, we calculate the edit distances using an extension of the Ukkonen's algorithm by Bergel and Roach [2].

In addition to the above statistics, we generate weekly:

- minutes spent programming
- % of programming done during night (22-07)
- number of compilation errors
- number of style- and programming-related issues.

The number of compilation errors is gathered by compiling the program code for each snapshot, and gathering statistics out of the code compilation results. For identifying the style- and programming-related issues (e.g. wrong indentation, too long methods, copy-paste code, variables shadowing variables, and infinite loops), we utilize Checkstyle [3] and FindBugs [1].

If a student has not worked on the programming exercises during a specific week, values are entered as empty values. In our data, 21 of the students skipped at least one week of programming, and 9 of them ended up failing the mathematics course. We have purposefully left out the number of exercises each student has completed from the features, as our focus is on analysing the students' working behavior during the course.

3.1 Features

Each of the aggregated value is considered as a *feature*, and each week adds over 30 features that represent students' behavior during the specific week. Let us consider some of the features in more detail.

Figure 1 displays probability densities[1] for the standard deviation of snapshot time distances to deadline. Students that are more likely to fail the mathematics course have been working on the exercises during a smaller time period, e.g. during a single "crunch", while the students that are more likely to pass the mathematics course have worked on the exercises during several days.

The same "crunch" effect is visible in Figure 2, which displays probability densities for the maximum minutes between snapshots during week 3. Here, the students that take a larger pause (over 3.5 days) while working on the exercises are more likely to pass the mathematics course.

Figure 3 displays the amount of programming done during nights during week 6. The students that did not program between 22-07 hrs are slightly more likely to fail the mathematics course, and in our data, all the students that programmed more than 70% of their time during night passed the mathematics course.

[1]Note that the figures are plotted in R, and the used bandwidth for the density function is the default "nrd0". If a value is missing, i.e. has a value NA, it is removed from the dataset prior to plotting.

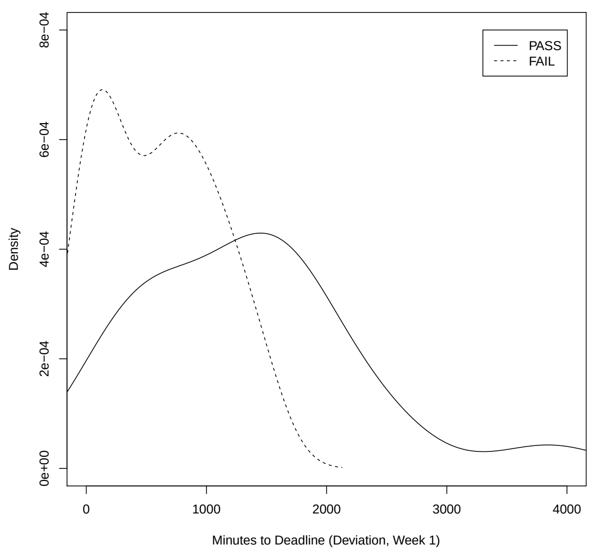

Figure 1: "Minutes to Deadline (Deviation, Week 1)" displayed using probability densities for groups that have passed and failed the course Introduction to University Mathematics.

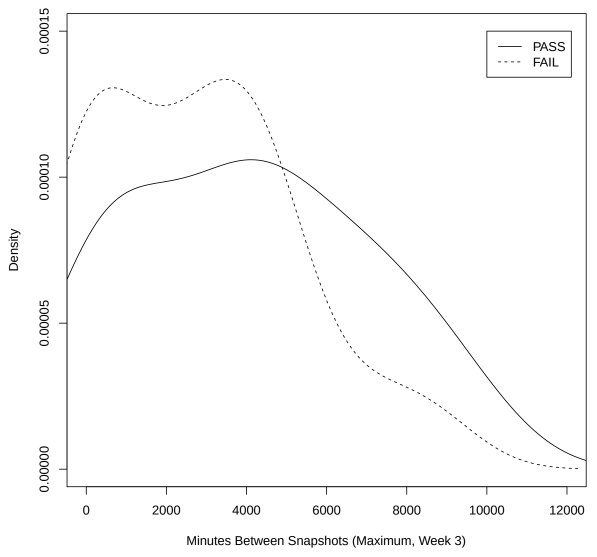

Figure 2: "Minutes Between Snapshots (Maximum, Week 3)" displayed using probability densities for groups that have passed and failed the course Introduction to University Mathematics.

4. METHODOLOGY AND RESULTS

We consider identifying the students' that are likely to succeed or fail their introductory mathematics course a classification problem. The course result (pass/fail) is used as the class to predict, and each feature vector contains the aggregated values from a student's snapshots. In total there are

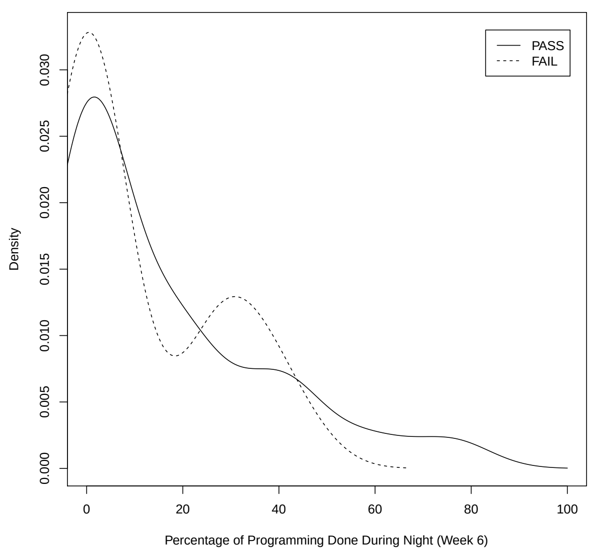

Figure 3: "Percentage of Programming Done During Night (Week 6)" displayed using probability densities for groups that have passed and failed the course Introduction to University Mathematics.

52 feature vectors with class labels, one for each student. We utilize a non-parametric Bayesian network tool called B-Course [10] for both modeling the dependencies between the features in the data, and for building the classifiers.

Validation is performed using student level leave-one-out cross-validation, and if a feature value is missing, i.e. a student has not programmed during a specific week, B-Course ignores the value when calculating the class probabilities.

We have built 12 separate classifiers for our data. Half of the classifiers are built using data from only the students' behavior, without results from static and dynamic code analysis, i.e. compilation errors and style- and programming-related issues, and the other half include static and dynamic code analysis results. Each of the classifier represents students' behavior up to a specific week in the programming course. The features for separate weeks are currently the same; students' weekly behavior is aggregated to feature values.

The following results contain accuracy, precision, recall, and the weighted harmonic mean of precision and recall (F-measure). Precision, recall, and F-measure are calculated assuming that we are predicting success in the mathematics course.

Results for the classifiers that did not include code analysis results are described in Table 1. Already after one week of programming, we are able to identify students' that are likely to fail the mathematics course with an `84.6%` accuracy. After 5 weeks of programming, the accuracy is `98.1%`.

Table 2 describes the results where dynamic and static code analysis results are included in addition to the students' be-

Week	Accuracy	Precision	Recall	F-Measure
1	84.6 %	85.7 %	85.7 %	85.7 %
2	88.5 %	92.3 %	85.7 %	88.9 %
3	92.3 %	92.3 %	92.3 %	92.3 %
4	94.2 %	100 %	89.3 %	94.3 %
5–6	98.1 %	100 %	96.4 %	98.2 %

Table 1: Results for data that includes students' programming behavior, and excludes compilation errors and style- and programming-related issues.

havior. After a single week of programming, the accuracy is 88.5%, and at the end of fourth week, we are able to identify the students' at risk with 100% accuracy.

Week	Accuracy	Precision	Recall	F-Measure
1	88.5 %	92.3 %	85.7 %	88.9 %
2	94.2 %	96.3 %	92.9 %	94.5 %
3	96.2 %	96.4 %	96.4 %	96.4 %
4–6	100 %	100 %	100 %	100 %

Table 2: Results for data that includes students' programming behavior as well as compilation errors and style- and programming-related issues.

5. DISCUSSION AND FUTURE WORK

With our current dataset, we are able to predict the students' success and failure in a 14-week introductory mathematics course already after a few weeks of their studies based on their programming behavior. Our current data indicates that computer science freshman that have a tendency to "crunch" their exercises and start working close to the deadline are at a higher risk of failing their introductory mathematics course than the students that work during a longer time interval.

There is a need for intervention at an early part of the at-risk students' studies, which would direct the students towards more successful learning styles. However, we do not know if there is a direct causality between the working habits, and cannot tell if our subjects would really perform better by e.g. simply starting to work on their assignments earlier. Our current number of samples (52) is relatively small, and we need more data from future students.

Our current plan is to evaluate the students' working process during Fall 2013, and perform intervention(s) to a subset of the population that our current classifier indicates being at risk. We are also seeking more descriptive behavioral indicators from the programming data in addition to our current features. Overall, we are not only interested in a single course or a single semester, but the students' success in their whole studies.

6. REFERENCES

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, 2008.

[2] H. Berghel and D. Roach. An extension of Ukkonen's enhanced dynamic programming ASM algorithm. *ACM Trans. Inf. Syst.*, 14(1):94–106, Jan. 1996.

[3] O. Burn. Checkstyle homepage. *URL http://checkstyle.sourceforge.net/*, 2001–2012.

[4] M. R. Clark. Negotiating the freshman year: Challenges and strategies among first-year college students. *Journal of College Student Development*, 46(3):296–316, 2005.

[5] A. Collins, J. Brown, and A. Holum. Cognitive apprenticeship: Making thinking visible. *American Educator*, 15(3):6–46, 1991.

[6] A. Collins and J. G. Greeno. Situative view of learning. In V. G. Aukrust, editor, *Learning and Cognition*, pages 64–68. Elsevier Science, 2010.

[7] K. A. Ericsson, R. T. Krampe, and C. Tesch-romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, pages 363–406, 1993.

[8] J. Helminen, P. Ihantola, V. Karavirta, and L. Malmi. How do students solve parsons programming problems?: an analysis of interaction traces. In *Proceedings of the 9th annual international conference on International computing education research*, ICER '12, pages 119–126, New York, NY, USA, 2012. ACM.

[9] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the 2nd international workshop on Computing education research*, ICER '06, pages 73–84, New York, NY, USA, 2006. ACM.

[10] P. Myllymäki, T. Silander, H. Tirri, and P. Uronen. B-course: A web-based tool for bayesian and causal data analysis. *International Journal on Artificial Intelligence Tools*, 11(03):369–387, 2002.

[11] 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.

[12] S. B. Robbins, K. Lauver, H. Le, D. Davis, R. Langley, and A. Carlstrom. Do psychosocial and study skill factors predict college outcomes? a meta-analysis. *Psychological bulletin*, 130(2):261–288, 2004.

[13] J. Spacco, D. Hovemeyer, and W. Pugh. An eclipse-based course project snapshot and submission system. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, eclipse '04, pages 52–56, New York, NY, USA, 2004. ACM.

[14] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the 7th international workshop on Computing education research*, ICER '11, pages 85–92, New York, NY, USA, 2011. ACM.

[15] A. Vihavainen, M. Luukkainen, and M. Pärtel. Test my code: An automatic assessment service for the extreme apprenticeship method. In *2nd International Workshop on Evidence-based Technology Enhanced Learning*, pages 109–116. Springer, 2013.

[16] A. Vihavainen, M. Paksula, M. Luukkainen, and J. Kurhila. Extreme apprenticeship method: key practices and upward scalability. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 273–277, New York, NY, USA, 2011. ACM.