Early performance prediction for CS1 course students using a combination of machine learning and an evolutionary algorithm

Filipe D. Pereira (filipe.dwan@ufrr.br)
Department of Computer Science
Federal University of Roraima
Boa Vista, Brazil

Elaine Oliveira, David Fernandes
Computer Institute
Federal University of Amazon
Manaus, Brazil

Alexandra Cristea
Department of Computer Science
Durham University
Durham, United Kingdom

***Abstract*—** Many researchers have started extracting student behaviour by cleaning data collected from web environments and using it as features in machine learning (ML) models. Using log data collected from an online judge, we have compiled a set of successful features correlated with the student grade and applying them on a database representing 486 CS1 students. We used this set of features in ML pipelines which were optimised, featuring a combination of an automated approach with an evolutionary algorithm and hyperparameter-tuning with random search. As a result, we achieved an accuracy of 75.55%, using data from only the first two weeks to predict the student final grades. We show how our pipeline outperforms state-of-the-art work on similar scenarios**.**

Keywords-component; online judge, learner analytics, genetic algorithms, machine learning, data-driven, code metrics

I. INTRODUCTION

Educational researchers point to the need of identifying students with difficulties as early as possible [4,6]. A remarkable study [6] states that, whilst there are many predictive methods available, these works need to be replicated since many works focus on simplistic metric analysis. Additionally, our work focus on Programming Online Judge (POJ) systems, which are more complex and rich in interaction types than regular online learning environments. We have identified features that are being used in state-of-the-art researches targeting the same problem and, in this paper, we compile the best ML attributes found, as well as propose new ones, in order to find the best combination of ML features. We called this feature set the *programming profile* and we apply it to a database of behaviour collected from 486 CS1 students.

The programming profile is then applied, in order to produce ML models, which aim at predicting whether the student would achieve a grade less than 5 or not in a set of seven exams applied throughout the course and in the final examination (with ’5’ being the passing grade). Additionally, in order to perform optimisation in the ML pipelines, we used a combination of an automated approach with an evolutionary algorithm and hyperparameter-tuning with random search.

II. METHODOLOGY

The method proposed in this research uses evidence drawn from students' attempts to solve programming problems presented as problem lists, performed in a POJ called CodeBench, which was implemented by one of the authors. These activities are followed by an exam. We call each pair formed of a programming problem list and an exam a *session*. In total, 7 sessions (S1 to S7) were held throughout the course, lasting a little over 2 weeks each. For the training of machine learning algorithms, we use only the data from the *first session* (first two weeks), since the purpose of this study is to investigate early predictors. We hypothesised that the programming student behaviour from the first session (variables and sequential structure) is enough to achieve a desired accuracy throughout the CS1 course. To do so, we conducted two experiments. In Experiment 1 (E1), we used only data from S1 to predict the exams grades in all subsequent sessions. In addition, we predicted the final grade of the students using only the data from S1, which we called Experiment 2 (E2).

A ‘programming profile’ for each student in each session was constructed by using twenty code metrics (M1 to M20) which represented metrics proposed by state-of-the-art studies, and others proposed by ourselves. This 'student programming profile' was then digitally represented as a feature matrix. The features are listed below, and sources are given; the ones without sources are self-devised: **comment_num** (M1): average number of comments for each submitted code [10]; **blank_num** (M2): average blank lines for each submitted code [10]; **loc** (M3): average number of lines for each submitted code [10]; **lloc** (M4): average number of logical lines for each submitted code [10]; **IDE_time** (M5): total time spent, in minutes, by the students solving problems in the embedded IDE (counted only when the student was typing); **code_ratio** (M6): ratio between M3 and M5 [10]; **log_lines** (M7): average log lines on attempt to solve problems. To illustrate, each time the student presses a button in the embedded IDE of the POJ, this event is stored as a line in a log file (Adapted from [4, 8]); **correctness** (M8): average of test cases passed for each problem [4]; **correctness_with_effort** (M9): represents the same as M8, but in this case we considered correct only student solutions with more than 50 *log_lines*; **access_num** (M10): number of student logins between the beginning and end of a session; **attempts_num** (M11): average of submission attempts for each problem [4]; **coefficient_of_variation** (M12): ratio between the standard deviation and the mean of the number of submissions for each problem; **engagement** (M13): a binary feature (0 when the M11 decreases between two sessions and otherwise 1); **tests** (M14): average number of tests; **prop_paste** (M15): proportion between pasted characters (‘ctrl+V’) and characters typed; **keystroke_latency** (M16): keystroke latency of the students when typing in the embedded IDE (adapted from [8]); **delete_average** (M17): average of deleted characters for each problem; **success_prop** (M18): ratio between the number of solutions

accepted and the *num_attempts* [4]; **syntax_error** (M19): ratio between the submissions with the syntax error and the *num_attempts* (Adapted from [7])); **difficulty_reported** (M20): difficulty level reported [1-3] by the students when they were solving the problems.

III. Results and Discussion

We used an automated ML approach Tree-based Pipeline Optimization Tool (TPOT), which is an off-the-shelf tool to construct and optimise an entire ML pipeline. TPOT uses a genetic algorithm over existing implementations of the well-reputed ML library scikit-learn. It provides automatic preprocessing, feature construction, feature selection, model selection and hyperparameter tuning. Thus, the main idea of TPOT is to find, in a large search space, a good ML pipeline, which best fits the data, by using a guided search based on a version of the famous multi-objective genetic algorithm NSGA-II. The output of the TPOT is a configuration of a ML model that can be used to predict the targets of new data. In spite of the fact that TPOT could provide the above described optimisation, we noticed that for our database, the exported pipelines had some room for improvements. Thus, we propose and performed a second step in the process of ML pipeline optimisation: we applied random search, to find the best models' hyperparameters, after the predictive model was obtained by TPOT.

As the database was unbalanced, we used cost-sensitive classifiers in the random search hyperparameter optimisation step. Models were evaluated using these statistical measures: True Positive Rate (TPR), True Negative Rate (TNR), and accuracy (Acc.) All these aforementioned performer metrics were calculated using 30% of data separated for validation. We performed the optimisation process with the remaining 70% of the data with cross-validation (10 folds).

Table 1 shows the evaluation of the best models found from E1. On the other hand, in E2 we achieved an accuracy of 75.55%. Comparatively, prior predictive study [8] obtained their highest accuracy using data from a larger amount of time (week 1 to 8), being 71.8% and [4] achieved 72.90% of accuracy using data from the first four weeks. Whilst these were applied to different databases, they were performed in similar educational settings to our study

TABLE I. Results from Experiment 1.

Session	Acuracy	TPR	TNR	Model
S2	77.29%	83.63%	62.39%	RF
S3	71.50%	71.15%	71.95%	XGB
S4	69.52%	77.31%	54.47%	ERT
S5	67.77%	73.26%	60.68%	RF
S6	71.51%	78.23%	60.97%	ERT
S7	68.40%	69.06%	67.85%	RF

TPR e TNR values showed that the estimators segregated the students well into those who were struggling and who were not. Since the performance of the models keep reasonable through the sessions, we can state that the code metrics presented in our programming profile have a short-term and long-term relationship with the student performance and can be employed to design early predictors. Note that if the instructor monitors the student performance in the first two weeks, some precautions could be taken, in order to avoid students failing in the exams and, hence, potentially failing in the course. Furthermore, the instructor may create exams more suited to the reality of his class and possibly tailor problem lists to individual student needs.

In addition, we observed that the importance of the features of the programming profile varies across the sessions. Thus, we cannot extract a homogeneous set of the most important features for the entire course. Instead, for each session, we can point out the most important ones. To do this, we used the best classifiers found for each session and applied feature selection embedded methods during the execution of the estimators to analyse five most important features. In overall, M8, M9 and M11 (witch quantifies success on the programming problem lists) appear with more frequency in the set of most important features. Indeed, M11 occurred in all the sessions. However, we have found, additionally, that there were some occurrences of metrics M17 and M19 in the top five. Such metrics superficially quantify students' errors, so they can be useful for identifying students with difficulty. M7 is shown in the top five for sessions S1 and S4, which supports the importance of the number of log lines as a relevant metric to predict the student performance and, hence, the importance of a keystroke level data collection.

Concluding, we believe that these top features might be generalised for other contexts, as they do not depend on nuances of a given compilation message, or a specific programming language or web-based system. Furthermore, these features may not have a high relevance in isolation; however, together they have a reasonable predictive power.

References

[1] K. Castro-Wunsch, A. Ahadi, and A. Petersen. 2017. Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education. ACM, 111–116.

[2] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, et al. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. In Proceedings of the 2015 ITiCSE on Working Group Reports. ACM, 41–63.

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

[4] Leinonen, J., Longi, K., Klami, A., & Vihavainen, A. "Automatic inference of programming performance and experience from typing patterns." Proceedings of the 47th ACM Technical Symposium on Computing Science Education. ACM, 2016.

[5] R. S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. 2016. Evaluation of a tree-based pipeline optimization tool for automating data science. In Proceedings of the 2016 on Genetic and Evolutionary Computation Conference. ACM, 485–492.

[6] Dwan, Filipe, Elaine Oliveira, and David Fernandes. "Predição de Zona de Aprendizagem de Alunos de Introdução à Programação em Ambientes de Correção Automática de Código." Brazilian Symposium on Computers in Education. Vol. 28. No. 1. 2017.

[7] J. Otero, L. Junco, R. Suarez, A. Palacios, I. Couso, and L. Sanchez. 2016. Finding informative code metrics under uncertainty for predicting the pass rate of online courses. 373 (2016), 42–56.