Investigating Automated Student Modeling in a Java MOOC

Michael Yudelson
Carnegie Learning, Inc.
437 Grant St.
Pittsburgh, PA 15219, USA
myudelson@carnegielearning.com

Roya Hosseini
Intelligent Systems Program
University of Pittsburgh
210 South Bouquet Street
Pittsburgh, PA,
roh38@pitt.edu

Arto Vihavainen
Dep. of Computer Science
University of Helsinki
P.O. Box 68 FI-00014
avihavai@cs.helsinki.fi

Peter Brusilovsky
University of Pittsburgh,
135 North Bellefield Ave.,
Pittsburgh, PA 15260, USA
peterb@pitt.edu

ABSTRACT
With the advent of ubiquitous web, programming is no longer a sole prerogative of computer science schools. Scripting languages are taught to wider audiences and programming has become a flag post of any technology related program. As more and more students are exposed to coding, it is no longer a trade of the select few. As a result, students who would not opt for a coding class a decade ago are in a position of having to learn a rather difficult subject. The problem of assisting students in learning programming has been explored in several intelligent tutoring systems. The key component of such systems is a student model that keeps track of student progress. In turn, the foundation of a student model is a domain model – a vocabulary of skills (or concepts) that structures the representation of student knowledge. Building domain models for programming is known as a complicated task. In this paper we explore automated approaches for extracting domain models for learning programming languages and modeling student knowledge in the process of solving programming exercises. We evaluate the validity of this approach using large volume of student code submission data from a MOOC on introductory Java programming.

Keywords
Big Data, MOOC, Student Modeling, Automated Domain Model Construction.

1. INTRODUCTION
Today, information and computer technology is all around us. Programming is not an art accessible to the few and taught at select computer science schools anymore. Scripting and programming languages are taught to wider student audiences and programming courses have become a flag post of any technology related program. As more and more students are taking on programming, it becomes a universal skill, a necessity for every student studying increasingly computerized technology. As a result, the distribution of talent in programming classes shifts from the mathematically gifted to the overall population mean.

There have long existed a number of educational systems that have served the purpose of teaching students an abundance of programming languages and since then have greatly advanced the field of online learning. LISPTUTOR – a system teaching students a language of LISP – was the precursor of the modern intelligent tutoring systems [1] and SQL-tutor – a constraint-based system that instructed students who learned SQL [6], to name just a few.

A classical educational system always has a user model – an integral component responsible for keeping track of student progress. The core of a student model is a vocabulary of skills (concepts) that structure the representation of student knowledge. Conceptualizing a set of skills is a hard task in and of itself. However, programming is an inherently structured domain. The basis of a programming language is the grammar that imposes a structure on any code that compiles.

There were several attempts to exploit the inherent structure of the programming language with respect to student modeling tasks. For example, authors of [7] used a parsed concept map of C and Java to perform cross-adaptation of the content while [11] and [4] used the concept structure of parameterized questions for C and Java to provide within-domain adaptive navigation support.

Until now, to the best of our knowledge, there were no attempts to utilize an auto-parsed structure of the code as a substitute for a conceptualization of the knowledge model. The benefits of such automation with respect to programming are many. First of all, it is inherently transferrable to any programming or scripting language: one just has to have a parser for that language. Second, given the parsed concepts, student modeling can be done on the fly. Third, with recent popularity of massive open online courses, there are volumes of data potentially available to experiment.

The challenge of this approach is that, besides relative easiness of extraction, when programs start to get more complex so grows the volume of concepts parsed and the signal becomes noisier. Additionally, identifying programming constructs essential for passing a particular test is not trivial. And finally, high accuracy of such models can ensure help is given to a student while selecting the next problem, while a model's capacity to aid students during problem solving requires a different form of validation.

In this paper, we report on our investigation of automatically generated user models for the assignment-grading system deployed in a set of introductory programming classes. The data intensity of the code submission stream makes the task of knowledge modeling truly a "big data" problem. Results of our retrospective analysis demonstrate that the models created automatically can successfully support students during problem solving activity.

2. DATA
To explain our idea and a set of explored user modeling approaches, it is important to start with a description of data that we had at our disposal and how the data was processed for our studies. Our data

came from three introductory programming courses organized by the University of Helsinki; two local courses held during Fall 2012 and Fall 2013, and a MOOC held during Spring 2013. Although the MOOC course lasted 12 weeks in total (see [9] for details of an instance offered in 2012), we included only the first six weeks in the analysis to be able to compare the results directly with the introductory programming courses.

The courses emphasized students' personal effort and constant practice. New topics were always accompanied by a set of programming exercises where the first tasks provided clear guidelines that outlined both the required program structure and required functionality, and latter ones were open-ended, giving students more freedom on the application design. During the six weeks, the students worked on over 100 exercises that were further split into a total of some 170 tasks. All exercises were done using an industry-strength IDE with a plugin that provided textual on-demand feedback that had been encoded into the tasks, records students' progress, and allowed students to submit their solutions for grading directly from within the IDE [10].

Table 1. Overall student population statistics.

Course	N. students (M/F)	Age: Avg./ Med./Max.	N. snapshots: all/median
Fall 2012: Introduction to Programming	185(121/64)	18/22/65	204460 / 1131
Fall 2013: Introduction to Programming	207(147/60)	18/22/57	263574 / 1126
Spring 2013: MOOC on Programming[1]	683(492/60)	13/23/75	842356 / 876

Due to this unique problem-solving approach and careful data archiving, each of the three courses produced a unique picture of student behavior. For each exercise and for each student, the system stored a relatively large sequence of code snapshots that were taken on save, compile and run -events, each representing a complete or incomplete attempt to solve the programming problem. Moreover, since each snapshot was tested on a set of tests designed for the corresponding exercise, information on tests that passed and failed was available. This data provides an excellent start for exploring various approaches to student modeling.

Student population statistics are given in Table 1. Each in-class version of the course had about one half of the MOOC's attendance. All three courses were mostly taken by male students (more so in the case of the MOOC). Age distribution was roughly the same. Around 40% of the students were CS major (in-class courses) and around half of the students had previous programming experience (MOOC). In terms of student activity (the number of snapshots) student medians were quite close for the two in-class course, and for the MOOC this number is lower due to the dropout rates.

3. APPROACH

We investigated an automated approach to creating concept models and student modeling for the domain of introductory Java programming. Our approach was based on two principal ideas: (1) modeling knowledge behind every program submission using the inherent structure of the programming language and (2) automatic testing of program correctness using a set of tests. In brief, we considered every program submitted or saved as a solution of a programming exercise as an application of a range of concepts that were present in the submitted code. Once this program passed one or more tests, we considered it as a *successful* application of these concepts in an absolute sense or relative to the earlier submission. In the specific case explored in the paper, concept extraction from the body of submitted program was done by our concept extraction tool "Java Parser" [3] while the correctness of the submitted student code was determined by the system infrastructure introduced above. Thus, the main body of the paper is focused not on the tools, but on using the large body of collected data to explore the plausibility of the approach - the correctness of the student model itself and its usefulness in assisting students while they work on the code.

3.1 Data Preprocessing

For our analysis we preprocessed the raw student submissions. First the code was compiled and run against the suite of tests recording which tests passed. Each snapshot was also analyzed using JavaParser [3]. The extracted concepts were recorded both as an exhaustive list of all concepts in the snapshot and as a difference from the previous snapshot accounting for additions and removals (initial snapshot copied in full). An additional data-thinning procedure removed all snapshots that had an empty list of concept changes to filter out insignificant changes to the code.

3.2 Hypotheses

First, it is possible to model student knowledge acquisition (models can detect learning). Second, only a subset of code constructs is important for solving a particular problem. Third, constructed models are useful beyond modeling student knowledge acquisition and can be used as a basis for creating a recommendation component to help students with the code.

4. MODELS

We chose a set of models that are widely used in the field of student modeling. We first set the modeling lower boundary with the *Null model* (the majority class model). The next model of our choice was the *Rasch model* (1PL IRT) [5]. Although the Rasch model does not capture learning by definition, it is frequently used in psychometrics and would set a baseline for us. The model is given in Eq. (1). Here, *Pr* denotes probability of student i to correctly solve problem j. *Inverse.logit* is the sigmoid function, θ_i is the student proficiency parameter, and β_j is the item complexity parameter. Since the result of compiling and running a problem is a binary mask of passed and failed tests, we treated the problem-test tuples as unique items. We broke each student transaction from the data into n, where n is the number of tests submission is checked against. Passed tests would yield a result of 1, failed a result of 0. Student and concept data were copied across the broken transactions accordingly. We fit Rasch model using mixed effect regression, treating both student and item complexity parameters as random factors.

$$\Pr_{ij} = \Pr(Y_{ij} = 1 \mid \theta, \beta) = inverse.\log it(\theta_i + \beta_i) \quad \text{Eq. (1)}$$

$$\Pr_{ij} = \Pr(Y_{ij} = 1 \mid \theta, \beta, \delta, \gamma) = inverse.\log it(\theta_i + \beta_i + \sum_k (\delta_{kj} + \gamma_{kj} t_{ikj}) \quad \text{Eq. (2)}$$

To actually model student learning we would use a variant Additive Factors Model (AFM) [2]. In addition to the parameters in Rasch model, AFM (Eq. (2)) has skill complexity – δ_{kj} (intercept), and skill learning rate – γ_{kj} (slope). Although standard AFM does not have item complexity, we will have it in our AFM models to account for item variability. For each student submission we will count the number of prior attempts to use a particular coding construct – t_{ikj}.

[1] We only include data on the students that answered the survey

In AFM it is customary to fit concept intercepts and slopes across all items. We will treat concepts as within-item effects.

When the standard AFM model is used, for each item or problem step a set of relevant concepts is known. Often, a table relating concepts to items is called a Q-matrix. We do not have information on what programming constructs are relevant for the successful passing of the tests. We used three different rules to select concepts. Rule A selects all concepts that were parsed from the student code. Rule B uses the concepts that were different from the previous code snapshot (added or removed alike). Rule C used concept differences just like Rule B, but treating addition and removal as different instances of one concept (appending a suffix to the concept identifier in case of concept removal).

First, the AFM model is to use all parsed concepts or concepts difference lists. It is, however, safe to assume that not all concepts are relevant for solving a problem and different subsets of concepts could be relevant for each particular test the problem is verified against. To set aside the concepts that have a significant influence on the successful passing of the problem's test, we used a PC algorithm for systematic conditional independence search implemented in the Tetrad – a data-mining tool developed at Carnegie Mellon University [8]. For each problem in our three datasets we composed a data-mining problem for the PC algorithm to find a bipartite graph where arcs go from concepts to tests denoting causal links (but not between tests or concepts). We admittedly violate i.i.d. assumptions and, although we are mining for these graphs across multiple students, we are using multiple data points from the same student. However, we are not going to draw causal conclusions on the included arcs and are only using the results of the algorithm to filter out concepts. For the tests of independence we used a p-value of 0.05. Our experimentation with different p-values did not result in tangible changes of the output.

One important phenomenon we noticed in the data is that students have different submission speeds. One student might submit one code snapshot per 10-20 minutes of work, while the other would submit every change to the code with several submissions per minute. As a result, the number of attempts per code construct per unit of time would vastly differ across students and the estimations of the concept learning rates would be extremely noisy. To compensate for these differences, we applied natural logarithm function to the student opportunity counts (t_{ikj}).

Four different versions of AFM models were constructed by turning on and off of the two features: whether or not to filter concepts, and whether or not to log counts of concept opportunities, together with one Rasch and one null model, give us 14 models in total. In order to go beyond model-fitting accuracy and to check our third hypothesis and to make sure that our models can potentially serve as a basis for a component to recommend changes to the code, we ran a specialized validation procedure. In this procedure we distinguished four changes between passing and failing of a particular problem's test in successive code snapshots. Namely, from fail to fail (NN – not passing to not passing), from pass to pass (YY – passing to passing), from pass to fail (YN), and from fail to pass (NY). In each of the four cases we looked at which concepts students added and which concepts they removed between the snapshots. For additions and removals, we computed support scores – sums of concept slopes in the model giving us model's judgment in favor of all addition and all removals. These two sums were either positive (P), negative (N), or zero (0), giving us 9 different combinations. Thus each successive code snapshot was assigned a 4-letter code. For example, NYP0 would denote that a student went from failing to passing a test and the model has a positive support score for concept addition and a neutral 0-score for concepts removal. Based on these codes, for each of our models we computed four conditional probabilities.

Probability A: the non-negative support of the changes to the concepts in cases of two successful passes of the test. *Rationale*: Since in two consecutive attempts student's code passed the test, model negative support of code changes is undesirable.
Probability B: negative code changes support in the case of pass changes to fail. *Rationale*: Since students apparently made the code worse, we want the model to vote against it.
Probability C: non-positive support for the code changes in the case of two successive fails. *Rationale*: The code did not improve and the model should not support any changes made.
Probability D: positive support for the changes made between a failure and a success. *Rationale*: When a student is on the right path, the model should be supportive of that.

We performed validation with respect to the three rules of the concept selection (A – all concepts, B – changed concepts, and C – changed accounting for removals and additions) as well as filtering of the concepts (only considering slopes for concepts that were selected by the PC algorithm).

5. RESULTS

Table 2 is a summary of the model fitting and validation results for the 14 models we discussed. The dataset was balanced: with the majority class model performing only a little better than chance. The Rasch model that assumes no learning is a tangible improvement with 71% accuracy. AFM models perform better with respect to accuracy. Models considering all concepts in the snapshot (A) are doing better, and models considering changes on concepts distinguishing additions and removals (C) being second. Filtering concept lists using PC algorithm improves model accuracies, while taking logs of opportunity counts does a little bit of the opposite. Out of the top three models with respect to accuracy, two are picking all concepts available and two are using PC algorithm for concept filtering.

An important consideration is the size of the input data. More data complicates training the models as well as online-prediction of potential modifications to the code. Models using concept selection, rule A, are more data hungry. Applying the PC algorithm to only leave influential concepts reduces the data requirement. Logging opportunity counts increases the data requirement mostly due to the text representation of our data. Model accuracy and data requirements together paint a mixed picture.

Reviewing the validation columns of Table 2, We see in the average validation probabilities columns, probabilities A and C described model quality with respect to situations when a student neither improves the code nor makes it worse (in terms of passing the tests). In these cases, we would like our models to not discourage changes when students' code did not improve beyond an already passing rating (probability A) and we would like models to not support changes when students do not improve their code and the tests still fail to pass (probability C). Arguably, A and C are secondary to probabilities B and D, where we want them to positively reinforce changes from pass to fail (probability D) and negatively reinforce changes from fail to pass (probability B). In an attempt to make model selection more rigorous we take an average of all probabilities (A through D), and an average of the columns of the primary interest (B and D).

Looking at validation results alone, models with logged opportunity counts using concept selection rules A and B are in the lead, model AFM B +PC+Log has a slight edge (third and first with respect to

the two averages of the conditional probabilities). This model also has a top average rank overall. It is only 5% over the accuracy of the Rasch model, but it is quite low on data requirements and performs well in the validation.

Table 2. Summary of model fitting and validation statistics. Models ranked among top three in each category are bold faced.

Model	Accuracy & rank*		File size, Mb & rank		Avg. validation prob. & rank				Overall rank
					A-D		B, D		
Null	.56	-	-	-	-	-	-	-	-
Rasch	.71	-	49	-	-	-	-	-	-
AFM A	.81	4	1312	11	.61	5	.39	7	6.75
AFM B	.73	11	446	8	.62	4	.39	8	7.75
AFM C	.78	6	445	7	.59	9	.23	12	8.50
AFM A+PC	**.84**	**1**	1528	12	.57	11	.34	10	8.50
AFM B+PC	.77	7	526	9	.60	7	.44	4	6.75
AFM C+PC	**.83**	**2**	530	10	.56	12	.30	11	8.75
AFM A+Ln	.75	10	242	5	**.62**	**2**	**.45**	**3**	5.00
AFM B+Ln	.71	12	**123**	**1**	**.63**	**1**	.43	5	**4.75**
AFM C+Ln	.77	8	**139**	**2**	.60	6	.35	9	6.25
AFM A+PC+Ln	**.82**	**3**	284	6	.59	8	**.47**	**2**	**4.75**
AFM B+PC+Ln	.75	9	**141**	**3**	**.62**	**3**	**.49**	**1**	**4.00**
AFM C+PC+Ln	.78	5	161	4	.58	10	.40	6	6.25

* Null and Rasch models are not ranked and given as a reference

It is particularly interesting whether accuracy, data requirements, and validation conditional probabilities correlate. Naturally, accuracy grows with the data necessary to fit the model and explains 35% of its variance. The average of four conditional probabilities is negatively related to the accuracy and explains 71% of its variance. However, despite the fact that the average of negative support for going from pass to fail and positive support for going from fail to pass, respectively correlates with model accuracy negatively, the percent of variance explained is low.

6. DISCUSSION

In this work we investigated the value of using student models for programming domain without a priori conceptualization of the problem domain. We hypothesized that, thanks to the inherent structure of the programming language, it could be possible to skip tedious development of a concept vocabulary overall.

Serving as a basis for navigation support, the models of student knowledge that we built could be used for recommending the next problem to solve. However, an arguably more interesting feature is to reuse the models for within-problem support. As we have shown in our validation, even in the absence of a formal conceptual domain structure, just relying on the code parser and concept selection and filtering algorithms, our models can be useful.

Based on the model accuracy, data requirement, and validation, we were able to select a model that has a promise to be accurate both modeling student knowledge and suggesting students what concepts to address in their code. The choice, however, has a number of tradeoffs. Depending on model accuracy, computational complexity of model fitting (size of the data required), and validation characteristics (potential accuracy of recommendation) one might opt to select a different model. The trade-off between modeling accuracy and validation accuracy is particularly sharp, because these two metrics are negatively correlated.

In our models, we only accounted for the presence of programming language constructs in the code, completely ignoring the number of times they were used. One particular roadblock that exists on the path toward incorporating problem-concept counts is that it would be not possible to use the PC algorithm anymore. The PC algorithm is intended for binary data only (passing of the test and presence of the concept). There are few empirically verified structural search algorithms that use block-of-conditional-independence-tests that handle hybrid data (binary and count data together).

In addition, when looking at the code, we only looked at the list of concepts and not at the structure of the code. We were able to detect certain strategies that students employed while solving the problems. In our future work, we plan to exploit these findings to improve our model's prediction and validation scores.

7. REFERENCES

[1] Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. Journal of the Learning Sciences, 4(2), 167-207.

[2] Cen, H., Koedinger, K.R., Junker, B. (2008) Comparing Two IRT Models for Conjunctive Skills. In 9th International Conference On Intelligent Tutoring Systems. (pp. 796–798). Springer, Heidelberg

[3] Hosseini, R., & Brusilovsky, P. (2013). JavaParser: A Fine-Grain Concept Indexing Tool for Java Problems. In The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (pp. 60-63).

[4] Hsiao, I.-H., Sosnovsky, S., and Brusilovsky, P. (2010) Guiding students to the right questions: adaptive navigation support in an E-Learning system for Java programming. Journal of Computer Assisted Learning, 26 (4), 270-283.

[5] van der Linden, W.J., Hambleton, R.K. (eds.): Handbook of Modern Item Response Theory. (1997) Springer, New York

[6] Mitrovic, A. (2003). An Intelligent SQL Tutor on the Web. International Journal of Artificial Intelligence in Education, 13(2-4), 173-197.

[7] Sosnovsky, S. A., Dolog, P., Henze, N., Brusilovsky, P., & Nejdl, W. (2007). Translation of Overlay Models of Student Knowledge for Relative Domains Based on Domain Ontology Mapping. In 13th International Conference on Artificial Intelligence in Education (pp. 289-296)

[8] Spirtes, P., Glymour, C., and Scheines, R. (2000) Causation, Prediction, and Search, 2nd Ed. MIT Press, Cambridge MA.

[9] Vihavainen, A., Luukkainen, M., & Kurhila, J. (2012). Multi-faceted support for MOOC in programming. In Proceedings of the 13th annual conference on Information technology education (pp. 171-176).

[10] Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013). Scaffolding students' learning using test my code. In Proceedings of the 18th ACM conference on Innovation and technology in computer science education (pp. 117-122)

[11] Yudelson, M. & Brusilovsky, P. (2005). NavEx: Providing Navigation Support for Adaptive Browsing of Annotated Code Examples. In 12th International Conference on Artificial Intelligence in Education (pp. 710-717).