Measuring Transfer of Data-Driven Code Features Across Tasks in Alice

Nicholas Diana
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
ndiana@cmu.edu

Michael Eagle
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
meagle@cs.cmu.edu

John Stamper
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
john@stamper.org

Shuchi Grover
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
shuchig@cs.stanford.edu

Marie Bienkowski
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
marie.bienkowski@sri.com

Satabdi Basu
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
satabdi.basu@sri.com

ABSTRACT

Previous research has demonstrated that low-level log data from introductory programming environments like Alice can be used to predict student outcomes and generate data-driven rubric criteria. The success of these methods suggests that the features used in these predictive models may be crude representations of more fundamental introductory programming skills. In this experiment we first replicated previously reported methods using a novel dataset. Then, we tested whether or not the features predictive of success on one task are also predictive of other tasks. We found that we could, with reasonable accuracy, use a model trained on data from one task to predict a separate task. Our results suggest that the features of successful predictive models contain some amount of information that transfers across tasks.

Keywords

Alice, Introductory Programming, Transfer

1. INTRODUCTION

Log data derived from students working through programming exercises has the potential to aid educators working in computer science education. Recent work has shown that student learning outcomes can be predicted from low-level log data long before students actually complete and submit their final program code [2]. However, it is unclear if the data-driven features of these models are task specific or generalizable across activities. While there has been some work focused on gleaning pedagogical value from the features of predictive models [1], there remains a disconnect between the nature of human versus data-driven grading. In an ideal world, a project grade in an introductory computer science course is not only an assessment of the student's performance on the project, but, more fundamentally, is an assessment of their progress toward mastering the key skills of introductory programming. For human graders, course projects simply provide opportunities for students to practice and demonstrate their skills. Data-driven graders, on the other hand, are often limited in scope to the specifics of a particular project (i.e., are by definition "driven by the data"). Whether the features predictive of success in one task are representative of more fundamental skills that transfer across tasks remains to be seen. We hypothesize that a subset of the features in a predictive model trained on one task will be relevant for predicting other tasks, and that we will be able to make predictions across tasks with an accuracy comparable to within-task predictions.

1.1 Related Work

Log data from Alice can be used to support learning in a number of ways. First, previous research has demonstrated that a machine learning model trained on Alice log data could be used to predict grades with reasonable accuracy. Diana et al. [2] first broke down Alice log data into a vocabulary of short terms, and then counted the number of times each term appeared in a given student program. These counts were used as features in a linear regression trained to predict student grades. In addition to showing that grades could be predicted using this method, the researchers also integrated this predictive model into a prototype of an instructor dashboard. Using the dashboard, instructors could monitor, in real-time, each student's progress and their predicted grade (given their work up to that point). Instructors could use these predicted grades to inform which students they might assist, or which students might make good peer tutors. The researchers found that the model's performance stabilized after only 9 minutes into a 30-minute class (though this may be more reflective of the task students were asked to do than the researchers' method).

In addition to predicting student grades, log data has also be used to model how students are progressing through a task. For example, Hint Factory [9], uses an interaction network created from previous student data to train a Markov Decision Process (MDP) of student problem-solving approaches

to serve as a domain model for automatic hint generation. In the computer science domain, Rivers and Koedinger [8] demonstrated that programming data can be used to automatically generate feedback. Building off of this work, Diana et al. [3] used Alice log data to demonstrate that low-performing students could be paired with high-performing peer tutors based on the similarity of their code, and how those pairings could more than double the number of students who were able to receive help in a typical class period.

In their most recent work, Diana et al. [1] found that both the accuracy and interpretability of the features of their model could be improved by using a more nuanced method of feature-generation. Rather than breaking a student's program into arbitrary terms, the researchers used the presence or absence of small snippets of code (referred to as code-chunks) as features in the predictive model. They found that this new method not only increased accuracy (reducing root mean square error (RMSE) by 38%), but also seemed to produce features that were more closely related to the human-generated rubric criteria.

Though these results are promising, their generalizability is limited by the fact that each of these advancements were made using the same dataset and the same task. Demonstrating that meaningful insights can be gleaned from Alice log data requires that these methods be replicated with another dataset, and preferably with one or more different tasks. In the current study, we attempted to replicate the above results, applying the above methods to two different samples of students and three different tasks. We hypothesized that student grades in the new dataset could be predicted using both the Natural Language Processing and Code-Chunk methods (with accuracies comparable to the original results).

Replicating these methods on a series of novel tasks also provides us with an opportunity to test another critical hypothesis. Previous research has suggested that a model's ability to identify code-chunks that are predictive of learning outcomes may indicate that highly predictive code-chunks are crude manifestations of key component skills. If we assume that some of these key component skills transfer across tasks, then it may be possible to predict a student's performance on a new task based on the student's use of those component skills in a previous task. We hypothesize that models trained on one task could be used to predict another task (perhaps with a slight decrease in accuracy). If shown to be true, these results would bolster the generalizability and value of data-driven methods in the context of a larger curriculum.

2. METHODS

2.1 The Alice Programming Environment

Alice is a block-based introductory programming environment (similar to Scratch or Blockly) that allows students to create 3D animations and games while learning programming fundamentals such as conditionals, iteration, and parallel processing [6]. Using a drag-and-drop interface, students can "snap" blocks of code together to build chains of actions and events. The drag-and-drop scaffolding limits improper syntax and the frustration that accompanies it. Alice is widely used in classrooms throughout the United States.

2.2 Data Collection

The log data used in this experiment were generated by 94 students. From those 94 students, 359,573 transactions were collected. A transaction is logged each time a student makes a change to their program, so each transaction represents the state of a student's program code up to that point, or what we call a *code-state*. In Alice, students are the creators of a digital world, where characters and objects move, speak, and interact with one another. A code-state is essentially an object that contains the properties and values of those characters and their behaviors. For example, let's assume a student decides to make a frog character turn to the left. The corresponding code-state will include: 1) an object for the frog, 2) a property for the action the frog takes (e.g., "turn to the left"), and a value for that action indicating how far to the left the frog should turn (e.g., "90 degrees"). This representation captures the product of student actions (rather than the actions themselves). However, because we capture a code-state after each student action, we can infer these actions by examining the changes in code-states over time.

Capturing the student's progression from the beginning of the task to their final product has proven to be valuable for some methods (e.g., peer-tutor matching), however for the purposes of this experiment we will be focusing on only the student's final code-states (i.e., the last transactions logged for each student). These final code-states represent the program in its finished state (i.e., the program that students turn in for a grade).

Over the course of a semester, students were asked to complete three tasks using the Alice software. These tasks were designed to test a student's knowledge of introductory programming concepts like conditionals and loops [5]. Because students began and finished working on the tasks at different points throughout the semester, the number of students per task varies slightly (*Cat and Mouse*: n=56; *Midas Touch*: n=51; *Frog and Pond*: n=47). Additionally, while the criteria of these tasks were well-specified, the open-ended nature of the Alice environment resulted in students meeting the specified criteria in a number of different ways. As you might expect, some solutions completely satisfy the criteria and would receive full credit, while others may demonstrate the student's conceptual understanding, but lack technical correctness and would receive partial credit.

It is important to note that the Alice log data used in our models was collected using an improved version of the basic logging mechanism already present in Alice. The data generated by Alice's built-in logging system was intended to be used as a debugging tool for developers (not to be analyzed by learning scientists), and the process of converting the linear log data into a representation of the student's code required a great deal of reverse engineering. Even then, the built-in logging system failed to log steps that are crucial for reconstruction, such as undo and redo actions. To avoid these pitfalls, we decided to build a modified version of Alice that includes a more robust logging system. The output of this logging system mirrors the code-states that had previously been reverse-engineered. The main difference is a higher degree of confidence in the fidelity of the code-state.

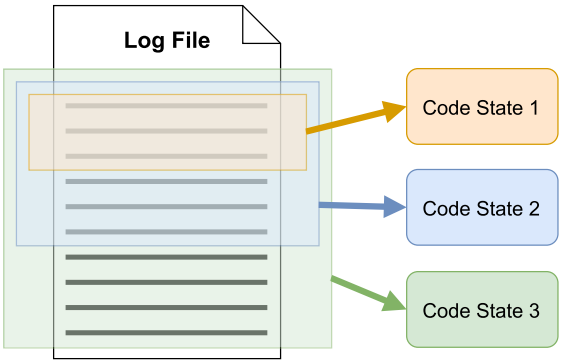

Figure 1: The conversion of low-level log data to code-states. Note how code-states are cumulative, with each successive code-state being a modification of the previous code-state.

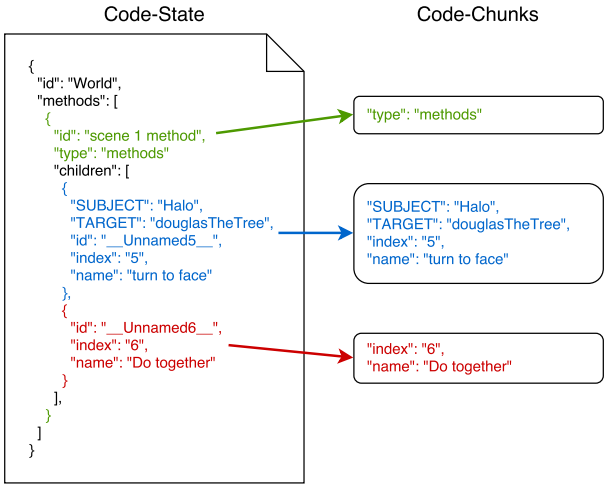

Figure 2: An example of how code-states are decomposed into code-chunks. Note that overly specific parameters like "id" are dropped. Additionally, parameters that represent lists of child nodes are also dropped. Instead, a new code-chunk is created for each of the child nodes.

Another significant improvement was how the data was collected from the students' individual machines. Alice has no built in networking capabilities, so in the past, data was collected manually from each machine by the researchers. Because we were interested in capturing code-states each time the user makes a change, collecting data manually would be impossible. Instead we integrated networking into our improved logging system. With this in place, each action a student takes inside Alice is silently logged and uploaded to a database hosted on the educational data repository and analysis service DataShop [10].

2.3 Code-state Bag-of-Words Method

Previous work on predicting student outcomes using Alice log data has utilized two methods to make predictions. The first method we will call the *Code-state Bag of Words* method. In this method, the log data are first converted into code-states (for a detailed description of this process, see Diana et al. [2]). Then, the code-states are broken down into a vocabulary of terms. The frequency of those terms are used as features in a ridge regression model that predicts student grades. In other words, this method suggests that some terms are more frequently used by high-performing students, and that even if we look at this simplified representation of a code-state's content (largely devoid of context and structure), we can still make reasonably accurate predictions about student performance.

Given the large number of features that would likely have little predictive power, the researchers opted to use a ridge regression. Ridge regression shrinks the coefficients of less important factors, which, at least in this case, improves prediction accuracy.

2.4 Code-chunk Frequency Method

The second method, which we will call the *Code-chunk Frequency* method, attempts to recapture some of the context stripped away in the *Code-state Bag of Words* method. Instead of using the frequencies of arbitrary terms, it uses the frequencies of small snippets of code, which we call *code-chunks*. To covert a code-state into a set of code-chunks, each level of the code-state object is translated into a single-level code-chunk. If a parameter represents a list of child nodes, that parameter is ignored. Instead, a new code-chunk is created for each of the child nodes. Additionally, overly specific parameters such as *"id"* are ignored. Including these parameters would give the code-chunk a degree of specificity that makes finding functionally identical code-chunks more difficult. See Figure 2 for an example conversion from code-state to code-chunks.

Once we have converted each code-state into a set of code-chunks, we can use the frequencies of each code-chunk (within the code-state) as features in our regression. We used L1 regularization to select features via a lasso regression. This process reduces features by encouraging weights to shrink to zero. Features with a weight of zero are effectively dropped from the model, reducing the number of features [4]. Note that the ridge regression used in the *Code-state Bag of Words* method and the lasso regression used in this method generally serve the same purpose, and the choice to use one over another in previous work was primarily determined by which regularization yielded the best performance. As this is primarily a replication study, for the sake of consistency we will also use a ridge regression for the first method and a lasso regression for the second.

To measure model performance, we first used a min-max scaler to normalize student grades on each of the tasks such that the data is scaled to a fixed range from 0 to 1. Then, we compared the predicted grades to the known actual grades to produce the Root Mean Square Error (RMSE). Each reported value is the average of a 10 fold Shuffle-Split Cross-Validation. For each fold, we randomly chose 10% of the students to use in the testing set. The remaining 90% of students were assigned to the training set. The python package scikit-learn was used for both cross-validation and regression [7]. In all methods, highly correlated features ($r > 0.9$) are

Table 1: RMSE values for each of the 9 possible training-task/testing-task pairs. Note: These values are relative to normalized task grades that range from 0 to 1.

		Testing-Task		
		Frog & Pond	Cat & Mouse	Midas Touch
Training Task	Frog & Pond	0.135	0.248	0.169
	Cat & Mouse	0.158	0.265	0.174
	Midas Touch	0.198	0.239	0.129

Table 2: A selection of code-chunks that were frequently important for predicting learning outcomes across different tasks. The similarity of many of these code-chunks to transferable introductory programming concepts is clear. Also listed is the percentage of all cross-task prediction folds (n=60) that selected the corresponding code-chunk as part of the regularization process. These relatively high percentages indicate that these code-chunks were frequently selected as important features for cross-task predictions.

Excerpt of Code-chunk	% Cross-Task Models that Selected this Code-Chunk
`{"Type": "DoInOrder"}`	65%
`{"Type": "LoopNInOrder", "count": "null", "index": "index", "end": "?", "start": "0", "increment": "1"}`	63%
`{"Type": "PointOfViewAnimation", "subject": "camera"}`	50%
`{"Type": "PropertyAnimation", "value": "0"}`	45%
`{"Type": "DoTogether"}`	33%
`{"Type": "IfElseInOrder", "condition": "edu.cmu.cs.stage3.alice.core.question..."}`	33%

removed from the feature set.

3. RESULTS & DISCUSSION

Before exploring whether or not code-chunks transfer across-tasks, we first replicated two prior analyses, the *Code-state Bag of Words* method and the *Code-chunk Frequency* method, using our dataset. Replicating these methods not only lays the groundwork for subsequent analysis, but also tests the degree to which these methods generalize to novel tasks.

3.1 Code-state Bag of Words Method

To replicate the *Code-state Bag of Words* method, we first generated a vocabulary of terms from the students' code-states, and then used the frequencies of those terms within a student's final code state as features in a ridge regression model. We found that, on average, this model's performance was comparable (RMSE=0.351) to previously reported performance (RMSE=0.384), with some variation across tasks (Frog and Pond RMSE=0.282; Cat and Mouse RMSE=0.324; Midas Touch RMSE=0.448).

3.2 Code-Chunk Method

To replicate the *Code-chunk Frequency* method, we first converted each code-state into a set of code-chunks, and then used the frequencies of those code-chunks as features in a lasso regression model (α=.01). We found that, on average, this model's performance was slightly better (RMSE=0.176) than the previously reported performance (RMSE=0.235), again with some variation across tasks (Frog and Pond RMSE= 0.135; Cat and Mouse RMSE=0.265; Midas Touch RMSE=0.129).

3.3 Cross-Task Predictions

The above results demonstrate that previously reported methods for analyzing Alice log data can be successfully applied to a variety of different tasks. Diana et al. [1] have argued that the ability of our models to identify code-chunks that are predictive of learning outcomes may indicate that highly predictive code-chunks are crude manifestations of key component skills. If we assume that some of these key component skills transfer across tasks, then we may be able to predict a student's performance on a new task based on the student's use of those component skills in a previous task. Because we have data from multiple tasks, we can test the degree to which the code-chunks that are predictive for one task are also predictive for others.

To test this, we trained our lasso regression using data from one task, and then tested how well the model could predict grades for a different task. We did this for each of the 9 possible combinations of training-task/testing-task pairs. Table 1 lists the RMSE values for each pair.

Note: To select subjects for the training and testing sets, we first selected only those students who had grades for both tasks (i.e., students who had completed both tasks. This list of users is split into training and testing sets. This eliminates

the possibility of choosing the same user for both the training and testing sets.

Discrepancies between scores may have to do with the nature or difficulty of the specific tasks. For instance, the model seems to have a greater difficulty predicting grades for the *Cat & Mouse*, even when trained on data from the same task. Future work will explore how task features interact with model performance.

The RMSE values reported in Table 1 demonstrate that a model trained on one task can be used to predict performance on a separate task with reasonable accuracy. At the very least, these results suggest 1) that there are some patterns of programming that are associated with learning outcomes, and 2) that these patterns are relevant to learning outcomes across different tasks. This runs contrary to the hypothesis that data-driven prediction methods are limited in scope to a specific course project, and instead implies that data-driven methods can be used to access crude representations of fundamental introductory programming skills.

An examination of the specific code-chunks points to a similar conclusion. Table 2 lists some of the code-chunks that were selected most frequently in cross-task predictions. In other words, within the 60 cross-task prediction folds (i.e., 6 cross-task predictions, 10 folds per task combination), these code-chunks were frequently selected during the regularization process. In these code-chunks, we see evidence of familiar introductory programming concepts like conditionals, (e.g., "IfElseInOrder"), iteration ("LoopNInOrder"), and even parallel processing (e.g., "DoTogether"). These concepts align closely to the programming concepts these tasks were originally designed to teach. Table 2 also lists the percentage of cross-task prediction folds that each of the code-chunks appeared in. High percentages indicate that these code-chunks were frequently selected as important features for cross-task predictions.

4. CONCLUSION

In this study, we first replicated two methods used to predict student outcomes using log data from the Alice introductory programming environment. We found that our RMSE values were comparable, or slightly better, than previously reported values. Much of the recent work on analyzing log data from Alice has made use of the same dataset [11], so demonstrating that these methods generalize to a novel dataset strengthens claims made in previous research.

In addition to replicating previous methods using a novel dataset, we expanded on prior work by testing a critical hypothesis: The features of a predictive model may be crude representations of more fundamental introductory programming skills. We demonstrated that features predictive of one task are also predictive of other tasks, suggesting that these features contain some information that transfers across tasks.

Taken together, these findings signify a small step toward uncovering evidence of general programming knowledge components in low-level log data, and underscore the value of analyzing introductory programming log data in computer science education.

5. REFERENCES

[1] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and S. Basu. Data-driven generation of rubric parameters from an educational programming environment. In E. André, R. Baker, X. Hu, M. M. T. Rodrigo, and B. du Boulay, editors, *Artificial Intelligence in Education*, pages 490–493, Cham, 2017. Springer International Publishing.

[2] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and S. Basu. An instructor dashboard for real-time analytics in interactive programming assignments. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, pages 272–279. ACM, 2017.

[3] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and B. Satabdi. Automatic peer tutor matching: Data-driven methods to enable new opportunities for help. In *Proceedings of the 10th International Conference on Educational Data Mining*, pages 372–373. ACM, 2017.

[4] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[5] S. Grover, S. Basu, M. Bienkowski, M. Eagle, N. Diana, and J. Stamper. A framework for using hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming environments. *ACM Transactions on Computing Education (TOCE)*, 17(3):14, 2017.

[6] R. Pausch, T. Burnette, A. C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. A brief architectural overview of alice, a rapid prototyping system for virtual reality, 1995.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] K. Rivers and K. R. Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, volume 50, 2013.

[9] J. Stamper, M. Eagle, T. Barnes, and M. Croy. Experimental evaluation of automatic hint generation for a logic tutor. *International Journal of Artificial Intelligence in Education (IJAIED)*, 22(1):3–18, 2013.

[10] J. Stamper, K. Koedinger, R. S. d Baker, A. Skogsholm, B. Leber, J. Rankin, and S. Demi. Pslc datashop: A data analysis service for the learning science community. In *International Conference on Intelligent Tutoring Systems*, pages 455–455. Springer, 2010.

[11] L. Werner, J. Denner, and S. Campe. The Fairy Performance Assessment : Measuring Computational Thinking in Middle School. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education - SIGCSE '12*, pages 215–220, 2012.