Predicting programming success using deep learning

Neeraj Mathur
Stanford University
`mathurn@stanford.edu`

Raejoon Jung
Stanford University
`raejoon@stanford.edu`

Ben Stenhaug
Stanford University
`stenhaug@stanford.edu`

Abstract

Deciding when to help a learner working on a task has been a decision that educators have been faced with for centuries. Now that some educational opportunities are moving online, it is the task of a computerized system. We apply deep learning to code.org submissions to predict if a student will continue to struggle with a task. We find that logistic regression outperforms deep learning methods on both a very basic and basic task. We hypothesize that the lack of lift from deep learning techniques is due to the basic structure of these block-based coding exercises, and that the value of deep learning will be significantly higher for more complex coding tasks.

1 Introduction

There is tremendous interest in building coding proficiency in students. One example is Obama's 2016 Computer Science for All Program and the development of Code.org where students can begin to learn computational thinking with block-based programming exercise. One of the challenges in this space is that learning happens best when students get responsive feedback on their work, including hints when they begin to go down an unproductive path. This is only possible at scale using automated algorithms that do not require human intervention.

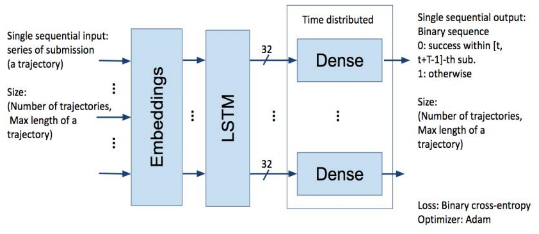
2 Related work

When to provide hints and what hint to provide has been a topic of educational interest for decades. This history was summarized and pushed into the age of deep learning in 2015 by Piech et al. [1]. They focus on which hint to provide in particular and describe their technique as a desirable path algorithm in which a student's submission history is analyzed and a hint's goal is to nudge a student onto the closest of one of a few paths to success.

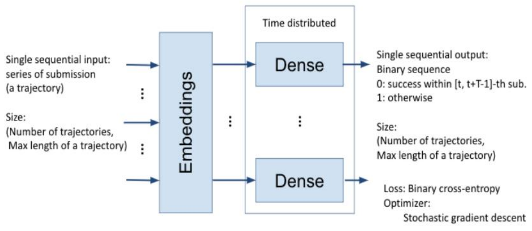
In 2017, Wang et al. pushed this work forward by contemplating optimal embeddings for block-based programming exercises. They distinguish two possible tasks: prediction of whether the student will be successful on the next programming exercise and based on a sequence of submissions predict whether a student will be successful on the current programming task. The latter is in someways preferable because it offers the ability to provide immediate help to students. They use an LSTM RNN architecture where each submission is represented most compactly as an abstract syntax tree (AST), which results in 96% hold-out accuracy (the majority class is 54%).

3 Dataset and Features

Code.org released the attempts from thousands of students for two exercises from the 2013 hour of code. [4] Dataset consists of every submissions from every students. The dataset has a set of



(a) LSTM model

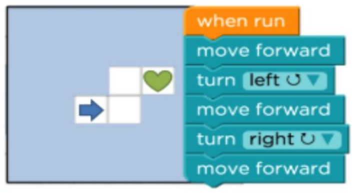


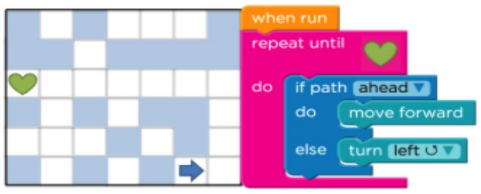
(b) Logistic regression model

Figure 1: Success prediction model

unique submissions and a set of trajectories. Each unique submission is represented by an abstract syntax tree (AST). A trajectory is an array of AST ids which represent the series of attempts from a student. A trajectory id is associated with each unique trajectories and student ids are mapped to these trajectory ids. Note that these mappings are many-to-one; two students can have the same trajectory.

The data comes from two simple block-coding exercises where the student uses blocks of code to navigate a blue arrow to a green heart. Note that the second exercises offers additional blocks including conditional logic and looping.





4 Methods

Given the submission data from `code.org`, our goal is to perform the following task: Given a partial knowledge of a trajectory from the beginning to timestep t, predict whether the student will succeed in the assignment within a time window of τ steps (i.e, in the range of timestep $[t, t+\tau-1]$).

In order to solve this problem, we use a many-to-many LSTM RNN model with the trajectories as inputs. Trajectories are represented as vectors containing the class identifiers of the ASTs. Therefore, the model requires *programming embeddings* to represent ASTs and to be fed in to LSTM layer. We experimented with two options; (1) create our custom embeddings using a separate model, (2) let the optimizer of the main prediction model optimize the weights of the embeddings.

We first describe our main success/failure prediction model and then present how we generated programming embeddings using a separate model.

Output labels are generated by passing a sliding window filter on the binary success/failure vector given a trajectory.

4.1 Main prediction model

The success prediction model using LSTM is presented in Figure 1a. A student's effort to solve an assignment is an iterative process where a student identifies mistakes from its previous code submissions and improve upon them. Because of the iterative nature, utilizing a recurrent layer component seems to be a suitable choice. Each input data is a sequence of AST ids of variable length. Since we want to vectorize the multiple dataset and use a model of fixed size, we set the width of the model to be the maximum length of a trajectory (which we denote as T_traj) and extend the input data to match the length by padding the last AST id in each trajectory.

The AST ids are passed to the embeddings layer to get a vector representation of each code submissions. The embeddings layer can be optimized jointly or can use an existing embeddings mapping generated from another model such as in Section 4.2 The embeddings are passed to a single LSTM

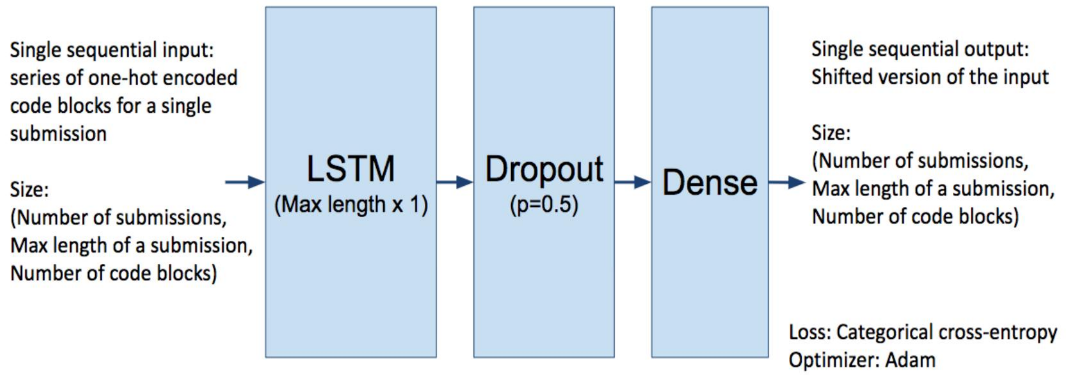

Single sequential input: series of one-hot encoded code blocks for a single submission
Size: (Number of submissions, Max length of a submission, Number of code blocks)
LSTM (Max length x 1)
Dropout (p=0.5)
Dense
Single sequential output: Shifted version of the input
Size: (Number of submissions, Max length of a submission, Number of code blocks)
Loss: Categorical cross-entropy
Optimizer: Adam

Figure 2: Next block prediction model to generate programming embeddings

model which outputs 32 hidden states for each timestep. These $32 \times T\_traj$ hidden states go through a time-distributed dense layer which are T separate dense layers, sharing identical parameters and take 32 inputs. The outputs of the dense layers pass sigmoid activation functions to generate values ranging from 0 to 1 which can be interpreted as probability of failure.[1]

To gauge the performance of the LSTM model, a baseline model using logistic regression in Figure 1b is implemented. The model is identical to the LSTM model with the exception of the missing LSTM layer. Note that although a single data point is a sequence of ASTs, each ASTs are directly fed into identical but separate dense layers.

Since this is a binary classification problem, we use a binary cross-entropy function as the loss function for both models.

$$\mathcal{L} = -\sum_{\substack{traj\_id \\ t \in [0, T_{traj}-1]}} y_{traj\_id,t} \log(\hat{y}_{traj\_id,t}) + (1 - y_{traj\_id,t}) \log(1 - \hat{y}_{traj\_id,t}) \qquad (1)$$

4.2 Embeddings generation model

There has been studies showing the generating separate programming embeddings can improve a prediction task for programming languages ([1,2]). From the insight of these studies, we created our own programming embeddings to be used in the main prediction model. In order to create the embeddings, we devised a model which predicts the next code block given a partial view of a serialized AST from the beginning to any given point. Our hypothesis is that the model performing the following task would reveal the high-level semantics of the input AST as analogous to the next word prediction task in natural language processing. The model for such task is presented in Figure 2.

Each input label is an AST serialized by performing a preorder traversal in the AST. When performing the traversal, we add two special blocks `call` and `return` to identify vertical traversals with traversals across siblings. [2] Also we add an `end` block to indicate the end of the program. This is required in order to pad these *end* blocks to the serialized ASTs to match the maximum AST length (which we denote as T_ast). The blocks are one-hot encoded and therefore the input data matrix is 3-dimensional (`AST_ids`, `timestep`, 0/1)

The output labels are generated by shifting the input labels by one timestep since we are prediciting the next block. Additional end block is appended at the end of the output label vector.

Given a bounded problem space in each assignment, our hypothesis was certain sequence of blocks will have correlation with appearance of a particular block in the immediate future in order to solve the assignment. Therefore, we suspected that an LSTM RNN model would also be appropriate for the next block prediction task. The one-hot encoded blocks (of total B unique blocks) are fed into the

[1]We label the failures as 1 in order to examine the recall of the predictions. It is important to properly predict most of the student which might fail in the context of student feedback.

[2]We omit these blocks in Hoc4 dataset since there are no control blocks and the program runs in a strictly serial manner.

LSTM layer with a single output value for each timestep. The colleciton of these values are fed into a dropout layer for regularization. The final layer is a single dense network which outputs $B \times T\_ast$ values after passing a softmax activations. Softmax activation is used since the task is a multi-class classification problem with B classes. We also use the categorical cross-entropy loss function for this purpose.

$$\mathcal{L} = - \sum_{\substack{ast\_id \\ t \in [0, T_{ast}-1]}} \sum_{c \in [0, B-1]} y_{ast\_id,t,c} \log(\hat{y}_{ast\_id,t,c}) \tag{2}$$

5 Experiments/Results/Discussion

Tables 1 and 2 provide results for both of the exercises. We find significantly better performance on Hoc4 than on Hoc18 across all models and performance measures. This makes sense as Hoc4 is the easier of the two exercises. More interestingly, we find that logistic regression without embeddings is the highest performing model in both cases. We hypothesize that logistic regression performs well in these cases because the exercises are straight-forward. Embeddings not being helpful could either also be the result of the exercises being straight-forward, or could be an indicator that the embeddings fail to usefully characterize the code submissions.

Table 1: Results for Hoc4 (window length 2)

	Accuracy	Precision	Recall	F1-Score
Logistic regression (w/o embeddings)	0.9276	0.9125	0.9579	0.9344
Logistic regression (w/ embeddings)	0.8557	0.8714	0.8627	0.8665
RNN (w/o embeddings)	0.9010	0.8768	0.9499	0.9112
RNN (w/ embeddings)	0.9255	0.9147	0.9515	0.9325

Table 2: Results for Hoc18 (window length 64)

	Accuracy	Precision	Recall	F1-Score
Logistic regression (w/o embeddings)	0.7625	0.6944	0.8627	0.7628
Logistic regression (w/ embeddings)	0.6894	0.6370	0.7512	0.6823
RNN (w/o embeddings)	0.7778	0.7876	0.7142	0.7424
RNN (w/ embeddings)	0.7784	0.7656	0.7523	0.7523

We trained and validated the baseline and target NN model for both Hoc4 and Hoc18 coding exercises. We ran them with and without embeddings, which led to training and validation of eight independent models in total.

The embeddings were generated once for both Hoc4 and Hoc18 using the embeddings generation model define in section 4. This model was trained with dropout of 0.5, using Adam optimizer and Cross-entropy loss. The model was trained with batch size of 16 for 50 epochs. The output weights of LSTM layer were used as embeddings. Once the embeddings were generated, Baseline model was trained and validated with and without embeddings for 10 epochs using mini batch of 32, Stochastic gradient descent optimizer, cross-entropy loss and window length of 2.

The target NN model was also trained and validated with and without embeddings for 50 epochs using mini batch of 32, Adam optimizer, binary cross-entropy loss and window length of 64. We've also attempted to train the target model with mini batch size of 16 and 64, however we didn't observe any significant difference in the results.

During the training of these models we found that it is critical to pay attention to the distribution of the data especially to address the data representation for the few students who make many attempts before they are successful with the given programming exercise. We also experienced that predicting the next block to generate code embeddings may not be useful as students tend to use different logic in coding during learning. Additionally, predicting for longer window length is harder since it needs to predict outcomes in further future.

After undergoing the complete process, there are still the following three key open questions

1. How to determine the optimal window length which is an interesting educational question as well that corresponds to how long to let a student struggle
2. What should be unit of analysis for the these programming exercises, would it be individual student or a unique trajectory submission by students
3. The baseline performs incredibly well for which we hypothesize that this is the result of the simplicity of the coding exercise.

We've identified **recall** as the key metric because intervention to help student is not costly and having a student give up is.

6 Conclusion/Future Work

In summary, the baseline model does incredibly well (better than our deep learning approaches). There are a couple of possibilities for why this might be the case. One possibility that, in contrast to NLP, it's possible that code embeddings are not useful in some cases because students might use different logic in coding during learning. Another possibility is that these coding exercises were so simple such that our baseline of logistic regression was able to capture the structure of the block-based coding exercise information.

This research opens up two interesting lines of future research. The first is with regard to automatic hint generation in online educational environments. The advent of learning platforms such as Khan Academy has an ever-increasing number of students learning from software. This will require models that can offer hints that are both effective and timely. The second is with regard to the use of deep learning to analyze coding both in an educational and professional setting. The core beauty of deep learning techniques is that ability to leverage complicated structure in data that previous models could not. Coding both block-based and text-based is a perfect opportunity for deep learning to offer both predictive and generative future abilities.

GitHub Repo: The code for this work is available at the following Github repo for future reference https://github.com/raejoon/cs230-bnr

7 Contributions

Raejoon and Neeraj performed the majority of model architecture and tuning. Ben created the input and output data for the embeddings and the core model. All authors contributed equally to the writing of this paper.

References

[1] Piech, Chris, et al. "Autonomously generating hints by inferring problem solving policies." Proceedings of the Second (2015) ACM Conference on Learning@ Scale. ACM, 2015.

[2] Wang, Lisa, et al. "Learning to represent student knowledge on programming exercises using deep learning." Proceedings of the 10th International Conference on Educational Data Mining; Wuhan, China. 2017.

[3] Vihavainen, Arto. "Predicting Students' Performance in an Introductory Programming Course Using Data from Students' Own Programming Process." Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on. IEEE, 2013.

[4] https://code.org/research