Modeling student traversal of partial solutions

Irving Rodriguez {irodriguez@stanford.edu}

December 17, 2016

Abstract

Modeling the way students work through assignments is critical in making online courses fully sustainable. A model which successfully captures this behavior opens the way to automating classroom tasks which traditionally have been filled by human teachers, including grading, providing hints, and giving personalized feedback. As such, we propose two models to describe the way in which students modify their partial solutions on two programming puzzles. We focus on making our models generalizable such that they can be easily applied to assignments with less structure than programming ones, such as essay writing and drawing. Our partial solution language model, learned via a stacked LSTM network, achieved an F1 score of 66.2% and 53.6 in our language model task for our two datasets. Our cognitive Bayesian Data Analysis model achieved an F1 score of 21% and 16.2, only slightly better than baseline. Despite the modest results in comparison to previous work on our dataset, our models illustrate the usefulness of successfully describing how students solve problems, especially in the context of designing massive online courses to provide education cheaply and to the vast number of students across the world without access to traditional classrooms.

1 Introduction

Massive open online courses are amplifying the reach of education. Many of these MOOCs are the exclusive sources of education for many people across the world. Because of limited resources and the scope of the populations which these MOOCs reach, making the courses self-sufficient is an important task.

Some key components of these courses include grading or evaluating submissions, providing personalized feedback to students, and providing hints when necessary. Automatic these tasks is a difficult but necessary step in making these courses successful and extending their reach to students without requiring human teachers or graders. Personalizing these tasks is also an important component of increasing the courses' teaching value and their ability to scale to large student populations.

Chris Piech et al tackle the task of generating hints for students based on Code.org's Hour of Code challenge[4]. Though this research was quite successful in predicting the partial solutions students were likely to attempt next, the

methodology is difficult to generalize to other problems. Namely, how would this method work for problems with much larger partial solution spaces? Programming problems (along with other quantitative problems, like math proofs) are well-structured and have easily comparable solutions. This search-based method for predicting partial solutions, however, becomes less tractable for problems with ill-defined transitions between partial solutions. Some examples include writing an essay, answering a set of multiple choice questions after a passage, finding the numerical answer for a physics problem, or programming problems written in a text editor instead of a code-block environment.

As such, we propose predicting partial solutions via a "partial solution" language model and using a recurrent neural network to learn the features of partial solutions that best model students' problem solving strategies. Formally, we are looking for a way to compute $P(s_i|s_1, ..., s_{i-1})$ where s_j denotes some partial solution feature vector in our set of possible partial solutions. The idea is that the most descriptive information is in how a student traverses the partial solution space, and we hope to implicitly learn this in the weights of our RNN.

In the interest of developing an approach that is applicable to a wide array of subjects and problems, we will also build a cognitive model to describe the decisions students make when altering their current solution. We follow the Bayesian data analysis framework described by Goodman et al in which we propose probabilistic models of these decisions and use our data to infer which of our models best captures the characteristics of our dataset [9].

2 Dataset

Code.org provides a dataset gathered from a previous Hour of Code challenge. The dataset contains information about student solutions to programming puzzles 4 and 18, which we label **HoC4** and **HoC18**[2]. In these challenges, students solve small puzzles by combining code blocks as described in Figure 1. Students attempt to make the blue arrow, serving as the agent, to the heart.

The dataset is composed of 1) partial solutions submitted by students and counts for the number of times these partial solutions were executed and 2) unique "trajectories" (sequences of partial solutions) that students took en route to their final solution, also with accompanying counts. We define partial solutions as the code blocks the students executed. There are over 1.1 million total trajectories for Problem 4, with $10k$ unique trajectories. There are over 1.2 million total trajectories for Problem 18, with nearly $80k$ unique trajectories.

For our language model, we use an 80%/10/10 training/validation/test set split for each of the two datasets. Our cognitive model uses the entire dataset for each respective case.

Some examples of how partial solutions and trajectories are stored in the dataset are in the appendix.

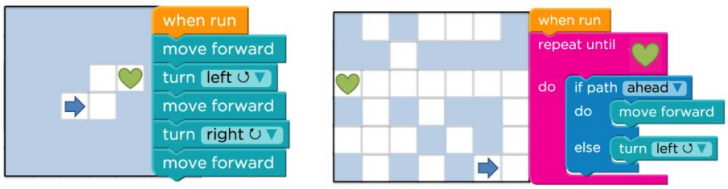


Figure 1: Hour of Code challenges 4 (left) and 18 (right) with accompanying solutions. The arrow is the agent and the heart is the goal.[2]

3 Partial Solution Language Model

With this data, we propose learning a mapping $T : S \rightarrow S$ from a partial solution to another partial solution based on the features of the current partial solution. This task should be more general than learning the transition probabilities from highly-dimensional partial solution space and should hence be more tractable for a wider array of problems. The mapping T is learned by calculating the language model probabilities $P(s_i|...)$ over the data set for each sub-trajectory in our data. The analog is that student trajectories represent the "sentences" of our language model while the partial solutions are the tokens.

As a baseline, we implement Naive Bayes to measure how much of the partial solution features can be captured by this naive (no pun intended) model. We take the Poisson path algorithm from Piech et al as our oracle[4]. This algorithm is able to correctly predict 95.9% of all ground truth transitions between partial solutions in **HoC4** and 84.6% of the transitions **HoC18**. As described in their paper, the Poisson path algorithm minimizes the amount of time it takes for the average student to reach the final solution. The amount of time it takes to generate a solution is modeled as a Poisson rate calculated from each partial solution's counts in the dataset.

To learn our mapping T, we implement a Long Short Term Memory (LSTM) network in Tensorflow[1]. The features of our partial solution are described below.

3.1 Partial Solution Features

We create our "word vector" analogs using indicator variable for the presence of unique code blocks in the given partial solution. We iterate over individual and nested features:

1. Individual features: We create an indicator variable for every unique code-block in the HoC puzzles. This includes blocks like "turnLeft" and control structures like "if path ahead."

2. Nested features: For each control structure, we create an indicator variable conditioning on other non-control structures that are nested inside. One

such variable is "if path ahead — turnleft" meaning there is a "turnLeft" block inside the control structure.

We abstain from engineering other features so that our results demonstrate the quality of prediction our models can achieve with general feature templates. This template works well for most programming assignments and similar approaches can easily be implemented in other subjects.

Iterating through the datasets for puzzle 4 and 18 yields feature vectors of size 11 and 33, respectively.

4 LSTM Network

4.1 Architecture

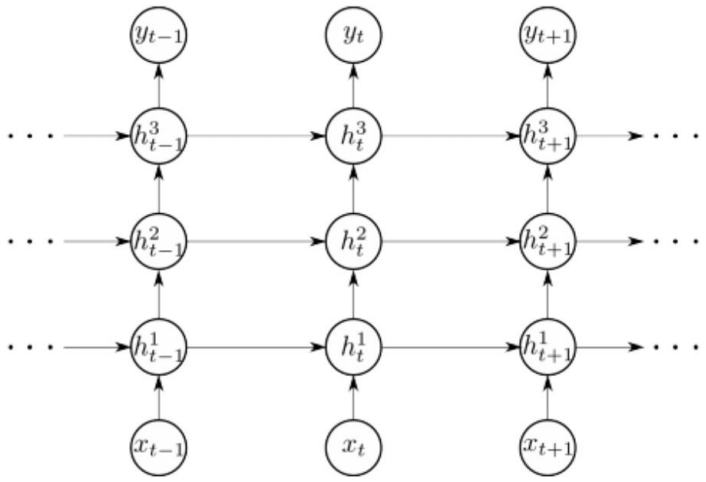


Figure 2: Illustration of stacked LSTM model for a given time step t. The superscript on the hidden states h denote the layer index. Note that we define the outputs at each time step to be $\hat{y}_t$ instead of y_t, as in the illustration. [3]

LSTM networks are popular models for solving traditional language model tasks [5] [6]. Based on our modeling of the partial solution traversal, the LSTM architecture is a natural fit for our problem.

The LSTM network consists of cell units which calculate the intermediate hidden states of the network at any given "time" step of the network (i.e. at the t^{th} input to the network.) Specifically, the cell contains a state C_t which process the input x_t based on four gated inputs, i_t, f_t, o_t, c_t, called the input,

forget, output candidate, and activation gates, respectively:

$$i_t = \sigma(W_i h_{t-1} + W_i s_t + b_i) \tag{1}$$
$$f_t = \sigma(W_f h_{t-1} + W_f s_t + b_f) \tag{2}$$
$$o_t = \sigma(W_o h_{t-1} + W_o s_t + b_o) \tag{3}$$
$$c_t = tanh(W_c h_{t-1} + W_i s_t + b_c) \tag{4}$$

where σ denotes the sigmoid function, h_{t-1} denotes the hidden state at the previous time step, s_t denotes the current partial solution feature vector that serves as input to the network, and the b, W's denote the biases and weights of each respective gate.

In essence, each of these gates calculates which components of the vectors within the cell carry meaningful information and updates the cell state accordingly. The forget gate scales down the components of the input state C_{t-1}, the input gate scales the components of w_t to be added to the cell state, the activation gate determines the new component values to be added to the cell state, and the output gate yields a vector to scale the cell's prediction.

We then update the cell state using these calculations and obtain our hidden state h_t:

$$C_t = f_t \times C_{t-1} + (1 - f_t) \times c_t \tag{5}$$
$$h_t = o_t \times tanh(C_t) \tag{6}$$

where $\times$ denotes element-wise multiplication. We then apply a softmax layer to compute our vector of posteriors over our partial solution space:

$$\hat{y} = softmax(W_s h_t + b_s) \tag{7}$$

For our task, we feed a sequence of partial solutions $(s_1, ..., s_N)$ for batch size N into the network and predict the partial solution with maximum likelihood, given by the index of the element with the highest probability in $\hat{y}$.

We stack these cells such that the output of the cell at a lower layer, h_t, becomes the input to the cell in the layer above, x_t. (Figure 2)

4.2 Optimization

We optimize our network by minimizing the cross-entropy loss:

$$J(y, \hat{y}) = -\Sigma_i y_i ln(\hat{y_i}) \tag{8}$$

where y denotes our one-hot label vector and $\hat{y}$ is our posterior probability vector, as outlined in Eq. 7. This loss has the desirable property that it scales with the residual of the prediction while simultaneously minimizing the difference between our predicted posteriors $\hat{y}$ and the true population distribution in partial solution space.

The cross-entropy loss is a convex function and can thus be optimized with gradient descent. We take advantage of the sequence nature of our architecture

and perform mini-batch SGD where our batch size depends on the length of the input sequence of our model at one iteration.

Given that our network can be quite large, we can implement dropout if we find that the model overfits. Dropout in an LSTM is accomplished by stochastic pruning of vertical connections *only* so that the recurrent update of the cell states between time steps is not disturbed [7]. (TensorFlow's dropout wrapper for LSTMs does exactly this.)

4.3 Hyperparameters

Aside from the free parameters associated with our cells and our softmax parameters, we have the hyperparameter set $\{N, H, L, \alpha\}$, denote the window size N of the input context into our model, the number of stacked layers of cells L, the number of hidden units H of our model (i.e. the dimensionality of C_t), and our SGD learning rate α. The size of N effectively serves as the n-gram parameter for our trajectories, so we fix it to 10 since most trajectories are on the shorter side. Tuning is done with dropout after noticing a large discrepancy between training and validation (We pad trajectories shorter than this with 0 vectors in Tensorflow, which are guaranteed to not cause updates during training, though they introduce a bias towards the blank solution.)

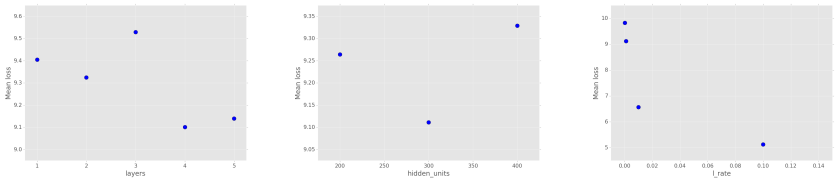


Figure 3: **Mean loss vs hyperparameter values** for the HoC4 dataset. Context size fixed to 10 for all experiments; default values are $[H = 200, L = 2, \alpha = 10^{(-3)}]$. Optimal values are $[H = 300, L = 4, \alpha = 0.1]$.

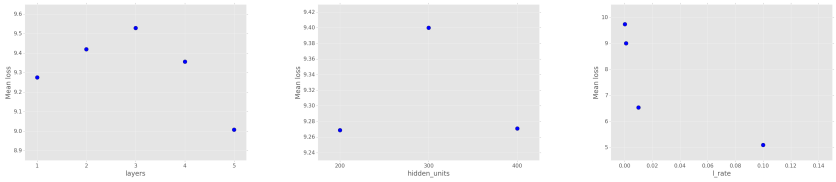


Figure 4: **Mean loss vs hyperparameter values** for the HoC18 dataset. Context size and default parameters are the same as tuning for the HoC4 dataset, Figure 3. Optimal values are $[H = 200, L = 5, \alpha = 0.1]$.

5 Cognitive Bayesian Data Analysis

One of the main disadvantages of a neural network approach is the unwieldy size of the parameter set, which can easily exceed millions of parameters depending on the architecture. (Indeed, each cell in our LSTM contains five $H \times H$ weight matrices alone.) It furthermore requires a vast amount of data to correctly learn students' behavior. If we instead hypothesize how students change the features of their partial solutions, we can use Bayesian Data Analysis to infer which of these hypotheses better explains our observations. This approach is more fundamental in the sense that we are explicitly modeling how a student makes the decision to change solution features based on what she sees while maintaining the ability to learn from data with a much more compact set of free parameters.

We use WebPPL, a publicly available language for performing probabilistic inference, to perform our Bayesian data analysis [8]. We adopt the framework outlined by Tessler and Tenenbaum [9]. We propose several hypotheses from a students' perspective, randomly sample from our set of generative models of these hypotheses, and condition on observing our Code.org data to learn the probability distribution over our hypothesis set. We then use the posterior distributions of the free parameters of each hypothesis to generate a posterior predictive distribution over our partial solution space, S, conditioned on our observed data. We can predict the next partial solution using this distribution. (We note that this is similar to the probability distribution generated in the softmax layer of our network.)

5.1 Student Traversal Hypotheses

We form a set of hypotheses by combining statistics of the Code.org data and intuition of students' perspective of the programming puzzles. The number of changes made is perhaps the most distinguishing feature during partial solution modification, so our hypotheses are created over this spectrum.

This hypothesis structure aligns well with teaching strategies suggested by the experts surveyed in Piech et. al. and in the introduction to the Code.org challenges. Namely, students are generally advised to execute their code frequently in between small changes to minimize bugs and ensure their program is behaving as expected [4], [2]. It is still common, however, for students to deviate from this strategy for several reasons. Students may be stuck, realize they have taken an erroneous approach, or are not satisfied with the progress of their current approach. In these cases, more dramatic changes may be justified, such as erasing a large part of their code or returning to a blank solution entirely. We can formally characterize changes between one partial solution s_1 and its consequent solution, s_2, via the L_2 norm of their feature vectors. That is, if $||s_2 - s_1||_2 \leq \delta$ for some δ, we can characterize the change as an incremental one, else it is a dramatic one. We choose $\delta = \sqrt{2}$, denoting the student made 1 or 2 changes to their code.

We build these five hypotheses as generative models in WebPPL and con-

struct a Bayesian data analysis model conditioned on the trajectories in our dataset. In order to properly traverse second degree solutions for hypothesis 2, we build a dictionary that stores each unique solution ID as a key and has a dictionary of (solutionID, counts) for each partial solution with a direct transition to the given solution.

Hypothesis 1, incremental changes (probabilities in parentheses):

1. Change deepest nested statements one at a time (ie change turn left to move forward) (P_{1H1})
2. Change nested control structures (ie from) ($P_{2H1} < P_{1H1}$)
3. Change entire control structure (ie from outer repeat loop to if statement) ($P_{3H1} < P_{2H1} < P_{1H1}$)

Hypothesis 1.5, uniform incremental changes:

1. ($P_{3H1} = P_{2H1} = P_{1H1}$)

Hypothesis 2, dramatic changes:

1. The student moves to a previously visited solution directly (without traversing the solutions in between) (P_{1H2})
2. The student moves to a solution similar to a previously visited one (ie 2 degrees away) ($P_{2H2} < P_{1H2}$)
3. The student moves to the blank solution ($P_{3H2} < P_{2H2} < P_{1H2}$)

Hypothesis 2.5, uniform dramatic changes:

1. ($P_{3H2} = P_{2H2} = P_{1H2}$)

Hypothesis 3, balanced changes:

1. With probability $1 - d$: do incremental change
2. With probability d: do dramatic change

6 Results

6.1 Language Model Results

The optimal hyperparameters are $[H = 300, L = 4, \alpha = 0.1]$ and $[H = 200, L = 5, \alpha = 0.1]$ for the **HoC4** and **HoC18** datasets, respectively[Figures 3, 4]. We cite only the results after implementing dropout in both hyperparameter tuning and training, as there was a large discrepancy in training and validation F1 score under our original experiment without dropout.

Training with the optimal set of parameters results in an F1 score of 66.2% and 53.6% on **HoC4** and **HoC18**. With early stopping, training concluded after 260 and 330 epochs, respectively.

	$\mathbf{HOC}_4$	$\mathbf{HOC}_{18}$
Naive Bayes	14.3%	10.7
Oracle	95.9	84.6

Table 1: **Baseline performance** from a Naive Bayes classifier using our featurized language model and the Poission path prediction algorithm from Piech et al's Markov Decision Process approach.

	Hidden Units	Layers	Learning Rate	Dropout
HoC 4	300	4	10^{-1}	True
HoC 18	200	5	10^{-1}	True

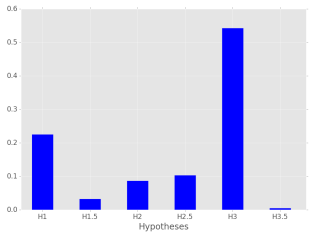
	Train F1	Validation	Test
HoC 4	75.6%	69.7	66.2
HoC 18	60.3	56.3	53.6

Figure 5: **LSTM training results.** Optimal hyperparameter values for both puzzle datasets (top), dropout added among vertical layers to reduce the large gap in training and validation F1 scores. F1 score of LSTM model on respective dataset splits (bottom).

6.2 Cognitive Model Results

The Bayesian data analysis model places most of the posterior probability mass, nearly 55%, on Hypothesis 3 (balanced changes). The median value of the branching parameter d is 64.3%.

The next most likely hypothesis is Hypothesis 1, with 23% of the posterior probability mass. The values of the free parameters for each of the hypotheses are shown in Figure 6. Using these median values, we are able to predict 21.0% and 16.2% of partial solution traversals in the **HoC4** and **HoC18** datasets.



	H1	H1.5	H2	H2.5	H3
P1H1	42.2	33.3			24.5
P2H1	39.4	33.3			24.1
P3H1	18.4	33.3			2.5
P1H2			46.5	33.3	23.7
P2H2			45.4	33.3	22.2
P3H2			8.0	33.3	3.1
d					64.3

Figure 6: **Bayesian data analysis results;** posterior distribution over the hypothesis space (left) and median values of posterior distributions over respective model parameters (right).

7 Discussion

	HoC4	HoC18	LSTM-4	LSTM-18	BDA-4	BDA-18
Incremental Traversals	59.5%	54.2	72.4	56.5	65.8	66.1
Dramatic Traversals	41.3	45.8	27.6	43.5	34.2	33.8

Figure 7: **Proportion of predictions** between incremental and dramatic changes for both the LSTM and cognitive Bayesian analysis model. True distribution from full datasets included.

7.1 Language Model

The stacked LSTM achieves modest results in comparison to Piech et al [Table 1]. Given the size of the dataset and of our optimal architecture, these results are encouraging. It appears that the model extracts useful information

from a partial solution's context to make a prediction, especially when compared to predictions made by Bayesian data analysis inference [Figure 7]. It is also promising that we achieved these results with a rather simple feature template that did not require much domain knowledge of the problem or subject of interest.

The final parameter set is nevertheless quite large. It certainly says a lot that the model initially overfit the training data, with train/validation splits of 81/54 for **HoC4** before dropout was added. The main tradeoff between problem simplicity and model size lies mostly in the scaling of the number of available traversals. **HoC4** has a small number of possible traversals with its 9 solution features, but even a few feature additions can quickly increase the traversal hypothesis space. If students were allowed to set variable values (say *moveForward(4 steps*) the traversal space can even become infinite. This likely explains why 300 hidden units can learn (and memorize) on a problem as simple as **HoC4:** contexts for different solutions remain varied and indeed increase in diversity as feature complexity increases. The obvious detriment of our model's scale is the necessary size of data required for training.

We note that the LSTM for both datasets interestingly over-estimates the number of traversals reached via incremental changes to the current solution. It is somewhat surprising that the gap for LSTM-4 between incremental and dramatic traversals in particular is so large given the simplicity of puzzle 4. Over the predictions made, 72.5% of LSTM-4's traversals were made with incremental changes to the current partial solution. That is, we expected that the first change students make is dramatic, since there are 4 code blocks to add to the blank solution to reach the correct one. Students would then make small tweaks if they made a mistake at that point. The model seems to miss this intuition for the simpler problem, though it is encouraging that the gap was much smaller for LSTM-18. This may be a feature (no pun intended) of the exclusively indicator-based feature vector we used in training.

Indeed, there is much subtle information that our feature vector does not fully quantify. Perhaps most obviously, certain featurizations of distinct solutions could in fact result in the same feature vector. For example, *if clear: move forward, while not heart: if clear: move forward* maps to the same feature vector as the solution excluding the first if-statement. This could skew the mapping T away from "better" solutions (like the former one) in order to account for different contexts that lead to the same path and thus introducing aberrations in the model's posterior probability over the solutions. We postulate that the greatest gains in performance can be made by exploring other feature templates which aim to extract more information from the current given solution.

Aside from modeling changes, there are several ways of improving the performance of the model as-is. These include tuning the context window size as an additional hyperparameter, adding more advanced SGD optimizations like Adagrad, and increasing the size of the neural net. Each of these attempts to squeeze more predictive power out of the outstanding model, though they would be largely unproductive efforts if the current performance was close to the minimum possible loss achievable from this architecture. The final cross-entropy loss

of 2.18 suggests there may be indeed be some significant difference between the posterior probability distributions over the number of unique solutions and the true population distribution for the dataset. Further experiments could focus on different initialization methods for the cell and softmax weights in addition to the additional optimizations. With more time, we would have liked to pay closer attention to the size of the gradient updates for each weight over the course of training, as these would provide insight into how much juice we're leaving on the table.

7.2 Cognitive Model

The cognitive model had poorer predictive performance than either the LSTM or the oracle, correctly predicting only 21% and 16.2% for the two datasets [Figure 6]. In both cases, it performed better than the Naive Bayes baseline.

We expected Hypotheses 1 to be the best m odel of student traversal, particularly because of the changes most commonly found in the datasets [Appendix IA]. When making incremental changes, students make the more minute changes (like changing the value of one nested block) over 30% of the time, suggesting that students were often able to reach a stable partial solution close to the correct final answer. Even so, the "balanced" hypothesis had over twice the probability mass as the first hypothesis, though with a stronger predilection for making incremental traversals than either dataset [Figure 7]. Eliminating the uniform sanity-check hypotheses resulted in moving most of the probability mass from these models (around 16%) to Hypothesis 3.

The posterior predictives from BDA likely do not match up well with data because of the heuristic hypotheses proposed even though the traversals available to the student in our model closely match the traversals that the average student made in the data. For example, the second and third hypothesis sufficiently made the blank solution available to students at any partial solution. At the same time, the model matched the frequency with which students start over (3.1% by Hypothesis 3 compared to $1.8, 5.7$ ground truths) relatively well, indicating that it likely made this traversal more probable for solutions that had strayed from a good path to the final solution. Other generative models which incorporate more sophisticated information, like student tendencies or demographics, may distribute the free parameters (in this case, the respective probabilities) such that they are closer to the ground truth distributions shown in Appendix IA.

This brings up the cognitive model's main strength in the face of its poor predictive power. Compared to learning MDP transitions as in Piech et al or context information as in our language model, BDA is more agile and much easier to iterate over. As mentioned in the previous paragraph, it is much simpler to incorporate additional parameters into the model and then to quickly see how these affect the posterior distributions of hypotheses and parameters. One example is the introduction of a "mistake tolerance" parameter: some students may only tolerate a few incorrect partial solutions before starting over altogether. Likewise, if a student is a visual learner, they may acquire more

information from seeing an incorrect partial solution execute on their screen and thus may make lots of dramatic changes in their traversal. It would be difficult to incorporate these explicitly into the bias terms of our LSTM, but would only require a few lines of code in WebPPL to incorporate into our BDA and to see its impacts.

Additionally, this model is much more robust to data size than either the LSTM or the oracle. Running BDA and posterior prediction with half of our dataset yields very similar distributions as in Figure 6. As such, this approach may be more tractable for less popular courses or as a "warm" model for new courses until enough data is collected for larger models to have the requisite descriptive power.

8 Conclusion

We have presented two possible methods for modeling how students solve problems and the strengths and weaknesses of each respective model. Our language model-LSTM approach achieved modest results, attaining an F1 score of 66.2% and 53.6% on our two datasets. Our cognitive model of students was less successful, with respective F1 scores of 21% and 16.2%.

In developing our framework, we especially emphasized the generalizability of our methods to other subjects and, by extension, to other types of problems. In particular, we'd be interested to see results that stem from our agile approach of using a general feature template and using this template to utilize both 1) the context of previous partial solutions and 2) a model of student decision making.

How much better can this system become? How well does it translate to problems with much larger feature spaces? Answering these questions successfully would go a long way in providing education to a wide swatch of students without the expensive use of traditional resources.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Code.org. Anonymized data for research purposes.

[3] C. et. al. Gated feedback recurrent neural networks.

[4] C. P. et. al. "autonomously generating hints by inferring student problem solving policies".

[5] S. et. al. Lstm neural networks for language modeling.

[6] T. et. al. Recurrent memory networks for language modeling.

[7] Z. et. al. Recurrent neural network regularization.

[8] N. D. Goodman and A. Stuhlmuller. The Design and Implementation of Probabilistic Programming Languages, 2014.

[9] N. D. Goodman and J. B. Tenenbaum. Probabilistic Models of Cognition, 2016.

Appendices

APPENDIX I

.1 Code.org Dataset Examples

In the data, "children" denotes code blocks embedded inside other code blocks. Using *pprint* makes it easy to see the control flow of the program.

Partial Solution Examples: HoC$_4$, Solution #997

This example solution shows the structure of this partial solution. Here, the student correctly went forward, turned left, went forward, and then turned right four times.

Raw json:

```
[u'type', u'id', u'children']
u'program'
u'15'
[{u'children': [{u'id': u'0', u'type': u'maze_moveForward'},
                {u'children': [{u'id': u'1', u'type': u'turnLeft'}],
                 u'id': u'2',
                 u'type': u'maze_turn'},
                {u'id': u'3', u'type': u'maze_moveForward'},
                {u'children': [{u'id': u'4', u'type': u'turnRight'}],
                 u'id': u'5',
                 u'type': u'maze_turn'},
                {u'children': [{u'id': u'6', u'type': u'turnRight'}],
                 u'id': u'7',
                 u'type': u'maze_turn'},
                {u'children': [{u'id': u'8', u'type': u'turnRight'}],
                 u'id': u'9',
                 u'type': u'maze_turn'},
                {u'children': [{u'id': u'10', u'type': u'turnRight'}],
                 u'id': u'11',
                 u'type': u'maze_turn'},
                {u'children': [{u'id': u'12', u'type': u'turnRight'}],
                 u'id': u'13',
                 u'type': u'maze_turn'}],
  u'id': u'14',
  u'type': u'statementList'}]
```

Partial Solution Example: HoC$_{18}$ Solution #10004

This example demonstrates how "embedded" code is represented in the dataset. The children of the "DO", "if", and "else" statements are explicitly shown prior to the control statement.

```
[{u'children': [{u'children': [{u'children': [{u'id': u'0',
            u'type': u'maze_moveForward'},
            {u'children': [{u'id': u'1',
                u'type': u'turnLeft'}],
            u'id': u'2',
            u'type': u'maze_turn'},
            {u'id': u'3',
            u'type': u'maze_moveForward'},
            {u'children': [{u'id': u'4',
                u'type': u'isPathRight'},
                {u'children': [{u'children': [{u'id': u'5',
                       u'type': u'turnLeft'}],
                    u'id': u'6',
                    u'type': u'maze_turn'}],
                u'id': u'7',
                u'type': u'DO'},
                {u'children': [{u'children': [{u'id': u'8',
                      u'type': u'turnRight'}],
                    u'id': u'9',
                    u'type': u'maze_turn'}],
                u'id': u'10',
                u'type': u'ELSE'}],
            u'id': u'11',
            u'type': u'maze_ifElse'}],
        u'id': u'12',
        u'type': u'statementList'}],
    u'id': u'13',
   u'type': u'DO'}],
u'id': u'14',
u'type': u'maze_forever'}]
```

.2 Trajectory Examples

The trajectory files are simply text files with a sequence of partial solution IDs that a student executed before submitting a final solution. Note that consequent duplicates indicate a student executed the code twice, perhaps to see the animation once more. This can be pre-processed in the data, though there is arguably some information contained in the act of having to see a partial solution twice before modifying.

Trajectory Example: HoC$_4$ Trajectory #9994
44, 44, 6, 6, 17

Trajectory Example: HoC_{18} Trajectory #9994
34, 6, 27, 27, 213, 43, 110, 110, 0

A Code.org Dataset Statistics

	HoC 4	HoC 18
Incremental Traversals	59.5%	54.2
Dramatic Traversals	41.3	45.8
1H1	59.5	30.4
1H2		19.7
1H3		3.1
2H1	25.6	20.6
2H2	13.9	19.5
2H3	1.8	5.7

Figure 8: **Proportion of solution change types**, enumerated by types of changes described in proposed hypotheses of partial solution traversals.