Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference

Mike Wu,[1] **Milan Mosse,**[1] **Noah Goodman,**[1,2] **Chris Piech**[1]
[1] Department of Computer Science, Stanford University, Stanford, CA 94305
[2] Department of Psychology, Stanford University, Stanford, CA 94305
{wumike,mmosse19,ngoodman,piech}@stanford.edu

Abstract

In modern computer science education, massive open online courses (MOOCs) log thousands of hours of data about how students solve coding challenges. Being so rich in data, these platforms have garnered the interest of the machine learning community, with many new algorithms attempting to autonomously provide feedback to help future students learn. But what about those *first* hundred thousand students? In most educational contexts (i.e. classrooms), assignments do not have enough historical data for supervised learning. In this paper, we introduce a human-in-the-loop "rubric sampling" approach to tackle the "zero shot" feedback challenge. We are able to provide autonomous feedback for the first students working on an introductory programming assignment with accuracy that substantially outperforms data-hungry algorithms and approaches human level fidelity. Rubric sampling requires minimal teacher effort, can associate feedback with specific parts of a student's solution and can articulate a student's misconceptions in the language of the instructor. Deep learning inference enables rubric sampling to further improve as more assignment specific student data is acquired. We demonstrate our results on a novel dataset from Code.org, the world's largest programming education platform.

Introduction

The need for high quality education at scale poses a difficult challenge. The price of education per student is growing faster than economy-wide costs (Bowen 2012), limiting the resources available to support student learning. When also considering the rising need to provide adult retraining, the gap between the demand for education and our ability to provide is especially large. In recent years, massively open online courses (MOOCs) from platforms like Coursera and Code.org have made progress by scaling the delivery of *content*. However, MOOCs largely ignore an equally important ingredient for learning: high quality *feedback*. The clear societal need, alongside massive amounts of data has led to a machine learning grand challenge: learn how to provide feedback for education at scale, especially in computer science due to its apparent structure and high demand.

Scaling feedback (a.k.a. "feedback" challenge) has proven to be a hard machine learning problem. Despite dozens of projects to combine massive datasets with cutting edge deep learning, current approaches fall short. Three issues emerge: (1) for even basic computer science education, homework datasets have statistical distributions with heavy tails similar to natural language; (2) hand labeling feedback is expensive, rendering supervised solutions infeasible; (3) in real world contexts feedback is needed for assignments with small (or zero) historical records of student learning. For the billions of learners around the world, most education and assessments have *at most* hundreds of records. Even if students use Code.org, assignments are constantly changing, making the small-data context perennial. It is a zero-shot solution that has potential to deliver enormous social impact.

We build upon a simple insight that enables us to move beyond the supervised paradigm: When experts give feedback, they are asked to perform the hard task of predicting misconception (y) given program (x). When breaking down the cognitive steps that experts go through, they often solve the inference by first thinking generatively $p(x, y)$. They imagine, "if a student were to have a particular set of misconceptions, what sorts of programs is he or she likely to produce." Thinking generatively is much easier: while there are a finite set of decomposable misconceptions, they combine into exponential amounts of unique solutions.

We formalize this intuition into a technique we call "rubric sampling" to elicit samples from an expert prior of the joint distribution $p(x, y)$ and use deep learning for inference $p(y|x)$. With no historical examples, rubric sampling enables feedback with accuracy close to the fidelity of human teachers, outperforming data-intensive state of the art algorithms. We case study this technique on Code.org, an online programming platform that has been used by 610 million students and has provided a full curriculum to 29 million students, equivalent to 39% of the US K12 population.

Specific contributions in this paper:

1. We introduce the Zero Shot Feedback Challenge on a dataset from 8 assignments from Code.org along with an evaluation set of 800 labels.
2. We articulate a novel solution: rubric sampling with deep learning inference which sets the new state of the art in code feedback prediction: F1 score doubled over baseline, approaching human level accuracy.

3. We introduce the ability to (i) attribute feedback to specific parts of code, (ii) trace learning over time and (iii) generate synthetic datasets.

The Zero Shot Feedback Challenge

The "Zero-Shot" Feedback Challenge is to infer the misconceptions that led to errors in a student's answer using zero historical examples of student work and zero expert annotations. Though this challenge is difficult, it is a task that humans find straightforward. Experts are especially adept at generalizing: an instructor does not need to see thousands of instances of a misunderstanding in order to understand it.

Why is zero-shot so important? Human annotated examples are surprisingly time consuming to acquire. In 2014, Code.org launched an initiative where hundreds of thousands of instructors were crowdsourced to provide feedback to student solutions[1]. Labeling was hard and undesirable work and the long tail of unique solutions meant that even after thousands of human hours of teacher work, the annotations were only scratching the surface of feedback. The initiative was cancelled after two years and the effort has not been reproduced since. For small classrooms and massive online platforms alike, it is infeasible to acquire the supervision required for contemporary nonlinear methods.

We foresee three main approaches: (1) learn to transfer information from one assignment to another, (2) learn to incorporate expert knowledge, and (3) form algorithms that can generalize from small amounts of human annotations.

Related Work

Education Feedback If you were to solve an assignment on Code.org today, the hints you would be given are generated from a unit test system combined with static analysis of the students solution. It has been a widely reported social-good objective to improve upon these hints (Price and Barnes 2017) especially since the state of the art is far from ideal (O'Rourke, Ballweber, and Popovií 2014). Achieving this goal has proved to be hard. Previous research on a more basic set of Code.org challenges (the "Hour of Code") have scratched the surface with respect to providing feedback at scale. Original work found that latent patterns in *how* students solve programming assignments have signal as to how he or she should proceed (Piech et al. 2015c). Applying a neural network improved prediction of feedback (Piech et al. 2015a) but models were (1) too far from human accuracy, (2) weren't able to explain its predictions and (3) required massive amounts of data. The current state of the art combines these ideas and provides some improvements (Wang et al. 2017a). In this paper we propose a method which uses less data, approaches human accuracy and works on more complex Code.org assignments by diverging from the classic supervised framework. Research on feedback for even more complex assignments such as medical education (Geigle, Zhai, and Ferguson 2016) and natural language questions (Bulgarov and Nielsen 2018) has also relied on data-hungry supervised learning and perhaps would benefit from a rubric sampling inspired approach.

[1]http://www.code.org/hints

Theoretical inspiration for our expert-based generative rubric sampling derives from Brown's "Repair Theory" which argues that the best way to help students is to understand the generative origins of their mistakes (Brown and VanLehn 1980). Simulating student cognition has been applied to simple arithmetic problems (Koedinger et al. 2015) and recent hard coded models have been very successful in inferring why students make subtraction mistakes (Feldman et al. 2018). Researchers have argued that such expert models are infeasible for assignments as complex as coding (Paaßen et al. 2017). However, the automated hierarchical decomposition achieved by (Nguyen et al. 2014) inspired us to develop rubric sampling, a simple expert model that works for programming assignments.

Zero Shot Learning There is a growing body of work in zero shot learning from the machine learning community, spawned by poor performance on unseen data.

The simplest approach is to include a human-in-the-loop. While labeling is one way human experts can "teach" a model, it is often not the most efficient. Instead, (Lee et al. 2017) leverages knowledge graphs build by humans to estimate a similarity score. Similarly, (Lake, Salakhutdinov, and Tenenbaum 2015) present a probabilistic knowledge graph (i.e. a Bayesian program) for generating symbols that outperform deep learning on out-of-sample alphabets. In this work, we employ a specific knowledge graph called a grammar, which we find to improve generalization.

A more complex approach (without human involvement) focuses on unsupervised algorithms to estimate the data distribution. (Verma et al. 2018; Wang et al. 2017b) train an adversarial autoencoder by generating *synthetic* examples and concurrently fitting a discriminator to classify between synthetic and empirical data. (Xian et al. 2018) propose a similar method for a CNN feature space. In this paper, we generalize this technique to a larger class of (non-differentiable) models: in lieu of a discriminator, we interpolate between synthetic and empirical data via a multimodal autoencoder.

The Code.org Exercises Dataset

Code.org is an online education platform for teaching beginners fundamental concepts in programming. Students build their solutions in a drag-and-drop interface that pieces together blocks of code. Its growing popularity since 2013 has captured a large audience, having been used by 610 million students worldwide. We investigate a single curriculum consisting of 8 exercises from Code.org's catalog. In this particular unit, students are learning to combine nested for loops with variables, and particularly the use of a for loop counter in calculations. The problems are all presented as drawing geometric shapes in a 2D coordinate space, requiring knowledge of angles. For instance, the curriculum begins with the task of drawing an equilateral triangle (see Figure 1).

The dataset is compiled from 54,488 students. Each time a student runs their code, the submission is saved. Each student is then associated with a *trajectory* of programs whose length depends on how many tries the student took. In total, there are 1,598,375 submissions. Since the exercises do

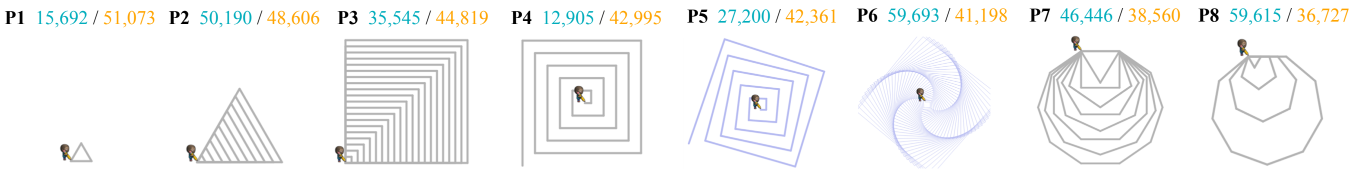


Figure 1: The curricula for learning nested for loops in Code.org. To provide intuition on the vast domain complexity, we show the number of unique solutions (blue) and the number of students (orange) who attempted the problem for each of the 8 exercises.

not have a bounded solution space, students could produce arbitrarily long programs. This implies that, much like natural language, the distribution of student submissions has extremely heavy tails. Figure 2 shows how closely the submissions follow a Zipf distribution. To emphasize the difficulty, even after a million students, there is still a 15% chance that a new student generates a solution never seen before.

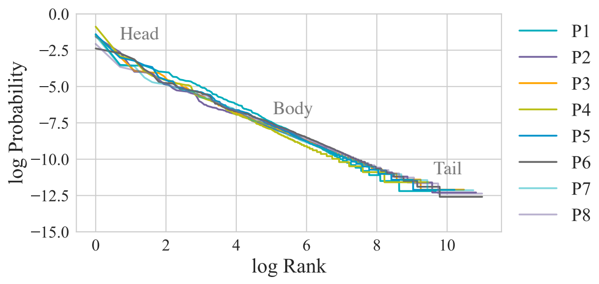


Figure 2: The distribution of programs for 8 problems from Code.org follow closely to a Zipf distribution, as shown by the linear relationship between the log probability of a program and the log of its rank in frequency. 5 to 10 programs dominate while the rest are in the heavy tails.

Evaluation Metric If we only cared about accuracy, we would prioritize the handful of 5 to 10 programs that make up the majority of the dataset. But given the infinite number of possible programs, struggling students who would benefit most from feedback will almost certainly not submit any one of the "most likely" programs. Knowing this, we define our evaluation metrics in terms of the Zipf itself: let the *head* (of the Zipf) refer to the top 20 programs ordered by frequency, the *tail* as any program with a frequency of three or less, and the *body* as everything in between. Figure 2 shows the rough placement of these three splits. When evaluating models, we ignore the head: these *very common* programs can be manually labeled. Instead, we will report two F1 scores[2]: one for programs in the body and one for the tail.

Human Annotations We collected fine-grained human annotations to over 800 *unique* solutions (chosen randomly from P1 and P8) from 7 teaching assistants from the Stanford introduction to programming course. We chose to label programs from the easiest (P1) and hardest (P8) exercises as they are most different. Intermediate exercises (P2 to P7) share many of same structural features. The annotations are binary labels of 20 misconceptions that cover geometric concepts (e.g. doesn't understand equilateral is 60 degrees) to control flow concepts (e.g. repeats code instead of using a loop). 200 annotations were used to measure inter-rater reliability and the remaining 600 were used for evaluation. We refer to this dataset as D. Labeling took 25.9 hours (117 seconds per program). At this rate, the entire dataset would take 9987 hours of expert time, over a year of continual work.

[2]We choose F1 score over accuracy as the labels are not close to balanced. Thus, accuracy tends to overinflate numbers.

Methods

We consider learning tasks given a dataset of n labeled examples, where each example (indexed by i) has an input string x_i and a target output vector $y_i = [y_{i,1}, ..., y_{i,l}]$ composed of l independent binary labels. In this context, we assume each string represents a block-based program in Lisp-like notation. Specifically, a program string is a sequence of T_i tokens, each representing either an operator (functions, *for* loop, *if* statements, etc.), an operand (i.e. variable), an open parenthesis "(", or a close parenthesis ")". See Listing 1 for an example program. Formally then, we describe the dataset as: $D = \{x_i, y_i\}_{i=1}^{n}$ where $x_i = [x_{i,1}, ..., x_{i,T_i}]$. The goal is to learn a function $\hat{y}_i = f(x_i)$ such that we minimize the error metric defined above, $err(\hat{y}_i, y_i)$. For supervised approaches, we split D into a training (D_{train}) and test set (D_{test}) via a 80-20 ratio. For unsupervised methods, we evaluate on the entire set D.

Listing 1: Example x_i from P1 with 51 tokens. The tokens *Program* and *WhenRun* serve the role of a start-of-sentence tokens. A model will receive each token in order.

```
( Program ( WhenRun ) ( Move ( Forward
   ) ( Value ( Number ( 50 ) ) ) ) (
   Repeat ( Value ( Number ( 3 ) ) ) (
   Body ( Turn ( Left ) ( Value (
   Number ( 120 ) ) ) ) ) ) )
```

Baselines

Majority Label As a sanity check, we can naively make predictions by voting for the majority label from D_{train}.

Predicting from Output The ubiquitous way to provide feedback is via unit tests: analyze the output of executing x_i. For Code.org, each execution trace results in a sequence

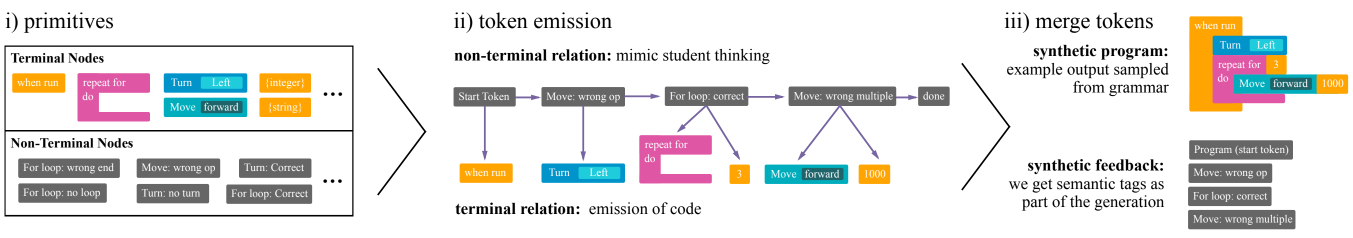


Figure 3: *Probabilistic grammar (PCFG) for synthetic block-based programs.* To generate a synthetic example, we sequentially choose a set of non-terminal nodes, each of which will emit a set of terminal nodes. The composition of these terminal nodes make up a program whereas the composition of non-terminal nodes make up the labels/feedback. The emission and transition probabilities are specified by a human designer (or learned via evolutionary strategies).

of output vectors $o_i = (o_{i,1}, o_{i,2}, ...)$ representing coordinates in 2D space where a line has been drawn. We train a recurrent neural network (RNN) to predict y_i from o_i. Unfortunately, this model requires x_i to compile.

Feedforward Neural Network To circumvent compilation, one can tackle the more difficult problem of predicting feedback from raw program strings (Piech et al. 2015b). We train a l-dimensional classifier composed of a RNN over tokens by minimizing the binary cross entropy between predictions $\hat{y}_i$ and ground truth y_i vectors.

$$\min \sum_{j=1}^{l} [-(y_{i,j} \log \hat{y}_{i,j}) + (1 - y_{i,j}) \log(1 - \hat{y}_{i,j})] \quad (1)$$

The model architecture borrows the sentence encoder (without any stochastic layers) from (Bowman et al. 2015) and concatenates a 3-layer perceptron with a softmax over l output dimensions. As we will reuse these architectures for other models, we refer to deterministic encoder as the *program network* and the latter MLP as the *feedback network*.

Trajectory Prediction No model so far uses the fact that each student submits many programs before either stopping or reaching the correct solution. In fact, the current SOTA (Wang et al. 2017a) is to associate a *trajectory* of k programs $(x_1, x_2, ..., x_k)$ with the label y_i corresponding to the last program, x_k. Then, for each program x_i, we train an embedding $e_i = f(z|x_i)$, where f is the program network. This results in a sequence of embeddings $(e_1, e_2, ..., e_k)$ for a single trajectory. We concurrently train a *second* RNN to compress the sequence to a single vector $z_i = \text{RNN}(e_1, e_2, ..., e_k)$. This is then provided as input to the feedback network. The hope is that structure induced by a trajectory implicitly provides labels that strengthen learning.

Deep Generative Model Finally, we present a new baseline that is promising in the context of limited data. If we consider programs and feedback as two modalities, one approach is to capture the joint distribution $p(x_i, y_i)$. Doing so, we can make predictions by sampling from the conditional having seen the program: $\hat{y}_i \sim p(y_i|x_i)$. To do this, we train a multimodal variational autoencoder, MVAE (Wu and Goodman 2018) with two channels. Succinctly, this generative model uses a product-of-experts inference network where the joint distribution factorizes into a product of distributions defined by two modalities: $q(z|x, y) = q(z|x)q(z|y)$ where x and y are two observed modalities and z is a latent variable. We optimize the multimodal evidence lower bound (Wu and Goodman 2018; Vedantam et al. 2017), which is a sum of three lower bounds:

$$\begin{aligned}
&\mathop{\mathbb{E}}_{q_{\phi_h}(z|x,y)} \Big[\sum_{h \in \{x,y\}} \lambda_h \log p_{\theta_h}(h|z)\Big] - \beta \text{KL}[q_{\phi_h}(z|x,y), p(z)] \\
&+ \mathop{\mathbb{E}}_{q_{\phi_x}(z|x)} [\log p_{\theta_x}(x|z)] - \beta \text{KL}[q_{\phi_x}(z|x), p(z)] \\
&+ \mathop{\mathbb{E}}_{q_{\phi_y}(z|y)} [\log p_{\theta_y} y|z)] - \beta \text{KL}[q_{\phi_y}(z|y), p(z)] \qquad (2)
\end{aligned}$$

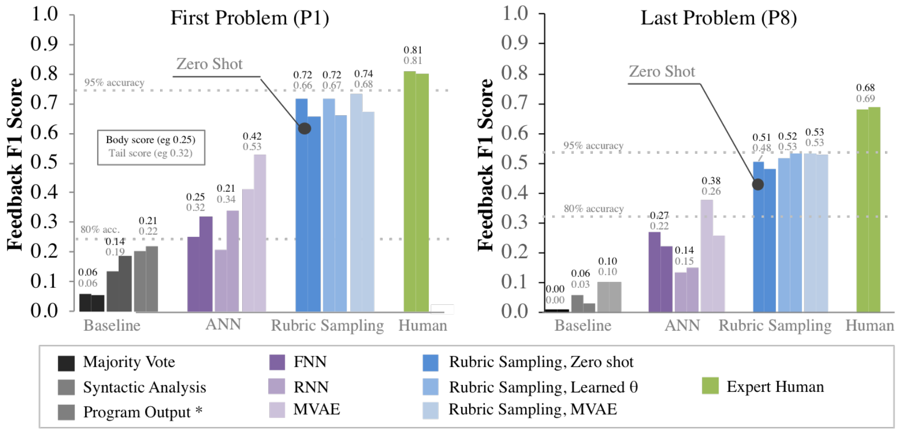
To parameterize q_{ϕ_x} and p_{θ_x}, we use architectures from (Bowman et al. 2015). For q_{ϕ_y} and p_{θ_y}, we use 3-layer MLPs[3]. To the best of our knowledge, this is the first application of a deep generative model to the feedback challenge.

Rubric Sampling

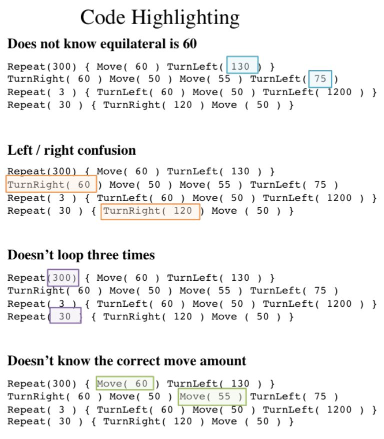
So far the models have been severely constrained by the number of labels. If we had a more efficient labeling strategy, we could better train these deep models to their full potential. For instance, imagine instead of focusing on individual programs, we ask experts to describe a student's thought process, enumerating strategies to get to a right or wrong answer. Given a detailed enough description, we can use it to label indefinitely. Granted, these labels will be noisy but the quantity should make up for any uncertainty. In fact, we can formalize such a "description" as a *context-free grammar.*

A context-free grammar (CFG) is composed of a set of acyclic production rules describing a space of output strings. As its name suggests, each rule is applied regardless of context (meaning no conditional arguments). Formally, a production rule is made up of *non-terminal* and *terminal* symbols. Non-terminal symbols are hidden and either produce another non-terminal symbol or a terminal one. Terminal symbols are made up of tokens that appear in the final output string. For instance, consider the following CFG: S $\rightarrow$ AA; A $\rightarrow$ α; A $\rightarrow$ β. S and A are non-terminal symbols whereas α and β are terminal. It is easy to see that this

[3] $q_{\phi_x}(z|x)$ is composed of the program network and a stochastic layer; $p_{\theta_y}(y|z)$ is equivalent to the feedback network.



(a)


Code Highlighting

Does not know equilateral is 60

```
Repeat(300) { Move( 60 ) TurnLeft( 130 ) }
TurnRight( 60 ) Move( 50 ) Move( 55 ) TurnLeft( 75 )
Repeat( 3 ) { TurnLeft( 60 ) Move( 50 ) TurnLeft( 1200 ) }
Repeat( 30 ) { TurnRight( 120 ) Move ( 50 ) }
```

Left / right confusion

```
Repeat(300) { Move( 60 ) TurnLeft( 130 ) }
TurnRight( 60 ) Move( 50 ) Move( 55 ) TurnLeft( 75 )
Repeat( 3 ) { TurnLeft( 60 ) Move( 50 ) TurnLeft( 1200 ) }
Repeat( 30 ) { TurnRight( 120 ) Move ( 50 ) }
```

Doesn't loop three times

```
Repeat(300) { Move( 60 ) TurnLeft( 130 ) }
TurnRight( 60 ) Move( 50 ) Move( 55 ) TurnLeft( 75 )
Repeat( 3 ) { TurnLeft( 60 ) Move( 50 ) TurnLeft( 1200 ) }
Repeat( 30 ) { TurnRight( 120 ) Move ( 50 ) }
```

Doesn't know the correct move amount

```
Repeat(300) { Move( 60 ) TurnLeft( 130 ) }
TurnRight( 60 ) Move( 50 ) Move( 55 ) TurnLeft( 75 )
Repeat( 3 ) { TurnLeft( 60 ) Move( 50 ) TurnLeft( 1200 ) }
Repeat( 30 ) { TurnRight( 120 ) Move ( 50 ) }
```

(b)

Figure 4: (a) The F1 scores for P1 and P8. We plot two bars for each model representing the F1 score on the body (left) and on the tail (right). Rubric sampling models perform far better than baselines and grow close to human-level. The "Zero Shot" marking refers to rubric sampling without fine-tuning. (b) Highlighting sub-programs conditioned on 4 feedback labels. The MVAE contains a modality for highlighting masks generated using the rubric. Imagine programming education where sections of a student's code can be highlighted along with helpful diagnostics.

CFG can only generate one of $\{\alpha\beta, \beta\alpha, \alpha\alpha, \beta\beta\}$. A *probabilistic* context-free grammar (PCFG) is a CFG parameterized by a vector θ where each production rule has a probability θ_i attached. For example, we can make our CFG from above probabilistic: S $\xrightarrow{1.0}$ AA; A $\xrightarrow{0.9}$ α; A $\xrightarrow{0.1}$ β. Now, the space of possible outputs has not changed but for instance, $\alpha\beta$ will be more probable than $\beta\alpha$.

For the feedback challenge, the non-terminal symbols are labels, y_i and the terminal symbols are programs, x_i. For example, a possible production rule might be:

$$\text{Correctly identified 45 degree angle} \xrightarrow{1.0} Turn(45)$$

With a PCFG, we can generate an infinite amount of synthetically labeled programs, $D_{\text{syn}} = \{x_i, y_i\}$, and use D_{syn} to train data-hungry models. We refer to this as *rubric sampling*. In practice, we sample a million synthetic examples. When training supervised networks, we only include unique examples to avoid prioritizing programs in the Zipf head.

Creating rubrics is surprisingly easy. For a novice (undergraduate) and an expert (professor), making a PCFG took 19.4 minutes. To make the process even easier, we developed a simple meta language for representing a grammar.[4]

Further Learning from Unlabeled Programs

As students use the platform, unlabeled submissions accumulate over time. We refer to the dataset as $D_{\text{unlabeled}}$.

Evolutionary Strategies We can use unlabeled data in rubric sampling to automatically learn θ. This means alleviating some of the burden for a human-in-the-loop, since choosing θ is often more difficult than designing the grammar itself. Intuitively, it is easier to reason about what mistakes a student can make than how often a student will make each mistake. But since a PCFG is discrete, we cannot differentiate. However, we can hope to approximate local gradients by sampling θ values within some ϵ-neighborhood and computing finite differences along these random directions (Salimans et al. 2017). If we repeatedly take a linear combination of the "best" samples as measured by a *fitness* function, then over many iterations, we expect the PCFG to improve. The challenge is in picking the fitness function.

A good choice is to pick θ whose generated distribution, D_{syn} is "closest" to $D_{\text{unlabeled}}$[5]. As both are Zipf-ian, we can properly measure "closeness" using a rank order metric (Havlin 1995), as rank is independent of frequency.

Rubric Sampling with MVAE Another way to service unlabeled data is to train with it: one of the features of the MVAE is that it can handle missing modalities. We can fit the MVAE with *two* data sources: D_{syn} and $D_{\text{unlabeled}}$.

In the case of missing labels, Equation 2 decomposes into the (unimodal) ELBO (Wu and Goodman 2018):

$$\mathop{\mathbb{E}}_{q_{\phi_x}(z|x)}[\log p_{\theta_x}(x|z)] - \beta \text{KL}[q_{\phi_x}(z|x), p(z)] \quad (3)$$

Thus, the MVAE is shown both a synthetic minibatch, $(x_i, y_i) \sim D_{\text{syn}}$, which is used to compute the multimodal elbo, and an unlabeled minibatch, $(x_i) \sim D_{\text{unlabeled}}$, which computes Equation 3. We can jointly optimize the two losses by summing the gradients prior to taking an optimization step. Intuitively, this interpolates between $D_{\text{unlabeled}}$ and D_{syn}, no longer completely relying on the PCFG. One can also interpret this as a soft-version of structure learning since using $D_{\text{unlabeled}}$ is somewhat akin to "editing" the PCFG.

[4]A Pytorch implementation, rubric grammars, along with data can be found at https://github.com/mhw32/rubric-sampling-public.

[5]In tuning θ, we consider all programs, not just the unique set (as doing so would not result in a sensible density estimator).

$$\begin{tikzcd}
D_{\text{unlabeled}} \arrow[d, "\text{log-zipf}"'] & D_{\text{generated}} \\
\tilde{D}_{\text{unlabeled}} \arrow[r, "\text{MVAE}"] & \tilde{D}_{\text{generated}} \arrow[u, "\text{exp-zipf}"']
\end{tikzcd}$$

Figure 5: *log-Zipf transformation*. Applying a log to frequencies of $x_i \sim D$ is like "tempering". This helps mitigate the effect of a few elements dominating the distribution.

Model	Amount of Correct Feedback
Predicting from output	1,483,157 (86.0%)
Rubric sampling with MVAE	**1,610,020** (93.7%)
Expert human	1,658,162 (96.2%)

Table 1: *Amount of correct feedback over the curriculum.* We ignore programs in the head of the Zipf as those can be manually labeled. With the best model, we could have provided 126,000 additional points of feedback.

Log-Zipf Transformation Capturing a Zipf is hard for a generative model since it is very easy to memorize the top 5 programs and very hard to capture the tail. To make it easier, we apply a *log* transformation,[6] e.g. if a program appears 10 times in D, it only appears once in the transformed dataset, $\tilde{D}$. Then, when generating with the MVAE, we invert the log by exponentiating the frequency of each unique program (exp-Zipf). Intuitively, log-Zipf is similar to "tempering" a distribution as it reduces any extreme peaks and troughs.

Results

Recreation of Human Labels

Figure 4 reports a set of F1 scores, including human-level performance estimated from annotations. Each model is given two bar plots, one for programs in the body (left) and one in the tail (right). First, we see that baselines have lower F1 scores compared to models that take advantage of synthetic data. That being said, the new baseline MVAE we introduced already performs far better than the previous SOTA. In P1, using rubric sampling increases the F1 score by 0.31 in the body and 0.13 in the tail (we find similar gains in P8). These promising results imply that the grammar indeed is effective. We also find that combining the MVAE with rubric sampling boosts the F1 by an additional 0.2, reaching 94% accuracy in P1 and 95% in P8. With these scores, we are reasonably confident that for a new student, despite the likelihood that he/she will submit a never-before-seen program, we will provide good feedback.

To put these results in terms of impact, Table 1 estimates the number of correct feedback we could have given to students in P1 to P8 based on projections from the annotated set. Over the full curriculum, our best model would have provided the correct feedback to an expected 126,000 additional programs compared to what Code.org currently uses, potentially helping thousands more students.

Tracing Knowledge Across Curricula

If we had a scalable way to provide feedback, what impact could we make to code education? We can begin to measure this impact using the full Code.org curriculum. Having demonstrated good performance on P1 and P8, we can confidently apply rubric sampling to P2 through P7. This is quite powerful as it allows us to estimate student understanding over a curricula. Critically, we can gauge the performance of both individual students and the student body as a whole. These sort of queries are valuable to teachers to be able to (1) measure a student's progress scalably and (2) judge how useful assignments and lessons have been.

In Figure 6, we analyze the average student's level of understanding over the 8 problems for two main concepts: loops and geometry (e.g. shapes, angles, movement). For each submission in a student's trajectory, we classify it as having either 1) no errors, 2) more loop errors, or 3) more geometry errors[7]. The figure shows the distribution of students in each of the three categories from the first 10 submissions. From looking at behavior within a problem and between problems, we can make the following inferences:

1. **Most students are successfully completing each problem.** In other words, the blue area is increasing over time. Equivalently, the number of students still struggling by the 10th submission approaches a small number.
2. **The difficulty of problems is not uniform.** P6 is much more difficult than the others as the fraction of students with correct solutions is roughly constant. In contrast, P1, P4, and P5 are easier, where students quickly cease to make mistakes. As a teacher, one could use this information to improve the educational content and better hone in on areas where more students struggle.
3. **Students are learning geometry better than looping.** The rate that the pink area approaches zero is consistently faster than that of the orange area. By P8, students are barely struggling with geometry but a fair proportion still find looping difficult. As the curriculum was intended to teach nested looping, one interpretation is that the drawing aspect was somewhat distracting.

Fine-grain Feedback: Code Highlighting

With most online programming education, the form of feedback is limited to pop-up tips or compiler messages. But, with generative models we can provide more fine-grain feedback through *highlighting* subsets of the program responsible for the model predicting each feedback label.

[6]We preserve examples that appear only once in D to $\tilde{D}$ i.e. $\tilde{x} = \min(\log(x), 1)$ where $x \in D$ and $\tilde{x} \in \tilde{D}$.

[7]We do so by comparing the summed predicted probabilities for all labels related to loops, $\hat{y}_{i,\text{loop}} = \sum_{j \in \text{loop}} \hat{y}_{i,j}$ and labels related to geometry, $\hat{y}_{i,\text{geom}} = \sum_{j \in \text{geom}} \hat{y}_{i,j}$. If both $\hat{y}_{i,\text{loop}} < 1$ and $\hat{y}_{i,\text{geom}} < 1$, then we classify this program as "no errors". Otherwise, we classify based on which quantity is larger.

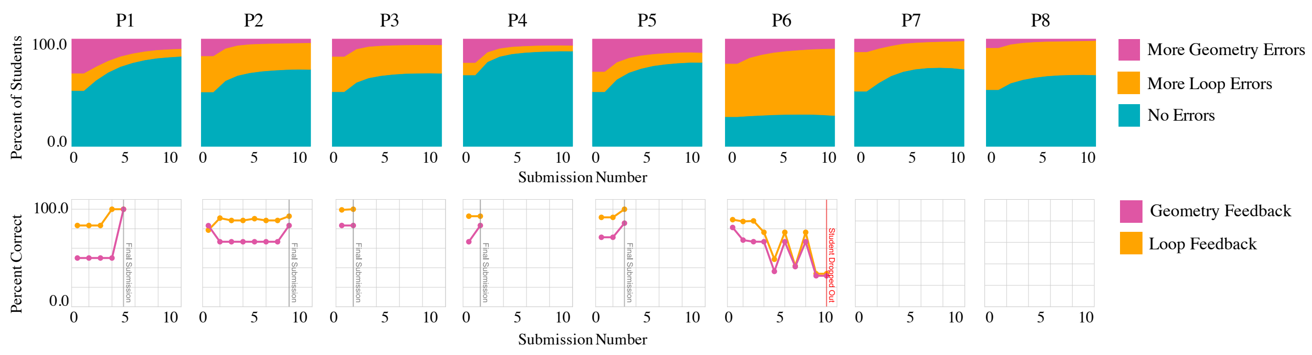
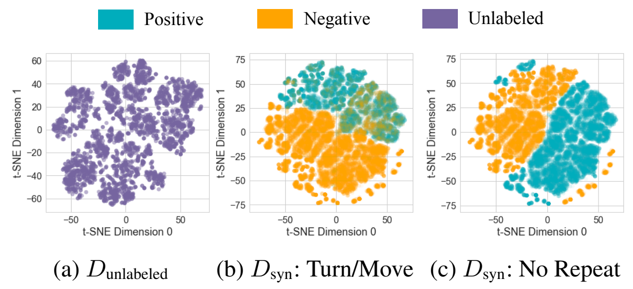


Figure 6: *Student understanding of loops and geometry across curricula*: (top row) We plot the percentage of students who are either doing perfect (cyan), struggling more with looping concepts (orange), or struggling more with geometry concepts (pink). In general the percentage of students with no errors increases over time as more students finish the problem. Additionally, we can extrapolate that students are more effectively learning geometry than looping, as the area covered by pink decreases faster and more consistently than the area covered by orange. We can also see that P6 is somewhat of an outlier, being much more difficult for students than any other problem. (bottom row) In addition to aggregate statistics, we can also track learning for individual students. We can infer that this particular student tries several attempts with P6 before dropping out.

First, if we consider a PCFG, the task of highlighting a program x_i is equivalent to finding the most likely parsing in a probabilistic tree that would generate x_i. In practice, we use the A* algorithm for fast Viterbi parsing (Klein and Manning 2003). Given the most likely parsing, we can follow the trajectory from root to leaf and record which subprograms are generated by non-terminal nodes. This has one major limitation: only programs within the support of the PCFG can be highlighted. To bypass this, we can curate a synthetic dataset with each program having a segmentation mask denoting which tokens to highlight. If we treat the mask as an additional modality, we can then learn the joint distribution over programs, labels, and masks. See (Wu and Goodman 2018) for details in defining a VAE with three modalities. In Figure 4b, we randomly sample 4 programs and show segmentation masks. The running time to compute a mask is negligible, meaning that this can be used for providing on-the-fly feedback to students. Moreover, highlighting provides a notion of interpretability (which is extremely important if we are working with students), much like Grad-Cam (Selvaraju et al. 2017) did for computer vision.

Clustering Students by Level of Understanding

With any latent generative model, the rich latent space provides a vector representation for unstructured data. Using the MVAE, we first sample $z_i \sim q(z_i|x_i)$ for all $x_i \in D_{\text{test}}$; we then train use t-SNE (Maaten and Hinton 2008) to reduce to two dimensions. In Figure 7b and c, we color each embedded program from D_{syn} by whether the true label is positive or negative. We find that the space is partitioned to group programs with similar feedback together. In Figure 7a, we see that $D_{\text{unlabeled}}$ is also organized into disjoint clusters. This implies that even with no data about a new student we can draw inferences from their neighbors in latent space.



(a) $D_{\text{unlabeled}}$ (b) D_{syn}: Turn/Move (c) D_{syn}: No Repeat

Figure 7: *Clustering students.* Using the inference network in the MVAE, we can embed a program in 2D. In $D_{\text{unlabeled}}$ (a), we see a handful of distinct clusters. In D_{syn} (b,c), we find meaningful clusters that are segmented by labels.

Discussion

We get closer to human level performance than previous SOTA. Any of the rubric sampling models beat the SOTA by at least 0.46 in P1 and 0.24 in P8, nearly tripling the F1 score in P1 and doubling in P8. In both exercises, our best model is just under 95% accuracy, which is encouraging for this to be implemented in the real world.

We can effectively track student growth over time. With a high performing model, we can analyze students over time. For Code.org, we were able to (1) gauge what individual students struggle with, (2) gauge what a collective of students struggle with, (3) identify the effectiveness of a curriculum, and (4) identify the difficulty of problems.

Making a rubric is neither difficult nor costly. It is not the case that only experts can make a good rubric. We also asked an undergraduate (novice) to make a rubric for P1 and found that while an experts' rubric averaged 0.69 ± 0.03 in F1 score, the novice's averaged 0.63 ± 0.01, both of which are much higher than baselines ($0.29\pm, 0.03$). Furthermore, we measured that it took a group of teaching assistants 24.9 hours to label 800 unique programs. In comparison, it took a novice an average of 19.4 minutes to make a rubric.

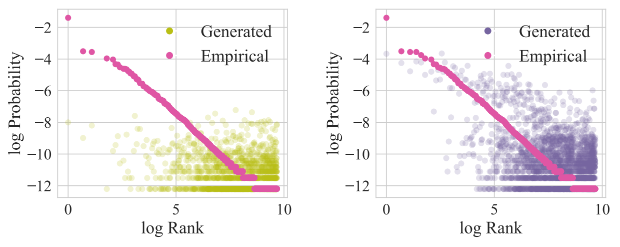
We provide feedback to programs that do not compile. Rubic sampling and MVAE make no assumptions on program behavior, working out-of-the-box from the 1st student.

We do not need to handpick θ when designing a rubric. In Figure 1, the PCFG uses hand-picked θ. However, one can argue that it is difficult to know how often students make mistakes and yet, the choice of θ is important: performance drops if we randomly choose. For example, in P1, using hand-picked θ has a 0.26 ± 0.002 increase over random θ in F1-score. In the absence of an expert, we can use evolutionary strategies to find a good minima starting from a random initialization. Over 3 runs, we found that learning θ reduces the difference to 0.007 ± 0.005 in P1, even beating expert parameters by 0.005 ± 0.009 in P8. With a large dataset, we only have to define the structure, not the probabilities.

The log-Zipf transform ensures that learning does not collapse to memorizing the most frequent programs. If we were to use the raw statistics of the student data, the MVAE would minimize loss by memorizing the most frequent programs. As evidence of this, when sampling from its generative model, we found that the MVAE only generated programs in the top 25 programs by frequency (even with 1 million samples). We propose the log-Zipf transformation as a general tool for parameter learning with Zipf-distributed data. Critically, the transform downweights the head and upweights the tail in an invertible fashion.

The MVAE interpolates between synthetic and empirical data. Unlike the PCFG, we can train the MVAE with multiple data sources. Effectively, the programs we show to the model is drawn from an interpolation between the synthetic distribution defined by rubric sampling and the true (unlabeled) data distribution. Intuitively, the increase in performance from the MVAE can be attributed to better capturing the true density of student programs (see Figure 9).

We can generate and open-source a large dataset of student programs. Datasets with student data are difficult to release to the open community. But large public datasets have been a dominant force in pushing the boundaries of research. Luckily, with generative models, we can curate our own "Imagenet" for education. But, we want to ensure that our dataset matches D in distribution. Intuitively, it is impossible that a PCFG can capture D since that would require production rules that span the entire tail of the Zipf. In fact, as shown in Figure 8, the PCFG is not that faithful. One remedy is to use the MVAE trained with $D_{\text{unlabeled}}$ as that is interpolating between distributions. Figure 8, confirms that the MVAE matches the shape of D much better.

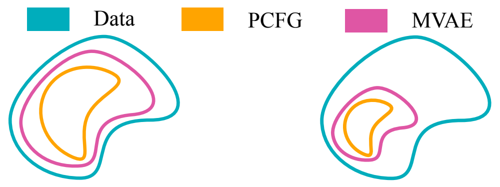


(a) P1 (PCFG) (b) P1 (MVAE)

Figure 8: We compare D_{syn}^{MVAE} and D_{syn}^{Rubric} to $D_{\text{unlabeled}}$. Programs from the MVAE cover $D_{\text{unlabeled}}$ much better than relying on synthetic data alone (PCFG).

Limitations and Future Work

The effectiveness of rubric sampling is largely determined by the complexity of the programming task. Although a block-based language like in Code.org already has an infinite number of possible programs, the set of probable student programs is much smaller than in a typical university level coding assignment. An important distinction is the introduction of variable and function names. As suggested by Figure 9, the version of rubric sampling used in this paper may have difficulty scaling to harder problems. As PCFGs are context-free, making a sufficiently expressive grammar for complex problems requires extremely large graphs with repetitive subgraphs. Future work could look to generalizing rubric sampling to arbitrary graphs with conditional branching, or investigate structural learning to improve graphs to cover more of the empirical data.



(a) Code.org (b) University classroom

Figure 9: The space of likely student programs in a block-based language can be covered by a PCFG. But in a higher-order language like Python, the space is much larger.

Conclusion

We introduce the zero shot feedback challenge. On a widely used platform, we show rubric sampling to far surpass the SOTA. We combine this with a generative model to cluster students, highlight code, and incorporate historical data. This approach can scale feedback for real world use.

Acknowledgments

The authors thank Lisa Wang, Baker Franke and the Code.org team for support, and guidance. MW is supported by NSF GRFP. NDG is supported by DARPA PPAML under FA8750-14-2-0006. We thank Connie Liu for edits.

References

Bowen, W. G. 2012. The cost disease in higher education: is technology the answer? *The Tanner Lectures Stanford University*.

Bowman, S. R.; Vilnis, L.; Vinyals, O.; Dai, A. M.; Jozefowicz, R.; and Bengio, S. 2015. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*.

Brown, J. S., and VanLehn, K. 1980. Repair theory: A generative theory of bugs in procedural skills. *Cognitive science* 4(4):379–426.

Bulgarov, F. A., and Nielsen, R. 2018. Proposition entailment in educational applications using deep neural networks. In *AAAI*.

Feldman, M. Q.; Cho, J. Y.; Ong, M.; Gulwani, S.; Popović, Z.; and Andersen, E. 2018. Automatic diagnosis of students' misconceptions in k-8 mathematics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 264. ACM.

Geigle, C.; Zhai, C.; and Ferguson, D. C. 2016. An exploration of automated grading of complex assignments. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*, 351–360. ACM.

Havlin, S. 1995. The distance between zipf plots. *Physica A: Statistical Mechanics and its Applications* 216(1-2):148–150.

Klein, D., and Manning, C. D. 2003. A parsing: fast exact viterbi parse selection. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, 40–47. Association for Computational Linguistics.

Koedinger, K. R.; Matsuda, N.; MacLellan, C. J.; and McLaughlin, E. A. 2015. Methods for evaluating simulated learners: Examples from simstudent. In *AIED Workshops*.

Lake, B. M.; Salakhutdinov, R.; and Tenenbaum, J. B. 2015. Human-level concept learning through probabilistic program induction. *Science* 350(6266):1332–1338.

Lee, C.-W.; Fang, W.; Yeh, C.-K.; and Wang, Y.-C. F. 2017. Multi-label zero-shot learning with structured knowledge graphs. *arXiv preprint arXiv:1711.06526*.

Maaten, L. v. d., and Hinton, G. 2008. Visualizing data using t-sne. *Journal of machine learning research* 9(Nov):2579–2605.

Nguyen, A.; Piech, C.; Huang, J.; and Guibas, L. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*, 491–502. ACM.

O'Rourke, E.; Ballweber, C.; and Popovií, Z. 2014. Hint systems may negatively impact performance in educational games. In *Proceedings of the first ACM conference on Learning@ scale conference*, 51–60. ACM.

Paaßen, B.; Hammer, B.; Price, T. W.; Barnes, T.; Gross, S.; and Pinkwart, N. 2017. The continuous hint factory-providing hints in vast and sparsely populated edit distance spaces. *arXiv preprint arXiv:1708.06564*.

Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; and Guibas, L. 2015a. Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning*.

Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; and Guibas, L. 2015b. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969*.

Piech, C.; Sahami, M.; Huang, J.; and Guibas, L. 2015c. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, 195–204. ACM.

Price, T. W., and Barnes, T. 2017. Position paper: Block-based programming should offer intelligent support for learners. In *Blocks and Beyond Workshop (B&B), 2017 IEEE*, 65–68. IEEE.

Salimans, T.; Ho, J.; Chen, X.; Sidor, S.; and Sutskever, I. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Selvaraju, R. R.; Cogswell, M.; Das, A.; Vedantam, R.; Parikh, D.; and Batra, D. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *ICCV*, 618–626.

Vedantam, R.; Fischer, I.; Huang, J.; and Murphy, K. 2017. Generative models of visually grounded imagination. *arXiv preprint arXiv:1705.10762*.

Verma, V. K.; Arora, G.; Mishra, A.; and Rai, P. 2018. Generalized zero-shot learning via synthesized examples. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Wang, L.; Sy, A.; Liu, L.; and Piech, C. 2017a. Learning to represent student knowledge on programming exercises using deep learning. In *Proceedings of the 10th International Conference on Educational Data Mining; Wuhan, China*, 324–329.

Wang, W.; Pu, Y.; Verma, V. K.; Fan, K.; Zhang, Y.; Chen, C.; Rai, P.; and Carin, L. 2017b. Zero-shot learning via class-conditioned deep generative models. *arXiv preprint arXiv:1711.05820*.

Wu, M., and Goodman, N. 2018. Multimodal generative models for scalable weakly-supervised learning. *arXiv preprint arXiv:1802.05335*.

Xian, Y.; Lorenz, T.; Schiele, B.; and Akata, Z. 2018. Feature generating networks for zero-shot learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.