Learning Program Embeddings to Propagate Feedback on Student Code

Chris Piech PIECH@CS.STANFORD.EDU
Jonathan Huang JONATHANHUANG@GOOGLE.COM
Andy Nguyen TANONEV@CS.STANFORD.EDU
Mike Phulsuksombati MIKEP15@CS.STANFORD.EDU
Mehran Sahami SAHAMI@CS.STANFORD.EDU
Leonidas Guibas GUIBAS@CS.STANFORD.EDU

Abstract

Providing feedback, both assessing final work and giving hints to stuck students, is difficult for open-ended assignments in massive online classes which can range from thousands to millions of students. We introduce a neural network method to encode programs as a linear mapping from an embedded precondition space to an embedded postcondition space and propose an algorithm for feedback at scale using these linear maps as features. We apply our algorithm to assessments from the Code.org Hour of Code and Stanford University's CS1 course, where we propagate human comments on student assignments to orders of magnitude more submissions.

1. Introduction

Online computer science courses can be massive with numbers ranging from thousands to even millions of students. Though technology has increased our ability to provide content to students at scale, assessing and providing feedback (both for final work and partial solutions) remains difficult. Currently, giving personalized feedback, a staple of quality education, is costly for small, in-person classrooms and prohibitively expensive for massive classes. Autonomously providing feedback is therefore a central challenge for at scale computer science education.

It can be difficult to apply machine learning directly to data in the form of programs. Program representations such as the *Abstract Syntax Tree (AST)* are not directly conducive to standard statistical methods and the edit distance metric between such trees are not discriminative enough to be used to share feedback accurately since programs with similar ASTs can behave quite differently and require different comments. Moreover, though unit tests are a useful way to test if final solutions are correct they are not well suited for giving help to students with an intermediate solution and they are not able to give feedback on stylistic elements.

There are two major goals of our paper. The first is to automatically learn a feature embedding of student submitted programs that captures functional and stylistic elements and can be easily used in typical supervised machine learning systems. The second is to use these features to learn how to give automatic feedback to students. Inspired by recent successes of deep learning for learning features in other domains like NLP and vision, we formulate a novel neural network architecture that allows us to jointly optimize an embedding of programs and memory-state in a feature space. See Figure 1 for an example program and corresponding matrix embeddings.

To gather data, we exploit the fact that programs are executable — that we can evaluate any piece of code on an arbitrary input (i.e., the precondition), and observe the state after, (the postcondition). For a program and its constituent parts we can thus collect arbitrarily many such precondition/postcondition mappings. This data provides the training set from which we can learn a shared representation for programs. To evaluate our program embeddings we test our ability to amplify teacher feedback. We use real student data from the Code.org Hour of Code which has been attempted by over 27 million learners making it, to the best of our knowledge, the largest online course to-date. We then show how the same approach can be used for submissions in Stanford University's Programming Methodologies course which has thousands of students and assignments that are substantially more complex. The programs we analyze are written in a Turing-complete language but do not allow for user-defined variables.

Our main contributions are as follows. First, we present a method for computing features of code that capture both functional and stylistic elements. Our model works by simultaneously embedding precondition and postcondition spaces of a set of programs into a feature space where programs can be viewed as linear maps on this space. Second,

Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37.

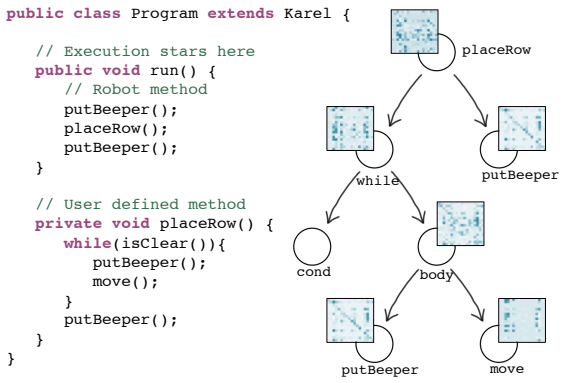

Figure 1. We learn matrices which capture functionality. Left: a student partial solution. Right: learned matrices for the syntax trees rooted at each node of placeRow.

we show how our code features can be useful for automatically propagating instructor feedback to students in a massive course. Finally, we demonstrate the effectiveness of our methods on large scale datasets. Learning embeddings of programs is fertile ground for machine learning research and if such embeddings can be useful for the propagation of teacher feedback this line of investigation will have a sizable impact on the future of computer science education.

2. Related Work

The advent of massive online computer science courses has made the problem of automated reasoning with large code collections an important problem. There have been a number of recent papers (Huang et al., 2013; Basu et al., 2013; Nguyen et al., 2014; Brooks et al., 2014; Lan et al., 2015; Piech et al., 2015) on using large homework submission datasets to improve student feedback. The volume of work speaks to the importance of this problem. Despite the research efforts, however, providing quality feedback at scale remains an open problem.

A central challenge that a number of papers address is that of measuring similarity between source code. Some authors have done this without an explicit featurization of the code — for example, the *AST edit distance* has been a popular choice (Huang et al., 2013; Rogers et al., 2014). (Mokbel et al., 2013) explicitly hand engineered a small collection of features on ASTs that are meant to be domain-independent.

To incorporate functionality, (Nguyen et al., 2014) proposed a method that discovers program modifications that do not appear to change the semantic meaning of code. The embedded representations of programs used in this paper also capture semantic similarities and are more amenable to prediction tasks such as propagating feedback. We ran feedback propagation on student data using methods from Nguyen et al and observe that embeddings enabled notable improvement (see section 6.3).

Embedding programs has many crossovers with embedding natural language artifacts, given the similarity between the AST representation and parse trees. Our models are related to recent work from the NLP and deep learning communities on recursive neural networks, particularly for modeling semantics in sentences or symbolic expressions (Socher et al., 2013; 2011; Zaremba et al., 2014; Bowman, 2013).

Finally, representing a potentially complicated function (which in our case is a program) as a linear operator acting on a nonlinear feature space has also been explored in different communities. The computer graphics community have represented pairings of nonlinear geometric shapes as linear maps between shape features, called *functional maps* (Ovsjanikov et al., 2012; 2013). From the kernel methods literature, there has also been recent work on representations of conditional probability distributions as operators on a Hilbert space (Song et al., 2013; 2009). From this point of view, our work is novel in that it focuses on the joint optimization of feature embeddings together with a collection of maps so that the maps simultaneously "look linear" with respect to the feature space.

3. Embedding Hoare Triples

Our core problem is to represent a program as a point in a fixed-dimension real-valued space that can then be used directly as input for typical supervised learning algorithms.

While there are many dimensions that "characterize" a program including aspects such as style or time/space complexity, we begin by first focussing on capturing the most basic aspect of a program — its function. While capturing the function of the program ignores aspects that can be useful in application (such as giving stylistic feedback in CS education), we discuss in later sections how elements of style can be recaptured by modeling the function of subprograms that correspond to each subtree of an AST. Given a program A (where we consider a program to generally be any executable code whether a full submission or a subtree of a submission), and a precondition P, we thus would like to learn features of A that are useful for predicting the outcome of running A when P holds. In other words, we want to predict a postcondition Q out of some space of possible postconditions. Without loss of generality we let P and Q be real-valued vectors encapsulating the "state" of the program (i.e., the values of all program variables) at a particular time. For example, in a grid world, this vector would contain the location of the agent, the direction the agent is facing, the status of the board and whether the program has crashed. Figure 2 visualizes two preconditions, and the corresponding postconditions for a simple program.

We propose to learn program features using a training set of

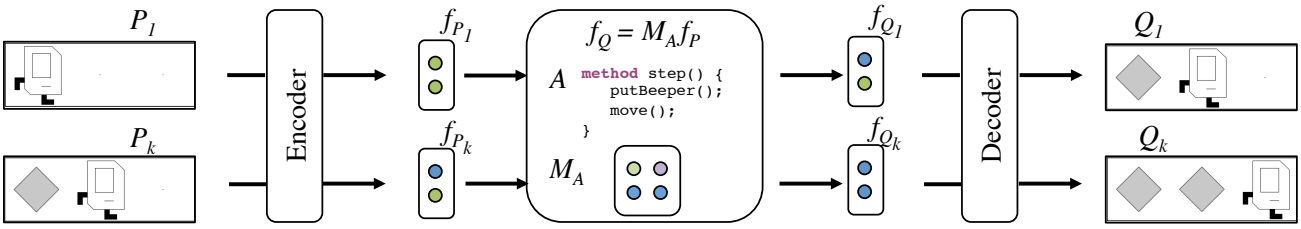

Figure 2. Diagram of the model for a program A implementing a simple "step forward" behavior in a small 1-dimensional gridworld. Two of the k Hoare triples that correspond with A are shown. Typical worlds are larger and programs are more complex.

(P, A, Q)-triples — so-called *Hoare triples* (Hoare, 1969) obtained via historical runs of a collection of programs on a collection of preconditions. We discuss the process by which such a dataset can be obtained in Section 5. The main approach that we espouse in this paper is to simultaneously find an embedding of states and programs into feature space where pre and postconditions are points in this space and programs are mappings between them.

The simple way that we propose to relate preconditions to postconditions is through a linear transformation. Explicitly, given a (P, A, Q)-triple, if f_P and f_Q are m-dimensional nonlinear feature representations of the pre and postconditions P and Q, respectively, then we relate the embeddings via the equation

$$f_Q = M_A \cdot f_P. \tag{1}$$

We then take the $m \times m$ matrix of coefficients M_A as our feature representation of the program A and refer to it as the *program embedding matrix*. We will want to learn the mapping into feature space f as well as the linear map M_A such that this equality holds for all observed triples and can generalize to predict postcondition Q given P and A.

At first blush, this linear relationship may seem too limiting as programs are not linear nor continuous in general. By learning a nonlinear embedding function f for the pre and postcondition spaces, however, we can capture a rich family of nonlinear relationships much in the same way that kernel methods allow for nonlinear decision boundaries.

As described so far, there remain a number of modeling choices to be made. In the following, we elaborate further on how we model the feature embeddings f_P, and f_Q of the pre and postconditions, and how to model the program embedding matrix M_A.

3.1. Neural network encoding and decoding of states

We assume that preconditions have some base encoding as a d-dimensional vector, which we refer to as P. For example, in image processing courses, the state space could simply be the pixel encoding of an image, whereas in the discrete gridworld-type programming problems that we use in our experiments, we might choose to encode the (x, y)-coordinate and discretized heading of a robot using a concatenation of one-hot encodings. Similarly, we assume that there is a base encoding Q of the postcondition.

We will focus our exposition in the remainder of our paper on the case where the precondition space and postcondition spaces share a common base encoding. This is particularly appropriate to our experimental setting in which both the preconditions and postconditions are representations of a gridworld. In this case, we can use the same decoder parameters (i.e., W^{dec} and b^{dec}) to decode both from precondition space and postcondition space — a fact that we will exploit in the following section.

Inspired by nonlinear autoencoders, we parameterize a mapping, called the *encoder* from precondition P to a nonlinear m-dimensional feature representation f_P. As with traditional autoencoders, we use an affine mapping composed with an elementwise nonlinearity:

$$f_P = \phi(W^{enc} \cdot P + b^{enc}), \tag{2}$$

where $W^{enc} \in \mathbb{R}^{m \times d}$, $b^{enc} \in \mathbb{R}^m$, and ϕ is an elementwise nonlinear function (such as $\tanh$). At this point, we can use the representation f_P to decode or reconstruct the original precondition as a traditional autoencoder would do using:

$$\hat{P} = \psi(W^{dec} \cdot f_P + b^{dec}), \tag{3}$$

where $W^{dec} \in \mathbb{R}^{d \times m}$, $b^{dec} \in \mathbb{R}^d$, and ψ is some (potentially different) elementwise nonlinear function. Moreover, we can push the precondition embedding f_P through Equation 1, and decode the postcondition embedding $f_Q = M_A \cdot f_P$. This mapping which reconstructs the postcondition Q, the *decoder*, takes the form:

$$\hat{Q} = \psi(W^{dec} \cdot f_Q + b^{dec}), \tag{4}$$

$$= \psi(W^{dec} \cdot M_A \cdot f_P + b^{dec}). \tag{5}$$

Figure 2 diagrams our model on a simple program. Note that it is possible to swap in alternative feature representations. We have experimented with using a deep, stacked autoencoder however our results have not shown these to help much in the context of our datasets.

3.2. Nonparametric model of program embedding

To encode the program embedding matrix, we propose a simple nonparametric model in which each program in the training set is associated with its own embedding matrix. Specifically, if the collection of unique programs is $\{A_1, \ldots, A_m\}$, then for each A_i, we will associate a matrix M_i. The entire parameter set for our nonparametric matrix model (henceforth abbreviated *NPM*) is thus: $\Theta = \{W^{dec}, W^{enc}, b^{enc}, b^{dec}\} \cup \{M_i : i = 1, \ldots, m\}$.

To learn the parameters, we minimize a sum of three terms: (1) a prediction loss ℓ^{pred} which quantifies how well we can predict postcondition of a program given a precondition, (2) an autoencoding loss ℓ^{auto} which quantifies how good the encoder and decoder parameters are for reconstructing given preconditions, and (3) a regularization term $\mathcal{R}$. Formally, given training triples $\{(P_i, A_i, Q_i)\}_{i=1}^n$, we can minimize the following objective function:

$$L(\Theta) = \frac{1}{n}\sum_{i=1}^{n} \ell^{pred}(Q_i, \hat{Q}_i(P_i, A_i; \Theta)) + \frac{1}{n}\sum_{i=1}^{n} \ell^{auto}(P_i, \hat{P}_i(P_i, \Theta)) + \frac{\lambda}{2}\mathcal{R}(\Theta), \tag{6}$$

where $\mathcal{R}$ is a regularization term on the parameters, and λ a regularization parameter. In our experiments, we use $\mathcal{R}$ to penalize the sum of the L_2 norms of the weight matrices (excluding the bias terms b^{enc} and b^{dec}).

Any differentiable loss can conceptually be used for ℓ^{pred} and ℓ^{auto}. For example, when the top level predictions, $\hat{P}$ or $\hat{Q}$, can be interpreted as probabilities (e.g., when ϕ is the Softmax function), we use a cross-entropy loss function.

Informally speaking, one can think of our optimization problem (Equation 6) as trying to find a good *shared* representation of the state space — shared in the sense that even though programs are clearly not linear maps over the original state space, the hope is that we can discover some nonlinear encoding of the pre and postconditions such that most programs simultaneously "look" linear in this new projected feature space. As we empirically show in Section 6, such a representation is indeed discoverable.

We run joint optimization using minibatch stochastic gradient descent without momentum, using ordinary backpropagation to calculate the gradient. We use random search (Bergstra & Bengio, 2012) to optimize over hyperparameters (e.g, regularization parameters, matrix dimensions, and minibatch size). Learning rates are set using Adagrad (Duchi et al., 2011). We seed our parameters using a "smart" initialization in which we first learn an autoencoder on the state space, and perform a vector-valued ridge regression for each unique program to extract a matrix mapping the features of the precondition to the features of the postcondition. The encoder and decoder parameters and the program matrices are then jointly optimized.

3.3. Triple Extraction

For a given program S we extract Hoare triples by executing it on an exemplar set of unit tests. These tests span a variety of reasonable starting conditions. We instrument the execution of the program such that each time a subtree $A \subset S$ is executed, we record the value, P, of all variables before execution, and the value, Q, of all variables after execution and save the triple (P, A, Q). We run all programs on unit tests, collecting triples for *all* subtrees. Doing so results in a large dataset $\{(P_i, A_i, Q_i)\}_{i=1}^n$ from which we collapse equivalent triples. In practice, some subtrees, especially the body of loops, generate a large (potentially infinite) number of triples. To prevent any subtree from having undue influence on our model we limit the number of triples for any subtree.

Collecting triples on subtrees, as opposed to just collecting triples on complete programs, is critical since it allows us to learn embeddings not just for the root of a program AST but also for the constituent parts. As a result, we retain data on how a program was implemented, and not just on its overall functionality, which is important for student feedback as we discuss in the next section. Collecting triples on subtrees also means we are able to optimize our embeddings with substantially more data.

4. Feedback Propagation

The result of jointly learning to embed states and a corpus of programs is a fixed dimensional, real-valued matrix M_A for each subtree A of any program in our corpus. These matrices can be cooperative with machine learning algorithms that can perform tasks beyond predicting what a program does. The central application in this paper is the force multiplication of teacher-provided feedback where an active learning algorithm interacts with human graders such that feedback is given to many more assignments than the grader annotates. We propose a two phase interaction. In the first phase, the algorithm selects a subset of exemplar programs for graders to apply a finite set of annotations. Then in the second phase, the algorithm uses the human provided annotations as supervised labels with which it can learn to predict feedback for unlabelled submissions. Each program is annotated with a set $H \subset L$ where L is a discrete collection of N possible annotations. The annotations are meant to cover a range of comments a grader could apply, including feedback on style, strategy and functionality. For each ungraded submission, we must then decide which of the N labels to apply. As such, we view feedback propagation as N binary classification tasks.

One way of propagating feedback would be to use the el-

ements of the embedding matrix of the root of a program as features and then train a classifier to predict appropriate feedback for a given program. However, the matrices we have learned for programs and their subtrees have been trained only to predict functionality. Consequently, any two programs that are functionally indistinguishable would be given the same instructor feedback under this approach, ignoring any strategic or stylistic differences between the programs.

4.1. Incorporating structure via recursive embedding

To recapture the elements of program structure and style that are critical for student feedback, our approach to predict feedback uses the embedding matrices learned for the NPM model, but incorporates all constituent subtrees of a given AST. Specifically, using the embedding matrices learned in the NPM model (which we henceforth denote as M_A^{NPM} for a subtree A), we now propose a new model based on recursive neural networks (called the *NPM-RNN* model) in which we parametrize a matrix M_A in this new model with an RNN whose architecture follows the abstract syntax tree (similar to the way in which RNN architectures might take the form of a parse tree in an NLP setting (Socher et al., 2013)).

In our RNN based model, a subtree of the AST rooted at node j is represented by a matrix which is computed by combining (1) representations of subtrees rooted at the children of j, and (2) the embedding matrix of the subtree rooted at node j learned via the NPM model. By incorporating the embedding matrix from the NPM model, we are able to capture the function of every subtree in the AST.

Formally, we will assume each node is associated with some *type* in set $\mathcal{T} = \{\omega^1, \omega^2, \dots\}$. Concretely, the type set might be the collection of keywords or built-in functions that can be called from a program in the dataset, e.g., $\mathcal{T} = \{\textbf{repeat}, \textbf{while}, \textbf{if}, \dots\}$. A node with type ω is assumed to have a fixed number, a_ω, of children in the AST — for example, a **repeat** node has two children, with one child holding the body of a repeat loop and the second representing the number of times the body is to be repeated.

The representation of node j with type ω is then recursively computed in the NPM-RNN model via:

$$a^{(j)} = \phi\left(\sum_{i=1}^{a_\omega} W_i^\omega \cdot a^{(c_i[j])} + b^\omega + \mu M_j^{NPM}\right), \qquad (7)$$

where: ϕ is a nonlinearity (such as $\tanh$), $c_i[j]$ indexes over the a_ω children of node j, and M_j^{NPM} is the program embedding matrix learned in the NPM model for the subtree rooted at node j. We remind the reader that the activation $a^{(j)}$ at each node is an $m \times m$ matrix. Leaf nodes of type ω are simply associated with a single parameter matrix W^ω.

In the NPM-RNN model, we have parameter matrices $W^\omega, b^\omega \in \mathbb{R}^{m \times m}$ for each possible type $\omega \in \mathcal{T}$. To train the parameters, we first use the NPM model to compute the embedding matrix M_j^{NPM} for each subtree. After fixing M_j, we optimize (as with the NPM model) with minibatch stochastic gradient descent using backpropagation through structure (Goller & Kuchler, 1996) to compute gradients. Instead of optimizing for predicting postcondition, for NPM-RNN, we optimize for each of the binary prediction tasks that are used for feedback propagation given the vector embedding at the root of a program. We used hyper-parameters learned in the RNN model optimization since feedback optimization is performed over few examples and without a holdout set.

Finally, feedback propagation has a natural active learning component: intelligently selecting submissions for human annotation can potentially save instructors significant time. We find that in practice, running k-means on the learned embeddings, and selecting the cluster centroids as the set of submissions to be annotated works well and leads to significant improvements in feedback propagation over random subset selection. Surprisingly, having humans annotate the most common programs performs worse than the alternatives, which we observe to be due to the fact that the most common submissions are all quite similar to one another.

Statistic	Ω_1	Ω_1	Ω_1
Num Students	>11 million	2,710	2,710
Unique Programs	210,918	6,674	63,820
Unique Subtrees	311,198	15,550	198,918
Unique Triples	5,334,452	476,502	4,211,150
Unique States	149	1,399	114,704
Unique Annotations	15	12	14

Table 1. Dataset summary. Programs are considered identical if they have equal ASTs. Unique states are different configurations of the gridworld which occur in student programs.

5. Datasets

We evaluate our model on three assignments from two different courses, Code.org's Hour of Code (HOC) which has submissions from over 27 million students and Stanfords Programming Methodology course, a first-term introductory programming course, which has collected submissions over many years from almost three thousand students. From these two classes, we look at three different assignments. As in many introductory programming courses, the first assignments have the students write standard programming control flow (if/else statements, loops, methods) but do not introduce user-defined variables. The programs for these assignments operate in maze worlds where an agent can move, turn, and test for conditions of its current location. In the Stanford assignments, agents can also put down and pick up beepers, making the language Turing complete.

Specifically, we study the following three problems:

Ω_1: The 18th problem in the Hour of Code (HOC). Students solve a task which requires an if/else block inside of a while loop, the most difficult concept in the Hour of Code.

Ω_2: The first assignment in Stanford's course. Students program an agent to retrieve a beeper in a fixed world.

Ω_3: The fourth assignment in Stanford's course. Students program an agent to find the midpoint of a world with unknown dimension. There are multiple strategies for this problem and many require $O(n^2)$ operations where n is the size of the world. The task is challenging even for those who already know how to program.

In addition to the final submission to any problem, from each student we also collect partial solutions as they progress from starter code to final answer. Table 1 summarizes the sizes of each of the datasets. For all three assignments studied, students take multiple steps to reach their final answer and as a result most programs in our datasets are intermediate solutions that are not responsive to unit tests that simply evaluate correctness. The code.org dataset is available at `code.org/research`.

For all assignments we have both functional and stylistic feedback based on class rubrics which range from observations of solution strategy, to notes on code decomposition, and tests for correctness. The feedback is generated for all submissions (including partial solutions) via a complex script. The script analyzes both the program trees and the series of steps a student took to assign annotations. In general, a script, no matter how complex, does not provide perfect feedback. However the ability to recreate these complex annotations allows us to rigorously evaluate our methods. An algorithm that is able to propagate such feedback should also be able to propagate human quality labels.

6. Results

We rely on a few baselines against which to evaluate our methods, but the main baseline that we compare to is a simplification of the NPM-RNN model (which we will call, simply, *RNN*) in which we drop the program embedding terms M_j from each node (cf. Eqn. 7).

The RNN model can be trained to predict postconditions as well as to propagate feedback. It has much fewer parameters than the NPM (and thus NPM-RNN) model being a strictly parametric model, and is thus expected to have an advantage in smaller training set regimes. On the other hand, it is also a strictly less expressive model and so the question is: how much does the expressive power of the NPM and NPM-RNN models actually help in practice? We address this question amongst others using two tasks: predicting postcondition and propagating feedback.

Algorithm	Ω_1	Ω_2	Ω_3
NPM	95% (98%)	87% (98%)	81% (94%)
RNN	96% (97%)	94% (95%)	46% (45%)
Common	58%	51%	42%

Table 2. Test set postcondition prediction accuracy on the three programming problems. Training set results in parentheses.

6.1. Prediction of postcondition

To understand how much functionality of a program is captured in our embeddings, we evaluate the accuracy to which we can use the program embedding matrices learned by the NPM model to predict postconditions — note, however, that we are not proposing to use the embeddings to predict post-conditions in practice. We split our observed Hoare triples into training and test sets and learn our NPM model using the training set. Then for each triple (P, A, Q) in the test set we measure how well we can predict the postcondition Q given the corresponding program A and precondition P. We evaluate accuracy as the average number of state variables (e.g. row, column, orientation and location of beepers) that are correctly predicted per triple, and in addition to the RNN model, compare against the baseline method "Common" where we select the most common postcondition for a given precondition observed in the training set. As our results in Table 2 show, the NPM model achieves the best training accuracy (with 98%, 98% and 94% accuracy respectively, for the three problems). For the two simpler problems, the parametric (RNN) model achieves slightly better test accuracy, especially for problem Ω_2 where the training set is much smaller. For the most complex programming problem, Ω_3, however, the NPM model substantially outperforms other approaches.

6.2. Composability of program embeddings

If we are to represent programs as matrices that act on a feature space, then a natural desiderata is that they "compose well". That is, if program C is functionally equivalent to running program B followed by program A, then it should be the case that $M_C \approx M_B \cdot M_A$. To evaluate the extent to which our program embedding matrices are *composable*, we use a corpus of 5000 programs that are composed of a subprogram A followed by another subprogram B (Compose-2). We then compare the accuracy of postcondition prediction using the embedding of an entire program M_C against the product of embeddings $M_B \cdot M_A$. As Table 3 shows, the accuracy using the NPM model for predicting postcondition is 94% when using the matrix for the root embedding. Using the product of two embedding matrices, we see that accuracy does not fall dramatically, with a decoding accuracy of 92%. When we test programs that are composed of three subprograms, A followed by B, then C (Compose-3), we see accuracy drop only to 83%.

Test	Direct	NPM	NPM-0	RNN	Common
Compose-2	94%	92%	87%	42%	39%
Compose-3	94%	83%	72%	28%	39%

Table 3. Evaluation of composability of embedding matrices: Accuracy on 5k random triples with ASTs rooted at **block** nodes. NPM-0 does not jointly optimize.

By comparison, the embeddings computed using the RNN, a more constrained model, do not seem to satisfy composability. We also compare against NPM-0, which is the NPM model using just the weights set by the smart initialization (see Section 3.2). While NPM-0 outperforms the RNN, the full nonparametric model (NPM) performs much better, suggesting that the joint optimization (of state and program embeddings) allows us to learn an embedding of the state space that is more amenable to composition.

6.3. Prediction of Feedback

We now use our program embedding matrices in the feedback propagation application described in Section 4. The central question is: given a budget of K human annotated programs (we set $K = 500$), what fraction of unannotated programs can we propagate these annotations to using the labelled programs, and at what precision? Alternatively, we are interested in the "force multiplication factor" — the ratio of students who receive feedback via propagation to students to receive human feedback.

Figure 3 visualizes recall and precision of our experiment on each of the three problems. The results translate to $214\times$, $12\times$ and $45\times$ force multiplication factors of teacher effort for Ω_1, Ω_2 and Ω_3 respectively while maintaining 90% precision. The amount to which we can force multiply feedback depends both on the recall of our model and the size of the corpus to which we are propagating feedback. For example, though Ω_2 had substantially higher recall than Ω_1, in Ω_2 the grading task was much smaller. There were only 6,700 unique programs to propagate feedback to, compared to Ω_1 which had over 210,000. As with the previous experiment, we observe that for both Ω_1 and Ω_2, the NPM-RNN and RNN models perform similarly. However for Ω_3, the NPM-RNN model substantially outperforms all alternatives.

In addition to the RNN, we compare our results to three other baselines: (1) Running unit tests, (2) a "Bag-of-Trees" approach and (3) k-nearest neighbor (KNN) with AST edit distances. The unit tests unsurprisingly are perfect at recognizing correct solutions. However, since our dataset is largely composed of intermediate solutions and not final submissions (especially for Ω_1 and Ω_3), unit tests are not a particularly effective way to propagate annotations. The Bag-of-Trees approach, where we trained a Naïve Bayes model to predict feedback conditioned on the set of subtrees in a program, is useful for feedback propagation but we observe that it underperforms the embedding solutions on each problem. Moreover, we extended this baseline by amalgamating functionally equivalent code (Nguyen et al., 2014). Using equivalences found using similar amount of effort as in previous work, we are able to achieve 90% precision with recall of 39%, 48% and 13%, for the three problems respectively. While this improves the baseline, NPM-RNN obtains almost twice as much recall on all problems. Finally, we find KNN with AST edit distances to be computationally expensive to run and highly ineffective at propagating feedback — calculating edit distance between all trees requires 20 billion comparisons for Ω_1 and 1.5 billion comparisons for Ω_3. Moreover, the highest precision achieved by KNN for Ω_3 is only 43% (note that the cut-off for the x-axis in Figure 3 is 80%) and at that precision only has a recall of 1.3%.

The feedback that we propagate covers a range of stylistic and functional annotations. To further understand the strengths and weaknesses of our solution, we explore the performance of the NPM-RNN model on each of the nine possible annotations for Ω_3. As we see in Figure 4(c), our model performs best on functional feedback with an average 44% recall at 90% precision, followed by strategic feedback and performs worst at propagating purely stylistic annotations with averages of 31% and 8% respectively. Overall propagation for Ω_3 is 33% recall at 90% precision.

6.4. Code complexity and performance

The results from the above experiments are suggestive that the nonparametric models perform better on more complex code while the parametric (RNN) model performs better on simpler code. To dig deeper, we now look specifically into how our performance depends on the complexity of programs in our corpus — a question that is also central to understanding how our models might apply to other assignments. We focus on submissions for Ω_3, which cover a range of complexities, from simple programs to ones with over 50 decision points (loops and if statements). The distribution of *cyclomatic complexity* (McCabe, 1976), a measure of code structure, reflects this wide range (shown in gray in Figures 4(a),(b)). We first sort and bin all submissions to Ω_3 by cyclomatic complexity into ten groups of equal size. Figures 4(a),(b) plot the results of the post-condition prediction and force multiplication experiments run individually on these smaller bins (still using a holdout set, and a budget of 500 graded submissions). While the RNN model performs better for simple programs (with cyclomatic complexity ≤ 6), both train and test accuracies for the RNN degrade dramatically as programs become more complicated. On the other hand, while the NPM model overfits, it maintains steady (and better) performance in test accuracy as complexity increases. This pattern may help to

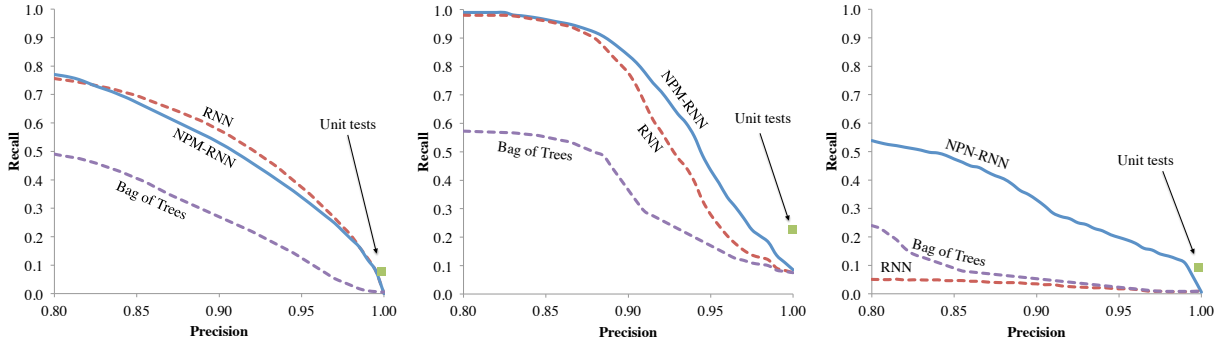

Figure 3. Recall of feedback propagation as a function of precision for three programming problems: (a) Ω_1, (b) Ω_2, and (c) Ω_3. On each, we compare our NPM-RNN against the RNN method and two other baselines (bag of trees and unit tests).

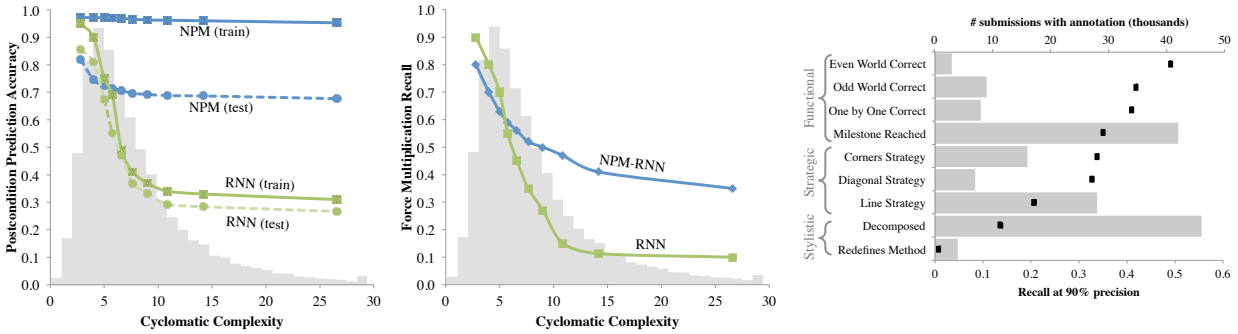

Figure 4. (a) NPM and RNN postcondition prediction accuracy as a function of cyclomatic complexity of submitted programs; (b) NPM-RNN and RNN feedback propagation recall (at 90% precision). Note that the ratio of human graded assignments to number of programs is much higher in this experiment than Figure 3; (c) A breakdown of the accuracy of the nonparametric model by feedback type for Ω_3 (black dots). The gray bars histogram the feedback types by frequency.

explain our observations that the RNN is more accurate for force multiplying feedback on simple problems.

7. Discussion

In this paper we have presented a method for finding simultaneous embeddings of preconditions and postconditions into points in shared Euclidean space where a program can be viewed as a linear mapping between these points. These embeddings are predictive of the function of a program, and as we have shown, can be applied to the the tasks of propagating teacher feedback. The courses we evaluate our model on are compelling case studies for different reasons. Tens of millions of students are expected to use Code.org next year, meaning that the ability to autonomously provide feedback could impact an enormous number of people. The Stanford course, though much smaller, highlights the complexity of the code that our method can handle.

There remains much work towards making these embeddings more generally applicable, particularly for domains where we do not have tens of thousands of submissions per problem or the programs are more complex. For settings where users can define their own variables it would be necessary to find a novel method for mapping program memory into vector space. An interesting future direction might be to jointly find embeddings across multiple homeworks from the same course, and ultimately, to even learn using arbitrary code outside of a classroom environment. To do so may require more expressive models. From the standpoint of purely predicting program output, the approaches described in this paper are not capable of representing arbitrary computation in the sense of the Church-Turing thesis. However, there has been recent progress in the deep learning community towards models capable of simulating Turing machines (Graves et al., 2014). While this "Neural Turing Machines" line of work approaches quite a different problem than our own, we remark that such expressive representations may indeed be important for statistical reasoning with arbitrary code databases.

For the time being, feature embeddings of code can at least be learned using the massive online education datasets that have only recently become available. And we believe that these features will be useful in a variety of ways — not just in propagating feedback, but also in tasks such as predicting future struggles and even student dropout.

Acknowledgments

We would like to thank Kevin Murphy, John Mitchell, Vova Kim, Roland Angst, Steve Cooper and Justin Solomon for their critical feedback and useful discussions. We appreciate the generosity of the Code.Org team, especially Nan Li and Ellen Spertus, who providing data and support. Chris is supported by NSF-GRFP grant number DGE-114747.

References

Basu, Sumit, Jacobs, Chuck, and Vanderwende, Lucy. Powergrading: a clustering approach to amplify human effort for short answer grading. *Transactions of the Association for Computational Linguistics*, 1:391–402, 2013.

Bergstra, James and Bengio, Yoshua. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

Bowman, Samuel R. Can recursive neural tensor networks learn logical reasoning? *arXiv preprint arXiv:1312.6192*, 2013.

Brooks, Michael, Basu, Sumit, Jacobs, Charles, and Vanderwende, Lucy. Divide and correct: Using clusters to grade short answers at scale. In *Proceedings of the first ACM conference on Learning@ scale conference*, pp. 89–98. ACM, 2014.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Goller, Christoph and Kuchler, Andreas. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pp. 347–352. IEEE, 1996.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Hoare, Charles Antony Richard. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Huang, Jonathan, Piech, Chris, Nguyen, Andy, and Guibas, Leonidas J. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *The 16th International Conference on Artificial Intelligence in Education (AIED 2013) Workshop on Massive Open Online Courses (MOOCshop)*, 2013.

Lan, Andrew S, Vats, Divyanshu, Waters, Andrew E, and Baraniuk, Richard G. Mathematical language processing: Automatic grading and feedback for open response mathematical questions. *arXiv preprint arXiv:1501.04346*, 2015.

McCabe, Thomas J. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

Mokbel, Bassam, Gross, Sebastian, Paassen, Benjamin, Pinkwart, Niels, and Hammer, Barbara. Domain-independent proximity measures in intelligent tutoring systems. In *Proceedings of the 6th International Conference on Educational Data Mining (EDM)*, 2013.

Nguyen, Andy, Piech, Christopher, Huang, Jonathan, and Guibas, Leonidas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International World Wide Web Conference (WWW 2014)*, Seoul, Korea, 2014.

Ovsjanikov, Maks, Ben-Chen, Mirela, Solomon, Justin, Butscher, Adrian, and Guibas, Leonidas. Functional maps: a flexible representation of maps between shapes. *ACM Transactions on Graphics (TOG)*, 31(4):30, 2012.

Ovsjanikov, Maks, Ben-Chen, Mirela, Chazal, and Guibas, Leonidas. Analysis and visualization of maps between shapes. In *Computer Graphics Forum*, volume 32, pp. 135–145. Wiley Online Library, 2013.

Piech, Chris, Sahami, Mehran, Huang, Jonathan, and Guibas, Leonidas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, pp. 195–204. ACM, 2015.

Rogers, Stephanie, Garcia, Dan, Canny, John F, Tang, Steven, and Kang, Daniel. *ACES: Automatic evaluation of coding style*. PhD thesis, Masters thesis, EECS Department, University of California, Berkeley, 2014.

Socher, Richard, Pennington, Jeffrey, Huang, Eric H, Ng, Andrew Y, and Manning, Christopher D. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 151–161. Association for Computational Linguistics, 2011.

Socher, Richard, Perelygin, Alex, Wu, Jean Y, Chuang, Jason, Manning, Christopher D, Ng, Andrew Y, and Potts, Christopher. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1631–1642. Citeseer, 2013.

Song, Le, Huang, Jonathan, Smola, Alex, and Fukumizu, Kenji. Hilbert space embeddings of conditional distributions with applications to dynamical systems. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 961–968. ACM, 2009.

Song, Le, Fukumizu, Kenji, and Gretton, Arthur. Kernel embeddings of conditional distributions: A unified kernel framework for nonparametric inference in graphical models. *Signal Processing Magazine, IEEE*, 30(4):98–111, 2013.

Zaremba, Wojciech, Kurach, Karol, and Fergus, Rob. Learning to discover efficient mathematical identities. In *Advances in Neural Information Processing Systems*, pp. 1278–1286, 2014.