Neuro-Symbolic Program Corrector for Introductory Programming Assignments

Sahil Bhatia
Netaji Subhas Institute of Technology
Delhi, India
sahilbhatia.nsit@gmail.com

Pushmeet Kohli
Google Deepmind
London, UK
pushmeet@google.com

Rishabh Singh
Microsoft Research
Redmond, USA
risin@microsoft.com

ABSTRACT

Automatic correction of programs is a challenging problem with numerous real world applications in security, verification, and education. One application that is becoming increasingly important is the correction of student submissions in online courses for providing feedback. Most existing program repair techniques analyze Abstract Syntax Trees (ASTs) of programs, which are unfortunately unavailable for programs with syntax errors. In this paper, we propose a novel Neuro-symbolic approach that combines neural networks with constraint-based reasoning. Specifically, our method first uses a Recurrent Neural Network (RNN) to perform syntax repairs for the buggy programs; subsequently, the resulting syntactically-fixed programs are repaired using constraint-based techniques to ensure functional correctness. The RNNs are trained using a corpus of syntactically correct submissions for a given programming assignment, and are then queried to fix syntax errors in an incorrect programming submission by replacing or inserting the predicted tokens at the error location. We evaluate our technique on a dataset comprising of over 14,500 student submissions with syntax errors. Our method is able to repair syntax errors in 60% (8689) of submissions, and finds functionally correct repairs for 23.8% (3455) submissions.

CCS CONCEPTS

• **Computing methodologies** → *Neural networks*; • **Software and its engineering** → *Functionality*; *Search-based software engineering*;

KEYWORDS

Neural Program Correction, Automated Feedback Generation, Neural guided search

ACM Reference Format:
Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180219

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180219

1 INTRODUCTION

The increasing importance of computing has resulted in a dramatic growth in the number of students interested in learning programming. Initiatives such as edX and Coursera attempt to meet this demand by providing Massive Open Online Courses (MOOCs) that are easily accessible to students worldwide. While MOOCs have numerous benefits, their effectiveness over the traditional classroom setting is limited by the challenge of providing quality feedback to students on their submissions to open-response programming assignments. We present a learning based technique to automatically generate corrections for student submissions that in turn can be used to generate feedback.

Most existing systems for automated feedback generation are based on the functional correctness and style characteristics of student programs. For instance, the AutoProf [27] system uses constraint-based synthesis to find the minimum number of changes to an incorrect student submission that would transform it to become functionally equivalent to a reference teacher implementation. In contrast, the Codewebs system [17] adopts a search based approach where feedback generated by teachers on a handful of submissions is propagated to provide feedback on thousands of submissions by querying the dataset using code phrases. Codewebs allows querying the dataset of student submissions using "code phrases", which are subgraphs of AST in the form of subtrees, subforests, and contexts. These systems assume the availability of program ASTs, which unfortunately do not exist for programs with syntax errors. As an example, almost 20% of submissions in a dataset we obtained from an introductory Python course on edX had syntax errors. In this paper, we propose a novel Neuro-symbolic program correction approach that overcomes this problem by using a hybrid approach that combines deep neural networks with constraint-based reasoning. While the neural networks are able to correct syntactic problems with student submissions, the constraint-based synthesis techniques allow for finding semantic corrections. A particularly interesting aspect enabled by our hybrid approach is that of generating the correct usage of infrequent variable names in a student program.

There are two key steps in our approach. For a given programming problem, we first use the set of student submissions without syntax errors to learn a model of token sequences, which is then used to hypothesize possible fixes to syntax errors in a student submission. Our system enumerates over the set of possible modifications to the incorrect program in an ordered fashion to compute the syntax error fixes. We use a Recurrent Neural Network (RNN) [25] to learn the token sequence model that can learn large contextual dependencies between tokens. In the second step, we use constraint-based program synthesis (the Sketch solver [28])

to find minimal changes to the student program such that it becomes functionally equivalent to a reference implementation. Note that the small size of student programs in this domain of introductory programming (around 10-20 LOC) allows for performing sophisticated search-based correction techniques.

Our approach is inspired from the recent pioneering work on learning probabilistic models of source code from a large repository of code for many different applications [1–4, 13, 19, 24]. Hindle et al. [13] learn an n-gram language model to capture the repetitiveness present in a code corpora and show that n-gram models are effective at capturing the local regularities. They used this model for suggesting next tokens that was already quite effective as compared to the type-based state-of-the-art IDE suggestions. Nguyen et al. [19] enhanced this model for code auto-completion to also include semantic knowledge about the tokens (such as types) and the scope and dependencies amongst the tokens to consider global relationships amongst them. The Naturalize framework [3] learns an n-gram language model for learning coding conventions and suggesting changes to increase the stylistic consistency of the code. More recently, some other probabilistic models such as conditional random fields and log bilinear context models have been developed for suggesting names for variables, methods, and classes [4, 24]. We also learn a language model to encode the set of valid token sequences, but our approach differs from previous approaches in four key ways: i) our application of using a language model learnt from syntactically correct programs to fix syntax errors is novel, ii) since we cannot obtain the AST of the programs with syntax errors, many of these techniques that use the AST information for learning the language model are not applicable, iii) we learn a recurrent neural network (RNN) that can capture more complex dependencies between tokens than the n-gram or logbilinear neural networks, and finally iv) instead of learning one language model for the complete code corpus, we learn individual RNN models for different programming assignments so that we can generate more precise repairs for different problems.

We evaluate our system on student solutions obtained from 9 programming problems taken from the Intro to Programming class (6.00x) offered on edX. We consider 74818 student submissions in total, out of which 14526 (19.4%) submissions have syntax errors. Our technique can find repairs for fixing syntax errors for 59.8% (8689/14526) of these submissions and finds semantic repairs for 23.8% (3455) of the submissions. To summarize, the paper makes the following key contributions:

- We formalize the problem of syntax corrections in programs as a token sequence prediction problem using the recurrent neural networks (RNN).
- We present the SynFix algorithm that uses the predicted token sequences for finding syntax repairs using a search procedure. The algorithm then uses constraint-based synthesis to find minimal repairs for semantic correctness.
- We evaluate the effectiveness of our system on more than 14, 500 student submissions.

2 MOTIVATING EXAMPLES

In this section, we present a sample of different types of corrections our system is able to generate as shown in Figure 2. The example programs, shown in Figure 2, come from the student submissions for different problems taken from the Introduction to Python Programming MOOC (6.00x) on edX.

Our syntax correction algorithm considers two types of parsing errors in Python programs: i) syntax errors, and ii) indentation errors. It uses the offset information provided by the Python compiler to locate the potential error locations, and then uses the program tokens from the beginning of the function to the error location as the prefix token sequence to query the RNN model to predict the correcting token sequences. The algorithm enumerates subsequences of predicted sequence in a ranked order to generate proposals for insertion or replacements at the error location. However, there are many cases such as the ones shown in Figure 2(c) where the compiler is not able to accurately locate the exact offset location for the syntax error. In such cases, our algorithm ignores the tokens present in the error line and considers the prefix ending at the previous line. Using the prefix token sequence, the algorithm uses a neural network to perform the prediction of next k tokens that are most likely to follow the prefix sequence, which are then either inserted at the error location or are used to replace the original token sequence.

A sample of repairs generated by our algorithm (emphasized in red) based on inserting the predicted tokens at the offset location is shown in Figure 2(a). The kinds of errors in this class typically include inserting unbalanced parenthesis, completing partial expressions (such as exp- to exp-1), adding syntactic tokens such as : after if and else, etc. Some example syntax errors that require replacing the original tokens in the incorrect program with the predicted tokens are shown in Figure 2(b). These errors typically include replacing an incorrect operator with another operator (such as replacing = with *, = in comparisons with ==), deleting additional mismatched parenthesis etc.

Since the compiler might not always accurately identify the error location, our system predicts the complete replacement line for such cases as shown in Figure 2(c). These errors typically include wrong spelling of keywords (e.g. retrun instead of return), wrong expression for the return statement etc. For fixing such syntax errors, our algorithm generates the prefix token sequence that ends at the last token of the line previous to the error line. It then queries the model to predict a token sequence that ends at a new line, and replaces the error line completely with the predicted line. A sample of indentation errors is shown in Figure 2(d) and submissions that require multiple fixes are shown in Figure 2(e). After fixing a syntax error, the algorithm iteratively calls itself to fix other errors.

The fixes to syntax errors in the code may or may not correspond to the correct *semantic* fix, i.e. the fixed program may not compute the desired result. An end-to-end example of semantic repair is shown in Figure 1. Since the Python compiler does not predict the correct location for syntax error for this case, the RNN model uses the previous line method to insert a new line while a%t != 0 or b%t !=0 as the suggested fix. This results in fixing the syntax error, but introduces an unknown variable t. In the second phase, our system uses an error model consisting of a set of rewrite rules to modify different program expressions, and then uses the Sketch solver to efficiently search over this large space to compute minimal modifications to program such that it becomes functionally equivalent with respect to a teacher's implementation. For this example, we

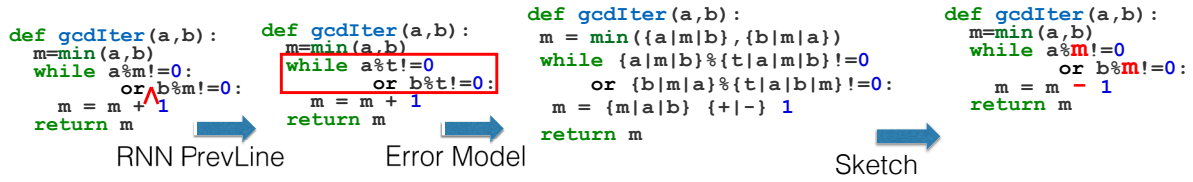


Figure 1: An end-to-end correction example: the RNN model first fixes the syntax error by replacing line 3 but introduces an unknown variable t. The second phase applies an error model to introduce various replacement choices and uses Sketch to find minimal repair for the input program to be functionally correct. An expression of the form {a|b|c} denotes a choice expression, where the Sketch compiler can choose any expression amongst them. The first option in the choice expression denotes the default expression with cost 0, while all other options result in a cost of 1.

```
def gcdRecur(a, b):
    if b == 0:
        return a
    x,y=max(a,b),min(a,b)
    return gcdRecur(x-y,y)
```

```
def recPower(b, e):
    if e <= 0:
        return 1
    return b * recPower(b, e-1)
```

(a) SyntaxError - Insert Token

```
def isWordGuessed(sWord, lGuess):
    result=True
    for s in sWord:
        if ! not(s in lGuess):
            result=False
    return result
```

```
def rP(b, e):
    t = b
    if(e==0):
        return t
    else:
        t *=b
        return t+rP(b,e-= 1 1) )
```

(b) SyntaxError - Replace Token

```
def recPower(b, e):
    f e == 1:
    if e == 1:
        return b
    return b * recPower(b, e - 1)
```

```
def recPower(b, e):
    if e == 1:
        return b
    if e > 1:
        return e -= 1
        return b * recPower(b, e-1)
```

(c) SyntaxError - Previous Line Insert

```
def recPower(b, e):
    if e == 0:
    return 1
      return 1
    return b * recPower(b,e-1)
```

```
def recPower(b, e):
    x = b
    while(e > 0):
        x *= b
        -= 1
        e -= 1
    return b
```

(d) Indentation Error - Insert Token

```
def recPower(b, e):
    if e == 1:
       return e
    return b + recPower(b, e- - 1)
```

```
def gcdIter(a, b):
    mi=min(a,b)
    while a%mi!=0 : or b%mi!=0:
        mi-=1
    return mi
```

(e) Multiple Errors - Combination of Insert, Replace, Previous Line

Figure 2: A sample of example syntax fixes generated by the RNN model for the student submissions with syntax errors. The fix suggestions are emphasized in red font, whereas the expressions removed are emphasized in blue with a frame box.

use a simple error model for brevity that can replace a variable in assignments and conditionals with any other variable, and replace an operator with + or -. Sketch is then able to identify the correct semantic fix that replaces t with mi and changes + to - in line 4.

3 NEURO-SYMBOLIC REPAIR ALGORITHM

An overview of the workflow of our system is shown in Figure 3. For a given programming problem, we first use the set of all syntactically correct submissions to learn a generative token sequence

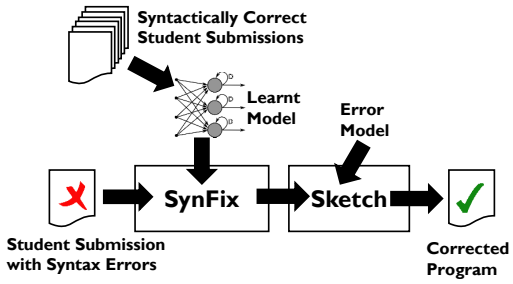


Figure 3: An overview of the system workflow.

model using an RNN [30]. The sequence models are problem-specific and we learn different models for different problems. We then use the SynFix algorithm to find small syntactic corrections to a student submission by modifying the submission with the token sequences predicted by the learnt model. Finally, we use the Sketch [28] synthesis system to find minimal modifications (defined by rewrite rules) to the syntactically-repaired program such that the program becomes functionally correct with respect to a reference implementation. We now briefly describe the three key phases in our workflow: i) the training phase, and ii) the SynFix algorithm, and iii) Sketch-based synthesis.

3.1 RNN-based Language Model

The Recurrent Neural Network (RNN) is a generalization of feedforward networks that can encode sequences of variable length and RNNs have been shown to be effective in many domains such as machine translation [31] and speech recognition [7]. We first use the syntactically correct student submissions to obtain the set of all valid token sequences and then use a threshold frequency value to decide whether to relabel a token to a general IDENT token for handling rarely used tokens (such as infrequent variable/method names). A token is encoded into a fixed-length one-hot vector, where the size of the vector is equal to the size of the training vocabulary. In the training phase, we provide the token sequences to the input layer of the RNN and the input token sequence shifted left by 1 as the target token sequence to the output layer as shown in Figure 4(a). After learning the network from the set of syntactically correct token sequences, we use the model to predict next token sequences given a prefix of the token sequence to the input layer as shown in Figure 4(b). The first output token is predicted at the output layer using the input prefix sequence. For predicting the next output token, the predicted token is used as the next input token in the input layer as shown in the figure.

We first present a brief overview of the computational model of a simple RNN with a single hidden layer. Consider an input sequence of length L and an RNN with I number of inputs, a single hidden layer with H number of hidden units, and k output units. Let $x_t \in R^I$ denote the input at time step t (encoded as a vector), $h_t \in R^H$ denote the hidden state at time step t, $W \in R^{H \times I}$ denote the weight matrix corresponding to the weights on connections from input layer to hidden layer, $V \in R^{H \times H}$ be the weight matrix from hidden to hidden layer (recursive), and $U \in R^{I \times H}$ be the weight matrix from hidden to the output layer. The computation model of the RNN can be defined as following:

$$h_t = f(W * x_t + V * h_{t-1}); o_t = \text{softmax}(U * s_t)$$

The hidden state h_t at time step t is computed by applying a non-linear function f (e.g. tanh or sigmoid) to a weighted sum of the input vector x_t and the previous hidden state vector h_{t-1}. The output vector o_t is computed by applying the softmax function to the weighted state vector value obtained s_t obtained by an affine transformation $U * h_t$. The hidden units take the weighted sum as input and map it to a value in the set (-1,1) using the sigmoid function to model non-linear activation relationships. The RNNs can be trained using backpropagation through time (BPTT) [32] to calculate the gradient and adjust the weights given a set of input and output sequences.

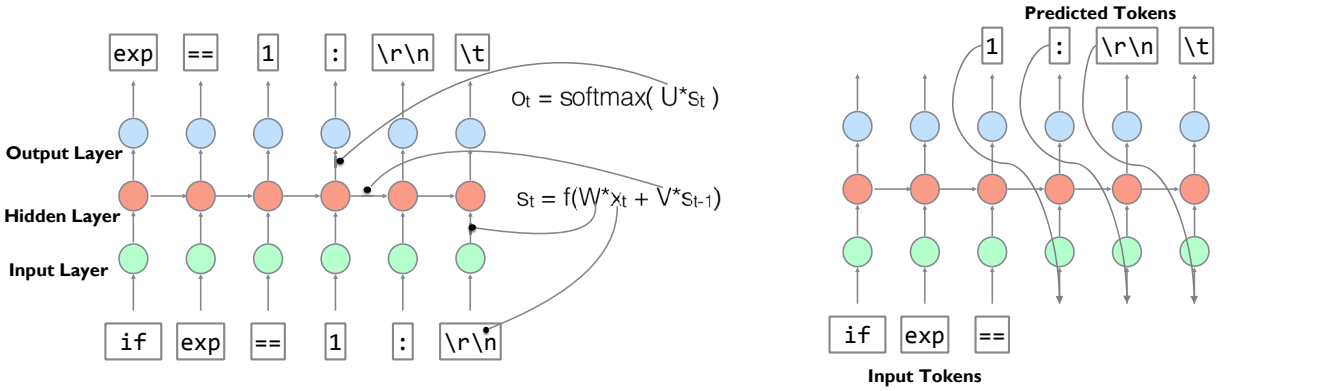
3.2 The SYNFIX Algorithm

We first present the SynFixOne algorithm that fixes the first syntax error in the student submission (if possible), and then present the SynFix algorithm to fix multiple syntax errors.

Fixing one syntax error: The SynFixOne algorithm, shown in Algorithm. 1, takes as input a program P (with syntax errors) and a token sequence model $\mathcal{M}$, and returns a program P' with its first syntax error on the error location fixed (if possible) or ϕ denoting that the syntax error can not be fixed. The algorithm first uses a parser to obtain the type of error err and the token location loc where the first syntax error occurs, and computes a prefix of the token sequence $\widetilde{\mathrm{T}}_{pref}$ corresponding to the token sequence starting from the beginning of the program until the error token location loc. We use the notation $a[i..j]$ to denote a subsequence of a sequence a starting at index i (inclusive) and ending at index j (exclusive). The algorithm then queries the model $\mathcal{M}$ to predict the token sequence $\widetilde{\mathrm{T}}_k$ of a constant length k that is most likely to follow the prefix token sequence.

After obtaining the token sequence $\widetilde{\mathrm{T}}_k$, the algorithm searches over token sequences $\widetilde{\mathrm{T}}_k[1..i]$ of increasing lengths ($1 \leq i \leq k$) until either inserting or replacing the token sequence $\widetilde{\mathrm{T}}_k[1..i]$ at the error location results in a program P' with no syntax error at the error location. If the algorithm cannot find a token sequence that can fix the syntax error at loc, the algorithm then creates another prefix $\widetilde{\mathrm{T}}_{pref}^{prev}$ of the original token sequence such that it ignores all previous tokens in the same line as that of the error token location. It then predicts another token sequence $\widetilde{\mathrm{T}}_k^{prev}$ using the model for the new token sequence prefix, and selects a subsequence $\widetilde{\mathrm{T}}_k^{prev}[1..m]$ that ends at a new line token. Finally, the algorithm checks if replacing the line containing the error location with the predicted token sequence results in fixing the syntax error. If yes, it returns the fixed program P'.

Example: Consider the Python program shown in Figure 5(a). The Python parser returns a syntax error in line 2 with the error offset corresponding to the location of the = token. The SynFix algorithm first constructs a prefix of the token sequence consisting of tokens from the start of the program to the error location such that $\widetilde{\mathrm{T}}_{pref}$ = ['def', 'recPower', '(', 'b', ',', 'e', ')', ':', '\r\n', '\t', 'if', 'e']. It then queries the learnt model to predict the most likely token sequence that follows the input prefix sequence. Let us assume k is 3 and the model returns the predicted token sequence



(a) Training Phase (b) Prediction Phase

Figure 4: The modeling of our syntax repair problem using an RNN with 1 hidden layer. (a) We provide input and output token sequences in the training phase to learn the weight matrices. (b) In the prediction phase, we provide a token sequence to the input layer of the RNN and generate the output token sequences using the learnt model.

Algorithm 1 SynFixOne

Input: buggy program P, token sequence model $\mathcal{M}$
(err,loc) := Parse(P); $\widetilde{\mathrm{T}}$:= Tokenize(P)
$\widetilde{\mathrm{T}}_{pref}$:= $\widetilde{\mathrm{T}}$[1..loc]; $\widetilde{\mathrm{T}}_k$:= Predict($\mathcal{M}, \widetilde{\mathrm{T}}_{pref}$)
for $i \in$ range$(1, k)$ **do**
  P'_{ins} := Insert(P, loc, $\widetilde{\mathrm{T}}_k[1..i]$)
  if Fixed(P'_{ins}, loc) **return** P'_{ins}
  P'_{repl} := Replace(P, loc, $\widetilde{\mathrm{T}}_k[1..i]$)
  if Fixed(P'_{repl}, loc) **return** P'_{repl}
end for
$\widetilde{\mathrm{T}}^{prev}_{pref}$:= $\widetilde{\mathrm{T}}[1..l_{\text{prev}}(loc)]$; $\widetilde{\mathrm{T}}^{prev}_k$:= Predict($\mathcal{M}, \widetilde{\mathrm{T}}^{prev}_{pref}$)
P'_{prev} := ReplaceLine(P, line(loc), $\widetilde{\mathrm{T}}^{prev}_k[1..m]$)
if Fixed(P'_{prev}, , loc) **return** P'_{prev} **else return** ϕ

Algorithm 2 SynFix

Input: buggy program P, sequence model $\mathcal{M}$, max fixes f
for $i \in$ range$(1, f+1)$ **do**
  P' := SynFixOne($P, \mathcal{M}$)
  if $P' == \phi$ **return** ϕ
  if Parse(P') = ϕ **return** P'
  else $P := P'$
end for
return ϕ

$\widetilde{\mathrm{T}}_k$ = ['==', '0', ':']. The algorithm first tries to use the smaller prefixes of the predicted token sequence (in this case '==') to see if the syntax error can be fixed. It first tries to insert the predicted token sequence '==' in the original program but that results in the expression if e == = 0:, which still results in an error. It then tries to replace the original token sequence with the predicted token sequence, which results in the expression if e == 0: that passes the parser check, and returns the fixed program.

Fixing multiple syntax errors: The SynFix algorithm shown in Algorithm. 2 takes as input the buggy student submission P, the token sequence model $\mathcal{M}$, and a parameter f denoting the maximum number of syntax corrections considered by the algorithm, and returns either the fixed program P' obtained using fewer than f number of corrections or ϕ otherwise. The algorithm iteratively calls the SynFixOne (a maximum of f times) to fix one syntax error in P at a time. Some example programs requiring multiple fixes are shown in Figure 2(e).

3.3 Constraint-based Semantic Repair

After obtaining a syntactically correct program using the SynFix algorithm, our system uses a technique similar to that of AutoProf to find minimal transformations to the student program such that it becomes functionally equivalent with respect to a reference implementation. The class of transformations are defined using a generic error model comprising of five different types of transformations $\mathcal{R}$ (rewrite rules): 1) Comparison operators: $c_{op} \rightarrow \{< \mid \leq \mid > \mid \geq \mid != \mid ==\}$ (i.e. a comparison operator can be modified to any of the operators in the right hand set), 2) Arithmetic operators: $a_{op} \rightarrow \{+|-|*|/|\%\}$, 3) Variable replacement: $v \rightarrow ?v$ on RHS of assignments and expressions, where $?v$ denotes the set of all variables live at the program location, 4) off-by-one errors: $a \rightarrow \{a+1, a-1, 0, 1\}$, and 5) Constant modification: $n \rightarrow ??$, where ?? denotes any integer constant value. For example, the first rewrite rule states that any comparison operator in the student program can be potentially rewritten to any of the operator in the set $\{< \mid \leq \mid > \mid \geq \mid != \mid ==\}$. Since this results in a huge search space of program modification, we use a constraint-based synthesis

solver Sketch to solve for minimal number of modifications such that the student program becomes functionally correct as follows.

$$\exists\, P' \,\forall\, \text{in} : P' = \text{Rewrite}(P, \mathcal{R}) \wedge P_T(in) = P'(in) \wedge \min(\text{Cost}(P, P')) \quad (1)$$

The above constraint requires Sketch to find a program P' in the space of programs obtained from the rewrite rules $\mathcal{R}$ such that P' is functionally equivalent with respect to the reference implementation P_T and P' requires minimal number of applications of the rewrite rules. More details about how Sketch solves the above constraint using the Cegis algorithm can be found in [28].

Consider the program shown in Figure 5(b), which requires 4 corrections. The program uses an undefined variable t and uses the wrong comparison expressions a%t != 1 instead of a%mi != 0. Given this program, Sketch uses the generic error model to identify the correct loop comparison expression a%mi != 0 or b%mi != 0, which results in a functionally correct program.

4 EVALUATION

We evaluate our system on 74, 818 Python submissions obtained from the Intro to Programming in Python course (6.00x) on the edX MOOC platform. Our benchmark set consists of student submissions for 9 programming problems, which asks students to write iterative or recursive functions over integer, string, and list data types. The number of submissions for each problem in our evaluation set is shown in Table 1 and a significant fraction of student submissions have syntax errors (19.41%). Using the evaluation, we aim to answer the following research questions: 1) How well our algorithm is able to repair syntax and semantic errors?, 2) What algorithmic choices are chosen in the SynFix algorithm for fixing the errors?, 3) How do problem-specific models compare with general models?, and finally 4) How does the RNN language model compares with an n-gram baseline model.

4.1 Benchmarks

Our benchmark set consists of student submissions for 9 programming problems recurPower, iterPower, oddTuples, evalPoly, compDeriv, gcdRecur, hangman1, hangman2, and gcdIter taken from the edX course. These problems ask students to write iterative or recursive functions over integer, string, and list data types, and requires students to use different language constructs such as conditionals, nested loops, functions, list comprehensions etc to solve the problems. The recurPower problem asks students to write a recursive function that takes as input a number base and an integer exp, and computes the value base^{exp}. The iterPower problem has the same functional specification as the recurPower problem but asks students to write an itervative solution instead of a recursive solution. The oddTuples problem asks students to write a function that takes as input a tuple l and returns another tuple that consists of every other element of l. The evalPoly problem asks students to write a function to compute the value of a polynomial on a given point, where the coefficients of the polynomial are represented using a list of doubles. Finally, the compDeriv problem asks students to write a function to return a polynomial (represented as a list) that corresponds to the derivative of a given polynomial. The distribution of the lines of code (LOC) for different benchmark problems is shown in Figure 6. The overall mean is 7.13 LOC with a standard deviation of 3.52.

The number of student submissions for each problem in our evaluation set is shown in Table 1. In total our evaluation set consists of 74, 818 student submissions. The number of submissions for the evalPoly and compDeriv problems are relatively fewer than the number of submissions for the other problems. This is because these problems were still in the submission stage at the time the data snapshot was obtained from the edX platform. But this also gives us an opportunity to evaluate how well our technique performs when we have thousands of correct attempts in the training phase as compared to only a few hundred attempts. Another interesting fact to observe from the table is that a significant fraction of student submissions have syntax errors (19.41%).

Training Phase: The token sequence model for a given problem is learnt over all the problem submissions without syntax errors. The submissions are first tokenized into a set of sequence of tokens (with tokens occurring below a threshold renamed as ident), which are then fed into the RNN. We used a learning rate of 0.002, a sequence length of 10, and a batch size of 50. We use the batch gradient descent method with rmsprop (decay rate 0.97) to learn the weights, where the gradients were clipped at a threshold of 1. We use RNN with LSTM cells consisting of 2 hidden layers each with 128 hidden units and train the model for 50 epochs. We also varied the number of hidden layers (1/2), the hidden units(128/256), and the type of RNN cells (RNN/LSTM), but did not observe significant changes in correction accuracy.

4.2 Number of Corrected Submissions

The overall results of our system is shown in Table 1. The SynFix algorithm is able to generate repairs to fix the syntax errors in 8689 (59.8%) programs. Amongst these repaired programs, 2051(14.12%) of the programs are also semantically correct – i.e. the repaired programs exhibit the desired functional behavior. Using the Sketch system with a generic error model, our system is able to generate semantically correct repairs for 3455(23.78%) of the submissions. The average time taken by the SynFix to repair syntax errors is 1.15s and by Sketch to generate semantic repair is 2.7s.

We observe that even with relatively fewer number of total attempts for the evalPoly and compDeriv problems, the system is able to repair a significant number of syntax errors (50.3% and 55.73% respectively). The average number of tokens per problem is 331, 458, whereas the average vocabulary size for training RNNs is only 626. This implies that there are a large number of common identifier names shared across large number of student submissions. For the results shown in the table, we use a vocabulary threshold value of $t = 4$. Additionally, a large fraction of syntax errors can be fixed using only 1 repair (6966), but there are still a significant number of submissions (1723) that are repaired by considering additional changes.

Another interesting observation is that the difference between vocabulary size (number of unique tokens) and the training vocabulary obtained after removing tokens below the threshold is not very large – implying that there are many popular identifier names that are shared across submissions that the language model can use for prediction. Finally, the number of semantic corrections is the

```
def recPower(b, e):
    if e = 0:
      return 1;
    else:
      return b*recPower(b,e-1)
```

(a)

```
def gcdIter(a, b):
    mi=min(a,b)
    while a%t!=1 or b%t!=1:
        mi-=1
    return mi
```

(b)

Figure 5: (a) A submission with syntax error in line 2 (= instead of ==), (b) A syntactically correct but semantically incorrect program generated by SynFix algorithm.

Problem	Total Attempts	Syntax Errors (Percentage)	Total Tokens	Training Vocab	Syntax Fixed	Semantic Fixed	RNN + Sketch Semantic Fix
recurPower	10247	2071 (20.21%)	3389858	117	1309 (63.21%)	663 (32.01%)	955 (46.11%)
iterPower	11855	2661 (22.45%)	3558849	526	1864 (70.05%)	401 (15.07%)	667 (25.07%)
oddTuples	29120	4905(16.84%)	1167877	1053	2976 (60.67%)	165 (3.37%)	654 (13.34%)
evalPoly	1148	324 (28.22%)	55370	276	163 (50.31%)	39 (12.04%)	67 (20.68%)
compDeriv	528	323 (61.18%)	18557	150	180 (55.73%)	54 (16.72%)	84 (26.00%)
gcdRecur	7751	1421(18.33%)	275476	274	829 (58.34%)	426 (29.98%)	484 (34.06%)
hangman1	2040	192(9.41%)	106970	398	79 (41.14%)	14 (7.29%)	36 (18.75%)
hangman2	1604	101(6.29%)	108662	546	53 (52.48%)	8 (7.92%)	23 (22.77%)
gcdIter	10525	2528(24.01%)	552405	741	1236 (48.89%)	281(11.15%)	485 (19.18%)
Total	74818	14526(19.4%)	2983124	453	8689(59.8%)	2051(14.1%)	3455(23.8%)

Table 1: The overall results of the SynFix algorithm on 9 benchmark problems.

Problem	Incorrect Attempts	Syntactically Fixed					Num. of Fixes	
		Offset		Offset-1		PrevLine	f=1	f=2
		Insert	Replace	Insert	Replace			
recurPower	2071	55	55	574	832	1128	1022	287
iterPower	2661	15	14	746	964	1488	1559	305
oddTuples	4905	198	192	1279	1765	2106	2311	665
evalPoly	324	8	6	49	48	147	135	28
compDeriv	323	2	2	53	80	147	129	51
gcdIter	2528	34	44	637	640	927	1013	223
hangman1	192	1	7	24	43	64	65	14
hangman2	101	2	3	23	18	44	47	6
gcdRecur	1421	20	19	332	620	732	685	144
Total	14526	335	342	3717	5010	6783	6966	1723

Table 2: The detailed breakdown of the algorithmic choices taken by SynFix.

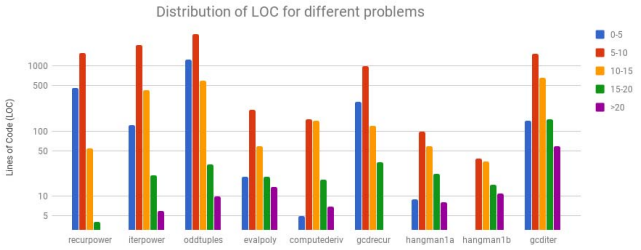


Figure 6: The distribution of lines of code (LOC) for different benchmark problems.

highest for recursive problems as compared to other problems that ask for imperative solutions.

A detailed breakdown of the choices taken by the SynFix algorithm is shown in Table 2. The table reports the number of cases for which the syntax errors were fixed by the predicted token sequences using different algorithmic choices: i) **Offset:** the prefix token sequence is constructed from the start of the program to the error token reported by the parser, ii) **Offset-1:** the prefix token sequence is constructed upto one token before the error token, iii) **PrevLine**: the prefix token sequence is constructed upto the previous line and the error line is replaced by the predicted token sequence, (iv) **Insert**: the predicted token sequence is inserted at the Offset location, and (v) **Replace**: the original tokens starting at

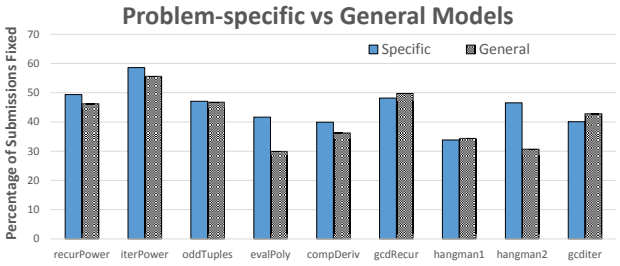


Figure 7: The performance of problem-specific models vs general models trained on all the problems.

the Offset location are replaced by the predicted token sequence. The table also reports the number of repairs that require 1 and 2 changes. We can observe that there is no one single choice that works better than every other choice. This motivates the need of the SynFix algorithm that tries all of the different algorithmic choices in the order of first finding an insertion or a replacement fix using the predicted token sequences of increasing length and then using the Previous Line method. We use this ranking order over the choices to prefer repairs that result in smaller changes to the original program.

First, we can observe that generating the prefix token sequences for querying the language model that end at one token earlier than the error token (**Offset-1**) performs significantly better than using prefix sequences that end at the error token (**Offset**). Second, the **PrevLine** choice performs the best compared to other choices. The reason for this is that the algorithm has more freedom to make larger changes to the original program by completely replacing the error line, but it may sometimes also lead to undesirable semantic changes. The Previous line changes are explored only after trying out the Insertion/Replace choices in the SynFix algorithm. The replacement of original tokens with the predicted token sequences performs a little better than the insertion choice. Finally, we observe that there are many student submissions that are fixed uniquely by each one of the 5 choices, and the algorithm therefore needs to consider all of the choices.

We also report the number of submissions that need 1 and 2 changes. A large fraction of submissions can be fixed using only 1 repair (6966), but there are still a significant number of submissions (1723) that are repaired by considering additional changes. We observe a small insignificant increase in the number of additional repaired programs when considering $f > 2$ number of changes.

4.3 General vs Problem Specific Models

We now present an experiment to evaluate whether we need to train separate token sequence models per problem or one global general model can perform equally well. For evaluating this question, we perform two sets of experiments. For the first experiment, we train a general model using the set of all correct submissions for all the benchmark problems, and compare its performance with that of problem-specific models that are trained individually only on the correct solutions for a given problem. For the second experiment, we train a model only on the correct submissions for a particular benchmark problem (recurPower) to correct the remaining problems, and compare its performance with the problem-specific models. The second experiment is performed to evaluate how well the token models learnt from one problem (with a large number of diverse solutions) generalize to other problems. The comparisons are performed based on the number of submissions that are syntactically fixed with one ($f = 1$) fix.

The results for the first experiment are shown in Figure 7. Overall, the general model performs comparable to the problem-specific models. In total, the general model fixes syntax errors for 6833 student submissions compared to 6966 fixed by the problem-specific models. Moreover, for two problems, gcdRecur and gcdIter, the general model performs a little better than the problem-specific models. This result shows that it might not be necessary to maintain and train separate models per problem, and a global general model trained on all the problems together can work almost as well in most of the cases.

For investigating how well the token sequence models learnt from one problem generalize to another problem, we perform the second experiment. We train the RNN model on the syntactically correct submissions for the recurPower problem and use it to perform corrections on submissions to other problems. The results for this experiment are shown in Figure 8(a). We can observe that the problem-specific model consistently fixes more number of syntax errors as compared to the recurPower language model. In total, the recurPower model fixes 5217 submissions whereas the problem-specific models repair about 14% more number of submissions (5944). In addition, we also empirically observe that the fixes generated by problem-specific models resulted in significantly higher semantically correct fixes in comparison to the fixes generated by the recurPower model. This result shows that for the RNN model to perform well on a larger number of submissions to a new problem, it needs to be trained on at least some correct programs from the new problem.

4.4 Comparison with N-gram Baseline

We compare the effectiveness of using an RNN model to learn token sequences over using an N-gram language model with N=5. The 5-gram model first queries the model with a prefix of 4 tokens and returns the next token if one exists. Otherwise, it falls back to 3 size token sequence and so on until it finds one in the frequency dictionary. We perform the standard add-1 smoothing on 5-gram models. The results are shown in Figure 8(b). The RNN model significantly outperforms the N-gram model consistently across all problems. The N-gram model in total can fix syntax errors for 4016 problems (27.65%), whereas the RNN language model can fix errors for 8689 problems (59.81%). The RNN language model is able to capture and generalize long term dependencies between tokens as compared to a small context learnt by the N-gram model.

5 RELATED WORK

In this section, we describe several related work on learning language models for Big Code, automated code grading approaches, and machine learning based grading techniques for other domains.

Neural Networks for syntax correction: DeepFix [12] uses an attention-based seq2seq model for learning a token model from a synthetic set of buggy programs. The learnt model in DeepFix first

Figure 8: (a) The performance of problem-specific vs model trained on recurPower. (b) Comparison of the performance of RNN token sequence model vs a 5-gram baseline.

predicts the buggy line in the program and replaces the complete line with the statement predicted by the model. Sk_p [22] uses a skip-gram neural network model to predict a program statement using the lines before and after the erroneous line. It enumerates over all lines in the program and their potential replacements until finding one program that is correct. Unlike these techniques that always replace the complete statement (line) in a student program, our system uses an iterative algorithm that is able to generate fine-grained token-level fixes to generate small repairs, which are more likely to correspond to student's intent. Moreover, our system uses constraint-based synthesis to perform semantic repairs and complement the syntactic repairs found by the RNN token model. Finally, the model in DeepFix is trained on a synthetic dataset obtained by mutating correct student submissions using an error model, whereas our token sequence model is trained directly on the correct submissions. The synthetic dataset generation not only requires knowing some error model upfront, but also the considered error model might not correspond to the distribution of real-world student errors.

Language Models for Big Code: Our work is inspired from the work on capturing naturalness (in terms of syntactic patterns) of code in large repositories [1, 3, 4, 13, 19, 24]. Unlike these techniques that analyze large code repositories, our system analyzes student submissions that are typically much smaller and therefore also more amenable to more expensive analyses such as enumerative and constraint-based synthesis for program correction. Hindle et al. [13] use an n-gram model to capture the regularity of local project-specific contexts in software and use it for next token suggestions. Nguyen et al. [19] extended this model to also include semantic token annotations that describe the token type and their semantic role. Allamanis et al. [1] showed that the n-gram models can significantly increase their predictive capabilities when learnt on a large corpus containing 350 million lines of code. Naturalize [3] is a language-independent framework that uses the n-gram language model to learn the coding convention from a project codebase and suggests revisions such as identifier names and formatting conventions to improve stylistic consistency. Allamanis et al. [4] recently presented a technique for suggesting method and class names from its body and methods respectively using a neural probabilistic language model. JSNice [24] uses structured prediction with CRFs for predicting identifier names and type annotation of variables in JavaScript programs.

Our technique is inspired from these previous works, but uses problem-specific RNN language models to compute fixes to syntax errors in student submissions. Campbell et al. [6] presented a technique to use unnaturalness of code with syntax errors to locate the actual source of errors. Ray et al. [23] recently showed that even syntactically correct buggy code are typically unnatural, which can then be used to assist bug repair methods. Our system currently uses the Python compiler to locate error locations, but these techniques can be used to complement and enhance our technique to increase the robustness of locating errors in submissions.

Code Grading and Feedback: The problem of providing feedback on programming assignments has seen a lot of recent interest. These approaches can be categorized into two broad categories – peer-grading [14, 15] and automated grading techniques [8, 17, 26, 27, 29]. A recently proposed approach uses neural networks to simultaneously embed both the precondition and postcondition spaces of a set of programs into a feature space, where programs are considered as linear maps on this space. The elements of the embedding matrix of a program are then used as code features for automatically propagating instructor feedback at scale [21]. However, these techniques rely on the ability to generate the program AST and cannot repair programs with syntax errors.

Machine learning for Grading in Other domains: There have been similar automated grading systems developed for domains other than programming such as Mathematics [16] and short answer questions [5]. The Mathematical Language Processing (MLP) [16] framework leverages solutions from large number of learners to evaluate correctness of student solutions to open response Mathematical questions. It first converts an open response to a set of numerical features, which are then used for clustering to uncover structures of correct, partially-correct, and incorrect solutions. A teacher assigns a grade/feedback to one solution in a cluster, which is then propagated to all solutions in the cluster. Basu et al. [5] present an approach to train a similarity metric between short answer responses to United States Citizenship Exam, which is then used to group the responses into clusters and subclusters for powergrading. The main difference between our technique and these techniques is that we use RNNs to learn a language model

for token sequences unlike machine learning based clustering approaches used by these techniques. Moreover, we focus on giving feedback on syntax errors whereas these techniques focus on semantic correctness of the student solutions.

Automated Program Repair: The research in automated program repair focuses on automatically modifying incorrect programs such that the modified program meets some desired specification [11]. GenProg [10] uses an extended form of genetic programming to search for a source-level patch by evolving a program variant to fix some defects in the original buggy program while maintaining functionality with respect to a set of test cases. SemFix [18] uses a combination of symbolic execution, constraint solving, and program synthesis to automatically derive repair expressions for a buggy program given a set of testcases. It first uses statistical fault localization techniques to locate the error location and then uses layered component based synthesis techniques to generate repair expressions of varying complexity. Qlose [8] also uses program synthesis techniques to perform automated repair by optimizing a multi-objective constraint of minimizing both syntactic and semantic distances between the original buggy program and the fixed program. These systems focus on fixing semantic bugs in programs by encoding and analyzing program ASTs, whereas our system focuses on repairing programs with syntax errors for which program ASTs can not be obtained.

Neural Program Synthesis: There are some recent neural architectures developed for program synthesis that learn to search over a discrete space of programs [9, 20]. Unlike learning to search over programs from scratch, we instead search over minimal modifications to student programs and use the prefix context in student submissions to guide the search more efficiently. We believe our technique of learning sequence models over corpus of real programs can complement program synthesis techniques as well.

6 LIMITATIONS AND FUTURE WORK

Our system currently only uses the prefix token sequence for predicting the potential token sequences for syntax repair. For the program shown in Figure 9, the SynFix algorithm suggests the fix corresponding to the expression exp==0. If the algorithm also took into account the token sequences following the error location such as return base, then it could have predicted a better fix corresponding to the token sequence exp == 1, and could have produced a semantically correct program. Our system currently uses Sketch to perform such semantic edits, but in future we would like to extend the RNNs to also encode the suffix sequences to potentially generate more semantic repairs. Another limitation of our system is that we currently depend on the Python interpreter to provide the error location for syntax errors, which is sufficient for most but not all cases. In future, we would like to train a separate neural model that also learns to predict such error locations in an end-to-end manner.

There is another interesting research question on how to best translate the generated repairs into good pedagogical feedback. Some syntax errors are simply typos such as mismatched parenthesis/operators, for which the feedback generated by our technique should be sufficient. But there are syntax errors that point to deeper misconceptions in the student's mind. Some examples of such errors include assigning to return keyword e.g. return = exp, performing an assignment inside a parameter value of a function call e.g. recurPower(base,exp-=1), etc. We plan to build a system on top of our technique that can first distinguish small errors from deeper misconception errors, and then translate the suggested repair fix accordingly so that students can learn the high-level concepts for correctly understanding the language syntax.

```
def recurPower(base, exp):
    if exp < 0:
        return 1
    if exp = 0:
        return base
    return base * recurPower(base, exp-1)
```

Figure 9: The SynFix algorithm suggests the fix exp == 0 using the prefix sequence.

7 THREATS TO VALIDITY

A threat to internal validity is that the syntax error repairs generated by the SynFix algorithm might not be natural or desirable, since the algorithm selects any repair that passes the compiler check. To mitigate this issue, we perform two steps. First, we use the Sketch solver to compute semantic repairs that reduces the potential chances to computing an undesirable syntax repair as it will likely preclude the corresponding semantic repair. Second, we also randomly sampled 200 syntactically-fixed programs for 2 assignments, and manually checked that 74% of the fixed programs corresponded to desirable syntax repairs. Another threat to internal validity is that we did not comprehensively evaluate all different neural network configurations and parameter values due to compute resource constraint. However, we sampled some configurations from the space using a limited parameter sweep. A threat to external validity of our results is that we have only evaluated our framework on small Python programs obtained from an introductory course, which might not be representative of other programming courses taught in different languages.

8 CONCLUSION

We presented a technique to combine RNNs with constraint-based synthesis to repair programs with syntax errors. For a programming assignment, our technique first uses the set of all syntactically correct student submissions to train an RNN for learning the token sequence model, and then uses the trained model to predict token sequences for finding syntax repairs for student submissions. It then uses constraint-based synthesis techniques to find minimal semantic repairs based on a set of rewrite rules. We show the effectiveness of our system on a large set of student submissions obtained from edX. We believe that this combination of RNNs with constraint-based synthesis can provide a basis for providing effective feedback on student programs with syntax errors.

REFERENCES

[1] Miltiadis Allamanis and Charles A. Sutton. Mining source code repositories at massive scale using language modeling. In *MSR*, pages 207–216, 2013.

[2] Miltiadis Allamanis and Charles A. Sutton. Mining idioms from source code. In *FSE*, pages 472–483, 2014.

[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Learning natural coding conventions. In *FSE*, pages 281–293, 2014.

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Suggesting accurate method and class names. In *FSE*, pages 38–49, 2015.

[5] Sumit Basu, Chuck Jacobs, and Lucy Vanderwende. Powergrading: a clustering approach to amplify human effort for short answer grading. *TACL*, 1:391–402, 2013.

[6] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In *MSR 2014*, pages 252–261, 2014.

[7] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems*, pages 577–585, 2015.

[8] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *CAV*, pages 383–401, 2016.

[9] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *ICML*, pages 990–998, 2017.

[10] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1): 54–72, 2012.

[11] Claire Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, September 2013.

[12] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, 2017.

[13] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *ICSE*, pages 837–847, 2012.

[14] Chinmay Kulkarni, Michael S. Bernstein, and Scott R. Klemmer. Peerstudio: Rapid peer feedback emphasizes revision and improves performance. In *Learning @ Scale,*, pages 75–84, 2015.

[15] Chinmay Eishan Kulkarni, Pang Wei Wei, Huy Le, Daniel Jin hao Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R. Klemmer. Peer and self assessment in massive online classes. *ACM Trans. Comput.-Hum. Interact.*, 20(6): 33, 2013.

[16] Andrew S. Lan, Divyanshu Vats, Andrew E. Waters, and Richard G. Baraniuk. Mathematical language processing: Automatic grading and feedback for open response mathematical questions. In *Learning@Scale*, pages 167–176, 2015.

[17] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas J. Guibas. Codewebs: scalable homework search for massive open online programming courses. In *WWW*, pages 491–502, 2014.

[18] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *ICSE*, pages 772–781, 2013.

[19] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[20] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *ICLR*, 2017.

[21] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. Learning program embeddings to propagate feedback on student code. In *ICML*, pages 1093–1102, 2015.

[22] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. *CoRR*, abs/1607.02902, 2016.

[23] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. In *ICSE*, pages 428–439, 2016.

[24] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *POPL*, pages 111–124, 2015.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, 1986.

[26] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. Question independent grading using machine learning: The case of computer program grading. In *KDD*, pages 263–272, 2016.

[27] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[28] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[29] Shashank Srikant and Varun Aggarwal. A system to grade computer programming skills using machine learning. In *KDD*, pages 1887–1896, 2014.

[30] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *ICML*, pages 1017–1024, 2011.

[31] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[32] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990.