Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks

Sahil Bhatia SAHILBHATIA.NSIT@GMAIL.COM
Netaji Subhas Institute of Technology, Delhi, India

Rishabh Singh RISIN@MICROSOFT.COM
Microsoft Research, Redmond, WA, USA

Abstract

We present a technique for providing feedback on syntax errors that uses Recurrent neural networks (RNNs) to model syntactically valid token sequences. Syntax errors constitute one of the largest classes of errors (34%) in our dataset of student submissions obtained from a MOOC course on edX. For a given programming assignment, we first learn an RNN to model all valid token sequences using the set of syntactically correct submissions. Then, for a student submission with syntax errors, we query the learnt RNN model with the prefix token sequence to predict token sequences that can fix the error by either replacing or inserting the predicted token sequence at the error location. We evaluate our technique on over $14,000$ student submissions with syntax errors.

1. Introduction

With the ever-increasing role of computing, there has been a tremendous growth in interest in learning programming and computing skills. The computer science enrollments in universities has been growing steadily and it is becoming more and more challenging to meet this increasing demand. Recently, several online education initiatives such as edX, Coursera, and Udacity have started providing Massive Open Online Courses (MOOCs) to tackle this challenge of providing quality education at scale that is easily accessible to students worldwide. One important drawback of MOOCs is that students typically do not get quality feedback for assignments since it is prohibitively expensive to hire enough instructors and teaching assistants to provide individual feedback to thousands of students. In this paper, we address the problem of providing automated feedback on *syntax errors* in programming assignments using machine learning techniques.

The problem of providing feedback on programming assignments at scale has seen a lot of interest lately. These approaches can be categorized into two broad categories – peer-grading (4) and automated grading techniques (8; 5). While providing feedback on functional and stylistic elements of student submissions is important, a significant fraction of submissions (more than 34% in our dataset) comprise of syntax errors and providing feedback on syntactic errors has largely been unexplored.

In this paper, we present a technique to automatically provide feedback on student programs with syntax errors leveraging the large dataset of correct student submissions. Our hypothesis is that even though there are thousands of student submissions, the diversity of solution strategies for a given problem is relatively small and the fixes to syntactic errors can be learnt from correct submissions. For a given programming problem, we use the set of student submissions without syntax errors to learn a sequence model of tokens, which is then used to hypothesize possible fixes to syntax errors in a student solution. Our system incorporates the suggested changes to the incorrect program and if the modified program passes the compiler syntax check, it provides those changes as possible fixes to the syntax error. We use a Recurrent Neural Network (RNN) (7) to learn the token sequence model that can learn large contextual dependencies between tokens.

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 (3; 6; 2; 1). Hindle et al. (3) 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. The NATU-

Appearing in Proceedings of the 2^{nd} Indian Workshop on Machine Learning, IIT Kanpur, India, 2016.

RALIZE framework (1) learns an n-gram language model for learning coding conventions and suggesting changes to increase the stylistic consistency of the code.

We evaluate the effectiveness of our system on over $14,000$ student submissions from an online introductory programming class. Our system can completely correct the syntax errors in 31.69% of the submissions and partially correct the errors in an additional 6.39% of the submissions.

2. Motivating Examples

We now present a few examples of the different types of syntax errors we encounter in student submissions from our dataset and the repair corrections our system is able to generate using the token sequence model learnt from the syntactically-correct student submissions. The example corrections are shown in Figure 1 for the student submissions for the recPower problem taken from the Introduction to Programming MOOC (6.00x) on edX. The recPower problem asks students to write a recursive Python program to compute the value of $\mathsf{base}^{\mathsf{exp}}$ given a real value base and an integer value exp as inputs.

A sample of syntax errors and the fixes generated by our algorithm (emphasized in boldface red font) based on different code transforamtions is shown in Figure 1. 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 locations for syntax errors, and then uses the program statements from the beginning of the function to the error location as the prefix token sequence for performing the prediction. However, there are many cases such as the ones shown in Figure 1(c) where the compiler is not able to accurately find 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 at the error location.

3. Approach

An overview of the workflow of our system is shown in Figure 2. For a given programming problem, we first use the set of all syntactically correct student submissions to train a neural network in the training phase for learning a token sequence model for all valid token sequences that is specific to the problem. We then use the SYNFIX algorithm to find small corrections to a student submission with syntax errors using the token sequences predicted from the learnt model. These corrections are then used for providing feedback in terms of potential fixes to the syntax errors. We now describe the two key phases in our workflow: i) the training phase, and ii) the SYNFIX algorithm.

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 3(a). The figure also shows the equations to compute the output probabilities for output tokens and the weights associated with connections from input to hidden layer, hidden to hidden layer, and hidden to output layer. 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 3(b). The first output token is predicted at the output layer using the input token 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.

The SYNFIX algorithm, shown in Algorithm. 1, takes as input a program P (with syntax errors) and a token sequence model $\mathcal{M}$, and returns either a fixed program P' (if possible) or ϕ denoting that the program cannot be fixed. The algorithm first uses a parser to obtain the type of error err and the token location where the error occurs loc, and computes a prefix of the token sequence $\widetilde{\mathrm{T}}_{prefix}$ 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 iteratively tries 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 fixed program P' with no syntax errors. If the algorithm cannot find a token sequence that can fix the syntax errors in the program P, the algorithm then creates another prefix $\widetilde{\mathrm{T}}_{prefix}$ 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 no syntax errors. If yes, it returns the fixed program P'. Otherwise, the algorithm returns ϕ denoting that no fix can be found for the syntax error in P.

(a) SyntaxError - Insert Token

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

```
def recPower(base, exp):
    if exp > 1:
        return base * recurPower(base, exp-1)
    else:
        return 1
```

(b) SyntaxError - Replace Token

```
def recurPower(base, exp):
    total = base
    if(exp==0):
        return total
    else:
        total*=base
        return total+recurPower(base,exp-=11)
```

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

(c) SyntaxError - Previous Line Insert

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

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

Figure 1. Some examples of the fixes suggested by our system for the recurPower student submissions with syntax errors taken from edX. The suggestions are emphasized in red using larger font, whereas the modified expressions are emphasized in blue.

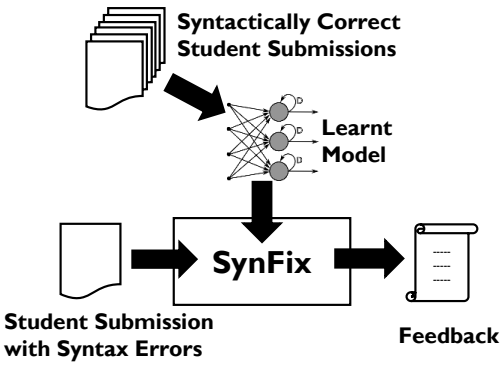


Figure 2. An overview of the workflow of our system.

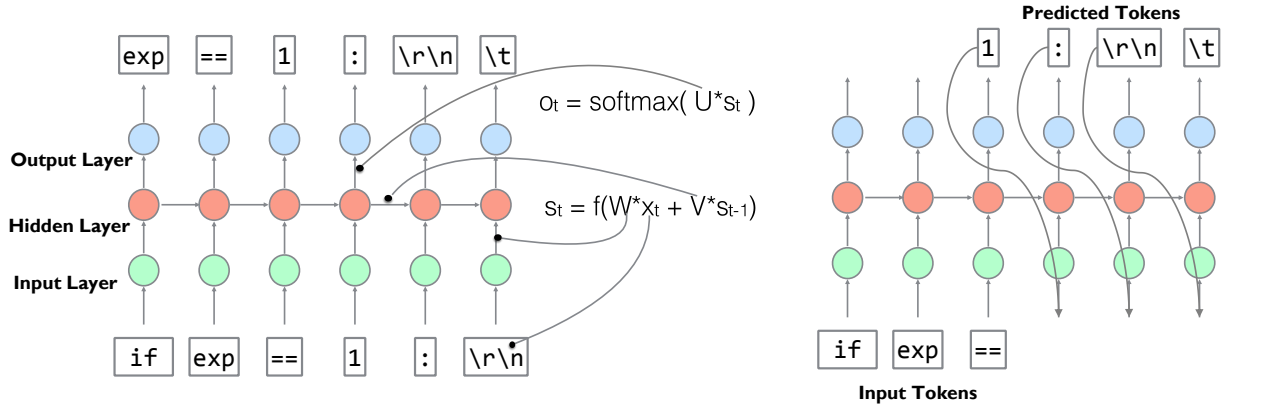
Algorithm 1 SYNFIX

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

4. Evaluation

We now present the evaluation of our system on $40,835$ Python submissions taken from the Introduction to Programming in Python course on the edX MOOC platform. Our benchmark set consists of student submissions to five programming problems recurPower, iterPower, oddTuples, evalPoly, and compDeriv taken from the edX course.

We first present the overall results of our system in terms of how many student submissions are corrected using the predicted tokens in Table 1. Since our algorithm currently considers only one syntax error in a student submission and there are many submissions with multiple syntax errors, we also report the number of cases where the suggested correction fixes the first syntax error but the submission isn't completely fixed because of other errors. We call this class of programs as *Fixed(Other)*. In total, our system is able to provide suggestions to completely fix the syntax error in 31.69% of the cases. Additionally, it is able to fix the first syntax error on a given error line without fixing other syntax errors on future lines in 6.39% of the cases. The system



(a) Training Phase (b) Prediction Phase

Figure 3. 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.

Problem	Incorrect Attempts	Completely Fixed	Fixed (Other)
recurPower	2071	1061 (51.23%)	281 (13.57%)
iterPower	2661	1599 (60%)	276 (10.37%)
oddTuples	8824	1575 (17.85%)	303 (3.43%)
evalPoly	324	131 (40.43%)	38 (11.73%)
compDeriv	323	135 (41.79%)	10 (3.09%)
Total	**14203**	**4501 (31.69%)**	**908 (6.39%)**

Table 1. The number of submissions that are completely fixed and partially fixed (error in another line) by our system using the LSTM-(2,128) neural network.

isn't able to provide any fix to the errors for the remaining 61.92% of the submissions. The number of programs that are completely and partially fixed for each individual problem is also shown in the table.

5. Limitations and Future Work

There are several limitations in the presented algorithm that we would like to extend in future work. One limitation of our technique is that it currently handles only one syntax error in the student program. We plan to extend our algorithm to also handle multiple syntax errors by automating the process of fixing the first syntax error found in the program using the SYNFIX algorithm and then calling the algorithm again recursively on the next error found in the updated program. We would like to build a system on top of our technique that can first distinguish small syntax 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.

References

[1] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton. Learning natural coding conventions. In *FSE*, 2014.

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

[3] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In *ICSE*, pages 837–847, 2012.

[4] C. E. Kulkarni, P. W. Wei, H. Le, D. J. hao Chia, K. Papadopoulos, J. Cheng, D. Koller, and S. R. Klemmer. Peer and self assessment in massive online classes. *TOCHI*, 20(6):33, 2013.

[5] A. Nguyen, C. Piech, J. Huang, and L. J. Guibas. Codewebs: scalable homework search for massive open online programming courses. In *WWW*, pages 491–502, 2014.

[6] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. FSE, 2013.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, 1986.

[8] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.