

Syntax Error Detection and Correction in Python Code using ML

Dr. Deipali Vikram Gore¹, Mahima Binoj², Sayali Borate³, Ritu Devnani⁴ and Sakshi Gopale⁵

¹⁻⁵PES Modern College of Engineering, Department of Computer Engineering, Pune, India

Email: {deipali.gore, mahima.binoj, sayali.borate, ritu.devnani, sakshi.gopale}@moderncoe.edu.in

Abstract—A lot of methodologies have been developed to automatically localize and correct a code written in certain programming languages. Requirements of such automatic program repair for senior programmers allow them to utilize their time effectively on the logic of the code and not in manually localizing the syntactic bugs within the code. Machine Learning models hence are one of the major contender technologies being used for achieving the goal. Hence, a brief walk through various developments in syntactic correction would help in noticing the efforts made by the industry. Many researchers from Microsoft have also contributed to this development. This list comprises models developed using OpenAI Codex, the encoder decoder transformer, recurrent neural networks or extending the spelling correction algorithms for the target purpose. An inspiration from all these allowed us to build a model to correct the syntax for the Python codes. Our model combines the two different techniques, a model for generating real world buggy Python codes to enhance the accuracy and prefix and suffix consideration for prediction of the next expected token. A practical usage for the programmers, the extension of the model for various other highly used programming languages in future would have varied applications.

Index Terms— Machine Learning, Encoder Decoder Transformer, Long Short Term Memory model, Recurrent Neural Networks, syntax error.

I. INTRODUCTION

A. What is a Syntax Error?

A syntax error is an error in the syntax of a coding or programming language, entered by a programmer. Syntax errors are caught by a software program called a compiler, and the programmer must fix them before the program is compiled and then run. Though caught by the compiler, syntax errors are hard to understand to novice programmers.

B. Types of Syntax Error

- Indentation Error
- Unbalanced Parenthesis
- Missing symbols
- Misspelled keywords

C. Our Contribution

Correcting syntax errors indicated by the compiler is time-consuming and frustrating. Sometimes, the location indicated by the compiler is not the actual error location. Hence, to avoid the amount of time wasted in correcting syntax errors can be reduced if the process of correcting syntax errors is automated. In this article, we propose a deep learning-based approach to detect and correct syntax errors automatically. We use the encoder-decoder transformer which consists of LSTM(Long Short Term Memory) cells. LSTM is an extension of RNN(Recurrent Neural Networks). RNNs are capable of remembering the sequence of previous data which is essential in predicting the next token. LSTM cells are capable of remembering the sequence of tokens for a longer amount of time as compared to RNNs. This is essential in correcting the unbalanced parenthesis errors in codes.

II. LITERATURE REVIEW

In this section, we discuss the existing literature and recent advancements in syntax error detection and correction.

A. Synfix

In this paper, they have proposed a novel Neuro-symbolic approach that combines neural networks with constraint-based reasoning. Specifically, it first uses a Recurrent Neural Network (RNN) to perform syntax repairs for the buggy programs; then, 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. They have presented a technique to combine RNNs with constraint-based synthesis to repair programs with syntax errors. For a programming assignment, the 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. They show the effectiveness of the system on a large set of student submissions obtained from edX.

B. Deepfix

Fixing Common C Language Errors by Deep Learning was another paper referred to. It presents an end-to-end solution that does not rely on any external tool to locate or fix errors. It only uses a compiler to validate DeepFix's suggested fixes. DeepFix is powered by a multi-layered sequence-to-sequence neural network with attention which consists of an encoder recurrent neural network (RNN) for processing input and a decoder RNN for generating output. The network has been trained to predict an incorrect programme location as well as the correct statement. DeepFix uses it iteratively to correct multiple errors in the programme one at a time. DeepFix can be used to solve any previously unknown programming task. Unlike DeepFix, neural networks used in some approaches are incapable of localizing errors on their own. Bhatia and Singh used compiler-involved prefixes to perform a brute force, enumerative search for error localization. DeepFix captures long-term dependencies using an attention-based sequence-to-sequence model. The result was, DeepFix corrects multiple errors by invoking a trained neural network iteratively.

C. Open AI's Codex

An Artificial Intelligence research company, OpenAI has developed Codex which is a machine learning model and a descendant of Generative Pre trained Transformer. Codex is hence pre trained with abundant data extracted from the internet and it returns a code written in a programming language that a user wants based on the input is natural language. There have been varied applications of this model which have been possible because of the availability of about 175 billion parameters for the model. Having such capabilities and easy access through a prompt, Codex was evaluated for the syntax correction. The methodology indicated the access of the directly available model without its fine tuning for the target task and the input being provided with a lot of variations. The variety of input variations included - a buggy code, a buggy code with doc strings containing the correct functioning of the code, codes with some hints in the comments, correct code. Several parameters, such as the maximum number of tokens, stopping criteria, etc were provided to the model along with the input. The model was then evaluated by the comparison of the output with expected codes or through test cases available. The performance was notable but several factors such as the hints within comments were not highly effective. The

results yielded without fine tuning for the target tasks were quite effective but fine tuning would further enhance its capabilities.

D. CoEdit

A novel error correction mechanism in compilers using spelling correction algorithms was one of the papers referred to. In that, the misspelling correction process necessitates the use of a dictionary to compare the incorrect word to find a match. In most cases, the dictionary includes a list of proper words. Proper words are those that are defined in a specific natural language. The Java programming language is used as an example in this work. As a result, a dictionary containing words useful in this work, such as Java keywords, predefined methods, predefined variables, abstracts, and packages, is required. Furthermore, users can define their own variables and methods, which should be included in the dictionary. Because these differ from one user to the next, they are only included when a specific source code is run. As a result, another method was used to compile a list of top misspelled English words, according to several English dictionaries such as Oxford Dictionary. These resources provide the correct form of commonly misspelled words. This enables it to be used as real-world test data. To investigate the behavior of the algorithms used, the test data is divided into several categories. Three different types of experiments are carried out: the first compares the Four Way algorithm to the Editex algorithm without including the CoEdit algorithm. The second algorithm is used to demonstrate the behavior of CoEdit when it is implemented with the Four Way algorithm as well as the Editex algorithm. The third and final experiment investigates the effect of the size of the candidate list used in the CoEdit approach on accuracy and execution time. The experiment concluded that when the candidate size is 10, CoEdit with Editex produced the best trade-off between accuracy and time, with 97.67% and 0.96 s, respectively.

E. GrammarGuru

In [1], Santos et. al. introduce a model, GrammarGuru, which is based on RNNs. GrammarGuru is used to correct errors in JavaScript. It proposes an approach that uses LSTM, an extension of RNN which is capable of remembering the context of the sequence of tokens for longer sequences. The input to LSTMs is bad code (i.e. code containing syntax errors) and the output is the corresponding good code. The model employs two LSTM networks working in opposite directions but trained on the same data:

1. Forward LSTM which predicts a token based on the prefix sequence
2. Backward LSTM which predicts a token based on the suffix sequence.

Using the predictions of the two language models, GrammarGuru suggests the fixes for single-token syntax errors. Data extracted from Github is used to train and test the model. The results show that the model predicts the correct location of the syntax error 54.74% in its top 4 suggestions and produces an exact fix up to 35.50% of the time. The drawback of this model is that it can fix only one syntax error per code file.

F. Break-It-Fix-It

The break-it-fix-it model or the BIFI-model consists of the critic, the breaker and the fixer. The critic acts as a compiler that assesses the quality of the input of code whereas the aim of the fixer is to convert a code with syntax errors to one without syntax errors i.e. bad code to good code. Input to be given will be unlabeled data and a critic that acts as a compiler to assess if the code has errors. The task of the fixer is to repair the code or in other words fix the syntax errors in the bad code. The BIFI model uses a dataset with both good and bad codes to train the breaker and fixer accordingly. Both the breaker and fixer are used iteratively to generate more synthetically produced paired data. BIFI-model can also be used for turning raw unlabeled data into usable paired data with the help of a critic. It can be used in areas such as molecular design, text editing and machine translation.

III. OUR APPROACH

Our approach is based on the combination of the models proposed in the following two papers:

1. Finding and correcting syntax errors using recurrent neural networks by Eddie Antonio Santos, Joshua Charles Campbell, Abram Hindle, and Jose Nelson Amaral
2. Break-It-Fix-It(BIFI): Unsupervised Learning for Program Repair by Michihiro Yasunaga & Percy Liang

[1] suggests an approach that takes into account the prefix and suffix content and generates possible fixes for the same sequence of words. [2] uses an encoder-decoder model and proposes using a machine learning-based approach to generate realistic, bad code instead of introducing random errors. Combining these two techniques improves model accuracy by providing two distinct ways to improve the model. We build three encoder-decoder

models: a forward model to generate the next token based on prefix sequence, a backward model to generate the next token based on suffix sequence and a model to generate bad code. The forward and backward models (Correctors) are used for detecting and correcting the syntax errors in python codes. The input for these two models is bad code and the output is the corresponding bad code. The third encoder-decoder model (Code Corruptor) used, takes good code as input and introduces errors in this code which are similar to the errors made by real-life programmers. Fig 1 shows the workflow of the methodology.

We use the Python dataset used in [2] for training and testing the model. Training the encoder-decoder models requires a pair of good and bad code. Initially, synthetic mutation methods used in [4], [5] are applied to good code(i.e. code without syntax errors) for generating bad code. The Correctors will be trained on this data. For training the Code Corruptor, the output returned by the corrector is used as the training dataset.

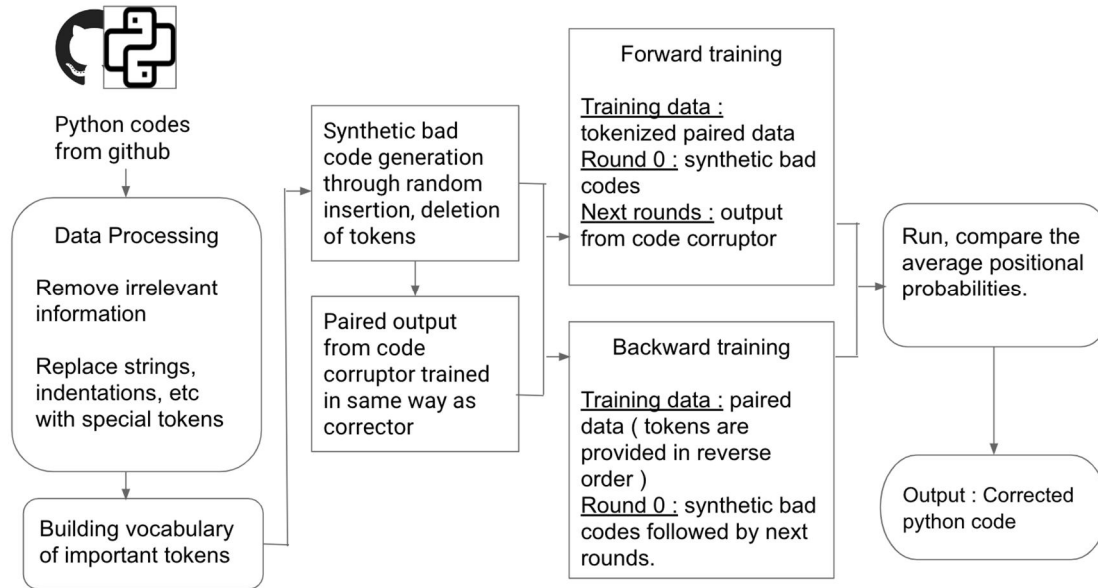


Figure 1. Workflow of the model

This cycle of training can be repeated to enhance the accuracy. The test dataset (syntactically incorrect Python code) can then be applied to the corrector in the testing phase..

Also, the model is able to fix multiple syntax errors. This removes the drawback of [1], which is able to correct single-token errors.

A. Methodology

1. **Data Collection:**
Collect syntactically correct Python codes from public Github repositories. These codes have been collected for training the correctors.
2. **Data Preprocessing:**
Clean and pre-process the data, removing any irrelevant information. Replace strings, variables, indentation with special tokens. Eg: Replace a string with special token <STRING>, indentation space with <INDENT>, variables with <unk>.
3. **Build a vocabulary:**
Vocabulary consists of the commonly used tokens in a language. A python vocabulary consists of python keywords(if, for, import) and symbols(“ , (, [)
4. **Generate bad codes:**
Syntactically incorrect codes are generated by performing mutation in syntactically correct codes. Mutation can be done using[4][5]:
 - **Random Deletion:** a token chosen at random from the input source file is deleted.
 - **Random Replacement:** a token was chosen at ran-dom and replaced with a random token found from vocabulary.
 - **Random Insertion:** a location in the source file is chosen at random and a random token found is inserted there.

5. Train and Run the Correctors:

An encoder-decoder transformer requires paired data. For the forward and backward corrector, a pair of bad and good code is given as input. We use the Fairseq toolkit for training the models. Fairseq is a sequence modeling toolkit for training custom models for translation, summarization, and other text generation tasks. After training the correctors, we provide bad code as input to these models and the output is the corresponding good code.

6. Train and Run the Code Corruptor:

Same as Corrector training, a pair of good and bad code is provided as input to the encoder-decoder. After training the code corruptor, we provide good code as input to this model and the output is the corresponding bad code generated by introducing random errors.

The process of training and testing the correctors and code corruptor is repeated until the required accuracy is obtained. The code from correctors is given as input to code corruptors and vice versa.

B. Evaluation Matrix

Accuracy of the model will be calculated to evaluate the model. Syntactical correctness of code will be ascertained by the compiler. Accuracy will be calculated as the ratio of corrected codes to total codes.

IV. RESULTS

The forward corrector model, the backward corrector model and the corruptor model have been used for the purpose of syntax correction and there is about 76% accuracy for the overall model built. The main idea behind these three models within the main model is that the data generated by one model will be utilized further for the next model and various rounds of this cyclic process are continued to enhance the accuracy. The cyclic process initially starts with the data collected from the Github repository in this case. After the first training of both correctors, new paired data(good and bad code) is generated by the corrector. After generation of paired data, the corruptor is trained. The corruptor generates bad codes from good codes. These bad codes are closer to real life coding errors done by programmers. After training the corruptor, the corruptor is used to generate the paired data for the corrector.

V. CONCLUSIONS

We presented a deep learning-based approach to detect and correct syntax errors automatically for which we used a combination of two techniques which consists of the forward and backward model and the encoder-decoder model to repair programs with syntactic errors. The two models are combined for generating real world buggy Python codes to enhance the accuracy and prefix and suffix consideration for prediction of the next expected token. We collected correct Python codes from GitHub which we used as datasets for building a vocabulary and for training. The Corrector and Code Corruptor are trained and run iteratively until the required accuracy level is obtained. We believe that this can potentially be a very powerful and useful tool for both novice and experienced programmers alike.

VI. FUTURE WORK

Currently, the system is unable to correct keyword errors. The wrongly written keywords are considered as variables and hence are replaced with the token <unk> during tokenization. Before replacement, the wrongly written keywords can be recognized to remove this drawback. This could be done by using Levenshtein's distance formula. Another drawback of the system is its inability to correct long codes. The length of the codes being corrected by the model depends on the length and the type of the codes used for training. Hence, this drawback can be removed by changing the training data.

REFERENCES

- [1] Eddie Antonio Santos, Joshua Charles Campbell, Abram Hindle and Jose Nelson Amaral, "Finding and correcting syntax errors using recurrent neural networks" from PeerJ Preprints during August, 2017.
- [2] Michihiro Yasunaga and Percy Liang, "Break-It-Fix-It: Unsupervised Learning for Program Repair" on the proceedings of the 38th International Conference on Machine Learning, PMLR 139,2021.
- [3] Julian Aron Prenner, Romain Robbes, "Automatic Program Repair with OpenAI's Codex- Evaluating QuixBugs" from arXiv e-prints on 6 November, 2021.

- [4] Rahul Gupta, Soham Pal, Aditya Kanade, Shirish Shevade, “DeepFix: Fixing Common C Language Errors by Deep Learning” on the proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)
- [5] Michihiro Yasunaga, Percy Liang, “Graph-based, Self-Supervised Program Repair from Diagnostic Feedback” on the proceedings of the 37 th International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.
- [6] Sahil Bhatia, Pushmeet Kohli and Rishabh Singh, “Neuro-Symbolic Program Corrector for Introductory Programming Assignments” in 2018 ACM/IEEE 40th International Conference on Software Engineering.
- [7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser and Illia Polosukhin. “Attention is All You Need” presented at the 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.
- [8] Abdulelah Alwabel, “CoEdit: A novel error correction mechanism in compilers using spelling correction algorithms”, Journal of King Saud University – Computer and Information Sciences.