Finding Syntax Errors using Deep Learning

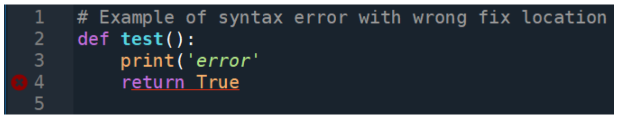
Project Report for Analyzing Software using Deep Learning

ABSTRACT

The goal of this project is to design and build a neural network-based analysis tool, that is able to correctly identify and fix syntax errors in simplified Python source code. Our approach for this project is a tool that is composed of three models each of which solve one subtask that is required to fix syntax errors. Those tasks are locating the error, predicting the necessary action to solve the error, and if necessary predict the token that needs to be inserted. Combining these models results in an analysis tool that is able to fix syntax errors with high accuracy.

1 INTRODUCTION

Syntax errors prevent programs from being compiled or interpreted correctly, thus they slow down program development. On top of that, finding syntactic errors is not always simple, especially in complex source code. Even though modern compilers and IDE's try to present the error location to the developer, this is not always easy and intuitive for the developer.



```python
# Example of syntax error with wrong fix location
def test():
    print('error'
    return True
```

1: Python code snippet that shows wrong error location.

One example of such a presentation can be seen in Fig. 1. As can be seen, the error is a missing closing parenthesis on line 3, however the IDE (in this example Spyder [1] v4.2.5) marks and underlines the line below that. In more complex cases, errors like that can consume a lot of time.

1.1 Project Description

This project aims to make a step towards solving the just described problem. The goal is to design, implement, and evaluate a neural network-based program analysis tool, which localizes syntactic errors in Python source code and tries to provide a fixed version.

To make this task easier we make two simplifications. Firstly, we only consider a simplified version of Python source code, i.e., all literals are replaced by the token 'LIT', all comments are replaced by the token '#COMMENT', and all identifiers (variable names, function names, etc.) are replaced by the token 'ID'. This simplification solves the vocabulary problem, since all tokens that are provided by developers are being replaced by fixed tokens. Thus, the vocabulary becomes a relatively small finite set of tokens. Examples of such simplified Python source code can be seen in Fig. 2.

Additionally, for the scope of this project, we only differentiate between three different types of syntax errors: a single missing token, a single superfluous token, and a single incorrect token. Especially, this means that syntax errors that consist of more than one erroneous token are not considered.

1.2 Project Data

In order to work on this project we are provided with the data necessary for training the neural network-based model. This means we do not have to collect, aggregate, and process data, which is a tremendous part of the work of training a neural network-based model, but this also means we have no control over the training data.

The provided training data consists of 100 JSON files, each JSON file consists of 500 samples, resulting in a total of 50,000 training samples. Each sample is a Python function with additional information. The structure of the samples can be seen in Fig. 2. The samples consist of multiple fields:

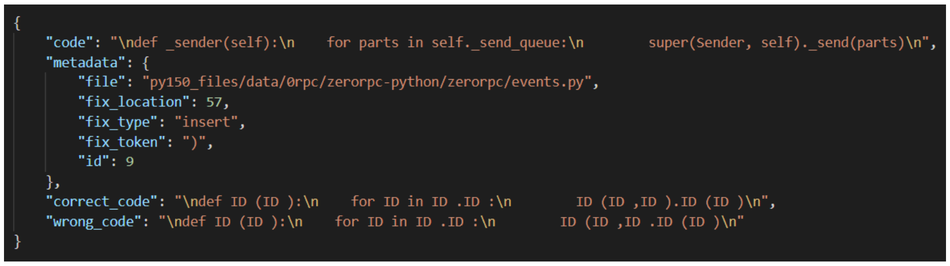
- code - the original source code.
- metadata - additional data about the sample, consisting of the name of the file, the fix location, the fix type, the fix token and a unique ID.
- correct_code - the abstracted version of the source code.
- wrong_code - the abstracted version of the source code, mutated to contain one syntax error of a previously described error type.

Next we want to talk about the attributes fix_location, fix_type, and fix_token in more detail. fix_location is the character location of the syntax error present in wrong_code. Note that every character is counted including whitespace, also note that '\n' counts as one character. The fix_type attribute has three possible values corresponding to error types mentioned earlier. The three types are 'insert', 'delete', and 'modify'. The fix_token attribute is only present when the fix_type is not 'delete', this attribute specifies the token that needs to be inserted or modified to in order to fix the syntax error.

2 APPROACH

In this section we talk about how we solve the task of this project. For this we split it into three subtasks. First, we try to localize the syntax error, i.e., find the exact error location in source code. Then, we try to predict the type of the error, i.e., how to fix the error ('insert', 'delete', and 'modify'). Lastly, in case the fix type is 'insert' or 'modify', we try to predict the token that fixes the syntax error. This means we train a model for each subtask instead of training one model that tries to solve all subtasks at once.

Before we describe how we solve the individual subtasks, we show characteristics that they have in common. In order to represent the code in a way that the models can work with, we split the code into tokens, ignoring spaces but preserving things like newlines and indents as they carry a syntactical meaning in the Python programming language. In total our vocabulary consists of 86 tokens (including two special tokens, which we will describe



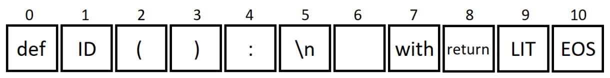
```
{
    "code": "\ndef _sender(self):\n    for parts in self._send_queue:\n        super(Sender, self)._send(parts)\n",
    "metadata": {
        "file": "py150_files/data/0rpc/zerorpc-python/zerorpc/events.py",
        "fix_location": 57,
        "fix_type": "insert",
        "fix_token": ")",
        "id": 9
    },
    "correct_code": "\ndef ID (ID ):\n    for ID in ID .ID :\n        ID (ID ,ID ).ID (ID )\n",
    "wrong_code": "\ndef ID (ID ):\n    for ID in ID .ID :\n        ID (ID ,ID .ID (ID )\n"
}
```

2: Structure of training samples of provided data.

later). We obtained the vocabulary by tokenizing every sample of the training data, while also looking at the documentation of Python keywords and operators [2], [3]. Furthermore, we perform one-hot-encoding to transform every token to a tensor. Note that unknown tokens, i.e., tokens that are not part of our vocabulary, are mapped to a zero tensor.

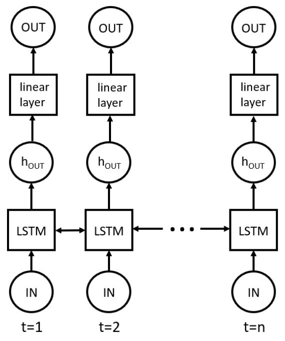
2.1 Fix Location

The goal of this subtask is to predict the location of the syntax error. As previously mentioned the training data contains the location of the syntax error on a character level, meaning that it represents the number of characters after which the error is located. However, as just described our models operate on a token level. This means that working with the fix location is more complicated for the models, since the exact location on a character level is also dependent on whitespace, but this is omitted in our token representation. To solve this problem, we transform the fix location from a character level to a token level, such that a value of 7 means that the token at position 7 causes the syntax error. The details on how we perform this transformation can be found in the source code of this project. However, there is one problem remaining, namely what happens if the syntax error is that at the very end of the code a token is missing? This would mean that the code up to the final token position is correct and the expected value would be the token position after that, however this position does not exist because the sequence of tokens has ended. To solve this problem, we introduce the first special token 'EOS', which represents the end of the sequence of tokens and is always appended to end of the tokens. An example of the token representation can be seen in Fig. 3.



3: Token representation of source code. Note that token at position 6 is an indent (four spaces).

Now that we know how to represent the code and the fix location, we can continue and explain the architecture of our model that is responsible for predicting the fix location. A simplified representation of this model can be found in Fig. 4. Since the input data for our model is a sequence, it makes sense to use a recurrent neural network (RNN). In particular, we use a Long Short-Term Memory [4] (LSTM) model, a special kind of RNN that is capable of storing "long-term" information. This property makes it especially well suited for working with languages [5]. For the subtask of finding the fix location, we use a many-to-many approach, meaning that we use the output of every time step of the LSTM. Note that one time step corresponds to one token of the input sequence. So for the example seen in Fig. 3, the source code has been transformed to eleven tokens, which means the LSTM will have eleven outputs one for each token. This also means that the number of outputs of the LSTM step is variable and depends on the number of tokens the input source code produces. After the input has passed through the



4: Simplified representation of the model used for predicting the error location.

LSTM, we send each output through a fully connected linear layer to map it to a single value. This final single value represents the likelihood that this token position is the error location. To achieve this desired behaviour we use the Cross-Entropy-Loss as our loss function[6].

The only missing detail now is that we do not use a simple single direction LSTM, instead we use a bidirectional LSTM, meaning that we process the data once from left to right and once from right to left. To understand why we use a bidirectional LSTM, we again use the example in Fig. 3. here the error is a superfluous token 'with' at position 7. However, if we go through the sequence token by token from left to right, by the time we are at position 7 we have no way of telling that this is indeed the position the syntax error is located. For all we know the sequence could continue in a way such that the 'with' token is syntactically correct. But, given the information of which tokens occur after the token at position 7, it is clear that this is in fact the error position. So to make sure our model is also able to deal with such cases, we provide it with information from both directions by using a bidirectional LSTM.

2.2 Fix Type

Now that we have a model for predicting the fix location, the next subtask we need to solve is predicting the fix type. The overall architecture of the model for predicting the fix type remains almost the same as for predicting the fix location. We still use a bidirectional LSTM as the core of the model, but this time we use a many-to-one approach meaning that we only use the output of the final time step of the LSTM for further processing. The idea is that the final time step contains information about the complete sequence, thus is able to infer the correct fix type. Similar to the previous model, we again use a fully connected linear layer to map the output of the LSTM to the final output. This time the output is a tensor of three values, such that each of the three fix types can be represented. Each value corresponds to one fix type and indicates how likely it is, that this is the action that is required to fix the syntax error. Unlike the model for the first subtask, in this model the data is sent through a drop-out layer before being send through the fully connected linear layer, this is done to reduce the effect of overfitting.

So far we have not used the additional information we gained from the previous subtask, namely the position of the fix location. In an effort to also make the fix position available to our model, we change the input, i.e., the sequence of tokens in a way such that it also includes this information. For this reason, we introduce the second special token called 'FIX_ME'. This token is simply inserted at the predicted fix location. The idea behind this is that the model learns that this token carries a special meaning and puts extra emphasis around the position of this token.

2.3 Fix Token

For the last subtask, finding the fix token, we adjust the input in way such that it reflects the previous acquired knowledge, similar to the previous subtask. To achieve this we again insert the special 'FIX_ME' token at the fix location. However, how exactly the special token is inserted depends on the previously predicted fix type. For this we only consider the cases where the fix type is either 'insert' or 'modify', since if the fix type is 'delete' we do not need to predict a fix token.

If the fix type is 'insert' it means that no token of the sequence is wrong, rather a token is missing. In this case we simply insert 'FIX_ME' at the fix location exactly like it is done in the previous subtask. But if the fix type is 'modify', it means that the token at the fix location is erroneous and needs to replaced by a different token. So in this case instead of simply inserting the special token, we replace the erroneous token with 'FIX_ME'. By doing this we simplified both cases to a slot filling problem, i.e., we have sequence with a blank (in our case the 'FIX_ME' token) and need to find the correct token to insert into this blank. We can then view this as a classification problem by trying to classify the token of our vocabulary that needs to be inserted into the blank. This is only possible thanks to the simplification of the source code, which leads to a small enough vocabulary.

The overall architecture of the model is identical to the one for finding the fix type. The only difference is how the input is changed to reflect the known information and the size of the final output of the fully connected linear layer. The final output now contains 86 values one for each token of the vocabulary, similar to the previous subtasks the value indicates the likelihood of this token to be the token that needs to be inserted into the slot in order to fix the syntax error.

3 IMPLEMENTATION

In this section we briefly describe the implementation. We focus in particular on how the training is implemented.

The models described in the previous section are implemented using Python version 3.8 and Pytorch version 1.8.1. No further external libraries are used besides Pytorch.

We train each model separately. For calculating the loss we use the correct data from the training samples, e.g., for training the model that predicts the fix type we use the correct fix location present in the training data as opposed to a fix location that has been predicted by another model (e.g. the model for predicting the fix location). The reason for this is to decrease the number of incorrect samples part of the training data, even though in practice the models only have access to previously predicted data. Another added benefit of using only the correct data from the training samples is that training each model can happen independent from each other. This means that the order in which we train the models does not matter.

Since the training of each model is almost identical we only describe the training process once, but point out the differences between the models. The first step is loading the data, in total the provided dataset contains 50,000 samples, however, we split the data into training and evaluation data, with a ratio of 7:3. So we end up with 35,000 samples for training and 15,000 samples for evaluation. Keep in mind that for training and evaluating the fix token prediction model, we can only use samples whose fix type is not 'delete', so we have less data for this model. We train each model for 20 epochs, meaning that we go through all training samples 20 times. Note that we shuffle the training data after each epoch, in an effort to avoid the model inferring information from the order of samples.

The parameters of the models are updated after every sample. Because of this, we need a low learning rate to avoid large jumps. The model for predicting the fix type and the one for predicting the fix token use the Adam optimizer [7] to update the parameters, while the fix location model simply adds the gradient multiplied with the learning rates to its parameters. The learning rate for the models which use Adam is 0.001, while the learning rate for the model which uses simple gradient descent is 0.005.

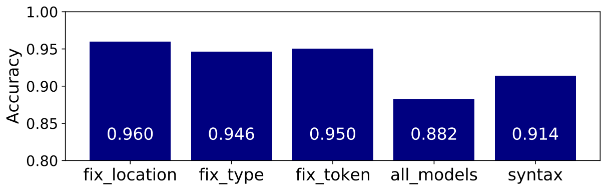
We set the size of the hidden layer to 128 across all models, but note that since we use bidirectional LSTM's, the number of values in the hidden layer is twice as large.

We trained the models on a machine with a NVIDIA GeForce RTX 2060 GPU, an Intel(R) Core(TM) i7-8750H @ 2.20GHz CPU, and 8.00GB of RAM. The total time required for training using the GPU is 3 hours and 12 minutes, from which 59 minutes are used for training the fix location model, 1 hour and 26 minutes are used for training the fix type model, and 47 minutes are used for training the model for predicting the fix token. Note that the amount of time it takes for training is strongly dependent on the state of the machine the training is running on, so training time might vary a lot.

4 RESULTS

The final section of this report focuses on the results obtained from evaluating the models independently and together. For this we first look at the prediction accuracy of the models and afterwards we look at samples that provide interesting results.

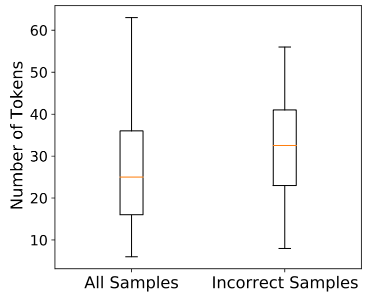
As previously mentioned we split the data into data for training and data for evaluation, so the evaluation of the models is done with data the models have never seen before. The resulting accuracy of our models is shown in Fig. 5. The values for the models are



5: Bar chart showing the accuracy of the different models.

obtained by evaluating the models in isolation. This is done to get an idea on how well each model is able to solve its assigned task. As can be seen the accuracy of the models is high, with the lowest accuracy being 94.6% for the fix type model. Note that for evaluation we only check whether the predicted result matches the expected result provided by the evaluation samples, however there might be more than one correct way to fix the syntax error. Additionally, we evaluate how well the models perform together, i.e., each model uses the output from the previous model as their input (as it is done in practice). The result of this can be seen in Fig. 5 under 'all_models'. The last bar seen in Fig. 5 shows the accuracy of the combined models only considering whether the resulting predicted code, i.e., the sample code with the syntax error fixed, is indeed syntactically correct now. However, fixing the syntax error is not easy even with the correct fix location, the correct fix type, and the correct fix token. This is because we fix the syntax error on a token level, but ignoring whitespace can lead to syntactical errors. Nonetheless, our implementation tries to account for this.

Next we show where our models make mistakes. In total 1762 out of the 15,000 samples we use for evaluation, are predicted wrongly at one step (i.e. the prediction does not match the expected one from the evaluation data). Out of those 745 are incorrect because the first model predicted the fix location wrongly. The model for predicting the fix type is accounting for 696 of the incorrect samples and the remaining 321 wrong samples are incorrect because a wrong fix token is predicted even though both the fix location and the fix type has been correctly predicted for those cases. Additionally, we want to take a closer look at the samples that have been predicted wrongly. From the incorrect samples 491 have an expected fix type of 'insert', 731 have an expected fix type of 'modify', and 540 incorrect samples have a fix type of 'delete'. So the fix types are evenly distributed which means, that not one fix type is problematic and causes most of the incorrect results. Furthermore, we take a look at the length of the code to see if this impacts the accuracy of our models. For this we provide two boxplots in Fig. 6, one that shows the length distribution across all evaluation data and one that shows it across the incorrect samples. As can be seen in Fig. 6, the samples that are predicted wrongly tend to be longer. We also look at if it makes a difference whether the syntax error is at the beginning, in the middle, or at the end of the sequence. However, we do not observe a clear pattern, which suggests that the error location does not have a significant impact on the prediction accuracy.



6: Boxplots that show the length distribution across all samples and across the incorrectly predicted samples.

Now that we have an idea on how accurate the predictions of the models are, we want to take a closer look on samples that provide interesting results. An interesting pattern we observe the results from the evaluation data is that if the expected fix token is 'LIT' or 'ID', the predicted fix token sometimes uses the other one, e.g., if the expected token is 'LIT' in some cases the models predict 'ID' and vice versa. This makes sense, since in a lot of places in source code one can use literals or identifiers in the form of variables interchangeably, at least syntactically speaking.

Another interesting aspect we observe during evaluation is that at the beginning and at the end of the code, the predicted fix type and fix token sometimes differs from what is expected from the evaluation data, but is still syntactically correct. The differences here are that one wants to modify a token at the very beginning or at the very end to '\n', while the other simply wants to delete this token or vice versa. Again syntactically speaking both methods are equivalent since a newline at the beginning or the end of the code does not matter. Interestingly, we notice that our models tend to delete the token rather than changing it to a '\n' if the code already has another '\n' at the beginning, so the model prefers one newline (if any) at the start of the code as opposed to two newlines.

The previously explained observations show that the predicted results are often still valid fixes for the given syntax error, even though they do not necessarily match the expected ones. Considering this the accuracy for the 'all_models' case in Fig. 5 would be even higher when we account for this. Since this does not effect the 'syntax' accuracy, it explains the difference in accuracy between the 'all_models' and 'syntax' case.

REFERENCES

[1] https://www.spyder-ide.org/
[2] https://docs.python.org/3/library/keyword.html
[3] https://docs.python.org/3/library/operator.html
[4] Sepp Hochreiter, Jürgen Schmidhuber: Long Short-Term Memory. Neural Comput. 9(8): 1735-1780 (1997)
[5] Martin Sundermeyer, Ralf Schlüter, Hermann Ney: LSTM Neural Networks for Language Modeling. INTERSPEECH 2012: 194-197
[6] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
[7] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html