

# Towards Intelligent Error Prediction and Rectification in IDEs

Pritish Kishore Kumar

## 1 Introduction

Code prediction is a technology that is primarily used in autocompletion techniques in intelligent IDEs. Autocomplete is very useful to developers as it serves to ease their effort in writing good code, one important use case being the prediction of the next code token that the developer could potentially use. Another use case is that it does away with the requirements for developers to remember API calls and function names.[2]

However, while autocompletion serves to ease a developer's task of writing good code, this unfortunately only applies to typing syntactically or contextually correct code. If the code being typed is contextually or syntactically incorrect, then the autocomplete feature obviously doesn't offer up any suggestions. Some IDEs however offer some intuitive indicators towards such errors such as underlining the error in red etc or offering a very rudimentary tooltip informing the user that a symbol or token can't be resolved. This is not as helpful or intuitive as it's supposed to be.

We propose to enhance these error rectification mechanisms with AI enabled suggestions. For example:

- The current error tooltip is rudimentary at best with very little information about what could possibly cause the error. This can be enhanced with either context based suggestions. For example in the Java instantiation `FirebaseOptions options = new fb`. The current tooltip would only suggest that it couldn't resolve the symbol 'fb'. Instead it could be enhanced to atleast suggest the token 'FirebaseOptions' (or any other suggestion from the current context). Suggestions could be based on context (after the token 'new' only a new instance of 'FirebaseOptions' is applicable and should be suggested as a fix.
- Another intelligent error rectification suggestion is to suggest StackOverflow links intelligently. Imagine that a program/project is syntactically correct and compiles correctly. However, it produces a runtime error. The basic instinct is to check the Stacktrace and Google it, which would most probably result in Stack Overflow links. We propose to have a tooltip which will either appear on hovering/select a portion of the stacktrace and right-click (maybe?) which would intelligently suggest StackOverflow links. This would require mining of StackOverflow links.

## 2 Context based suggestions and Transformer based result ranking

### 2.1 Limitations

As mentioned previously, the most popular autocomplete techniques utilize popular machine learning models such as Type-based inference or sequential Recurring Neural Network (SeqRNN) ones. However, this paper points major limitations in the those two: Type-based models such as Jedi<sup>1</sup> or Eclipse<sup>2</sup> recommended and rank

the next token in a very naive way (usually alphabetically), while their research has showed very clearly that popular machine learning based code prediction models such as SeqRNN recommends the next token in such a way that the correct result is present in approximately the top 2.7 results, leaving a definite margin for improvement.[2]

### 2.2 Improvements

While the SeqRNN ML model yields impressive results in terms of next token recommendation, this paper's research found that using Transformer-based models for code prediction (instead of RNN) improved the ranking statistics: Correct results were displayed to developers within the first 1.5 - 2 results, an improvement of about 0.5 to 1 ranks higher.[2]

A Transformer is a machine learning model first introduced by **GoogleBrain**.<sup>3</sup> It is a type of model which processes input as a whole (for instance an entire sentence) and does not depend on past hidden states to retain past information (like Recurrent Neural Networks (RNN) do).[4] This has the advantage that Transformers don't risk losing past information.

An abstract syntax tree (AST) is generally defined as the abstract syntactic representation of source code.[1] Within this representation, each node of the AST represents a construct/token of the source code. The research in this paper postulates the above mentioned improvement in accuracy by feeding ASTs into Transformers. However, since Transformers are sequential models, only partial structures of the AST can be fed in at one point in time. Their solution to this was to explore three different methods to this end:[2]

- Decompose source code into partial programs and consequently each partial program into a sequential stream of source code tokens (SeqTrans)
- Decompose the AST into multiple paths and subsequently feed each path into the Transformer (PathTrans)
- Decompose the AST into its various traversal orders and subsequently feed each traversal into the Transformer (TravTrans)

### 2.3 SeqTrans

Since a Transformer is designed to take only sequential input, the input for this particular method works out of the box with the Transformer. A partial program (which is effectively a sequence of source code tokens) is fed as input to the Transformer, which in turn is expected to yield a single source code token as output. This method was already proven to be better than SeqRNN, as well as provided a platform for the following two AST based methods.

### 2.4 PathTrans

This method works on the concept of root paths: A root path is the path from a leaf node of an AST to its root. Such an example is depicted in Fig 2 If an AST has multiple leaf nodes, there will be multiple root paths. Since each root path forms a sequence of internal nodes of the AST to each leaf token, it would ideally capture all the syntactical information along this path. Consequently, the

<sup>1</sup><https://github.com/davidhalter/jedi>

<sup>2</sup>[https://www.eclipse.org/pdt/help/html/working with code assist.htm](https://www.eclipse.org/pdt/help/html/working%20with%20code%20assist.htm)

SE4AI WiSe'21, Winter Semester 2021, Darmstadt, Germany  
2021.

<sup>3</sup><https://research.google/teams/brain/>



**Figure 1.** Rootpath concept



**Figure 2.** AST as input to TravTrans

root paths help in leaf token prediction when they are fed into the Transformer.

## 2.5 TravTrans

This method works on the concept of tree traversals;ie;feeding a sequence of nodes of the AST in a certain tree traversal order. The popular tree traversal methods include pre-order, post-order and in-order and can be applied to tree structures such as AST. The method involves feeding code token sequences which are obtained from pre-order traversals (in Depth First Search order since pre-order traversal is a kind of DFS) of the AST to optimize leaf token prediction. Fig 3 represents an AST. If we consider node 29 for example (atoi), the the node chain in DFS order preceding it would be:

**Call -> NameLoad -> Map -> AttributeLoad -> NameLoad -> string -> Attr**

## 3 Implementation and Datasets

The project mainly used two types of datasets:[2]

- External dataset comprising of 150k Python 2 source code files across several external GitHub repositories decomposed into ASTs. This resulted in a overall count of roughly 16 million leaf nodes.
- Internal dataset from GitHub repositories internal to Facebook. These also consisted of similar Python 2 source code files decomposed into ASTs. However, since the coding style was distinct, the main aim here was to ascertain consistency in prediction across disjoint projects. This also resuted in additional 1 million leaf nodes.

As for the implementation of the Transformers(SeqTrans, PathTrans and TravTrans), the project mainly used the GPT-2 small

implementation[3] based on the PyTorch library<sup>4</sup>. The schematic of a GP2 Transformer is depicted in Fig 4



**Figure 3.** GP2 Transformer

<sup>4</sup><https://github.com/graycode/gpt-2-Pytorch>

## 4 Result

The token prediction results are generally expressed in terms of **mean reciprocal rank** percentages. As mentioned earlier, for typical RNN based methods, this percentage is about **37**. However, their most efficient method, **TravTrans**, which functions on feeding AST tree traversal sequences to Transformers, was found to increase this percentage by about **18**. Since 37 roughly equates to the correct token being present in the first 2.7 results, this 18 percent increase meant that now the correct token could be present anywhere between the first 1.5 and 2 results.

## References

- [1] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. 2010. Code Comparison System based on Abstract Syntax Tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. 668–673. <https://doi.org/10.1109/ICBNMT.2010.5705174>
- [2] et al. Kim, Seohyun. 2021. Code prediction by feeding trees to transformers. *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021).
- [3] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
- [4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).